ASSIGNMENT- 1.

WAP to find addition, substraction, multiplication and division using numpy.

```
de :- import numpy as np
       V1 = np.array ([4,2,7])
       V2 = np.array ([1,3,5])
       Print ("vector addition:", np.add (V1,V2))
       print ("vector substraction:", np.sub (V1,V2))
       print ("vector multiplication:", np.multiply (V1,V2))
       print ("vector division:", np.divide (V1,V2)).
```

put :
```
vector addition : [5 5 12]
vector substraction: [3 -1 2]
vector multiplication (element-wise) : [4. 6 35]
vector division : [4. 0.66666667 1.4]
```

WAP to find matrix using numpy.

```
:- import numpy as np
   m1 = np.array ([[2,4],[3,1], [4,1]])
   m2 = np.array ([[5,6], [3,4], [2,3]])
   m_add = np.add (m1, m2)
   print ("matrix addition :\n", m_add)
   m_sub = np.sub (m1, m2)
   print ("matrix subtraction :\n", m_sub)
```

ut :-
```
matrix addition :
[[7 10]
 [6 5]
 [6 4]]
matrix subtraction :
[[-3 -2]
 [0 -3]
 [2 -2]].
```

WAP to check whether all elements in a vector are non zero or ero using np.all()?

```
e :- import numpy as np
     V1 = np.array ([4,2,7])
     print ("vector :", V1)
     V1 = np.array ([0,4,2])
     Print (np.all (V1))
```

output :- vector : [4 2 7]
False.

WAP to Create an Array of size 10 with each element of it set to value 3.

```
e :- import numpy as np
     A = np.full(10, 3)
     A.
```

tput :- array([3,3,3,3,3,3,3,3,3,3])

WAP to find the memory size of this array and its individual elements.

```
e :- my_array = np.array([1,2,3,4,5])
     print(my_array.size)
     print(my_array)
```

tput :- 5
[1 2 3 4 5].

WAP to Create an array of size 10 with values ranging from 0 to 90 evenly spaced.

de :- Without using numpy :-

```
my_array = [i*10 for i in range(10)]
my_array
```

tput :- [0, 10, 20, 30, 40, 50, 60, 70, 80, 90].

de :- using numpy :-

```
import numpy as np
array_evenly_spaced = np.linspace(0, 90, 10)
array_evenly_spaced.
```

tput :- array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90]).

Reverse the above array using numpy.

de :-

```
import numpy as np
array_evenly_spaced_reversed = np.flip(array_evenly_spaced)
array_evenly_spaced_reversed.
```

utput :- array([90, 80, 70, 60, 50, 40, 30, 20, 10, 0]).

WAP to add array A and B, store the result in array C.

```
e:- Import numpy as np
    A = np.array ([1,2,3])
    B = np.array ([4,5,6])
    C = A+B
    print ("Array A :", A)
    print ("Array B :", B)
    print ("Array C (A+B) :", C)
```

put:- Array A: [1 2 3]
Array B: [4 5 6]
Array C (A+B) : [5 7 9]

WAP to create two arrays and concatenate it.

```
e:- Import numpy as np
    V1 = np.array ([3,6,4])
    V2 = np.array ([1,5,7])
    np.concatenate ((V1,V2))
```

put:- array ([3,6,4,1,5,7]).

18/7/25

ASSIGNMENT - 02.

1 Write a program in python to perform BFS.

Code :-

```
def bfs(graph, start):
    visited = set()
    queue = [start]
    while queue:
        node = queue[0]
        queue = queue[1:]
        if node not in visited:
            print(node, end = ' ')
            visited.add(node)

            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'];
    'F': []
}
print("Breadth-First Search starting from node A:")
bfs(graph, 'A')
```

Output :-

Breadth-First search starting from node A:

A B C D E F.

Creating Depth-First Search (DFS) using adjacency list and recursion.

Code :-

```
def dfs(graph, start, visited = None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end = ' ')
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
print("Depth-First Search starting from node A:")
dfs(graph, 'A')
```

Output :-

Depth-First Search starting from node A:
A B D E F C.

25/7/25

Write a Code in Jupyter to execute TICTAETOee.

```
de :- def print_board(board):
        print(f"\n{board[0]}|{board[1]}|{board[2]}")
        print("---+---+---")
        print(f"{board[3]}|{board[4]}|{board[5]}")
        print("---+---+---")
        print(f"{board[6]}|{board[7]}|{board[8]}");
    def check_win(board, player):
        win_conditions = [[0,1,2], [3,4,5], [6,7,8],
                          [0,3,6], [1,4,7], [2,5,8],
                          [0,4,8], [2,4,6]]
        for condition in win_conditions:
            if board[condition[0]] == board[condition[1]]
               == board[condition[2]] == player:
                return True
        return False
    def check_draw(board):
        return all(cell in ['x','O'] for cell in board)
    def play_game():
        board = [' ' for _ in range(9)]
    Initialize the board
        current_player = 'x'
        while True:
            print_board(board)
            try:
                move = int(input(f"player {current_player}, enter
                your move (1-9): "))-1
                if move <0 or move>=9 or board[move] != ':
                    print("Invalid move. Try again.")
                    continue
                except ValueError:
```

P.T.O

```
            print("Invalid input. Please enter a number
                        between 1 and 9.")
            Continue
        board[move] = current_player
        if check_win(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break
        elif check_draw(board):
            print_board(board)
            print("It's a draw!")
            break
    current_players = 'O' if current_player == 'X' else 'X'
if __name__ == "__main__":
    play_game().
```

OUTPUT :-

```
   |   |
---+---+---
   |   |
---+---+---
   |   |
```

Player X, enter your move (1-9) : 2

```
   | X |
---+---+---
   |   |
---+---+---
   |   |
```

Player O, enter your move (1-9) : 5

```
   | X |
---+---+---
   | O |
---+---+---
   |   |
```

Player X, enter your move (1-9) : 3

```
   | X | X
---+---+---
   | O |
---+---+---
   |   |
```

Player O, enter your move (1-9) : 4

```
   | X | X
---+---+---
 O | O |
---+---+---
   |   |
```

Player X, enter your move (1-9) : 1

```
 X | X | X
---+---+---
 O | O |
---+---+---
   |   |
```

Player X wins!

creating TICTACTOEE manually where value set by the user.

```
e :- import math
     def print_board (board):
         print (f" \n{board[0]} | {board[1]} | {board[2]}")
         print ("--+--+--")
         print (f" {board[3]} | {board[4]} | {board[5]} ").
         print ("--+--+--")
         print (f" {board[6]} | {board[7]} | {board[8]}")
     def check_win (board, player):
         win_conditions = [[0,1,2], [3,4,5], [6,7,8],
                           [0,3,6], [1,4,7], [2,5,8],
                           [0,4,8], [2,4,6]].
         for condition in win_conditions:
             if board[condition[0]] == board[condition[1]] ==
                                       board[condition[2]] == player:
                 return True
         return False
     def check_draw (board):
         return all (cell in ['x','0'] for cell in board)
     def minimax (board, depth, is_maximizing):
         if check_win (board,'0'):
             return 1
         if check_win (board,'x'):
             return -1
         if check_draw (board):
             return 0
         if is_maximizing:
             best_score = -math.inf
             for i in range (9):
                 if board[i] == '':
                     board[i] = '0'
                     score = minimax (board, depth+1, False)
                     board[i] == '':
                     best_score = max (best_score
```

A7.0

```
        return best_score
def find_best_move(board):
    best_move = None
    best_score = -math.inf
    for i in range(9):
        if board[i] == ' ':
            board[i] = 'O'
            move_score = minimax(board, 0, False)
            board[i] = ' '
            if move_score > best_score:
                best_score = move_score
                best_move = i
    return best_move
def play_game():
    board = [' ' for _ in range(9)]
Initialize the board
    current_player = 'X'
    while True:
        print_board(board)
        if current_player == 'X':
            try:
                move = int(input(f"player {current_player}, enter
                                 your move (1-9): ")) - 1
                if move < 0 or move >= 9 or board[move] != ' ':
                    print("Invalid move. Try again.")
                    continue
                except ValueError:
                    print("Invalid input. please enter b/w 1 and 9.")
                    continue
                else:
                    move =
        find_best_move(board)
                    print(f"Computer chooses position {move+1}")
                board[move] = current_player
                if check_win(board, current_player):
                    print_board(board)
```

```
            print(f"Player {current_player} wins!")
            break
        elif check_draw(board):
            print_board(board)
            print("It's a draw!")
            break
        current_player = 'O' if current_player == 'X' else 'X'

if __name__ == "__main__":
    play_game()
```

PUT

```
   |   |
--+---+--
   |   |
--+---+--
   |   |
```

ayer X, enter your move (1-9): 5

```
   |   |
--+---+--
   | X |
--+---+--
   |   |
```

mputer places O in position 1.

```
 O |   |
---+---+--
   | X |
---+---+--
   |   |
```

ayer X, enter your move (1-9): 2

```
 O | X |
---+---+--
   | X |
---+---+--
   |   |
```

mputer places O in position 9

```
 O | X |
---+---+--
   | X |
---+---+--
   |   | O
```

Player X, enter your move (1-9): 8

```
 O | X |
---+---+--
   | X |
---+---+--
   | X | O
```

Player X wins!

ASSIGNMENT-04.

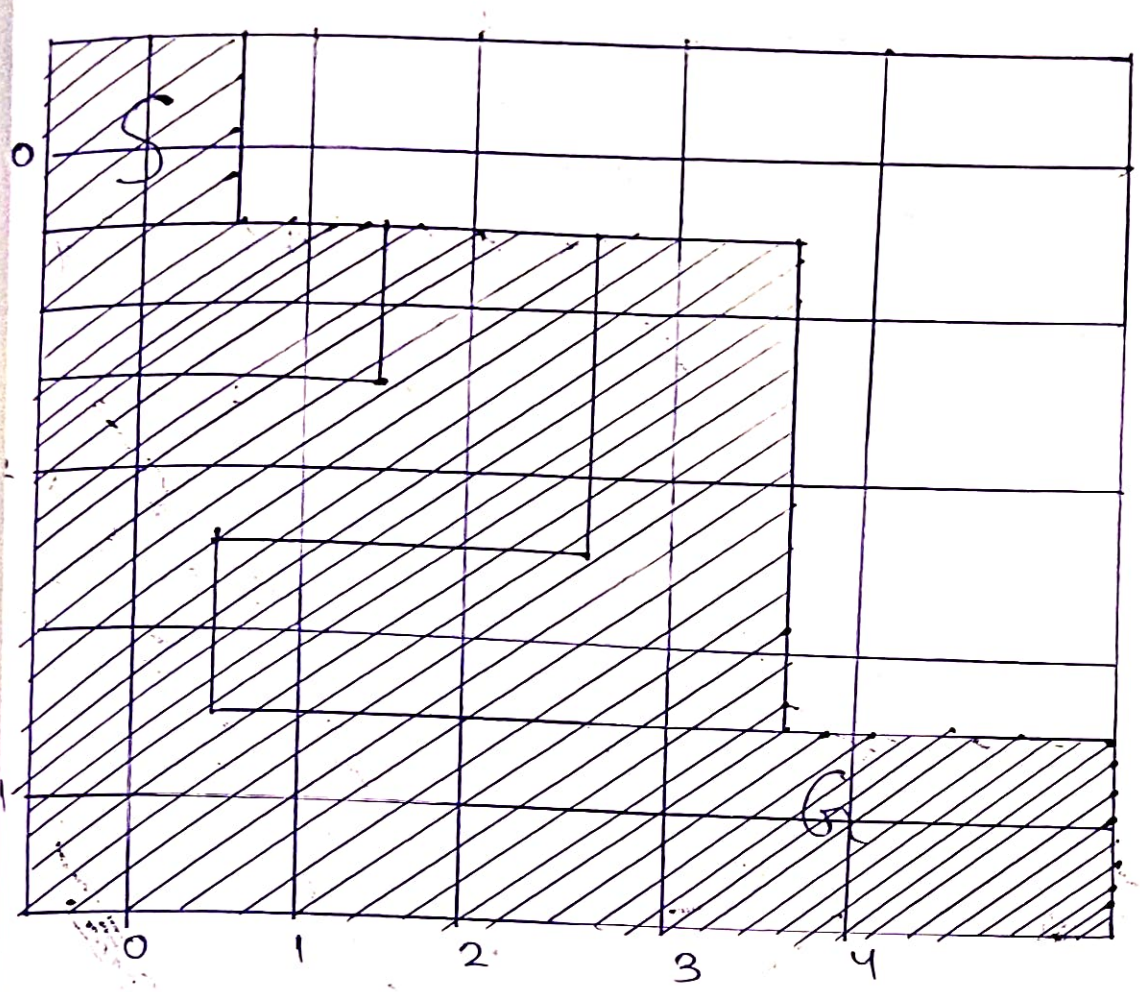
Implement the A* Search Algorithm by using python code.

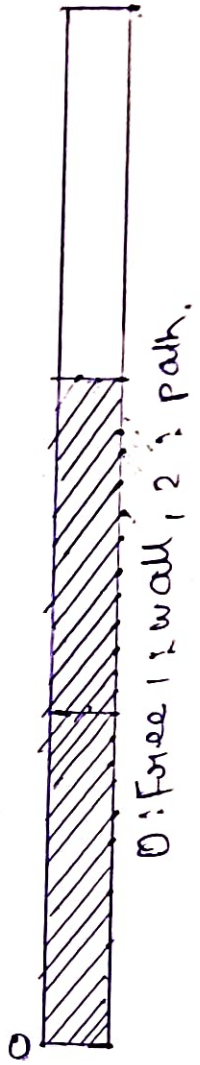
```
import heapq.
import matplotlib.pyplot as plt
Import numpy as np
clam Node:
    def - init - (Self, position, parent = None):
        Self.position = position
        Self. parent = Parent
        Self. g = 0
        Self.h = 0
        self.g = 0
    def - lt - (self, other):
        return self.f < other.f
    def heuristic (a,b):
        return abs (a[0] - b[0] + abs (a[i] - b [i])
    def aster (grid, start, end):
        open - node = [ ]
        closed - set = set()
        Start - node = Node (Start)
        goal - node = Node (end)
        heapq. heappush (open - list, start - node)
while open - list:
    Current - node = heapq. heappop (open - list)
    closed - set .add (Current - node. position)
    if Current - node. position == goal - node. position
    Path [ ]
    total - cost = current - node : path. append (& current - node. position)
Current - node = current - node. parent
return path [::-1], total cost
(x, y) = current - node, position
neighbors = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
for. next - pas
if nx < 0 or ny < 0 nx > = len (grid) or ny > = len (grid [0]):
Continue
if grid [nx] [ny] 1 = 0
Continue
nerghbor - node = Node (next - pos, Current - node)
nerghbor - node. g = Current - node. g + 1
neighbor - node. h = heuristic (next - pos, godl - node. position)
```

```
neighbor_node.f = neighbor_node.g + neighbor_node.h for open_list):
continue
heapq.heappush(open_list, neighbor_node)
return None, float('inf')
def plot_grid(grid, path, start, end):
    grid_display = np.array(grid)
    for (x,y) in path:
        if (x,y) != start and (x,y) != end
            grid_display[x][y] = 2
    plt.figure(figsize=(6,6))
    plt.imshow(grid_display, cmap=plt.cm.get_cmap('Accent',3))
    plt.xticks(range(len(grid[0])))
    plt.yticks(range(len(grid)))
    plt.grid(True)
    plt.text(start[1], start[0], 'S', ha='center', va='center', color
                                    ='black', fontsize_weight='bold')
    plt.title('A* Pathfinding visualization')
    plt.colorbar(ticks=[0,1,2], label='0: Free, 1: Wall, 2: Path')
    plt.show()
grid = [
        [0,0,0,0,0],
        [1,1,0,1,0],
        [0,0,0,1,0],
        [0,1,1,1,0],
        [0,0,0,0,0]
    ]
start = (0,0)
end = (4,4)
path, cost = astar(grid, start, end)
if path:
    print("Path found:", path)
    print("Minimum Cost:", cost)
    plot_grid(grid, path, start, end)
else:
    print("No path found")
```

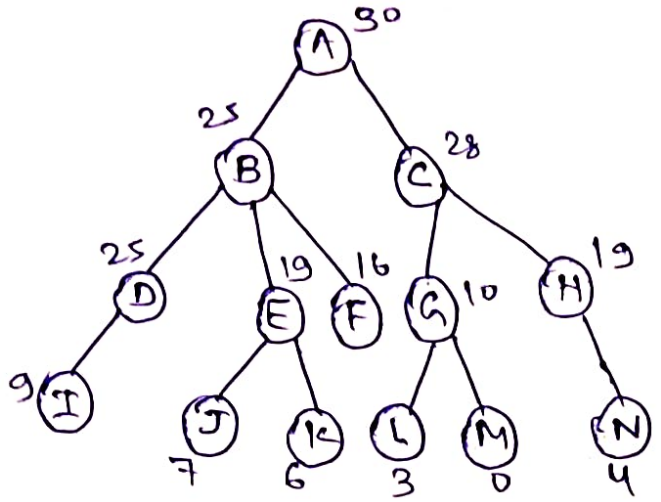
Output: Path found : [(0,0), (0,1), (0,2), (0,3), (0,4), (1,4), (2,4), (3,4), (4,4)]

Minimum Cost : 8


S
G
0
0
1
2
3
4


0: Free 1: wall, 2: path.
0

Best First search.



[A]
[B, C]
[D, C]
[K, J, D, C]
[D, C]
[J, D, C]
[D, C]
[J, C]
[C]
[G, H]
[M, L, H]
↳ Goal node

de!

```
import heapq
    Best First Search:
def __init__(self, graph, heuristic):
        self.graph = graph
        self.heuristic = heuristic
def search(self, start, goal):
    visited = set()
    priority_queue = []
    heapq.heappush(priority_queue, (self.heuristic[start], start,
                                    [start]))
    while priority_queue:
        h_val, current, path = heapq.heappop(priority_queue)
        print(f"visiting node: {current} (h={h_val})")
        if current == goal:
            print("Goal reached!")
            print("path:", "->".join(path))
            return path
        visited.add(current)
```

```
        for neighbor in self.graph.get(current, []):
            if neighbor not in visited:
                heapq.heappush(priority_queue, (self.heuristic[neighbor],
                                                neighbor, path + [neighbor]))
        print("Goal not found")
        return None,
if __name__ == "__main__":
    graph = {'A': ['B', 'C', 'D'],
             'B': ['E', 'F'],
             'C': ['G', 'H'],
             'D': ['I'],
             'E': [],
             'F': [],
             'G': [],
             'H': [],
             'I': [],
            }
    heuristic = {
                 'A': 5, 'B': 4, 'C': 3, 'D': 7,
                 'E': 1, 'F': 4, 'G': 2, 'H': 6, 'I': 3
                }
    bfs = BestFirstSearch(graph, heuristic)
    bfs.Search('A', 'E')
```

Output:

```
Visiting Node : A (h=5)
Visiting Node : C (h=3)
Visiting Node : G (h=2)
Visiting Node : B (h=4)
Visiting Node : E (h=1)
Goal reached!
Path : A → B → E.
```

write a python code to demonstrate UCS.

```
e :- import heapq
    graph = { 'A': [(1,'B'),(4,'C')],
              'B': [(2,'D'),(5,'E')],
              'C': [(1,'F')],
              'D': [],
              'E': [(1,'G')],
              'F': [(3,'G')],
              'G': []
            }
    def uniform_cost_search(start, goal):
        queue = [(0,start,[start])]
        visited = set()
        while queue:
            cost, node, path = heapq.heapq(queue)
            if node == goal:
                return cost, path
            if node in visited:
                continue
            visited.add(node)
            for edge_cost, neighbour in graph.get(node,[]):
                if neighbor not in visited:
                    heapq.heappush(queue,(cost+edge_cost, neighbor, path
                                          + [neighbor]))
        return float('inf'), []
    cost, path = uniform_cost_search('A','G')
    print(f"Cost to reach G: {cost}")
    print(f"path : {path}")
```

utput :

```
Cost to reach G : 7
path : ['A','B','E','G']
```

ASSIGNMENT - 07.

write an example and also the python code to solve 8 puzzle problem.

ven, problem,

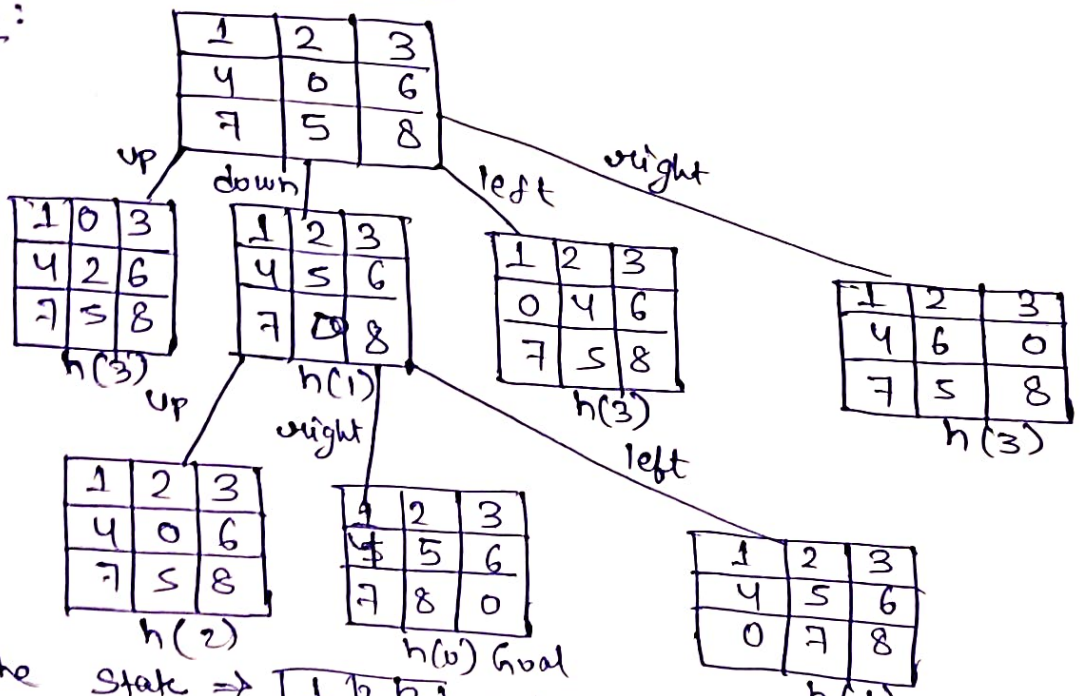
Initial state:

1	2	3
4	0	6
7	5	8

and given goal node / goal state as follows :

1	2	3
4	5	6
7	8	0

ps:



the state ⇒

1	2	3
4	5	6
7	0	8

Let's calculate the heuristic value for each element.

) = |(0-0)| + |(0-0)| = 0
) = |(0-0)| + |(1-1)| = 0
) = |(0-0)| + |(2-2)| = 0
) = |(1-1)| + |(0-0)| = 0
) = |(1-1)| + |(2-2)| = 0
) = |(1-1)| + |(2-2)| = 0
) = |(2-2)| + |(0-0)| = 0
) = |(2-2)| + |(2-2)| = 1.

∴ So, the futuristic value of the state is 0+0+0+0+0+0+0+1 = 1.

```
: goal = [[1,2,3], [4,5,6], [7,8,9]]
  moves = [(-1,0), (1,0), (0,-1), (0,1)]
  def heuristic (State):
    distance = 0
    for i in range(3):
    for j in range (3):
     val = State [i] [j]
     if val != 0:
       x,y = divmod (val - 1,3)
       distance += abs (x-i) + abs (y-j)
    return distance
```

```
ef to_tuple (state):
  return tuple (tuple (row) for row in state)
ef find_blank (state):
  for i in range (3):
    for j in range (3):
      if state [i] [j] == 0:
        return i, j
ef neighbors (state):
  x,y = find_blank (state)
  for dx, dy in moves:
    nx, ny = x+dx, y+dy
    if 0<=nx<3 and 0<=ny<3:
      new_state = [row] [i] for row in state]
      new_state = [x] [y], new_state [nx] [ny] = new_state
          [nx] [ny], new_state [x] [y]
      yield new_state
ef astar (start):
    pq = []
    heapq.heappush (pq, (heuristic (start), 0, start, []))
    visited = set()
    while pq:
        f,g, state, path = heapq.heapop (pq)
        if state == goal:
            return path + [state]
        visited.add (to_tuple (state))
        for neigh in neighbors (state):
    if to_tuple (neigh) not in visited:
      heapq.heappush (pq, (g+1 + heuristic (neigh), g+1, neigh, path + [state]))
start = [[1,2,3], [4,0,6], [7,5,8]]
solution = astar (start)
print ("solution found in ", len (solution)-1, "moves:")
for s in solution:
  for row in s:
    print (row)
  print()
```

tput : solution found in 2 moves :

```
[1,2,3]
[4,0,6]
[7,5,8]

[1,2,3]
[4,5,6]
[7,0,8]

[1,2,3]
[4,5,6]
[7,8,0]
```

ASSIGNMENT - 08

Implement Hill climbing search Algorithm with using python code.

```
import random
def objective_function(x):
    return -(x**2)+10
def hill_climbing(max_iterations=1000, step_size=0.1):
    current_x = random.uniform(-10,10)
    current_value = objective_function(current_x)
    for _ in range(max_iterations):
        left_x = current_x - step_size
        right_x = current_x + step_size
        left_value = objective_function(left_x)
        right_value = objective_function(right_x)
        if left_value > current_value or right_value > current_value:
            if left_value > right_value:
                current_x, current_value = left_x, left_value
            else:
                break
    return current_x, current_value
best_x, best_value = hill_climbing()
print(f"Best solution found: x = {best_x:.4f}, f(x) = {best_value:.4f}")
```

Output : Best solution found : xx = 0.0250, f(x) = 9.9994

Implement water jug problem using python code.

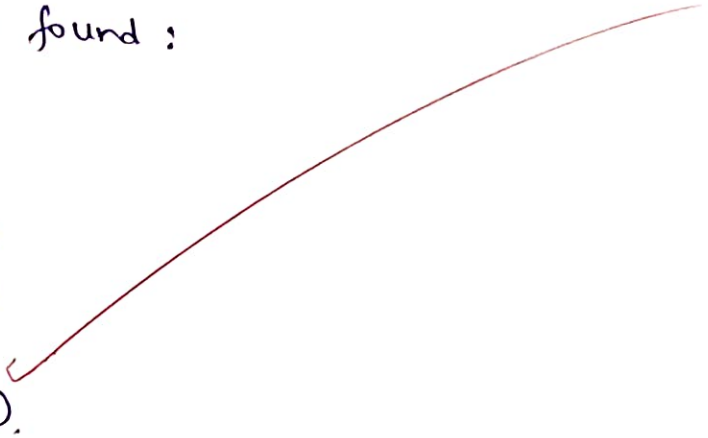
```
def water_jug_dfs(jug1, jug2, target, visited=None, path=None):
    if visited is None:
        visited = set()
    if path is None:
        path = []
    state = (jug1, jug2)
    if jug1 == target or jug2 == target:
        path.append(state)
        print("Solution found:")
        for step in path:
            print(step)
        return True
    if state in visited:
        return False
    visited.add(state)
    path.append(state)
    possible_moves = [(4, jug2),
                      (jug1, 3),
                      (0, jug2),
                      (jug1, 0),
                      (jug1 - min(jug1, 3-jug2), jug2 + min(jug1, 3-jug2))
```

```
(jug1 + min(jug2, 4-jug1), jug2 - min(jug2, 4-jug1)), ]
for next_state in possible moves:
  if water_jug_dfs(next_state[0], next_state[1], target, visited, path.copy()):
return True
return False

jug1 - capacity = 4
jug2 - capacity = 3
target = 2
if not water_jug_dfs(0, 0, target):
  print("No solution found:")
```

Output:

```
Solution found:
(0,0)
(4,0)
(4,3)
(0,3)
(3,0)
(3,3)
(4,2).
```



ASSIGNMENT-09

write a Python program to implement the Answer Extraction System.

```
knowledge_base = {
    "who is the president of India": "Draupadi Murmu",
    "what is the Capital of India": "New Delhi",
    "who is the prime minister of India": "Narendra Modi",
    "what is ai": "AI Stands for Artificial Intelligence",
    "what is machine learning": "Machine learning is a subset of
                                 AI allows to learn from data",
    "what is national animal of India": "Bengal Tiger" }
questions = input ("Ask a question:").lower().strip()
found = false
for key in question or question in key:
    print ("Answer:", knowledge_base[key])
    found = True
    break
if not found:
    print("Answer not found in knowledge base.")
```

Output: Ask a question: who is the president of India
Answer: Draupadi Murmu.

Write a code to implement Rule-based System.

```
def diagnose (symptoms):
    if "fever" in symptoms and "Cough" in Systems and "tired" in Symptoms
        return "You may have the flue".
    elif "headache" in symptoms and "nausea" in symptoms:
        return "You may hava a Migraine".
    elif "sneezing" in Symptoms and "runny nose" in symptoms:
        return "You may have common Cold".
    else:
        return "symptoms not relognized. Please consult a doctor"
Symptoms = input ("Enter your symptoms (Comma separated): ").
                  lower().replace(' ', '').split(",")
Symptoms = [S.strip() for s in symptoms]
print (diagnose (symptoms))
```

Output: Enter your symptoms (Comma Separated): Fever, Cough, tired
You may have the flue.

rite a python code to implement K-Nearest Neighbors using given dataset.

```
: dataset=[[2,4,'A'],
           [4,2,'A'],
           [4,4,'A'],
           [6,6,'B'],
           [6,8,'B'],
           [8,6,'B']]
row - point = [5,5]
distances = []
for point in dataset:
    x,y, label = point
    distance = ((x-new_point[0])**2+(y-new_point[1])**2)**0.5
    distances.append([distance, label])
distances.sort()
k=3
neighbors = distances[k]
votes = {}
for d, label in neighbors:
  if label in votes:
    votes[label]+=1
  else:
    votes[label]=1
predicted_label = max(votes, key=votes.get)
print("Distances to dataset points":, distances)
print("Nearest neighbors:", neighbors)
print("Predicted class for point", new_point, "is:", predicted_label)
```

put:

Distances to dataset points: [[1.4142135623730951, 'A'], [1.4142135622755
95', 'B'], [3.1622776601683795, 'A'], [3.1622776601683795, 'B'], [3.162277666
1683795, 'B']]

arest neighbors: [[1.4142135623730951, 'A'], [1.4142135623730951, 'B'],
[3.1622776601683795, 'A']]

edicted class for point [5,5] is A.

7/11/25