AI Problem Question Templates

This document outlines the question templates, parameters, and evaluation logic for the AI problem-solving application.

1. N-Queens Problem

Array Representation: A solution for N=4 is `[1, 3, 0, 2]`, meaning queens are at (0, 1), (1, 3), (2, 0), and (3, 2).

Template 1.1: Theory (Time Complexity)

- **ID:** `nq_theory_complexity`
- **Question Type:** Theory
- **Parameterized Question:** "What is the time complexity of the standard Backtracking algorithm for finding all solutions to the {N}-Queens problem?"
- **Parameter Generation:**
  - `{N}`: Random integer (e.g., 8, 10, 16). The answer is independent of N, but N makes it feel concrete.
- **Expected Answer Format:** "Please provide the time complexity in Big-O notation (e.g., `O(N^2)`)."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** "O(N!)"
  - **Evaluation:**
    1. Clean user input (remove spaces, case-insensitive).
    2. Check for exact match: `o(n!)`.
    3. **Score:** 100% for a correct match, 0% otherwise.
  - **Explanation:** "The correct answer is **O(N!)**. This is because the algorithm explores a search tree. In the first row, it tries N positions. In the second, it tries at most N-1, and so on, leading to a factorial-like upper bound on the number of nodes explored."

Template 1.2: Validation (Placement Viability)

- **ID:** `nq_validation_viability`
- **Question Type:** Validation
- **Parameterized Question:** "For a {N}-Queens problem, consider the partially filled board: `{partial_board}`. Is it viable to place the next queen (in row {next_row}) at column {col_to_check}?"
- **Parameter Generation:**
  - `{N}`: Random integer (e.g., 5 to 8).
  - `{partial_board}`: Generate a valid partial board (e.g., `[1, 3, 0]`).
  - `{next_row}`: `len(partial_board)` (e.g., 3).

    - `{col_to_check}` : Random integer from `0` to `N-1` .
- **Expected Answer Format:** "Please answer 'Yes' or 'No'."
- **Evaluation & Explanation Logic:**
    - **Correct Answer:** (Run validation logic) -> "Yes" or "No".
    - **Evaluation:**
        1. Run a validation function `is_safe(board, row, col)` .
        2. `correct_answer = "Yes"` if `is_safe` is true, else `"No"` .
        3. Check `user_answer.strip().lower() == correct_answer.lower()` .
        4. **Score:** 100% for correct, 0% for incorrect.
    - **Explanation:**
        - **(If correct: No):** "The correct answer is **No**. Placing a queen at ({next_row}, {col_to_check}) conflicts with the queen at ({conflict_row}, {conflict_col}) on [the same column / the same diagonal]."
        - **(If correct: Yes):** "The correct answer is **Yes**. The position ({next_row}, {col_to_check}) does not conflict on any row, column, or diagonal with the existing queens in `{partial_board}` ."

Template 1.3: Computation (Solution Count)

- **ID:** `nq_computation_count`
- **Question Type:** Computation (Async)
- **Parameterized Question:** "How many distinct solutions are there for the {N}-Queens problem?"
- **Parameter Generation:**
    - `{N}` : Random integer (e.g., 4 to 9). *Higher N is very slow.*
- **Expected Answer Format:** "Please enter a single integer. If no solution exists, write 'impossible'."
- **Evaluation & Explanation Logic:**
    - **Correct Answer:** (Run backtracking solver to find all solutions) -> `count` .
    - **Evaluation:**
        1. Handle "impossible" case: If `N=2` or `N=3` , `correct_answer = "impossible"` .
        2. Otherwise, `correct_answer = str(count_solutions(N))` .
        3. Check `user_answer.strip().lower() == correct_answer` .
        4. **Score:** 100% for correct, 0% for incorrect.
    - **Explanation:** "The correct answer is **{correct_answer}**. For N={N}, there are {count} distinct solutions. [If N is small, e.g., N=4, show all solutions: `[[1, 3, 0, 2], [2, 0, 3, 1]]`]"

Template 1.4: Computation (Find One Solution)

- **ID:** `nq_computation_find_one`
- **Question Type:** Computation
- **Parameterized Question:** "Provide one valid solution for the {N}-Queens problem."

- **Parameter Generation:**
  - `{N}` : Random integer (e.g., 4 to 8).
- **Expected Answer Format:** "Please provide the solution as an array of {N} indices (e.g., `[1, 3, 0, 2]`). If no solution exists, write 'impossible'."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** (Find one solution) -> `[1, 3, 0, 2]` .
  - **Evaluation:**
    1. Handle "impossible" case (N=2, 3).
    2. Parse the user's answer string into an array. If parsing fails, score 0%.
    3. Run a full validation function `is_solution_valid(user_board, N)` .
    4. **Score:** 100% if `is_solution_valid` is true. 0% otherwise.
  - **Explanation:**
    - **(If user is correct):** "Excellent! Your solution `{user_board}` is a valid solution."
    - **(If user is incorrect):** "Your solution `{user_board}` is not valid. [Reason, e.g., 'There is a conflict between the queen at (0, {user_board[0]}) and (2, {user_board[2]})']. A correct solution is **{correct_solution}**."
    - **(If user says 'impossible' correctly):** "Correct! It is impossible to solve {N}-Queens."

Template 1.5: Experimental (Algorithm Race)

- **ID:** `nq_experimental_race`
- **Question Type:** Experimental (Async)
- **Parameterized Question:** "For a {N}x{N} board (N={N}), which algorithm will find a *single* solution first: Backtracking, Hill Climbing, or Simulated Annealing?"
- **Parameter Generation:**
  - `{N}` : Random integer (e.g., 8, 16, 32).
- **Expected Answer Format:** "Please enter the name of the algorithm you think will be fastest (e.g., 'Hill Climbing')."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** (Run the race) -> `fastest_algo_name` .
  - **Evaluation:**
    1. Run all three algorithms in parallel/threaded, recording their finish times to find *one* solution.
    2. `times = {'Backtracking': 0.1s, 'Hill Climbing': 0.05s, 'Simulated Annealing': 0.08s}`
    3. `fastest_algo_name = 'Hill Climbing'` .
    4. Clean user input: `user_algo = user_answer.strip().lower()` .
    5. Check for a match using Levenshtein distance < 3 (e.g., "hill climbing", "hillclimbing", "hll climbing" would match).

6. **Score:** 100% if `user_algo` matches `fastest_algo_name`. 0% otherwise.

- **Explanation:** "The fastest algorithm for this instance was **{fastest_algo_name}**.
  - `Backtracking` : {time_bt}s
  - `Hill Climbing` : {time_hc}s
  - `Simulated Annealing` : {time_sa}s (Note: For N={N}, a {diff}s difference is [significant/negligible]. Results may vary on different runs.)"

2. Knight's Tour

Template 2.1: Theory (Warnsdorff's Rule Name)

- **ID:** `kt_theory_warnsdorff_name`
- **Question Type:** Theory
- **Parameterized Question:** "Consider the following heuristic for the Knight's Tour problem: 'Always move to the unvisited square that has the *fewest* valid onward moves.' What is the name of this heuristic?"
- **Expected Answer Format:** "Please enter the name of the rule (e.g., 'Smith's Rule')."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** "Warnsdorff's Rule"
  - **Evaluation:**
    1. Clean user input: `user_answer = user_answer.strip().lower()`.
    2. Check Levenshtein distance against `"warnsdorff's rule"` and `"warnsdorffs rule"`.
    3. **Score:** 100% if distance < 3, 0% otherwise.
  - **Explanation:** "Correct! The answer is **Warnsdorff's Rule**. It is a highly effective heuristic that guides the knight toward the 'harder' squares first, preventing it from getting trapped in a corner."

Template 2.2: Validation (Move Viability)

- **ID:** `kt_validation_viability`
- **Question Type:** Validation
- **Parameterized Question:** "On a {N}x{N} board, given the partial tour: `{move_list}`, is the move from `{last_move}` to `{next_move}` a valid next step?"
- **Parameter Generation:**
  - `{N}` : 5x5 or 8x8.
  - `{move_list}` : A list of `(x, y)` tuples.
  - `{last_move}` : The last tuple in `{move_list}`.
  - `{next_move}` : A randomly chosen `(x, y)` tuple.
- **Expected Answer Format:** "Please answer 'Yes' or 'No'."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** (Run validation logic) -> "Yes" or "No".

- **Evaluation:**
  1. `is_valid` = Check if `{next_move}` is a valid knight's L-shape move from `{last_move}`.
  2. `is_visited` = Check if `{next_move}` is already in `{move_list}`.
  3. `correct_answer = "Yes"` if `is_valid` and not `is_visited`. Else "No".
  4. Check `user_answer.strip().lower() == correct_answer.lower()`.
- **Explanation:**
  - **(If No, not L-shape):** "No. The move from {last_move} to {next_move} is not a valid L-shape knight's move."
  - **(If No, visited):** "No. The square {next_move} has already been visited in this tour."
  - **(If Yes):** "Yes. The move from {last_move} to {next_move} is a valid knight's move to an unvisited square."

Template 2.3: Computation (Find Tour)

- **ID:** `kt_computation_find_tour`
- **Question Type:** Computation (Async)
- **Parameterized Question:** "Find a *closed* Knight's Tour on a {N}x{N} board, starting at ({x}, {y})."
- **Parameter Generation:**
  - `{N}`: e.g., 6 or 8 (5x5 has no closed tour).
  - `{x}, {y}`: Random start (e.g., 0, 0).
- **Expected Answer Format:** "Please provide the solution as an array of {N*N} tuples (e.g., `[(0,0), (1,2), ...]`). If no solution exists, write 'impossible'."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** (Run backtracking/Warnsdorff's) -> `solution_list`.
  - **Evaluation:**
    1. Parse user's answer into a list of tuples.
    2. Check `len(user_list) == N*N`.
    3. Check all squares are unique.
    4. Check each move `[i]` to `[i+1]` is a valid knight's move.
    5. Check if the last move can reach the first move (for *closed* tour).
    6. **Score:** 100% if all checks pass. 0% otherwise.
  - **Explanation:** "Your tour was {status}. [Reason, e.g., 'Move 10 from (a,b) to (c,d) was invalid.' or 'Your tour was open, not closed.']. A correct closed tour is: **{correct_solution}**."

Template 2.4: Experimental (Algorithm Race)

- **ID:** `kt_experimental_race`
- **Question Type:** Experimental (Async)

- **Parameterized Question:** "For a {N}x{N} board (N={N}), which algorithm will find a *single* (open) tour first: standard Backtracking or Backtracking with Warnsdorff's Rule?"
- **Parameter Generation:**
  - `{N}` : e.g., 8.
- **Expected Answer Format:** "Please enter 'Backtracking' or 'Warnsdorff's Rule'."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** (Run the race) -> `fastest_algo_name` .
  - **Evaluation:**
    1. Run both algorithms, recording their finish times. Warnsdorff's will be *dramatically* faster.
    2. `fastest_algo_name = "Warnsdorff's Rule"` .
    3. Clean user input: `user_algo = user_answer.strip().lower()` .
    4. Check for a match using Levenshtein distance < 3.
    5. **Score:** 100% if `user_algo` matches `fastest_algo_name` . 0% otherwise.
  - **Explanation:** "The fastest algorithm was **{fastest_algo_name}**.
    - `Backtracking` : {time_bt}s
    - `Warnsdorff's Rule` : {time_warnsdorff}s (Note: For N={N}, the simple backtracking algorithm explores many dead ends, while Warnsdorff's heuristic is very effective at finding a solution quickly.)"

3. Graph Coloring

Template 3.1: Theory (Chromatic Number Definition)

- **ID:** `gc_theory_definition_name`
- **Question Type:** Theory
- **Parameterized Question:** "Consider a graph G. The 'minimum number of colors needed to color the vertices of G so that no two adjacent vertices share the same color' is known by what name?"
- **Expected Answer Format:** "Please enter the name (e.g., 'Color Count')."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** "Chromatic Number"
  - **Evaluation:**
    1. Clean user input: `user_answer = user_answer.strip().lower()` .
    2. Check Levenshtein distance against `"chromatic number"` .
    3. **Score:** 100% if distance < 3, 0% otherwise.
  - **Explanation:** "Correct! The answer is the **Chromatic Number**, often written as χ(G). It is a fundamental property of a graph."

Template 3.2: Computation (Find Chromatic Number)

- **ID:** `gc_computation_find_chi`

- **Question Type:** Computation (Async)
- **Parameterized Question:** "What is the chromatic number for the following graph: {graph_adj_list}?"
- **Parameter Generation:**
  - `{graph_adj_list}` : Generate a random graph (e.g., cycle, complete, bipartite). (e.g., `{'A': ['B', 'C'], 'B': ['A', 'C'], 'C': ['A', 'B']}` is K3).
- **Expected Answer Format:** "Please enter a single integer."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** (Run graph coloring algorithm) -> `min_colors` .
  - **Evaluation:**
    1. Try to parse `user_answer` to an integer. If fails, score 0%.
    2. `int(user_answer) == min_colors` .
  - **Explanation:** "The correct chromatic number is **{min_colors}**. This graph is a [Graph Type, e.g., K3 (triangle)], which always requires {min_colors} colors."

Template 3.3: Validation (Coloring Viability)

- **ID:** `gc_validation_viability`
- **Question Type:** Validation
- **Parameterized Question:** "For the graph `{graph_adj_list}` , is the following coloring valid: `{coloring_map}` ?"
- **Parameter Generation:**
  - `{graph_adj_list}` : A random graph.
  - `{coloring_map}` : A map of `{'A': 'Red', 'B': 'Green', ...}` . This map will be generated to be *invalid* (with a conflict) 50% of the time.
- **Expected Answer Format:** "Please answer 'Yes' or 'No'."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** (Run validation) -> "Yes" or "No".
  - **Evaluation:**
    1. Iterate through all edges `(u, v)` in the graph.
    2. If `coloring_map[u] == coloring_map[v]` , it's invalid.
    3. `correct_answer = "No"` if a conflict is found, else "Yes".
    4. Check `user_answer.strip().lower() == correct_answer.lower()` .
  - **Explanation:** "The correct answer is **{correct_answer}**. [If No: 'There is a conflict between vertex {u} and {v}, which are adjacent but both colored {color}.']"

Template 3.4: Experimental (Algorithm Race)

- **ID:** `gc_experimental_race`
- **Question Type:** Experimental (Async)

- **Parameterized Question:** "For the graph with {N} nodes and {M} edges: `{graph_adj_list}`, which algorithm will find a valid (not necessarily optimal) coloring *first*: Simple Greedy or Welsh-Powell (Largest-Degree-First)?"
- **Parameter Generation:**
  - `{graph_adj_list}`: Generate a random graph with `{N}` nodes and `{M}` edges.
- **Expected Answer Format:** "Please enter 'Simple Greedy' or 'Welsh-Powell'."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** (Run the race) -> `fastest_algo_name`.
  - **Evaluation:**
    1. Run both algorithms, recording their finish times.
    2. `fastest_algo_name = ...` (This is a true experiment; sometimes one is faster, sometimes the other).
    3. Clean user input: `user_algo = user_answer.strip().lower()`.
    4. Check for a match using Levenshtein distance < 3.
    5. **Score:** 100% if `user_algo` matches `fastest_algo_name`. 0% otherwise.
  - **Explanation:** "The fastest algorithm for this instance was **{fastest_algo_name}**.
    - `Simple Greedy`: {time_greedy}s
    - `Welsh-Powell`: {time_wp}s (Note: Simple Greedy is simpler, but Welsh-Powell must first sort the vertices, which adds overhead but can lead to a faster coloring on some graph structures.)"

4. Generalized Towers of Hanoi

Template 4.1: Theory (Standard 3-Peg Moves)

- **ID:** `hanoi_theory_3peg_moves`
- **Question Type:** Theory
- **Parameterized Question:** "What is the minimum number of moves required to solve the standard Towers of Hanoi problem with {N} disks and 3 pegs?"
- **Parameter Generation:**
  - `{N}`: Random integer (e.g., 3 to 10).
- **Expected Answer Format:** "Please enter a single integer."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** `(2**N) - 1`.
  - **Evaluation:**
    1. Try to parse `user_answer` to an integer. If fails, score 0%.
    2. `int(user_answer) == (2**N) - 1`.
  - **Explanation:** "Correct! The formula for {N} disks and 3 pegs is 2^N - 1. So, 2^{N} - 1 = **{correct_answer}**."

Template 4.2: Theory (Recursive Step)

- **ID:** `hanoi_theory_recursive_step`
- **Question Type:** Theory
- **Parameterized Question:** "In the optimal recursive solution for moving {N} disks from Peg A to Peg C (using Peg B as auxiliary), where must the top {N-1} disks be moved *first*?"
- **Parameter Generation:**
  - `{N}` : Random integer (e.g., 5 to 10).
- **Expected Answer Format:** "Please enter the destination peg ('A', 'B', or 'C')."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** "B"
  - **Evaluation:**
    1. `user_answer = user_answer.strip().upper()` .
    2. Check if `user_answer == 'B'` or `user_answer == 'AUXILIARY'` .
    3. **Score:** 100% for correct, 0% otherwise.
  - **Explanation:** "Correct. The first step is to move the top {N-1} disks to the **auxiliary peg (B)**. The 3-step solution is:
    1. Move {N-1} disks from Source (A) to Auxiliary (B).
    2. Move 1 disk (the largest) from Source (A) to Destination (C).
    3. Move {N-1} disks from Auxiliary (B) to Destination (C)."

Template 4.3: Validation (Move Viability)

- **ID:** `hanoi_validation_viability`
- **Question Type:** Validation
- **Parameterized Question:** "Consider a {M}-peg Hanoi game. The pegs are: {peg_state}. Is it a valid move to take the top disk from Peg '{peg_from}' and place it on Peg '{peg_to}'?"
- **Parameter Generation:**
  - `{M}` : 3 or 4.
  - `{peg_state}` : A dictionary (e.g., `{'A': [5, 3], 'B': [4, 2], 'C': [1]}`). (Disks are listed bottom-to-top).
  - `{peg_from}` : Random peg (e.g., 'A').
  - `{peg_to}` : Random peg (e.g., 'B').
- **Expected Answer Format:** "Please answer 'Yes' or 'No'."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** (Run validation) -> "Yes" or "No".
  - **Evaluation:**
    1. Check `peg_from` is not empty. If empty, `correct = "No"` .
    2. `disk_to_move = peg_state[peg_from][-1]` .

3. Check if `peg_to` is empty. If yes, `correct = "Yes"`.
4. `top_disk_on_to = peg_state[peg_to][-1]`.
5. `correct = "Yes"` if `disk_to_move < top_disk_on_to`, else `correct = "No"`.

- **Explanation:**
  - **(If No, from_empty):** "No. Peg {peg_from} is empty."
  - **(If No, larger on smaller):** "No. You cannot place disk {disk_to_move} (from Peg {peg_from}) on top of the smaller disk {top_disk_on_to} (on Peg {peg_to})."
  - **(If Yes):** "Yes. Moving disk {disk_to_move} (from {peg_from}) onto disk {top_disk_on_to} (on {peg_to}) is a valid move."

Template 4.4: Theory (Generalized Moves)

- **ID:** `hanoi_theory_4peg_effect`
- **Question Type:** Theory
- **Parameterized Question:** "Compared to the 3-peg problem, what effect does adding a 4th peg have on the minimum number of moves required to move {N} disks?"
- **Parameter Generation:**
  - `{N}`: A large number (e.g., 64).
- **Expected Answer Format:** "Please answer with 'Increases', 'Decreases', or 'No Effect'."
- **Evaluation & Explanation Logic:**
  - **Correct Answer:** "Decreases"
  - **Evaluation:**
    1. Clean user input: `user_answer = user_answer.strip().lower()`.
    2. Check `user_answer == 'decreases'`.
    3. **Score:** 100% for correct, 0% otherwise.
  - **Explanation:** "Correct! The number of moves **Decreases** dramatically. For {N} disks, 3 pegs takes 2^{N}-1 moves (which is huge), while the optimal 4-peg solution (Frame-Stewart algorithm) is much, much faster."

Template 4.5: Experimental (Algorithm Race)

- **ID:** `hanoi_experimental_race`
- **Question Type:** Experimental (Async)
- **Parameterized Question:** "To generate all {move_count} moves for a {N}-disk, 3-peg Hanoi problem, which algorithm will finish *first*: the standard Recursive algorithm or an Iterative (stack-based) algorithm?"
- **Parameter Generation:**
  - `{N}`: Random integer (e.g., 18-22).
  - `{move_count}`: `2**N - 1`.
- **Expected Answer Format:** "Please enter 'Recursive' or 'Iterative'."

- **Evaluation & Explanation Logic:**
  - **Correct Answer:** (Run the race) -> `fastest_algo_name` .
  - **Evaluation:**
    1. Run both algorithms, recording their finish times.
    2. `fastest_algo_name = ...` (This is a true experiment, will depend on Python's function call overhead vs. list manipulation).
    3. Clean user input: `user_algo = user_answer.strip().lower()` .
    4. Check for a match using Levenshtein distance < 3.
    5. **Score:** 100% if `user_algo` matches `fastest_algo_name` . 0% otherwise.
  - **Explanation:** "The fastest algorithm for this instance was **{fastest_algo_name}**.
    - `Recursive` : {time_recursive}s