BFS
1) start
2) Read start node and graph from user
3) visited = [ ], queue=[ ]
4) append startnode to queue
5) while (queue not empty)
5.1) node = queue. pop (0)
5.2) if node is visited
5.2.1) visited. add (node
5.2.2) queue. append (node)
6)print visited
7) stop

DFS
1) start
2) Stack = [ ]
3) for each vertex v, set visited [u] = false
4) push (s, v)
5)while (s is not empty) do
u = pop(s)
if (not visited [u]) then visited [u] = true
for each unvisited neighbor u of v
push (sw)
end if
end while
6) stop

4 Queens
1) start
2) def function solve NQueens ( )
2.1) col = set( )
2.2) pos Diag = set( )
2-3) neg Diag = set( )
2.4) res = [ ]
2.5) board = [["."] * n for i in range(n)]
3) def function backtrack (r):
3.1) if r==n:
copy = [" ". join (row) for row in board]
res append (copy)
return
3.2) for c in range (n):
if c in col or (r+c) in posdiag or (r-c)
in negDiag:
continue
col.add (c)
posDiag. add (r+c)
neg Diag.add (r-c)
backtrack (r+1)
col. remove (c)
pos Diag .remove (r+c)
neg diag.remove (r-c)
board [r] [c] = ". "
backtrack [0]
4) return res.

DIJKSTRA'S ALGORITHM
1) start
2) function dykstras (G,S)
for each vertex vin G distance [v] <-infinite
previous [V] <-NULL
if v!=s, add v to priority Queue Q.
distance [s] <-0
while Q IS NOT EMPTY
u<- Extract MIN from Q.
for each unvisited neighbour v of u
temp Distance<- distance [u]+ edge -
weight (U,V)
if temp Distance < distance [v]
distance [V] <-temp Distance
previous [v] <-u
return distance [], previous []
3) stop

WATER JUG
1)Water Jug DFS (initial state, sole, Max1, Max2):
visited= set( )
queue = dequeue( )
queue.append((initial_state, [ ]))
while queue
current-state, actions =queue. popleft( )
jug1, jug2 = current state
if jug1 = sol or jug2 = sol
return actions
if current state in visited:
continue
visited add current state to visited
2) # file Jug 1
queue.append (Max1,Jug2), action + ["fill Jug1"])
3) # fill Jug2
queue.append (Jug 1,Max2), action + ["Fill Jug 2"])
4) # transform from jug1 to jug 2
pour_amount = min (Jug1, Max2, jug 2)
queue.append ((jug1_pour_amount, jug2 + pour amount), actions + [Pour from jug1 to jug2].
5) # transfer from jug 2 to jug1
pour_amount = min (jug 2, max1, jug1)
queue. append (jug1 + pour_amount, jug 2_pour amounts), actions + [Pour from jug2 to jug1].
6) # Empty first jug
queue.append ((D, jug2), action + [' Empty Jug1'])
7) # Empty Second jug queue append (jugi, 0), actions ['Empty Jug 2'])
8) return None.
9) Main ( )
10) Input Maximum values for both the jugs.
11) Input Initial values for both jugs.
12) Enter for goal value.
13) Initial state = (initial_jug1, initial_jug2)
14) Solution = water_jug_dfs (initial_state, sol, Max1, Max 2)
15) if solution:
print ("Actions to measure" + str [sol]+" litres of water.")
for action in solution:
print (action)
elif (initial_jug1= sol or initial-jug 2 = sol)
if initial_jug1= sol
print("Jug1 is filled correctly")
else
print ("Jug 2 is filled correctly")
else
print (" No solution found")

8 puzzle heuristic
1) start
2) Create an open set containing the initial puzzle state
3) Initialize the come from map to track navigated nodes.
4) Initialize gscore and fscore dictionaries for tracking costs
5) while the open set is not empty
5.1) Select the node with lowest fscore value from the open set
5.2) if the current node is the goal state then reconstruct the path and return it
5.3) Remove, the current node from the open set.
5.4) Explore neighbour and update their scores if a better path is formed
6) Reconstruct path from the come from map starting from the goal state
7) Define the heuristic function, such as manhatten distance, to estimate the cost from the current state to the goal state
8) Calculate the manhatten distance between a title and its correct position
9) stop.

ALPHA BETA PRUNING.
1)Start
2) minimax (depth, Node. Index, max in P, values, alphabeta, max)
3) if depth == max depth then
3.1) return values [Node Index]
4) if max in P THEN
4.1) best = MIN
4.2, for i in range (2) Do
4.2.1) val = mini max (depth +1, Node Index=2+i, True, values, alphabeta,, max)
4-3) best = max (best, val)
4.4) alpha = max (best, alpha)
4.4.1) if beta <_alpha THEN
4-4-2) break
4.5) end for
4.6) return best
5) else
5.1) best = MAX
5.2) for i in range (2) Do
5.2.1) val = mimimax (depth +1), Node Index=2+i, True, values, alpha, beta)
5.4) beta = min (best, value)
5.5) beta = min (best, beta)
5.5) if beta <= alpha then
break
5.6) end for
5.7)return best
stop

8 PUZZLE PROBLEM USING DFS.
Input: Initial and goal state
Output: steps of transition from initial to goal state
1) start
2) function DFS (intial_state):
create a stack
push initial_state into stack
create a set to keep track of visited States.
while the stack #empty:
current Node = pop the node from stack.
add current Node state to visited[]
if current_node.state = goal state:
return path from initial to currentNode
successors = generate successor (Curr Node)
for each successor in successors:
if successor. state not in visited[]
push successor to stack
return NULL.
3) function generate_successor (node):
successors = empty list
empty_index = index(empty tile in node. state)
move = [(0,1), (0,-1), (1,0), (-1,0)]
for each move (dx, dy) in moves:
x,y = coordinates of empty tile after the move.
if x and y are within the puzzle:
new state = copy of node state
swap empty titee and title
at (x,y) in new_state.
successors. append (new Puzzle Node(
new-state, parent = node,
action = (dx, dy)))
return successors
class Puzzle Node
state, parent, action
function is_goal (state):
return state is sorted
if --name--=" main--":
initial_stale = [1,2,3,4,5,0,6,7,8]
Solution = DFS (initial_state)
if solution is not null:
print(" Solution found")

TIKTAKTOK
1. Start
Initialize the board with 9 empty positions.
2. Print Board (Function)
Display the current state of the board.
2.1. Print the first row: board[1] | board[2] | board[3].
2.2. Print a separator: -+-+-.
2.3. Print the second row: board[4] | board[5] | board[6].
2.4. Print a separator: -+-+-.
2.5. Print the third row: board[7] | board[8] | board[9].
3. Check if Space is Free (Function)
Check if a given position on the board is empty.
3.1. If board[position] == '':
        Return True.
3.2. Else:
        Return False.
4. Insert Letter (Function)
Place a player's symbol ('X' or 'O') in the specified position.
4.1. If spaceIsFree(position):
   4.1.1. Assign board[position] = letter.
   4.1.2. Print the updated board.
   4.1.3. If checkDraw():
        Print "DRAW" and exit.
   4.1.4. If checkWin():
        If letter == 'X':
            Print "BOT WINS" and exit.
        Else:
            Print "PLAYER WINS" and exit.
4.1.5. Return.
4.2. Else:
   4.2.1. Print "Incorrect position".
   4.2.2. Ask the user for a new position and retry.
5. Check for Win (Function)
Check if there is a winning combination on the board.
5.1. If any of the following combinations are met:
   - Horizontal: [1,2,3], [4,5,6], [7,8,9].
   - Vertical: [1,4,7], [2,5,8], [3,6,9].
   - Diagonal: [1,5,9], [7,5,3].
And the positions are not empty:
   Return True.
5.2. Else, return False.
6. Check for Draw (Function)
Check if all positions on the board are filled.
6.1. If any position is empty:
   Return False.
6.2. Else, return True.
7. Player Move (Function)
Let the player make a move.
7.1. Input the position from the player.
7.2. Call insertLetter('O', position).
8. Computer Move (Function)
Make the best move for the computer using the Minimax algorithm.
8.1. Initialize bestScore = -∞ and bestMove = 0.
8.2. For each empty position on the board:
   - Simulate the move by assigning board[key] = 'X'.
   - Call minimax(board, depth, False).
   - Undo the move.
   - If the score is greater than bestScore, update bestScore and bestMove.
8.3. Call insertLetter('X', bestMove).
9. Minimax (Function)
Determine the optimal move for the computer.
9.1. If checkWhichMarkWon('X'): Return 1.
9.2. If checkWhichMarkWon('O'): Return -1.
9.3. If checkDraw(): Return 0.
9.4. If isMax is True:
   - Initialize bestScore = -∞.
   - For each empty position:
      - Simulate the move with board[key] = 'X'.
      - Call minimax(board, depth + 1, False).
      - Undo the move.
      - Update bestScore with the maximum value.
   - Return bestScore.
9.5. Else (Minimizing player):
   - Initialize bestScore = ∞.
   - For each empty position:
      - Simulate the move with board[key] = 'O'.
      - Call minimax(board, depth + 1, True).
      - Undo the move.
      - Update bestScore with the minimum value.
   - Return bestScore.
10. Main Function
10.1. Initialize the board as a dictionary with keys 1-9 and empty values.
10.2. While not checkWin():
   - Call computerMove().
   - Call playerMove().
11. End

for move in solution:
if move == (0,1):
print("Move Right")
else if move == (0,-1):
print("Move Left")
else if move = = (1,0):
print(" Move down")
else if move == (-1,0):
print("Move up")
else:
print ("No Solution")

TIKTAKTOK
Initialize board as a 3x3 grid of empty spaces
Set current_player to "X"
While the game is not over:
    Print the current board state
    Prompt current_player to enter row and column
    If the cell is empty:
        Place current_player's symbol in the cell
        If current_player wins:
            Print the board and announce winner
            End game
        Else if the board is full (tie):
            Print the board and announce tie
            End game
        Switch current_player ("X" -> "O" or "O" -> "X")
    Else:
        Print message that the cell is already taken