Basic AI Programs.

Contents

1. Problem: Search in a List

Linear Search:

```
def linear_search(arr, target):
    for i, item in enumerate(arr):
        if item == target:
            return i
    return -1

# Example
my_list = [1, 2, 3, 4, 5]
target_value = 3
result = linear_search(my_list, target_value)
print(f"Linear Search: {target_value} found at index {result}")
```

2. Binary Search:

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

# Example (Note: Binary search requires a sorted list)
my_sorted_list = [1, 2, 3, 4, 5]
target_value = 3
result = binary_search(my_sorted_list, target_value)
print(f"Binary Search: {target_value} found at index {result}")
```

3. Problem: Finding the Factorial of a Number

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

# Example usage:
num = 5
fact = factorial(num)
print(f"The factorial of {num} is {fact}.")
```

4. Problem: Solve the 8-Puzzle

Depth-First Search (DFS):

```
from collections import deque

def dfs(initial_state, goal_state):
    stack = deque([(initial_state, [])])
    visited = set()

    while stack:
        current_state, path = stack.pop()
        if current_state == goal_state:
            return path
        if current_state not in visited:
            visited.add(current_state)
            for move in get_possible_moves(current_state):
                stack.append((move, path + [move]))

    return None

def get_possible_moves(state):
    # Implement the logic to generate possible moves based on the current state
    pass

# Example
initial_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
solution_path = dfs(initial_state, goal_state)
print("DFS Solution Path:", solution_path)
```

5. Problem: Solve the N-Queens Problem

Backtracking Algorithm:

```
def solve_n_queens(n):
    def is_safe(board, row, col):
        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 place_queen(board, row):
        if row == n:
            solutions.append(board.copy())
            return
        for col in range(n):
            if is_safe(board, row, col):
                board[row] = col
                place_queen(board, row + 1)

    solutions = []
    place_queen([-1] * n, 0)
    return solutions

# Example
n_queens_solutions = solve_n_queens(4)
print("N-Queens Solutions:", n_queens_solutions)
```