**`AI Problems and solves :**

**1.** **Write a program using PROLOG/LISP for addition and multiplication of two numbers.**

# Program to add and multiply two numbers with user input

def add\_two\_numbers(a, b):

return a + b

def multiply\_two\_numbers(a, b):

return a \* b

# Taking input from user

num1 = float(input("Enter the first number: "))

num2 = float(input("Enter the second number: "))

print("Addition of", num1, "and", num2, "is:", add\_two\_numbers(num1, num2))

print("Multiplication of", num1, "and", num2, "is:", multiply\_two\_numbers(num1, num2))  
result: Enter the first number: 4

Enter the second number: 6

Addition of 4.0 and 6.0 is: 10.0

Multiplication of 4.0 and 6.0 is: 24.0

=== Code Execution Successful ===

**2. Write a program using PROLOG/LISP for finding the sum of all numbers in a given list.**

# Program to find the sum of all numbers entered by the user

def sum\_of\_list(numbers):

return sum(numbers)

# Taking input from the user

user\_input = input("Enter numbers separated by spaces: ")

# Splitting input and converting to integers

my\_list = list(map(int, user\_input.split()))

# Calculating the sum

total = sum\_of\_list(my\_list)

print("The sum of the list is:", total)  
result: Enter numbers separated by spaces: 34 5 6 12 4

The sum of the list is: 61

**3. Write a program using PROLOG/LISP for finding largest and smallest element in a list.**

def find\_largest\_and\_smallest(numbers):

if not numbers: # Check if the list is empty

return None, None

largest = max(numbers)

smallest = min(numbers)

return largest, smallest

numbers = list(map(int, input("Enter the list of numbers separated by spaces: ").split()))

largest, smallest = find\_largest\_and\_smallest(numbers)

print(f"Largest element: {largest}")

print(f"Smallest element: {smallest}")

=== Code Execution Successful ===

**4. Write a program using PROLOG/LIST to count number of elements in a list.**

# Program to count the number of elements entered by the user

def count\_elements(my\_list):

return len(my\_list)

# Taking input from user

user\_input = input("Enter list elements separated by spaces: ")

my\_list = user\_input.split()

# Counting elements

total\_elements = count\_elements(my\_list)

print("The number of elements you entered is:", total\_elements)  
**results:**

Enter list elements separated by spaces: 34 56 7 1 4 5 2 7 9 90

The number of elements you entered is: 10

=== Code Execution Successful ===

**5. Write a program using PROLOG/LISP whether an element is a member of list.**

# Program to check if an element is a member of a list

def is\_member(element, my\_list):

if element in my\_list:

print(f"The element {element} is a member of the list.")

else:

print(f"The element {element} is NOT a member of the list.")

# Taking user input

user\_list = input("Enter list elements separated by spaces: ").split()

user\_element = input("Enter the element to check: ")

# Checking if the element is in the list

is\_member(user\_element, user\_list)  
**Result:**

Enter list elements separated by spaces: 45 67 23 4 5 7 89 10 32 12 34

Enter the element to check: 23

The element 23 is a member of the list.

=== Code Execution Successful ===

**6. Write a program using PROLOG /LISP to reverse a list.**

# Program to reverse a list with user input

def reverse\_list(my\_list):

return my\_list[::-1]

# Taking input from the user

user\_input = input("Enter list elements separated by spaces: ").split()

# Reversing the list

reversed\_list = reverse\_list(user\_input)

print("Reversed list:", reversed\_list)

**7. Write a program using PROLOG LISP to find out Union and Intersection of two lists.**

# Program to find Union and Intersection of two lists with user input

def find\_union(list1, list2):

return list(set(list1) | set(list2))

def find\_intersection(list1, list2):

return list(set(list1) & set(list2))

# Taking input from the user

list1 = input("Enter the first list elements separated by spaces: ").split()

list2 = input("Enter the second list elements separated by spaces: ").split()

# Converting the input to integers

list1 = list(map(int, list1))

list2 = list(map(int, list2))

# Finding union and intersection

union\_result = find\_union(list1, list2)

intersection\_result = find\_intersection(list1, list2)

print("Union of the lists:", union\_result)

print("Intersection of the lists:", intersection\_result)

**8. Write a program using PROLOG/LISP to determine the Greatest Common Divisor of two positive integer numbers.**

# Program to find GCD using Euclid's Algorithm with user input

def find\_gcd(a, b):

while b != 0:

a, b = b, a % b

return a

# Taking user input

num1 = int(input("Enter the first positive integer: "))

num2 = int(input("Enter the second positive integer: "))

# Finding the GCD

gcd\_result = find\_gcd(num1, num2)

print(f"The GCD of {num1} and {num2} is: {gcd\_result}")

Result:

Enter the first positive integer: 34

Enter the second positive integer: 12

The GCD of 34 and 12 is: 2

**9 . if a number is odd or even**  
# Taking user input

number = int(input("Enter a number: "))

# Checking if the number is odd or even

if number % 2 == 0:

print(f"{number} is an even number.")

else:

print(f"{number} is an odd number.")

**Section-B**

1. **Write a program using PROLOG/LISP to solve 4-queen problem.**

# Program to solve the 4-Queens problem using backtracking

def is\_safe(board, row, col):

# Check if there is a queen in the same column

for i in range(row):

if board[i] == col or board[i] - i == col - row or board[i] + i == col + row:

return False

return True

def solve\_n\_queens(board, row):

# If all queens are placed

if row == len(board):

print\_board(board)

return True

# Try placing the queen in all columns one by one

res = False

for col in range(len(board)):

if is\_safe(board, row, col):

board[row] = col

res = solve\_n\_queens(board, row + 1) or res

board[row] = -1 # Backtrack

return res

def print\_board(board):

for row in board:

line = ['Q' if i == row else '.' for i in range(len(board))]

print(" ".join(line))

print()

# Driver code to initiate the process

def solve():

n = 4

board = [-1] \* n # Initialize the board with -1 (no queen placed)

if not solve\_n\_queens(board, 0):

print("Solution does not exist.")

else:

print("Solutions found successfully.")

# Run the solver

solve()

1. **Write a program using PROLOG/LISP to solve 8-puzzle problem.**

import heapq

# Define the goal state for the 8-puzzle

goal\_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)

# Define the directions for moving the blank space (up, down, left, right)

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Calculate Manhattan distance for the heuristic

def manhattan\_distance(state):

return sum(abs((state[i] - 1) // 3 - i // 3) + abs((state[i] - 1) % 3 - i % 3)

for i in range(9) if state[i] != 0)

# Generate possible moves (child states)

def generate\_moves(state):

blank\_pos = state.index(0)

blank\_row, blank\_col = divmod(blank\_pos, 3)

children = []

for move in moves:

new\_row, new\_col = blank\_row + move[0], blank\_col + move[1]

if 0 <= new\_row < 3 and 0 <= new\_col < 3:

new\_blank\_pos = new\_row \* 3 + new\_col

new\_state = list(state)

new\_state[blank\_pos], new\_state[new\_blank\_pos] = new\_state[new\_blank\_pos], new\_state[blank\_pos]

children.append(tuple(new\_state))

return children

# A\* search algorithm

def a\_star\_search(start\_state):

open\_list = [(manhattan\_distance(start\_state), 0, start\_state, [])]

visited = {start\_state}

while open\_list:

f, g, state, path = heapq.heappop(open\_list)

if state == goal\_state:

return path + [state]

for child in generate\_moves(state):

if child not in visited:

visited.add(child)

heapq.heappush(open\_list, (g + 1 + manhattan\_distance(child), g + 1, child, path + [state]))

return None

# Function to print the board in a 3x3 grid format

def print\_board(state):

for i in range(0, 9, 3):

print(state[i:i+3])

print()

# Hardcoded start state

start\_state = (1, 2, 3, 4, 5, 6, 7, 0, 8)

# Main driver code

def main():

solution\_path = a\_star\_search(start\_state)

if solution\_path:

print(f"Solution found in {len(solution\_path) - 1} steps:")

for step in solution\_path:

print\_board(step)

else:

print("No solution found.")

# Run the main function

main()  
**Write a program using PROLOG/LISP to solve Tower of Hanoi problem.**

def tower\_of\_hanoi(n, source, target, auxiliary):

if n == 1:

print(f"Move disk 1 from {source} to {target}")

return

tower\_of\_hanoi(n - 1, source, auxiliary, target)

print(f"Move disk {n} from {source} to {target}")

tower\_of\_hanoi(n - 1, auxiliary, target, source)

# Driver code

n = 3 # Number of disks

tower\_of\_hanoi(n, 'A', 'C', 'B')  
Result :

Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C

**4. Write a program using PROLOG/LISP to solve Traveling Salesman problem.**import itertools

# Function to calculate the total distance of a given path

def calculate\_total\_distance(path, dist\_matrix):

total\_distance = 0

for i in range(len(path) - 1):

total\_distance += dist\_matrix[path[i]][path[i + 1]]

total\_distance += dist\_matrix[path[-1]][path[0]] # return to the starting point

return total\_distance

# Function to solve the Traveling Salesman Problem using brute force

def traveling\_salesman\_bruteforce(dist\_matrix):

n = len(dist\_matrix)

cities = list(range(n)) # List of cities to visit

# Generate all possible permutations of cities

all\_permutations = itertools.permutations(cities)

# Initialize the minimum distance as a large number

min\_distance = float('inf')

best\_path = None

# Evaluate the total distance for each permutation

for perm in all\_permutations:

total\_distance = calculate\_total\_distance(perm, dist\_matrix)

if total\_distance < min\_distance:

min\_distance = total\_distance

best\_path = perm

return best\_path, min\_distance

# Example distance matrix (symmetric, where dist\_matrix[i][j] is the distance from city i to city j)

dist\_matrix = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

# Solve TSP

best\_path, min\_distance = traveling\_salesman\_bruteforce(dist\_matrix)

# Output the result

print("Best path:", best\_path)

print("Minimum distance:", min\_distance)

**Write a program using PROLOG/LISP to implement Depth First Search.**

# Function to perform DFS

def dfs(graph, node, visited=None):

if visited is None:

visited = set() # A set to track visited nodes

# Mark the current node as visited

visited.add(node)

print(node, end=" ") # Print the node (or do any operation you want)

# Recur for all the adjacent nodes

for neighbor in graph[node]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

return visited

# Function to take user input and construct the graph

def create\_graph():

n = int(input("Enter the number of nodes: ")) # Number of nodes

graph = {i: [] for i in range(n)}

m = int(input("Enter the number of edges: "))

print("Enter the edges (u, v) where u and v are node indices:")

for \_ in range(m):

u, v = map(int, input().split()) # Read an edge

graph[u].append(v) # Add v to the adjacency list of u

graph[v].append(u) # Since it's an undirected graph, add u to the adjacency list of v

return graph

# Main program

graph = create\_graph()

# Start DFS traversal from the first node (or any node of your choice)

start\_node = int(input("Enter the starting node for DFS: "))

print("DFS traversal starting from node", start\_node, ":")

dfs(graph, start\_node)  
6. **Write a program using PROLOG/LISP to implement Breadth First Search.**

from collections import deque

def bfs(graph, start):

visited = set()

queue = deque([start])

visited.add(start)

print("BFS traversal starting from node", start, ":")

while queue:

node = queue.popleft()

print(node, end=" ")

for neighbor in graph[node]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

def create\_graph():

n = int(input("Enter the number of nodes: "))

graph = {i: [] for i in range(n)}

m = int(input("Enter the number of edges: "))

print("Enter the edges (u, v) where u and v are node indices:")

for \_ in range(m):

u, v = map(int, input().split())

graph[u].append(v)

graph[v].append(u)

return graph

graph = create\_graph()

start\_node = int(input("Enter the starting node for BFS: "))

bfs(graph, start\_node)  
Result:

Enter the number of nodes: 5

Enter the number of edges: 4

Enter the edges (u, v) where u and v are node indices:

1 3

2 4

4 1

3 2

Enter the starting node for BFS: 1

BFS traversal starting from node 1 :

1 3 4 2

**7. Write a program using PROLOG/LISP to implement Hill Climbing Algorithm.**import random

def hill\_climbing(objective\_function, start, step\_size, max\_iterations=100):

current\_solution = start

current\_value = objective\_function(current\_solution)

print(f"Starting point: x = {current\_solution}, f(x) = {current\_value}")

for \_ in range(max\_iterations):

neighbors = [current\_solution - step\_size, current\_solution + step\_size]

neighbor\_values = [objective\_function(neighbor) for neighbor in neighbors]

best\_neighbor\_value = max(neighbor\_values)

best\_neighbor = neighbors[neighbor\_values.index(best\_neighbor\_value)]

if best\_neighbor\_value > current\_value:

current\_solution = best\_neighbor

current\_value = best\_neighbor\_value

print(f"Moved to new solution: x = {current\_solution}, f(x) = {current\_value}")

else:

print("Local maximum reached.")

break

def user\_defined\_objective(x):

return -x\*\*2 + 5\*x

if \_\_name\_\_ == "\_\_main\_\_":

print("Welcome to the Hill Climbing Algorithm")

start = float(input("Enter the starting point (x): "))

step\_size = float(input("Enter the step size: "))

max\_iterations = int(input("Enter the maximum number of iterations: "))

hill\_climbing(user\_defined\_objective, start, step\_size, max\_iterations)  
**Result:**

Welcome to the Hill Climbing Algorithm

Enter the starting point (x): 0

Enter the step size: 0.5

Enter the maximum number of iterations: 15

Starting point: x = 0.0, f(x) = 0.0

Moved to new solution: x = 0.5, f(x) = 2.25

Moved to new solution: x = 1.0, f(x) = 4.0

Moved to new solution: x = 1.5, f(x) = 5.25

Moved to new solution: x = 2.0, f(x) = 6.0

Moved to new solution: x = 2.5, f(x) = 6.25

Local maximum reached.

=== Code Execution Successful ===

**8. sort a list**# Taking user input for the list of numbers

numbers = list(map(int, input("Enter the list of numbers separated by spaces: ").split()))

# Sorting the list in ascending order

numbers.sort()

print("Sorted list in ascending order:", numbers)

# Sorting the list in descending order

numbers.sort(reverse=True)

print("Sorted list in descending order:", numbers)

**8. Write a program using PROLOG/LISP to show Tic-tac-toe game for O and X.**def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 5)

def check\_winner(board, player):

for row in board:

if all([cell == player for cell in row]):

return True

for col in range(3):

if all([board[row][col] == player for row in range(3)]):

return True

if all([board[i][i] == player for i in range(3)]):

return True

if all([board[i][2 - i] == player for i in range(3)]):

return True

return False

def is\_board\_full(board):

return all([cell != ' ' for row in board for cell in row])

def play\_game():

board = [[' ' for \_ in range(3)] for \_ in range(3)]

current\_player = 'X'

print("Welcome to Tic-Tac-Toe!")

print\_board(board)

while True:

print(f"Player {current\_player}'s turn:")

row = int(input("Enter row (0-2): "))

col = int(input("Enter column (0-2): "))

if board[row][col] != ' ':

print("Cell already taken. Try again.")

continue

board[row][col] = current\_player

print\_board(board)

if check\_winner(board, current\_player):

print(f"Player {current\_player} wins!")

break

if is\_board\_full(board):

print("It's a draw!")

break

current\_player = 'O' if current\_player == 'X' else 'X'

if \_\_name\_\_ == "\_\_main\_\_":

play\_game()  
Result:

Welcome to Tic-Tac-Toe!

| |

-----

| |

-----

| |

-----

Player X's turn:

Enter row (0-2): 1

Enter column (0-2): 1

| |

-----

| X |

-----

| |

-----

Player O's turn:

Enter row (0-2): 1

Enter column (0-2): 2

| |

-----

| X | O

-----

| |

-----

Player X's turn:

Enter row (0-2): 2

Enter column (0-2): 2

| |

-----

| X | O

-----

| | X

-----

Player O's turn:

Enter row (0-2): 0

Enter column (0-2): 0

O | |

-----

| X | O

-----

| | X

-----

Player X's turn:

Enter row (0-2): 0

Enter column (0-2): 2

O | | X

-----

| X | O

-----

| | X

-----

Player O's turn:

Enter row (0-2): 1

Enter column (0-2): 0

O | | X

-----

O | X | O

-----

| | X

-----

Player X's turn:

Enter row (0-2): 2

Enter column (0-2): 0

O | | X

-----

O | X | O

-----

X | | X

-----

Player X wins!

=== Code Execution Successful ===