```
import random
```

```
#initialization
values  = [' ' for _ in range(9)]
freeLoc = [True for _ in range(9)]
player1, p1S = input("Enter Player name: "), input("Enter Symbol: ")
sym = {player1 : p1S, 'system' : 'X' if p1S != 'X' else 'O'}
player = [player1, "system"]
playerMoves = {player1 : [] , 'system' : []}
flag = False
```

```
Enter Player name: Ajith
Enter Symbol: X
```

```
def printT(state):
    m=3
    n=3
    print('\t '+'_____ '*m)
    for i in range(0,n):
      print('\t'+'|'+'     |'*m)
      print('\t'+'|'+('  {}  |'*m).format(state[i*m],state[i*m+1],state[i*m+2]))
      print('\t'+'|'+'_____|'*m)

printT(values)
```

```
         _____ _____ _____
        |     |     |     |
        |     |     |     |
        |_____|_____|_____|
        |     |     |     |
        |     |     |     |
        |_____|_____|_____|
        |     |     |     |
        |     |     |     |
        |_____|_____|_____|
```

```
cur_player = random.choice(player)
if cur_player == 'system':
  flag = True
cur_player
```

```
'system'
```

```
def game():
  if cur_player != 'system':
    takepos()
  else:
    print("Its System Turn, Thinking...")
    myalg()
def takepos():
  pos = int(input("Its {} turn. Enter the position:(press -1 to quit) ".format(player[0])))
  if pos == -1:
    return
  check(pos)
```

```
#handle the position taken
def check(pos):
    if cur_player != "system":
        if pos>9 or pos <1 :
          print("Error location. Try Again")
          takepos()
        if values[pos-1] != ' ':
          print("The position is occupied, enter again")
          takepos()
        else:
          setDetails(pos)
```

```
#to change player
def changeturn(cp):
  global cur_player
  if cp == 'system':
    cur_player  = player[0]
  else:
    cur_player = player[1]
```

```
#Algorithm to counter player moves
def myalg():
  opponent = player[0] if cur_player == 'system' else player[1]
  free = [i+1 for i in range(len(freeLoc)) if freeLoc[i]]
  edge = {1:[2,4],3:[2,6],7:[4,8],9:[6,8]}
  seq = {'mid':5,'edges':[1,3,7,9],'inbet':[2,4,6,8]}
  if values[5-1] == ' ':
    pos = 5
    setDetails(pos)
  else:
    lastMov = playerMoves[opponent][-1]
    Apos = aboutTowin()
    if lastMov in seq['inbet']:
      if lastMov in [2,8]:
        pos = lastMov+1 if lastMov+1 in free else lastMov-1
      else:
        pos = lastMov+3 if lastMov+3 in free else lastMov-3
      if Apos != -1:
        pos = Apos
      setDetails(pos)
    else:
      for i in edge[lastMov]:
        if i in free:
          pos = i
          if Apos != -1:
            pos = Apos
          setDetails(pos)
```

```
#To save the position and update the details
def setDetails(n):
    playerMoves[cur_player].append(n)
    freeLoc[n-1] = False
    values[n-1] = sym[cur_player]
    printT(values)
    win = checkWin([i+1 for i in range(len(freeLoc)) if freeLoc[i]])
    if not win:
        changeturn(cur_player)
        game()
    return
```

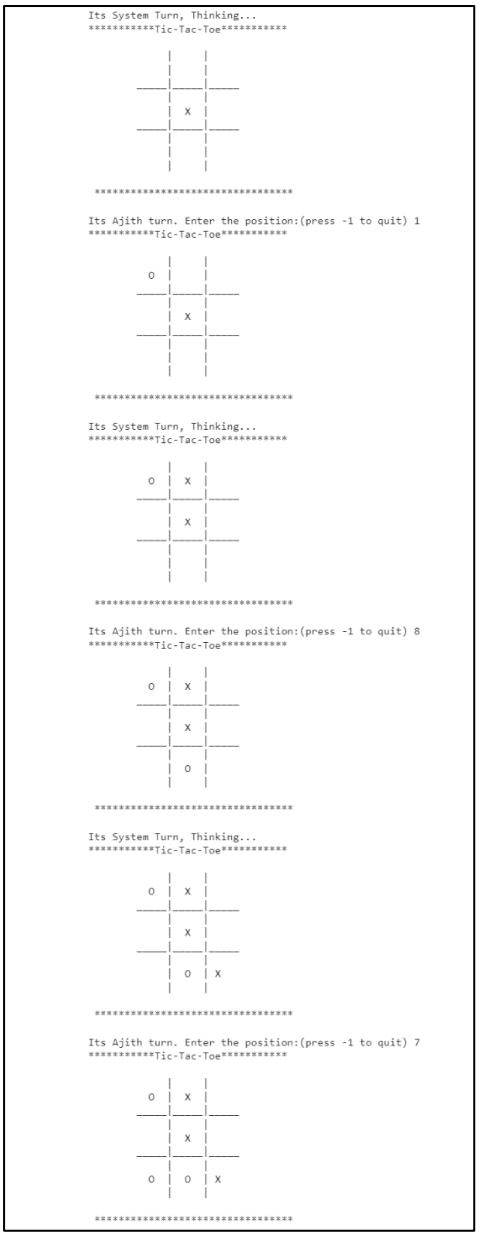
```
#to check win state
def checkWin(free):
    winCond = [[1,2,3],[4,5,6],[7,8,9],[1,4,7],[2,5,8],[3,6,9],[1,5,9],[3,5,7]]
    for i in winCond:
        wF = True
        for j in i:
            if j not in playerMoves[cur_player]:
                wF = False
        if wF:
             print("{} has Won!!".format(cur_player))
             return True
    if len(free) == 0:
        print("Its a tie....")
        return True
    return False
```

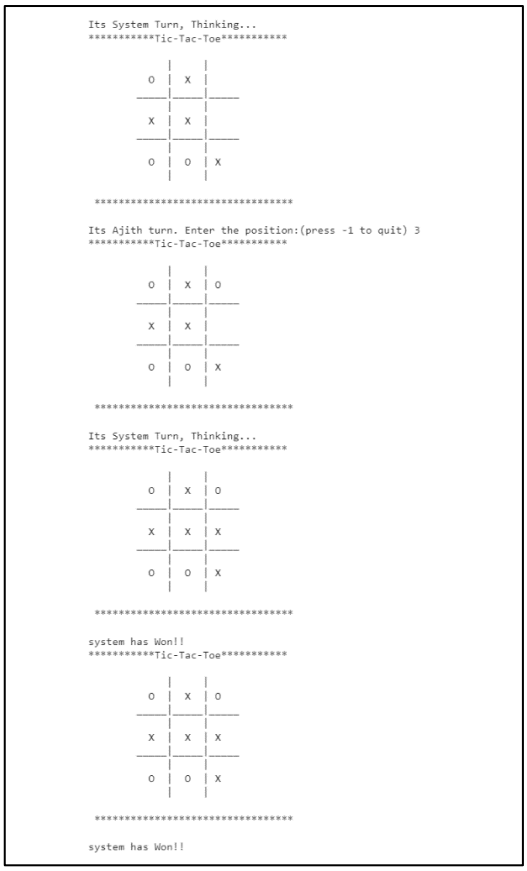
```
#to check if close to win by one move Or if the player is about to win in one move and counter it
def aboutTowin():
  winCond = [[1,2,3],[4,5,6],[7,8,9],[1,4,7],[2,5,8],[3,6,9],[1,5,9],[3,5,7]]
  free = [i+1 for i in range(len(freeLoc)) if freeLoc[i]]
  lastP = -1
  sysP = -1
  for i in winCond:
    sysP = set(i) - set(playerMoves['system'])
    if len(sysP) == 1:
        sysP = sysP.pop()
        if sysP and ( sysP in free ):
            return sysP

  for i in winCond:
    lastP = set(i) - set(playerMoves[player1])
    if len(lastP) == 1:
        lastP = lastP.pop()
        if lastP and lastP in free:
            return lastP
  return -1
```

```
changeturn(cur_player)
print(cur_player)
```

```
system
```

```
game()
```


Its System Turn, Thinking...
**********Tic-Tac-Toe**********
Its Ajith turn. Enter the position:(press -1 to quit) 1
**********Tic-Tac-Toe**********
Its System Turn, Thinking...
**********Tic-Tac-Toe**********
Its Ajith turn. Enter the position:(press -1 to quit) 8
**********Tic-Tac-Toe**********
Its System Turn, Thinking...
**********Tic-Tac-Toe**********
Its Ajith turn. Enter the position:(press -1 to quit) 7
**********Tic-Tac-Toe**********


Its System Turn, Thinking...
**********Tic-Tac-Toe**********
Its Ajith turn. Enter the position:(press -1 to quit) 3
**********Tic-Tac-Toe**********
Its System Turn, Thinking...
**********Tic-Tac-Toe**********
system has Won!!
**********Tic-Tac-Toe**********
system has Won!!

```
from collections import defaultdict
class Graph:
    def __init__(self, verticies):
        self.V = verticies
        self.graph = defaultdict(list)
    def addEdge(self,u,v):
            self.graph[u].append(v)
    def printG(self):
        for i in self.graph:
            print(i, "-->",self.graph[i])
    def DLS(self, src, target, maxDepth):
        if src == target : return True
        if maxDepth <=0 : return False
        for i in self.graph[src]:
            if(self.DLS(i, target, maxDepth-1)):
                return True
        return False
    def IDDFS(self, src, target, maxDepth):
        for i in range(1,maxDepth+1):
            if (self.DLS(src, target, i)):
                return True
        return False
```

```
NoOfVertices = int(input("Enter the number of vertices : "))
dirGraph =  Graph(NoOfVertices)
for _ in range(NoOfVertices):
    p, n = map(int,(input("Enter the Parent Node No and New Node No :")).split())
    dirGraph.addEdge(p,n)
```

```
Enter the number of vertices : 10
Enter the Parent Node No and New Node No :0 1
Enter the Parent Node No and New Node No :0 2
Enter the Parent Node No and New Node No :1 3
Enter the Parent Node No and New Node No :1 4
Enter the Parent Node No and New Node No :1 5
Enter the Parent Node No and New Node No :2 6
Enter the Parent Node No and New Node No :2 7
Enter the Parent Node No and New Node No :3 8
Enter the Parent Node No and New Node No :3 9
Enter the Parent Node No and New Node No :5 10
```

```
target, maxDepth = map(int, input("Enter the target, maxDepth: ").split())

if dirGraph.IDDFS(0, target, maxDepth) == True:
    print("Solution exists within the given search depth ")
else:
    print("Not within the depth")
```

```
Enter the target, maxDepth: 4 2
Solution exists within the given search depth
```

```
import copy
class child:
  def __init__(self, problem, parent, action):
    self.STATE = problem.RESULT(parent.STATE, action) if parent else problem
    self.PARENT = parent
    self.ACTION = action

class problem:
  def __init__(self,state,goal,actions):
    self.INITIAL_STATE = state
    self.GOAL = goal
    self.ACTIONS = actions

  def GOAL_TEST(self, state):
    return (state == self.GOAL)

  def RESULT(self,state,action):
    return self.swap(copy.deepcopy(state),state.index('O'),action)

  def swap(self,st,src,des):
    st[src], st[des] =st[des], st[src]
    return st
```

```
state = [1,2,3,4,7,5,6,'O',8]
goal = [1, 2, 3, 4, 5, 6,7,8 ,'O']
actions = {'l': -1,'r': +1,'u': -3,'d': +3}
P = problem(state,goal,actions)
```

```
def DLS(src, problem, maxDepth):
    if problem.GOAL_TEST(src.STATE) : return True
    if maxDepth <=0 : return False
    moves = [src.STATE.index("O") + problem.ACTIONS[a] for a in problem.ACTIONS if src.STATE.index("O") + problem.ACTIONS[a] in list(range(0,9)) ]
    for j in moves:
        if(DLS(child(problem,src,j), problem, maxDepth-1)):
            return True
    return False

def IDDFS(src,problem, maxDepth):
    for i in range(1,maxDepth+1):
        if (DLS(src, problem, i)):
            return True
    return False

def printB(state):
    m=3
    n=3
    print('\t '+'_____ '*m)
    for i in range(0,n):
      print('\t'+'|'+'     |'*m)
      print('\t'+'|'+('  {}  |'*m).format(state[i*m],state[i*m+1],state[i*m+2]))
      print('\t'+'|'+'_____|'*m)

def printSOL(f):
    if f.PARENT:
        printSOL(f.PARENT)
    else:
        return None
    printB(f.STATE)
```

```
maxDepth = int(input("Enter MAX depth: "))
src = child(state,None,None)
if IDDFS(src,P,maxDepth):
  print("\nSolution Exists!!")
  #t=finalANS
  #printSOL(t)
else:
  print("\nSolution doesn't exits in the given depth!!!")
```

```
Enter MAX depth: 10

Solution Exists!!
```

```
import copy
class child:
  def __init__(self, problem, parent, action):
    self.STATE = problem.RESULT(parent.STATE, action) if parent else problem
    self.PARENT = parent
    self.ACTION = action
    self.PATHCOST = ( parent.PATHCOST + 1 ) if parent else 0
    self.TOTALCOST = parent.PATHCOST + problem.STEPCOST(self.STATE) if parent else 0

class problem:
  def __init__(self,state,goal,actions):
    self.INITIAL_STATE = state
    self.GOAL = goal
    self.ACTIONS = actions

  def hFUNC(self, state):
    return sum([1 for i in range(1,9) if state.index(i)!=self.GOAL.index(i)])

  def GOAL_TEST(self, state):
    if state == self.GOAL:
      return True
    else:
      return False

  def RESULT(self,state,action):
    return self.swap(copy.deepcopy(state),state.index('O'),action)

  def STEPCOST(self,state):
    return self.hFUNC(state)

  def swap(self,st,src,des):
    st[src], st[des] =st[des], st[src]
    return st
```

```
state = [1,3,5,4,7,6,8,'O',2]
goal = [1, 2, 3, 4, 5, 6,7,8 ,'O']
#actions = {'l': [0,-1],'r': [0,+1],'u': [-1,0],'d': [+1,0]}
actions = {'l': -1,'r': +1,'u': -3,'d': +3}
P = problem(state,goal,actions)
```

```
def aStar(state,problem):
  global finalANS
  visited=[]
  solution=False
  if problem.GOAL_TEST(state):
        finalANS = newChild
        solution=True
  while not solution:
    if frontier == []:return False
    else:
      node = frontier.pop(Min(frontier))
      visited.append(node.STATE)
      #printB(node.STATE)
      src = node.STATE.index("O")
      moves = [src + actions[a]  for a in actions ]
      possible = list(range(0,9))
      possibleMoves = [i for i in moves if i in possible]
      for i in possibleMoves:
        newChild = child(problem,node,i)
        if newChild.STATE not in visited:
          frontier.append(newChild)
          #print(newChild.STATE, newChild.PATHCOST, newChild.TOTALCOST, newChild.PARENT.PATHCOST)
          if problem.GOAL_TEST(newChild.STATE):
            solution=True
            finalANS = newChild
            return True

def Min(F):
  min=0
  for j in range(0,len(frontier)):
    if frontier[j].TOTALCOST <= frontier[min].TOTALCOST:
      min=j
  return min

def printB(state):
    m=3
    n=3
    print('\t '+'_____ '*m)
    for i in range(0,n):
      print('\t'+'|'+'     |'*m)
      print('\t'+'|'+('  {}  |'*m).format(state[i*m],state[i*m+1],state[i*m+2]))
      print('\t'+'|'+'_____|'*m)

def printSOL(f):
    if f.PARENT:
        printSOL(f.PARENT)
    else:
        return None
    printB(f.STATE)
```

```
frontier = [child(state,None,None)]
finalANS=[]
visited=[]
if aStar(state,P):
  print("\nSolution Exists: as Before")
  t=finalANS
  printSOL(t)
else:
  print("\nSolution doesn't exits!!!")
```

Solution Exists: as Before

1	3	5
4	7	6
8	2	0

1	3	5
4	7	0
8	2	6

1	3	0
4	7	5
8	2	6

1	0	3
4	7	5
8	2	6

0	1	3
4	7	5
8	2	6

4	1	3
0	7	5
8	2	6

4	1	3
7	0	5
8	2	6

4	1	3
7	2	5
8	0	6

4	1	3
7	2	5
0	8	6

4	1	3
0	2	5
7	8	6

0	1	3
4	2	5
7	8	6

1	0	3
4	2	5
7	8	6

1	2	3
4	0	5
7	8	6

1	2	3
4	5	0
7	8	6

1	2	3
4	5	6
7	8	0

```
import copy
class child:
  def __init__(self, problem, parent,statePointer, action):
    self.STATE, self.POINTER  = problem.RESULT(copy.deepcopy(parent.STATE),parent.POINTER, action)  if parent else ( problem, startPointer )
    self.PARENT = parent
    self.ACTION = action
    self.PATHCOST = ( parent.PATHCOST + 1 ) if parent else 0
    self.TOTALCOST = self.PATHCOST + problem.STEPCOST(self.POINTER) if parent else 0

class problem:
  def __init__(self,state,goalP,actions):
    self.INITIAL_STATE = state
    self.GOALP = goalP
    self.ACTIONS = actions

  def hFUNC(self, stateP):
    return ((sum((stateP[i]-self.GOALP[i])**2 for i in range(2)))**.5)

  def GOAL_TEST(self,stateP):
    if stateP == self.GOALP:
      return True
    else:
      return False

  def RESULT(self,st,sP,action):
    if action != self.GOALP:
        st[action[0] * 4 + action[1]] ="."
    sP = action
    return st,sP

  def STEPCOST(self,stateP):
    return self.hFUNC(stateP)
```

```
def aStar(statePointer,problem):
  global finalSOL
  global visited
  solution=False
  if problem.GOAL_TEST(statePointer):
        solution=True
        finalSOL = problem.INITIAL_STATE
  while not solution:
    if frontier == []:return False
    else:
      node = frontier.pop(Min(frontier))
      visited.append(node.POINTER)
      #print("Visited ones: " , visited)
      #printB(node.STATE)
      src = node.POINTER
      moves = [[src[0]+actions[a][0], src[1]+actions[a][1] ] for a in actions ]
      possible = list(range(m))
      possibleMoves = [i for i in moves if i[0] in possible and i[1] in possible and node.STATE[i[0] * m + i[1]] != "*" ]
      #print(possibleMoves)
      for i in possibleMoves:
        newChild = child(problem,node,node.POINTER,i)
        if newChild.POINTER not in visited:
          frontier.append(newChild)
          #print("child POINTER = ",newChild.POINTER,"parent POINTER = ",newChild.PARENT.POINTER, "Path Cost= ", newChild.PATHCOST," Total cost path= "
          if problem.GOAL_TEST(newChild.POINTER):
            solution=True
            finalSOL = newChild
            return True

def Min(F):
  min=0
  for j in range(0,len(frontier)):
    if frontier[j].TOTALCOST <= frontier[min].TOTALCOST:
      min=j
  return min

def printB(state):
    print('\t '+'_____ '*m)
    for i in range(0,n):
      print('\t'+'|'+'     |'*m)
      print('\t'+'|'+('  {}  |'*m).format(state[i*m],state[i*m+1],state[i*m+2],state[i*m+3]))
      print('\t'+'|'+'_____|'*m)
```

```
import random
random.seed(3)
n,m = 4,4
state = [" "]*(n*m)
for i in range(6):
  state[random.randint(0,n*m - 1)] = "*"
state[0]="S"
state[3]="D"
printB(state)
actions = {'l': [0,-1],'r': [0,+1],'u': [-1,0],'d': [+1,0],'ul': [-1,-1],'ur': [-1,+1],'dl': [+1,-1],'dr': [+1,+1]}
startPointer = [[i // m,i % m] for i in range(len(state)) if state[i] == 'S' ][0]
goalPointer = [[i // m,i % m] for i in range(len(state)) if state[i] == 'D' ][0]
P = problem(state,goalPointer,actions)
```

```
        _____ _____ _____ _____
       |     |     |     |     |
       |  S  |     |  *  |  D  |
       |_____|_____|_____|_____|
       |     |     |     |     |
       |  *  |     |     |  *  |
       |_____|_____|_____|_____|
       |     |     |     |     |
       |     |     |     |  *  |
       |_____|_____|_____|_____|
       |     |     |     |     |
       |     |     |     |  *  |
       |_____|_____|_____|_____|
```

```
frontier = [child(state,None,startPointer,None)]
visited=[]
finalSOL=[]
if aStar(startPointer,P):
  print("\nSolution Exists: as followed")
else:
  print("\nSolution doesn't exits!!!")

printB(finalSOL.STATE)
print("Total COSTPATH = ", finalSOL.TOTALCOST)
```

```
Solution Exists: as followed
        _____ _____ _____ _____
       |     |     |     |     |
       |  S  |     |  *  |  D  |
       |_____|_____|_____|_____|
       |     |     |     |     |
       |  *  |  .  |  .  |  *  |
       |_____|_____|_____|_____|
       |     |     |     |     |
       |     |     |     |  *  |
       |_____|_____|_____|_____|
       |     |     |     |     |
       |     |     |     |  *  |
       |_____|_____|_____|_____|
Total COSTPATH =  3.0
```

```
state = ["*V", "*"]
vPointer = 0
def simpleAgent(vPointer):
    i=8
    while i:
        if state[vPointer].startswith("*"):
            state[vPointer]="V"
            print("CLEAN")
        elif state[vPointer].startswith("V"):
            if vPointer == 0:
                state[vPointer]=""
                vPointer += 1
                state[vPointer]= state[vPointer] + "V"
                print("MOVE RIGHT")
            elif vPointer == 1:
                state[vPointer]=""
                vPointer -= 1
                state[vPointer]= state[vPointer] + "V"
                print("MOVE LEFT")
        i -= 1
        print(state)
```

```
print(state)
simpleAgent(vPointer)
```

```
['*V', '*']
CLEAN
['V', '*']
MOVE RIGHT
['', '*V']
CLEAN
['', 'V']
MOVE LEFT
['V', '']
MOVE RIGHT
['', 'V']
MOVE LEFT
['V', '']
MOVE RIGHT
['', 'V']
MOVE LEFT
['V', '']
```

```
state = ["*", "*V"]
vPointer = 1

class modelBased:
    def __init__(self,state,vPointer):
        self.STATE = state
        self.ACTIONS = []
        self.MODEL = []
        self.STATEBOOL = [-1,-1]
        self.vPointer = vPointer

    def UPDATE_STATE(self,status, vPointer):
        self.STATE[vPointer] = status

    def WORKING(self,status,vPointer):
        self.UPDATE_STATE(status,vPointer)
        if  self.STATE[vPointer].startswith("*"):
            self.STATE[vPointer]="V"
            state[vPointer]="V" #syncing with env
            self.STATEBOOL[vPointer] += 1
            print("CLEANED ",self.STATE,state)
        elif  self.STATE[vPointer].startswith("V"):
            if vPointer == 0:
                self.STATE[vPointer]=""
                state[vPointer]=""   #syncing with env
                vPointer += 1
                self.STATE[vPointer] += "V"
                state[vPointer]+= "V"  #syncing with env
                print("MOVE RIGHT")
            elif vPointer == 1:
                self.STATE[vPointer]=""
                state[vPointer]=""   #syncing with env
                vPointer -= 1
                self.STATE[vPointer] += "V"
                state[vPointer] += "V" #syncing with env
                print("MOVE LEFT")
        if  sum([i for i in self.STATEBOOL]) < 0:
            print(state[vPointer],vPointer)
            return self.WORKING(state[vPointer],vPointer)
        else:
            return True
```

```
MB = modelBased([""]*2,vPointer)
s = state[vPointer]
if MB.WORKING(s,vPointer):
    print("Work Finished!!")
else:
    print("Some Error!!")
```

```
CLEANED  ['', 'V'] ['*', 'V']
V 1
MOVE LEFT
*V 0
CLEANED  ['V', ''] ['V', '']
Work Finished!!
```