1) Chabot :-

```
import random

responses = {
    "hello": ["Hi there!", "Hello!", "Hey!"],
    "how are you": ["I'm good, thanks!", "I'm just a chatbot, but I'm here to help."],
    "bye": ["Goodbye!", "See you later!", "Farewell!"],
}

def get_response(user_input):
    user_input = user_input.lower()
    for key in responses:
        if key in user_input:
            return random.choice(responses[key])
    return "I don't understand that. Please ask another question."
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        print("Chatbot: Goodbye!")
        break
    response = get_response(user_input)
    print("Chatbot:", response)
```

2) missionary and cannibals :-

```
from collections import deque

initial_state = (3, 3, 1)  # (Missionaries on the left, Cannibals on the left, Boat on the left)
goal_state = (0, 0, 0)     # (Missionaries on the left, Cannibals on the left, Boat on the left)

def is_valid(state):
    m, c, b = state
    if m < 0 or c < 0 or m > 3 or c > 3:
        return False
    if m < c and m > 0:
        return False
    if 3 - m < 3 - c and 3 - m > 0:
        return False
    return True

def successors(state):
    m, c, b = state
    possible_moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]
    valid_successors = []
```

```
    for move in possible_moves:
        if b == 1:  # Boat is on the left side
            new_state = (m - move[0], c - move[1], 0)
        else:
            new_state = (m + move[0], c + move[1], 1)

        if is_valid(new_state):
            valid_successors.append(new_state)

    return valid_successors

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

    while stack:
        current_state, path = stack.pop()

        if current_state == goal_state:
            return path + [current_state]

        if current_state not in visited:
            visited.add(current_state)
            for successor in successors(current_state):
                stack.append((successor, path + [current_state]))

    return None

if __name__ == "__main__":
    solution = solve()
    if solution:
        print("Solution found:")
        for state in solution:
            print(state)
    else:
        print("No solution found.")
```

3) tic-tac-toe.

```
import tkinter as tk
from tkinter import messagebox

root = tk.Tk()
root.title("Tic-Tac-Toe")


buttons = [[None, None, None], [None, None, None], [None, None, None]]
```

```
current_player = "X"
moves = 0

def button_click(row, col):
    global current_player, moves

    if buttons[row][col]["text"] == "" and not is_game_over():
        buttons[row][col]["text"] = current_player
        moves += 1

        if is_game_over():
            if check_winner(current_player):
                messagebox.showinfo("Game Over", f"Player {current_player} wins!")
            else:
                messagebox.showinfo("Game Over", "It's a draw!")
            root.quit()
        else:
            current_player = "O" if current_player == "X" else "X"

def check_winner(player):
    for row in range(3):
        if buttons[row][0]["text"] == buttons[row][1]["text"] == buttons[row][2]["text"] ==
player:
            return True

    for col in range(3):
        if buttons[0][col]["text"] == buttons[1][col]["text"] == buttons[2][col]["text"] == player:
            return True

    if buttons[0][0]["text"] == buttons[1][1]["text"] == buttons[2][2]["text"] == player:
        return True

    if buttons[0][2]["text"] == buttons[1][1]["text"] == buttons[2][0]["text"] == player:
        return True

    return False

def is_game_over():
    return moves == 9 or check_winner("X") or check_winner("O")

for row in range(3):
    for col in range(3):
        buttons[row][col] = tk.Button(root, text="", font=("Helvetica", 24), width=6, height=2,
                              command=lambda r=row, c=col: button_click(r, c))
```

```
        buttons[row][col].grid(row=row, column=col)

root.mainloop()
```

4) tower of Hanoi :-

```
def TowerOfHanoi(n , source, destination, auxiliary):
        if n==1:
                print ("Move disk 1 from source",source,"to destination",destination)
                return
        TowerOfHanoi(n-1, source, auxiliary, destination)
        print ("Move disk",n,"from source",source,"to destination",destination)
        TowerOfHanoi(n-1, auxiliary, destination, source)

n = 4
TowerOfHanoi(n,'A','B','C')
```

5) water-jug :-

```
from collections import deque

def water_jug_problem(capacity_x, capacity_y, target):
  visited_states = set()
  initial_state = (0, 0)
  queue = deque([(initial_state, [])])

  while queue:
    current_state, current_path = queue.popleft()
    visited_states.add(current_state)

    x, y = current_state

    if x == target or y == target:
      return current_path + [(x, y, "Goal")]

    next_states = [
      (capacity_x, y),
      (x, capacity_y),
      (0, y),
      (x, 0),
      (x - min(x, capacity_y - y), y + min(x, capacity_y - y)),
      (x + min(y, capacity_x - x), y - min(y, capacity_x - x))
    ]

    for state in next_states:
      if state not in visited_states:
        new_x, new_y = state
        queue.append((state, current_path + [(x, y, f"Pour {x}L and {y}L")]))
```

```
            visited_states.add(state)

    return "No solution found!"

capacity_x = 4
capacity_y = 3
target_amount = 2

result = water_jug_problem(capacity_x, capacity_y, target_amount)
if result == "No solution found!":
    print(result)
else:
    print("Solution:")
    for step in result:
        print(step)
```

6) depth first search :-

```
graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F'],

    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E']

}


visited = set()


def dfs(node):

    visited.add(node)

    print(node, end=' ')


    for neighbor in graph[node]:

        if neighbor not in visited:
```

```
        dfs(neighbor)


dfs('A')
```

7) **breadth first search :-**

```
from collections import deque

def bfs(graph, start, goal):
    queue = deque([(start, [start])])
    visited = set()

    while queue:
        current, path = queue.popleft()

        if current == goal:
            return path

        if current in visited:
            continue

        visited.add(current)

        for neighbor in graph[current]:
            if neighbor not in visited:
                queue.append((neighbor, path + [neighbor]))

    return None

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F', 'G'],
    'D': ['B'],
    'E': ['B', 'H'],
    'F': ['C'],
    'G': ['C', 'I'],
    'H': ['E'],
    'I': ['G']
}

start_node = 'A'
goal_node = 'I'

shortest_path = bfs(graph, start_node, goal_node)
```

```
if shortest_path:
    print("Shortest path from {} to {}:".format(start_node, goal_node), shortest_path)
else:
    print("No path found")
```

8) A* algorithm :-

```
import heapq

class Node:
    def __init__(self, x, y, cost, parent=None):
        self.x = x
        self.y = y
        self.cost = cost
        self.parent = parent

    def __lt__(self, other):
        return self.cost < other.cost

def astar(grid, start, goal):
    def heuristic(node):
        return abs(node.x - goal.x) + abs(node.y - goal.y)

    open_list = []
    closed_list = set()

    start_node = Node(start[0], start[1], 0)
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if (current_node.x, current_node.y) == goal:
            path = []
            while current_node:
                path.append((current_node.x, current_node.y))
                current_node = current_node.parent
            return path[::-1]

        closed_list.add((current_node.x, current_node.y))

        for neighbor in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            x, y = current_node.x + neighbor[0], current_node.y + neighbor[1]

            if (
                0 <= x < len(grid)
```

```
            and 0 <= y < len(grid[0])
            and grid[x][y] != 1
            and (x, y) not in closed_list
        ):
            new_cost = current_node.cost + grid[x][y]
            new_node = Node(x, y, new_cost, current_node)

            for node in open_list:
                if node.x == new_node.x and node.y == new_node.y:
                    if node.cost > new_node.cost:
                        open_list.remove(node)
                        heapq.heappush(open_list, new_node)
                    break
            else:
                heapq.heappush(open_list, new_node)

    return None

# Example usage:
grid = [
    [0, 0, 0, 0, 0],
    [0, 1, 1, 0, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0]
]

start = (0, 0)
goal = (4, 4)

path = astar(grid, start, goal)
if path:
    print("Shortest path:", path)
else:
    print("No path found")
```

9) travelling salesman problem

```
import itertools
def calculate_total_distance(path, cities):
    total_distance = 0
    for i in range(len(path) - 1):
        city1 = path[i]
        city2 = path[i + 1]
        total_distance += cities[city1][city2]
```

```
    return total_distance
def traveling_salesman(cities):
    num_cities = len(cities)
    city_indices = range(num_cities)
    shortest_path = None
    shortest_distance = float('inf')
    for path in itertools.permutations(city_indices):
        distance = calculate_total_distance(path, cities)
        if distance < shortest_distance:
            shortest_distance = distance
            shortest_path = path
    return shortest_path, shortest_distance
if __name__ == "__main__":

    cities = [
        [0, 29, 20, 21],
        [29, 0, 15, 18],
        [20, 15, 0, 25],
        [21, 18, 25, 0]
    ]
    shortest_path, shortest_distance = traveling_salesman(cities)
    print("Shortest Path:", shortest_path)
    print("Shortest Distance:", shortest_distance)
```