1) Towers of Hanoi (Recursion)

```python
def hanoi(n, source, target, auxiliary, moves):
    """Recursive solution for Towers of Hanoi."""
    if n == 0:
        return
    hanoi(n-1, source, auxiliary, target, moves)
    moves.append((n, source, target))
    hanoi(n-1, auxiliary, target, source, moves)

if __name__ == "__main__":
    n = 4  # number of disks
    moves = []
    hanoi(n, "A", "C", "B", moves)
    print(f"Total moves for {n} disks: {len(moves)}")
    for i, (disk, frm, to) in enumerate(moves, 1):
        print(f"{i}: Move disk {disk} from {frm} -> {to}")
```

---

2) Tic-Tac-Toe (Player vs Player)

```python
def print_board(b):
    for row in b:
        print(" | ".join(row))
```

```
        print("-"*9)

def check_win(b, player):
    for i in range(3):
        if all(b[i][j]==player for j in range(3)): return True
        if all(b[j][i]==player for j in range(3)): return True
    if b[0][0]==b[1][1]==b[2][2]==player: return True
    if b[0][2]==b[1][1]==b[2][0]==player: return True
    return False

def board_full(b):
    return all(b[i][j] != " " for i in range(3) for j in range(3))

def play():
    board = [[" "]*3 for _ in range(3)]
    current = "X"
    while True:
        print_board(board)
        move = input(f"Player {current}, enter row,col (1-3): ").split()
        if len(move) != 2 or not all(s.isdigit() for s in move):
            print("Invalid input.")
            continue
        r, c = int(move[0])-1, int(move[1])-1
        if not (0 <= r < 3 and 0 <= c < 3):
            print("Out of range.")
            continue
```

```
        if board[r][c] != " ":
            print("Cell occupied.")
            continue
        board[r][c] = current
        if check_win(board, current):
            print_board(board)
            print(f"Player {current} wins!")
            break
        if board_full(board):
            print_board(board)
            print("It's a draw.")
            break
        current = "O" if current == "X" else "X"

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

---

3) N-Queens Problem (Backtracking)

```
def solve_n_queens(n):
    solutions = []
    cols = set()
    diag1 = set()
```

```
    diag2 = set()
    board = [-1]*n

    def backtrack(r):
        if r == n:
            sol = []
            for i in range(n):
                row = ['.']*n
                row[board[i]] = 'Q'
                sol.append("".join(row))
            solutions.append(sol)
            return
        for c in range(n):
            if c in cols or (r-c) in diag1 or (r+c) in diag2:
                continue
            cols.add(c); diag1.add(r-c); diag2.add(r+c); board[r] = c
            backtrack(r+1)
            cols.remove(c); diag1.remove(r-c); diag2.remove(r+c); board[r] = -1

    backtrack(0)
    return solutions

if __name__ == "__main__":
    N = 8
    sols = solve_n_queens(N)
    print(f"Total solutions for N={N}: {len(sols)}")
```

```
    if sols:
        print("\n".join(sols[0]))
```

---

4) Random Maze Generator

```
import random
import numpy as np
import matplotlib.pyplot as plt

def make_maze(width, height):
    visual = np.zeros((2*height+1, 2*width+1), dtype=int)
    for i in range(height):
        for j in range(width):
            visual[2*i+1, 2*j+1] = 1

    visited = [[False]*width for _ in range(height)]
    stack = [(0,0)]
    visited[0][0] = True

    while stack:
        r,c = stack[-1]
        neighbors = []
        for dr, dc in [(0,1),(0,-1),(1,0),(-1,0)]:
```

```
            nr, nc = r+dr, c+dc
            if 0 <= nr < height and 0 <= nc < width and not visited[nr][nc]:
                neighbors.append((nr, nc, dr, dc))
        if neighbors:
            nr, nc, dr, dc = random.choice(neighbors)
            visual[2*r+1+dr, 2*c+1+dc] = 1
            visited[nr][nc] = True
            stack.append((nr, nc))
        else:
            stack.pop()
    return visual

def show_maze(visual):
    plt.figure(figsize=(6,6))
    plt.imshow(visual == 0, cmap="binary")
    plt.axis('off')
    plt.show()

if __name__ == "__main__":
    W, H = 20, 20
    vis = make_maze(W, H)
    show_maze(vis)
```

---

5) Research Note — Intelligent Agents and Their Types (Word Count ≈ 215)

Intelligent Agents

An intelligent agent is a system that perceives its environment through sensors and acts upon that environment using actuators to achieve specific goals. These agents can be simple, operating on pre-defined rules, or complex, incorporating perception, reasoning, learning, and decision-making capabilities.

The core components of an intelligent agent include:

Perception system: gathers data from the environment.

Decision-making mechanism: selects appropriate actions.

Actuators: execute the chosen actions.

Feedback loop: allows adaptation and learning.

Types of Intelligent Agents:

1. Simple Reflex Agents: Act only on the current percept without considering history. Example: thermostat.

2. Model-Based Reflex Agents: Maintain an internal model of the environment to handle partially observable situations.

3. Goal-Based Agents: Choose actions to achieve specific goals, often using search or planning algorithms.

4. Utility-Based Agents: Select actions based on a utility function that measures desirability among multiple possible outcomes.

5. Learning Agents: Improve their performance over time by learning from experience.

Applications of intelligent agents include chatbots, recommendation systems, autonomous vehicles, and industrial robots. With increasing complexity, modern agents integrate planning, probabilistic reasoning, and machine learning to operate in dynamic and uncertain environments. Ethical considerations, such as fairness and safety, are essential when deploying these systems in real-world settings.