1. A* Algorithm using Python

```
import heapq  # For using a priority queue (min-heap)

# Heuristic function: Manhattan distance
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

# A* search algorithm
def astar(grid, start, end):
    open_list = [(heuristic(start, end), 0, start, [start])]  # (f, g, current_node, path)
    visited = set()  # Track visited nodes

    while open_list:
        _, g, current, path = heapq.heappop(open_list)  # Get node with lowest f value
        if current == end:
            return path  # Goal reached
        if current in visited:
            continue
        visited.add(current)

        # Explore 4 directions (up, down, left, right)
        for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:
            x, y = current[0] + dx, current[1] + dy
            # Check boundaries and obstacles
            if 0 <= x < len(grid) and 0 <= y < len(grid[0]) and grid[x][y] == 0:
                next_node = (x, y)
                if next_node not in visited:
                    new_path = path + [next_node]
                    heapq.heappush(open_list, (
                        g + 1 + heuristic(next_node, end),
                        g + 1,
                        next_node,
                        new_path
                    ))
    return None  # No path found

# 0 = free, 1 = obstacle
grid = [
    [0, 1, 0, 0],
    [0, 1, 0, 1],
    [0, 0, 0, 0],
    [1, 1, 1, 0]
]

start = (0, 0)
```

```
        end = (3, 3)

        print("Path found:", astar(grid, start, end))
```

AO* Algorithm

```
# Graph represented with AND/OR costs
graph = {
    'A': [('B', 1), ('C', 1)],
    'B': [('D', 1), ('E', 1)],
    'C': [('F', 1)],
    'D': [],
    'E': [],
    'F': []
}

# Heuristic values for each node
heuristic = {
    'A': 3,
    'B': 2,
    'C': 4,
    'D': 0,
    'E': 0,
    'F': 0
}

solved = {}  # Stores solved nodes
path = []    # Final solution path
```

```
# AO* Algorithm function
def ao_star(node):
    print(f"Expanding: {node}")
    if node in solved:
        return 0
    if not graph[node]:  # If it's a leaf node
        solved[node] = True
        return heuristic[node]

    costs = []  # Store (cost, path) options
    for child in graph[node]:
        if isinstance(child, tuple):
            cost = child[1] + heuristic[child[0]]
            costs.append((cost, [child[0]]))

    if not costs:
        return heuristic[node]

    # Choose minimum cost path
    min_cost, best_path = min(costs, key=lambda x: x[0])
    heuristic[node] = min_cost
    solved[node] = True
    path.append((node, best_path[0]))  # Add to path
    return min_cost

ao_star('A')
```

```
print("\nSolved Heuristics:", heuristic)
print("Solution Path:", path)
```

Alpha-Beta Pruning

```
# Function to perform Alpha-Beta Pruning
def alphabeta(depth, index, isMax, values, alpha, beta):
    # Base case: If we reach depth 3 (leaf level), return the value at that node
    if depth == 3:
        return values[index]

    # If it's MAX player's turn
    if isMax:
        best = -9999  # Start with the worst possible value for MAX

        # Try both child nodes (left and right)
        for i in range(2):
            # Move to the next level, switch to MIN's turn
            value = alphabeta(depth + 1, index * 2 + i, False, values, alpha, beta)

            # Keep the maximum value seen so far
            best = max(best, value)

            # Update alpha (best value MAX can guarantee so far)
            alpha = max(alpha, best)

            # If MAX's best is greater than or equal to MIN's best option, stop exploring
            if beta <= alpha:
```

```
            break  # This is called pruning
        return best

    # If it's MIN player's turn
    else:
        best = 9999  # Start with the worst possible value for MIN

        # Try both child nodes (left and right)
        for i in range(2):
            # Move to the next level, switch to MAX's turn
            value = alphabeta(depth + 1, index * 2 + i, True, values, alpha, beta)

            # Keep the minimum value seen so far
            best = min(best, value)

            # Update beta (best value MIN can guarantee so far)
            beta = min(beta, best)

            # If MIN's best is less than or equal to MAX's best option, stop exploring
            if beta <= alpha:
                break  # This is called pruning
        return best


# Leaf node values at depth 3 (final possible results of the game)
leaf_values = [3, 5, 6, 7, 1, 2, 0, -1]
```

```
# Start the Alpha-Beta pruning from root node
# depth = 0, index = 0, it's MAX's turn
# Alpha = -infinity, Beta = +infinity
result = alphabeta(0, 0, True, leaf_values, -9999, 9999)

# Print the best value MAX player can achieve
print("Best value for MAX player:", result)
```

DFS and BFS

```
# Graph representation
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': [],
    'D': [],
    'E': []
}

# Depth-First Search (recursive)
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
```

```
# Breadth-First Search (queue-based)
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node)
            visited.add(node)
            queue.extend(graph[node])

print("DFS (Depth-First Search):")
dfs(graph, 'A')

print("\nBFS (Breadth-First Search):")
bfs(graph, 'A')
```

Hill climbing algorithm

```
import random

def hill_climbing():
    current = random.randint(0, 100)
```

```
    print("Starting at:", current)

    while True:
        neighbor = current + random.choice([-1, 1])  # Try neighbor
        if neighbor > current:
            current = neighbor  # Move if better
            print("Moving to:", current)
        else:
            print("Reached peak at:", current)
            break

hill_climbing()
```

tower of Hanoi

```
def tower_of_hanoi(n, source, helper, target):
    if n > 0:
        tower_of_hanoi(n-1, source, target, helper)
        print(f"Move disk {n} from {source} to {target}")
        tower_of_hanoi(n-1, helper, source, target)

tower_of_hanoi(3, 'A', 'B', 'C')
```

Tic tak toe 2 player game

```
board = [' ']*9
```

```
def print_board():
    for i in range(0, 9, 3):
        print(board[i], '|', board[i+1], '|', board[i+2])

def game():
    turn = 'X'
    for _ in range(9):
        print_board()
        move = int(input(f"{turn}'s turn (0-8): "))
        if board[move] == ' ':
            board[move] = turn
            turn = 'O' if turn == 'X' else 'X'
        else:
            print("Invalid move.")
    print_board()

game()
```

water jug problem

```
def water_jug():
    from collections import deque
    visited = set()
    queue = deque([(0, 0)])  # Initial state (0,0)

    while queue:
```

```
        a, b = queue.popleft()
        if (a, b) in visited:
            continue
        visited.add((a, b))
        print(f"Jug1: {a}L, Jug2: {b}L")
        if a == 4 or b == 4:  # Goal state
            break
        queue.extend([
            (5, b), (a, 7), (0, b), (a, 0),
            (min(a + b, 5), max(0, a + b - 5)),
            (max(0, a + b - 7), min(a + b, 7))
        ])

water_jug()
```

4 queens problem

```
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or abs(board[i]-col) == abs(i-row):
            return False
    return True

def solve_n_queens(n, board=[], row=0):
    if row == n:
        print(board)
        return
```

```
    for col in range(n):
        if is_safe(board, row, col):
            solve_n_queens(n, board + [col], row + 1)

solve_n_queens(4)
```

8 puzzle problem

```
from collections import deque

def is_goal(state):
    return state == '123456780'  # Goal configuration

def get_neighbors(state):
    neighbors = []
    i = state.index('0')
    moves = [(-1,0),(1,0),(0,-1),(0,1)]
    x, y = divmod(i, 3)
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            ni = nx * 3 + ny
            l = list(state)
            l[i], l[ni] = l[ni], l[i]
            neighbors.append(''.join(l))
    return neighbors
```

```
def solve_puzzle(start):
    visited = set()
    queue = deque([(start, [])])
    while queue:
        state, path = queue.popleft()
        if state in visited:
            continue
        visited.add(state)
        if is_goal(state):
            print("Solved:", path + [state])

            return
        for neighbor in get_neighbors(state):
            queue.append((neighbor, path + [state]))

solve_puzzle('125340678')
```

Monkey banana problem

```
def monkey_banana():
    monkey_has_banana = False
    monkey_on_box = False
    banana_is_hanging = True

    print("Monkey sees banana hanging from ceiling.")
    print("Monkey moves box under banana.")
    monkey_on_box = True
```

```
    print("Monkey climbs box.")
    print("Monkey grabs banana.")
    monkey_has_banana = True
    banana_is_hanging = False

    if monkey_has_banana:
        print("Monkey got the banana!")

monkey_banana()
```