DAA_Notes

https://github.com/dhananjay-deshmukh/SPPU_DAA_Pracs

Fibonacci

Q1. What is the Fibonacci series?

A: The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones.
Example: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Q2. What is the formula to find the nth Fibonacci number?

A:

$F(n)=F(n-1)+F(n-2)F(n) = F(n-1) + F(n-2)F(n)=F(n−1)+F(n−2)$

with base cases

$F(0)=0,F(1)=1F(0) = 0,\quad F(1) = 1F(0)=0,F(1)=1$

Q3. What are the first 10 Fibonacci numbers?

A: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Q4. What does “step count” mean in this program?

A: The step count represents the **number of computational steps** (like function calls or iterations) taken by the algorithm. It helps to analyze **time complexity**.

🔹 Code-related Questions

Q5. How can you calculate Fibonacci numbers in Python?

A: There are multiple ways:

1. **Recursive approach** – calls itself for `F(n-1)` and `F(n-2)`.
2. **Iterative approach** – uses a loop to build up results.
3. **Dynamic Programming / Memoization** – stores previous results to avoid recomputation.

Q6. Write a simple recursive Fibonacci function.

```
steps = 0

def fib(n):
    global steps
    steps += 1
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

After calling `fib(n)`, you can print the step count:

```
print("Steps:", steps)
```

Q7. What is the drawback of the recursive approach?

A: It recomputes the same subproblems many times, leading to **exponential time complexity** — approximately **$O(2^n)$**.
For example, `fib(5)` calls `fib(3)` twice, `fib(2)` three times, etc.

Q8. How can we improve the recursive version?

A: Use **memoization** (store computed values in a dictionary or array) or switch to an **iterative approach**, both reducing complexity to **O(n)**.

Q9. How does the iterative Fibonacci program work?

A:
It starts from the base cases (0 and 1) and iteratively computes each next Fibonacci number using a loop:

```
steps = 0
def fib_iterative(n):
    global steps
    a, b = 0, 1
    for i in range(n):
        steps += 1
        a, b = b, a + b
    return a
```

Q10. How do we find the step count in the iterative version?

A: The step count is equal to the number of loop iterations (`n` times).
So for `n = 10`, steps = 10.

🔹 Algorithm Analysis Questions

Q11. What is the time complexity of recursive Fibonacci?

A: O(2^n) — exponential growth, since each call spawns two more calls.

Q12. What is the time complexity of iterative Fibonacci?

A: O(n) — linear growth, since we loop only once per Fibonacci number.

Q13. What is the space complexity for both?

Method	Space Complexity	Reason
Recursive	O(n)	Call stack grows with recursive calls
Iterative	O(1)	Only uses a few variables (a, b, steps)

Q14. Why is counting steps useful?

A: Step counting gives a **practical measure** of how many operations are performed, helping to compare algorithm efficiency and verify theoretical complexity.

Q15. How many steps are required for `fib(5)` in recursion?

A: The recursive version makes **15 function calls** for `fib(5)`.
This shows exponential growth in calls as `n` increases.

🔹 Higher-Level / Conceptual Questions

Q16. What is dynamic programming in Fibonacci calculation?

A: It's a method where we store results of smaller subproblems (`F(0)`, `F(1)`, etc.) and reuse them to avoid recomputation — either through **tabulation (bottom-up)** or **memoization (top-down)**.

Q17. What is the difference between memoization and tabulation?

Feature	Memoization	Tabulation

Approach	Top-down	Bottom-up
Implementation	Recursive with cache	Iterative using loops
Storage	Dictionary or array	DP table
Example	Recursive Fibonacci with cache	Iterative DP Fibonacci

---

Q18. What are the base cases in Fibonacci?

A:

- F(0) = 0
- F(1) = 1

These stop the recursion from continuing infinitely.

Q19. Can Fibonacci numbers be generated using formulas?

A: Yes. The **closed-form (Binet's Formula)** is:

F(n)=(ϕn−(1−ϕ)n)5F(n) = \frac{(\phi^n - (1-\phi)^n)}{\sqrt{5}}F(n)=5(ϕn−(1−ϕ)n)

where
ϕ=1+52\phi = \frac{1 + \sqrt{5}}{2}ϕ=21+5 (the golden ratio).

Q20. What are some real-life applications of the Fibonacci sequence?

A:

- Nature (flower petals, shells, tree branching)
- Financial market analysis

- Computer algorithms (e.g., Fibonacci heap)
- Art and architecture (Golden ratio design)

🧠 Bonus: Expected Output Example

If you run:

```
n = 6
result = fib(n)
print("Fibonacci number:", result)
print("Step count:", steps)
```

You might see something like:

```
Fibonacci number: 8
Step count: 25
```

For the iterative version:

```
Fibonacci number: 8
Step count: 6
```

==========

Fractional Knapsack

Q1. What is this program about?

A: This program solves the **Fractional Knapsack Problem** using a **greedy algorithm**. It selects items to maximize total value without exceeding the knapsack's weight capacity. Fractions of items can be taken.

Q2. What is the difference between 0/1 Knapsack and Fractional Knapsack?

A:

- **0/1 Knapsack:** You must take the *whole item or none* — no fractions allowed.
- **Fractional Knapsack:** You *can take fractions* of items — e.g., half or one-third of an item's weight.

Q3. What algorithmic approach does the Fractional Knapsack use?

A: It uses the **Greedy approach** — always take the item with the **highest value-to-weight ratio** first until the knapsack is full.

Q4. What does the `Item` class represent?

A: The `Item` class represents an object with two attributes:

- `weight`: how heavy the item is.
- `value`: how valuable it is.

Q5. What is the purpose of the `__init__` method?

A: `__init__` is a **constructor** in Python that initializes object attributes (`weight` and `value`) when an `Item` object is created.

Q6. Why do we calculate the value-to-weight ratio?

A: The **value-to-weight ratio** determines how valuable each unit of weight is.
Items with a higher ratio should be chosen first to maximize total value.

⚙️ Intermediate Questions

Q7. Why do we sort items in decreasing order of value-to-weight ratio?

A: Sorting ensures we pick the *most efficient items first* — those that give the most value per unit weight. This guarantees an optimal solution for the fractional version.

Q8. What does this line mean?

```
value_weight_ratio = [(item.value / item.weight, item) for item in items]
```

A: It creates a list of tuples, where each tuple stores `(value/weight ratio, item)`.
This helps in sorting and iterating based on the ratio.

Q9. What happens if the capacity becomes zero in the loop?

A: When capacity becomes zero, the knapsack is full.
The `if capacity == 0:` check breaks the loop to stop adding more items.

Q10. Why do we use `min(item.weight, capacity)`?

A: Because:

- If the full item fits, take all of it.
- If not, take only the remaining capacity's worth (a fraction).

Q11. What is the output of this program?

A: It prints:

1. Which items (and what fraction of each) were selected.

2. The **maximum total value** achievable within the given capacity.

For example:

```
Item with weight 10 and value 60 (fraction taken: 1.0)
Item with weight 20 and value 100 (fraction taken: 1.0)
Item with weight 30 and value 120 (fraction taken: 0.6666)
Maximum value achievable: 240.0
```

Q12. What is the time complexity of this program?

A:

- Sorting step → **O(n log n)**
- Loop over items → **O(n)**
  So, total time complexity = **O(n log n)**

Q13. What is the space complexity?

A:
It uses additional space for storing `(ratio, item)` pairs → **O(n)**

Q14. What data structure is used for storing selected items?

A: A **list of tuples**, where each tuple stores `(item, weight_taken)`.

💡 Conceptual / Theory-Based Questions

Q15. Why does the Greedy approach work for Fractional Knapsack but not for 0/1 Knapsack?

A:
In fractional knapsack, we can take *part of an item*, so taking the highest ratio first always leads

to an optimal solution.
But in 0/1 knapsack, we can't take fractions — sometimes, a combination of smaller-ratio items may give a better total value.

Q16. What will happen if an item has weight = 0?

A: Division by zero occurs when calculating `item.value / item.weight`.
In a real program, we should check and skip items with zero weight to avoid errors.

Q17. Can this algorithm handle negative values or weights?

A: No. The algorithm assumes **positive weights and values**. Negative values or weights make no logical sense in the knapsack context.

Q18. What will happen if the capacity is larger than the total weight of all items?

A: The knapsack will take **all items completely**, and the total value will be the **sum of all values**.

Q19. What is the output type of the `fractional_knapsack` function?

A: A **tuple**:
`(total_value, knapsack)` where:

- `total_value` → float (maximum value achievable)
- `knapsack` → list of `(item, weight_taken)` tuples.

Q20. Why do we write this condition?

```
if __name__ == "__main__":
```

A: It ensures the example code runs **only when the script is executed directly**, not when imported as a module in another file.

🧠 Bonus (Application Questions)

Q21. Where is the Fractional Knapsack problem used in real life?

A:

- Resource allocation (e.g., bandwidth distribution, CPU scheduling)
- Portfolio optimization in finance
- Cargo loading in logistics
- Cutting stock problems in manufacturing

Q22. What modifications are needed to make this a 0/1 Knapsack program?

A:

- Remove the fractional part (you either take the item or skip it).
- Replace the greedy approach with **Dynamic Programming**, as the greedy strategy doesn't work optimally for 0/1 knapsack.

==========

0/1 Knapsack

Q1. What is the 0/1 Knapsack problem?

A: The 0/1 Knapsack problem is an optimization problem where we have a set of items, each with a weight and a value. We must choose items to include in a knapsack so that the total weight does not exceed the knapsack's capacity, and the total value is maximized. Each item can either be **included (1)** or **excluded (0)** — hence the name *0/1 Knapsack*.

Q2. Why is it called “0/1” Knapsack?

A: Because each item can be **either taken (1)** or **not taken (0)** — there’s no concept of taking fractional parts of an item.

Q3. What type of problem is the 0/1 Knapsack?

A: It’s a **combinatorial optimization problem** and a classic example of **Dynamic Programming (DP)**.

Q4. What is the difference between 0/1 Knapsack and Fractional Knapsack?

A:

Feature	0/1 Knapsack	Fractional Knapsack
Item selection	Either 0 or 1	Can take fractions
Approach	Dynamic Programming	Greedy Algorithm
Example	Items cannot be divided	Items can be divided
Complexity	O(n × W)	O(n log n) (due to sorting)

🔹 Code Understanding Questions

Q5. What does `dp[i][w]` represent in the code?

A: `dp[i][w]` represents the **maximum value** that can be obtained using the **first i items** with a **knapsack capacity of w**.

Q6. What is the base condition in this code?

A:
If `i == 0` or `w == 0`, then `dp[i][w] = 0` because:

- With zero items or zero capacity, the total value must be 0.

Q7. Explain the line:

```
dp[i][w] = max(values[i-1] + dp[i-1][w - weights[i-1]], dp[i-1][w])
```

A: This line chooses the better of two options:

1. **Include the item:** Add its value and use the remaining capacity (`w - weights[i-1]`).
2. **Exclude the item:** Keep the previous best value for capacity `w`.
   The `max` chooses whichever gives the higher total value.

Q8. Why do we use `i-1` in `weights[i-1]` and `values[i-1]`?

A: Because `dp` has `n+1` rows (including the 0th row for base case).
The items list is 0-indexed, so the i-th item in `dp` corresponds to index `i-1` in the `values` and `weights` lists.

Q9. What does the backtracking part of the code do?

A: It reconstructs which items were included in the optimal solution by checking where `dp[i][w]` differs from `dp[i-1][w]`.
If the values differ, it means item `i-1` was included.

Q10. Why do we reverse the `selected_items` list at the end?

A: Because we trace back from the last item to the first one, the order of items gets reversed. Reversing the list restores the original order.

🔹 Output and Analysis Questions

Q11. What will be the output for the given example?

A:
Values = [60, 100, 120]
Weights = [10, 20, 30]
Capacity = 50

Selected Items → Item 1 (100, weight 20) and Item 2 (120, weight 30)
Maximum value achievable: 220

Q12. What is the time and space complexity?

A:

- **Time Complexity:** O(n × capacity) → because two nested loops iterate over items and capacities.
- **Space Complexity:** O(n × capacity) → for the DP table.

Q13. Can the space complexity be reduced?

A: Yes. We can use a **1D DP array** of size `(capacity + 1)` and update it in reverse order for each item, reducing space to **O(capacity)**. However, this makes backtracking harder.

Q14. What happens if all item weights are greater than the knapsack capacity?

A: The algorithm will never include any item, so the maximum value will be **0**.

Q15. What if two different combinations give the same maximum value?

A: The DP table will still give one valid combination, but there might be multiple valid sets of items yielding the same maximum value.

🔹 Higher-Level / Conceptual Questions

Q16. Why is Dynamic Programming better than recursion for this problem?

A: A pure recursive solution recomputes many subproblems repeatedly.
Dynamic Programming stores results in the `dp` table and avoids recomputation, making it much faster.

Q17. What is the difference between "top-down" and "bottom-up" DP approaches?

A:

- **Top-down (Memoization):** Uses recursion + caching (stores already computed results).
- **Bottom-up (Tabulation):** Iteratively fills a table (as in this code).
  This code uses **bottom-up** DP.

Q18. Can the knapsack capacity or weights be fractional?

A: No, 0/1 Knapsack assumes **integer** weights and capacities. Fractional values are handled by **Fractional Knapsack**, which uses a greedy approach.

Q19. What is meant by "optimal substructure" in this context?

A: The optimal solution to the knapsack problem can be built from **optimal solutions to its subproblems** — for example, best solution using first `i` items depends on best solutions using first `i-1` items.

Q20. What is "overlapping subproblems" property?

A: While solving recursively, many subproblems (same item index and capacity) are solved multiple times. DP solves each subproblem only once and stores the result.

==========

Huffman Encoding

Q1. What is Huffman Encoding?

A: Huffman Encoding is a **compression algorithm** that assigns variable-length binary codes to characters based on their frequency.
Characters that occur **more frequently** are assigned **shorter codes**, and less frequent ones get **longer codes**, which minimizes the total encoded length.

Q2. Who developed Huffman Coding and when?

A: It was developed by **David A. Huffman** in **1952** as part of his term paper at MIT.

Q3. What is the main goal of Huffman Encoding?

A: To **minimize the total number of bits** required to represent a message, thereby achieving **data compression**.

Q4. Which algorithmic strategy is used in Huffman coding?

A: It uses a **Greedy strategy** — always take the two nodes with the smallest frequencies first and combine them.

Q5. Why is Huffman Encoding called a “lossless” compression algorithm?

A: Because the original data can be **perfectly reconstructed** from the encoded data — there is **no loss of information** during compression or decompression.

🔹 Code-Related Questions

Q6. What are the main steps in Huffman Encoding?

A:

1. Count the frequency of each character.
2. Create a leaf node for each character and build a min-priority queue (min-heap).
3. Repeat until only one node remains:
    - Extract the two smallest frequency nodes.
    - Create a new internal node with frequency equal to their sum.
    - Add it back to the queue.
4. Assign binary codes to each character:
    - ‘0’ for the left branch, ‘1’ for the right branch.

Q7. What data structure is commonly used in Huffman coding implementation?

A: A **min-heap (priority queue)** is used to efficiently extract the two nodes with the smallest frequencies.

Q8. What does each node in the Huffman tree represent?

A:

- **Leaf node:** Represents a character and its frequency.
- **Internal node:** Represents the sum of frequencies of its child nodes.

Q9. What happens when two characters have the same frequency?

A: The order of merging doesn’t affect the total cost or final encoded length; multiple valid Huffman trees can exist for the same input.

Q10. What is stored in the root node of the Huffman tree?

A: The **sum of frequencies** of all characters (i.e., total number of symbols in the text).

🔹 Example-Based Questions

Q11. Suppose we have the following frequencies:

```
A:5, B:9, C:12, D:13, E:16, F:45
```

A: Huffman tree merges smallest pairs step by step:

1. (A,B) → 14
2. (C,D) → 25
3. (14,25) → 39
4. (E,39) → 55
5. (55,F) → 100
   Resulting codes (one possible set):

```
A:1100, B:1101, C:100, D:101, E:111, F:0
```

Q12. What is the total encoded cost (bits)?

A: The cost = Σ(frequency × code length)
Used to measure compression efficiency.

Q13. How does Huffman Encoding achieve compression?

A: By replacing fixed-length codes (e.g., 8-bit ASCII) with **variable-length codes** where frequent characters get fewer bits.

Q14. Can Huffman encoding produce a unique code for each character?

A: Yes, and the codes satisfy the **prefix property** — no code is a prefix of another, ensuring unique decoding.

🔹 Algorithm & Complexity Questions

Q15. What is the time complexity of Huffman encoding?

A:

- Building the min-heap: **O(n)**
- Combining nodes (extract + insert operations): **O(n log n)**
  So overall **O(n log n)** where n = number of unique characters.

Q16. What is the space complexity?

A: O(n) for storing nodes and the Huffman tree.

Q17. Why is Huffman encoding considered a Greedy algorithm?

A: Because at each step, it **greedily picks the two smallest frequency nodes** to merge — this locally optimal choice leads to a globally optimal solution.

Q18. What property of Huffman Tree ensures unique decoding?

A: The **prefix-free property** — no code word is a prefix of another code word.

Q19. What is meant by “optimal prefix code”?

A: A prefix code with the **minimum total weighted path length** — Huffman coding guarantees this optimality.

Q20. Is Huffman encoding always optimal?

A: Yes, for a **set of independent symbols** with **known probabilities**. But it is not always optimal when symbol probabilities change dynamically or for large alphabets — in such cases, **arithmetic coding** can perform better.

🔹 Practical / Implementation Questions

Q21. Which Python module can be used to simplify Huffman implementation?

A: The `heapq` module can be used for creating and managing a **min-heap (priority queue)** efficiently.

Q22. How are codes generated from the tree?

A: By performing a **tree traversal**:

- Append `'0'` when going left,
- Append `'1'` when going right,
  until a leaf node (character) is reached.

Q23. What happens during decoding?

A: We traverse the Huffman tree bit by bit (0 → left, 1 → right) until we reach a leaf node; then we output that character and restart from the root for the next bits.

Q24. Can Huffman encoding handle Unicode characters?

A: Yes, as long as you can count frequencies of characters — the algorithm is independent of character encoding.

Q25. Give a real-life application of Huffman encoding.

A:

- **File compression:** ZIP, GZIP
- **Multimedia compression:** JPEG, MP3
- **Data transmission:** Efficient encoding in networks

🔹 Example Output (For Small Input)

Input String:
`ABRACADABRA`

Frequencies:
A:5, B:2, R:2, C:1, D:1

Huffman Codes (possible):

```
A: 0
B: 101
R: 100
C: 1110
D: 1111
```

Encoded string: `01011001111001011110...`

🔹 Bonus: Key Points Summary for Quick Revision

Concept	Description
Strategy	Greedy
Data Structure	Min-Heap

Encoding Type	Variable-Length, Prefix-Free
Time Complexity	O(n log n)
Compression Type	Lossless
Invented by	David Huffman, 1952
Real-World Use	ZIP, JPEG, MP3

==========

N-Qeens

♟️ Program Overview

The **8-Queens problem** asks you to place **8 queens on a chessboard (8×8)** such that **no two queens attack each other** — meaning:

- No two queens share the same **row**,
- No two queens share the same **column**,
- No two queens share the same **diagonal**.

Your program **starts with one queen already placed**, and uses **backtracking** to place the remaining 7 queens.

🔹 Basic Conceptual Questions

Q1. What is the 8-Queens problem?

A: It is a classic **constraint satisfaction problem** where the goal is to place 8 queens on an 8×8 chessboard such that no two queens threaten each other.

Q2. Why can’t two queens be in the same row, column, or diagonal?

A: Because in chess, a queen can attack any piece that lies in the same **row**, **column**, or **diagonal**.

Q3. What is the backtracking technique?

A: Backtracking is a **trial-and-error algorithmic technique** that incrementally builds a solution and **abandons (backtracks)** as soon as it determines that the current partial solution cannot lead to a valid complete solution.

Q4. Why is backtracking suitable for the 8-Queens problem?

A: Because we can place queens **one by one** in each row, and if a conflict occurs later, we **backtrack** and move a previously placed queen to a new position — exploring all possible configurations systematically.

Q5. What is the size of the matrix used in this problem?

A: An **8 × 8 matrix**, representing an 8×8 chessboard.

🔹 Algorithm & Logic Questions

Q6. What are the main steps in solving the 8-Queens problem using backtracking?

A:

1. Start with an empty 8×8 board.
2. Place the first queen in a given position (as per the problem).
3. Try to place the next queen in the next row.
4. For each column in that row:
    - Check if placing a queen there is safe (no attack).

- If safe, place it and move to the next row.
- If not safe, try the next column.

5. If a placement leads to no solution, backtrack — remove the last placed queen and try a new position.
6. Repeat until all 8 queens are placed successfully.

Q7. What function is commonly used to check if a position is safe?

A: A helper function like:

```
def is_safe(board, row, col):
```

It checks the **column**, **upper-left diagonal**, and **upper-right diagonal** for any existing queens.

Q8. Why don't we need to check the rows during the "safe check"?

A: Because we always place **one queen per row**, so no two queens can ever share the same row.

Q9. What does "backtracking" mean in this context?

A: If a queen placement leads to a dead end (no safe column for the next queen), the algorithm **removes** the last queen (backtracks) and **tries a new column** in the previous row.

Q10. What is the base condition for the recursive function?

A: When the function reaches beyond the last row (`row == 8`), it means all queens have been successfully placed, so we've found a solution.

🔹 Code-Related Questions

Q11. How is the board represented in the code?

A: Typically as a **2D list (matrix)** in Python:

```
board = [[0 for _ in range(8)] for _ in range(8)]
```

Here, `1` represents a queen, and `0` represents an empty cell.

Q12. How do we place the first queen?

A: If the problem specifies that the **first queen is already placed**, we manually set:

```
board[first_row][first_col] = 1
```

Then the algorithm starts placing the remaining queens from the next row.

Q13. How do we move to the next step after placing a queen?

A: By calling the recursive function for the **next row**, e.g.:

```
if solve_n_queens(board, row + 1):
    return True
```

Q14. How do we backtrack in the code?

A: If placing a queen in a column doesn't lead to a solution, we remove it:

```
board[row][col] = 0
```

and try the next column.

Q15. What output does the program generate?

A: The program prints or returns an **8×8 matrix** with `1`s marking queen positions and `0`s elsewhere.
Example:

```
[0, 0, 1, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
...
```

🔹 Example-Based Questions

Q16. If the first queen is placed at (0, 0), what happens next?

A: The algorithm starts placing the next queen in row 1 and checks for a safe column that is not attacked by the queen in (0, 0).
It continues recursively until all queens are placed or backtracks if stuck.

Q17. How many possible solutions exist for the 8-Queens problem?

A: There are **92 valid solutions** (unique placements of 8 queens).

Q18. Does the position of the first queen affect the final matrix?

A: Yes. The initial placement can change the pattern of the final valid solution, but the algorithm will still find at least one valid configuration (if possible).

Q19. What is printed if no valid arrangement exists (for smaller boards)?

A: The program prints a message like `"No solution exists"` if the backtracking search fails (common for N < 4).

Q20. What is the difference between 8-Queens and N-Queens problem?

A: The **8-Queens** problem is a specific case where N = 8, while the **N-Queens** problem generalizes it for any board size N.

🔹 Algorithm Analysis Questions

Q21. What is the time complexity of the 8-Queens backtracking algorithm?

A: The **worst-case time complexity** is approximately **O(N!)** since, in the worst case, the algorithm tries every possible arrangement of queens.
For N = 8, this is manageable due to pruning (safe checks).

Q22. What is the space complexity?

A: O(N) for recursion stack (since we place one queen per row).

Q23. What is the advantage of using backtracking here?

A: It avoids exploring invalid configurations early by pruning the search space when a conflict is detected.

Q24. What is pruning in backtracking?

A: Pruning means **cutting off (skipping)** branches of the search tree that cannot lead to a valid solution, improving efficiency.

Q25. Can the algorithm find all possible solutions?

A: Yes — by **not stopping at the first valid configuration** and continuing the recursion after backtracking, it can list all 92 solutions.

🔹 Practical / Real-World Questions

Q26. Where is the N-Queens problem used in real life?

A: It is mainly a **benchmark problem** used for:

- Testing backtracking algorithms,
- Constraint satisfaction solvers,
- Optimization and AI systems,
- Scheduling problems.

Q27. Can we solve this using other techniques?

A: Yes, using **genetic algorithms**, **hill climbing**, or **constraint programming**, but backtracking is the most classical and straightforward approach.

Q28. Why is this problem significant in AI and algorithms?

A: Because it demonstrates **search**, **constraint satisfaction**, and **optimization**, all fundamental concepts in Artificial Intelligence and algorithm design.

🔹 Expected Output Example

If the first queen is placed at (0, 0), one valid final matrix could be:

```
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
```

Each 1 marks a queen.

🧠 Quick Summary Table for Viva

Concept	Explanation
Algorithm type	Backtracking
Input	8×8 matrix with one queen placed
Output	Final 8×8 matrix with all queens placed safely
Constraint	No two queens attack each other
Time complexity	O(N!)
Space complexity	O(N)
No. of solutions for 8-Queens	92
Key functions	`is_safe()`, `solve_queens()`
Backtracking step	Remove the last queen and try next column
Real-world relevance	Constraint satisfaction, scheduling, AI search