Perfect, Prajakta 👏
You’ve written a **clean Python program** that prints the Fibonacci series using both **non-recursive (iterative)** and **recursive** methods.
Let’s go step by step — explanation + viva preparation.

🧩 Code Explanation (Step by Step)

🔹 1. Non-Recursive Function

```
def fibonacci(n):
    fib_series = []
    a = 0
    b = 1

    for i in range(n):
        fib_series.append(a)
        a, b = b, a + b

    return fib_series
```

🔸 **What it does:**

- Initializes two variables:
    - a = 0 (1st Fibonacci number)
    - b = 1 (2nd Fibonacci number)
- Loops n times and appends a to the list.
- Updates values in each iteration using tuple assignment:
- a, b = b, a + b

So the next number becomes the sum of the previous two.

🔸 **Example:**

If n = 6,
→ Series = [0, 1, 1, 2, 3, 5]

◆ **Time Complexity:**

- Each loop runs once for every term → **O(n)**

◆ **Space Complexity:**

- Stores all numbers → **O(n)**

◆ 2. Recursive Function

```
def fibonacci_recursive(n):
  if n <= 0:
    return []
  elif n == 1:
    return [0]
  elif n == 2:
    return [0, 1]
  else:
    fib_series = fibonacci_recursive(n - 1)
    fib_series.append(fib_series[-1] + fib_series[-2])
    return fib_series
```

◆ **What it does:**

- **Base cases:**
    - n <= 0: return an empty list
    - n == 1: return [0]
    - n == 2: return [0, 1]
- **Recursive case:**
    - First calls fibonacci_recursive(n - 1) — gets series up to n-1.

    - Then appends the next number (sum of last two) to the series.

🔸 Example:

If n = 5,

- fibonacci_recursive(4) → [0, 1, 1, 2]
- Append 2 + 1 = 3 → [0, 1, 1, 2, 3]

🔸 Time Complexity:

- Each call makes one recursive call → total **O(n)** calls.
- But if written in the traditional recursive formula fib(n-1) + fib(n-2), it would be **O(2^n)**. Here, we use recursion only for building the list, so **O(n)**.

🔸 Space Complexity:

- Due to recursive call stack → **O(n)**

🔹 3. Main Program

```
n = int(input("Enter total numbers to print in fibonacci series:\t"))

print("Fibonacci Series (non-recusive):\t", fibonacci(n))

print("Fibonacci Series (recusive):\t\t", fibonacci_recursive(n))
```

- Takes user input (n)
- Calls both functions
- Prints both series for comparison

💻 Example Output

```
Enter total numbers to print in fibonacci series:   7

Fibonacci Series (non-recursive):   [0, 1, 1, 2, 3, 5, 8]

Fibonacci Series (recursive):       [0, 1, 1, 2, 3, 5, 8]
```

Both produce the same result ✅

⚙️ Algorithm Summary

Method	Approach	Time Complexity	Space Complexity	Comment
Non-recursive	Iterative	O(n)	O(n)	Fast and efficient
Recursive	Recursive (build list)	O(n)	O(n)	Simple but uses stack space

---

🎓 Viva Questions & Answers

🔹 Basic Questions

1. **Q:** What is the Fibonacci series?
   A: It's a sequence where each number is the sum of the previous two numbers.
   Example: 0, 1, 1, 2, 3, 5, 8, …
2. **Q:** What are the first two Fibonacci numbers?
   A: 0 and 1.
3. **Q:** What is the formula for the nth Fibonacci number?
   A:
   (F(n) = F(n-1) + F(n-2))
   with (F(0) = 0, F(1) = 1)

🔹 About the Code

4. **Q:** What is the difference between recursive and non-recursive methods?
   A:
   - Recursive: Function calls itself repeatedly.
   - Non-recursive: Uses loops and variables directly without function calls.
5. **Q:** Why do we use tuple assignment a, b = b, a + b?
   A: To update both variables in one step without losing the previous value of a.
6. **Q:** What are the base cases in recursion?
   A:

    - For n <= 0: returns an empty list
    - For n == 1: returns [0]
    - For n == 2: returns [0, 1]

7. **Q:** What happens if there are no base cases in a recursive function?
   A: The function calls itself infinitely, causing a **stack overflow** error.
8. **Q:** What is the advantage of the iterative method?
   A: It is **faster**, uses **less memory**, and avoids the overhead of recursive calls.
9. **Q:** Which method is more efficient for large n?
   A: The **non-recursive** (iterative) method.

◆ **Complexity & Concepts**

10. **Q:** What is the time complexity of both methods here?
    A:
    - Recursive: **O(n)** (since it builds the list once per call)
    - Iterative: **O(n)**
11. **Q:** What is the space complexity of recursion and iteration?
    A:
    - Recursive: **O(n)** (due to call stack)
    - Iterative: **O(n)** (for storing list only)
12. **Q:** What is recursion?
    A: It is a programming technique where a function calls itself to solve smaller subproblems.
13. **Q:** What are the main parts of recursion?
    A:
    - **Base case:** condition to stop recursion
    - **Recursive case:** step where function calls itself
14. **Q:** What is the drawback of recursion?
    A: It uses more memory and can be slower due to repeated function calls.

🔹 Advanced / Bonus Questions

15. **Q:** Can Fibonacci be calculated without storing the series?
    A: Yes, by printing terms directly instead of storing them in a list.
16. **Q:** How can recursion be optimized in Fibonacci?
    A: By using **memoization** (storing previously computed values).
17. **Q:** What is tail recursion?
    A: A type of recursion where the recursive call is the last operation in the function — it can be optimized by the compiler to save stack space.
18. **Q:** Is Fibonacci series an example of dynamic programming?
    A: Yes — if we store previously computed values to avoid recomputation.

🧠 Quick Summary

Concept	Recursive	Iterative
Approach	Function calls itself	Uses loops
Base Case	Required	Not needed
Time Complexity	O(n)	O(n)
Space Complexity	O(n)	O(n)
Speed	Slower	Faster
Memory Usage	More	Less

Would you like me to give you a **short viva notes PDF (2-page format)** for this program — summarizing explanation, code, and Q&A neatly for submission or study?

#############################

🧩 1. What is Huffman Encoding?

Huffman Encoding is a **lossless data compression algorithm**.
It assigns **shorter binary codes** to **frequent characters** and **longer codes** to **less frequent characters** — reducing the overall size of data.

It uses a **greedy approach**:

Always pick the two characters with the smallest frequency and merge them until one tree remains.

⚙️ 6. Why It’s a Greedy Algorithm

At every step:

- We choose **two smallest frequencies** and combine them.
- This **locally optimal choice** (smallest first) leads to a **globally optimal code** (minimal weighted path length).

Hence, **Huffman Encoding uses a greedy strategy**.

⏱️ 7. Complexity Analysis

Type	Explanation	Complexity
Time Complexity	Each insertion/removal in a priority queue takes O(log n), done n times	**O(n log n)**
Space Complexity	Stores all tree nodes and codes	**O(n)**

🎓 8. Viva Questions

Question	Answer
Q1: What is Huffman Encoding?	A method of compressing data by assigning variable-length codes to characters based on frequency.

Question	Answer
Q2: Why is it called a Greedy algorithm?	Because it always merges the two smallest frequencies first, making the best local choice.
Q3: What data structure is used?	**Min-Heap / Priority Queue**.
Q4: What is the time complexity?	**O(n log n)**.
Q5: What property ensures unique decoding?	It’s a **prefix-free code** — no code is a prefix of another.
Q6: What are the applications?	Used in **JPEG, MP3, ZIP compression**, etc.
Q7: What happens if all frequencies are equal?	Each character gets the same length code.

🔍 Now your question: "When two frequencies are same, what happens?"

👉 When **two characters have the same frequency**, the **priority queue** can choose them in **any order**, depending on how Java’s PriorityQueue internally arranges elements of equal priority.

So for H(1), E(1), O(1) — any of them can be combined first.
That’s why:

- The **structure of the tree** might differ slightly.
- The **Huffman codes** (bit patterns) may differ,
- **But the encoded text will still decompress correctly** — because both the encoder and decoder use the same tree.

✅ **Important:**
Even if codes change, **compression efficiency remains same** — since frequency counts are the same.

🧠 Summary:

Concept	Explanation
Equal frequencies	Order of merging may change
Effect on codes	Bit patterns differ but valid
Effect on compression	None — total bits remain minimal
Example	H(1) and E(1) could be swapped, giving H→01 and E→00, but both correct

##############################

🧮 **(1) Fractional Knapsack using Greedy Method**
🎒 **(2) 0/1 Knapsack using Dynamic Programming**

Below are **important viva questions and answers** — categorized as **basic**, **conceptual**, and **code-specific** for each.

🧩 PART 1: Fractional Knapsack (Greedy Approach)

🔹 Basic Questions

1. **Q:** What is the knapsack problem?
   A: It is an optimization problem where we must maximize total value by selecting items within a given weight limit.
2. **Q:** What is the difference between 0/1 and Fractional Knapsack?
   A:
   - In **0/1 Knapsack**, each item must be taken **completely or not at all**.
   - In **Fractional Knapsack**, we can take **fractions** of items.
3. **Q:** What strategy is used in Fractional Knapsack?
   A: It uses the **Greedy approach** — always pick the item with the **highest value-to-weight ratio first**.

◆ Conceptual Questions

4. **Q:** Why is the greedy method optimal for the Fractional Knapsack problem?
   A: Because each choice (taking the next highest ratio item) does not affect future choices — so local optimum leads to global optimum.

5. **Q:** What happens if we sort items by value instead of ratio?
   A: It may not give the optimal solution, because a high-value item might have too much weight and lower overall efficiency.

6. **Q:** What is the time complexity of the Fractional Knapsack algorithm?
   A:
    - Sorting: **O(n log n)**
    - Selection loop: **O(n)**
    - Overall: **O(n log n)**

7. **Q:** What is the space complexity?
   A: O(n) — for storing ratios and item info.

8. **Q:** Can we use dynamic programming for fractional knapsack?
   A: Not required — because the greedy approach gives an optimal solution directly.

◆ Code-Specific Questions

9. **Q:** What does ratio[i][1] = val[i] / (double)weight[i]; do?
   A: It calculates the **value-to-weight ratio** for each item as a floating-point number.

10. **Q:** Why do we sort the array in ascending order and then loop backward?
    A: Because sorting ascending and looping from end gives items with the **highest ratio first**.

11. **Q:** Why is the final value a double?
    A: Because we may add **fractional values** (not always integers).

12. **Q:** How do you handle the fractional part in the code?
    A: When remaining capacity < current item's weight, we multiply ratio * remaining capacity.

🎒 PART 2: 0/1 Knapsack (Dynamic Programming)

◆ Basic Questions

1. **Q:** What is the 0/1 Knapsack problem?
   A: Given weights and values of items, find the maximum value that fits in a knapsack of capacity W, where each item can either be taken or not taken (no fractions).
2. **Q:** Why is this called “0/1”?
   A: Because each item can be chosen **0 times (excluded)** or **1 time (included)** only.
3. **Q:** What technique is used to solve it?
   A: Dynamic Programming (Bottom-Up approach).

◆ Conceptual Questions

4. **Q:** Why can’t we use a greedy approach for 0/1 knapsack?
   A: Because taking the highest ratio item first doesn’t always lead to the global optimum (some combinations may yield higher total value).
5. **Q:** What is the recurrence relation used?
   A:
   [
   dp[i][w] = \begin{cases}
   0 & \text{if } i = 0 \text{ or } w = 0 \
   dp[i-1][w] & \text{if } wt[i-1] > w \
   \max(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w]) & \text{otherwise}
   \end{cases}
   ]
6. **Q:** What do i and w represent in dp[i][w]?
   A:
   - i → number of items considered
   - w → current capacity of knapsack
7. **Q:** What is the time complexity of this approach?
   A: O(n × W) (where n = items, W = capacity).
8. **Q:** What is its space complexity?
   A: O(n × W) (can be optimized to **O(W)** using 1D array).

9. **Q:** What type of problem is the 0/1 Knapsack?
   A: It is a **combinatorial optimization** and **NP-complete** problem.

◆ Code-Specific Questions

10. **Q:** Why do we initialize first row and column of dp[][] to 0?
    A: Because with 0 items or 0 capacity, no value can be achieved.

11. **Q:** What does the statement

12. dp[i][w] = Math.max(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w]);

mean?
A: It chooses the **maximum value** between including or excluding the current item.

13. **Q:** Why do we use i-1 when accessing val[] and wt[]?
    A: Because i in DP represents items count, but array indices start from 0.

14. **Q:** What does dp[n][W] finally represent?
    A: The **maximum possible value** with n items and knapsack capacity W.

15. **Q:** What kind of Dynamic Programming approach is this — Top-down or Bottom-up?
    A: Bottom-up (iterative).

16. **Q:** Can we solve this recursively (without DP)?
    A: Yes, but it would be **exponential (O(2^n))**, so DP is preferred for efficiency.

▦ Bonus Conceptual Questions (Common for Both)

Question	**Short Answer**
What is Greedy algorithm?	An algorithm that makes the locally optimal choice at each step.
What is Dynamic Programming?	A method of solving complex problems by breaking them into overlapping subproblems and storing results to avoid recomputation.
What is the difference between Greedy and DP?	Greedy makes local choices; DP explores all subproblems and ensures a global optimum.

Question	**Short Answer**
Which approach is faster for large W?	Fractional (O(n log n)) is faster but only for fractional cases.
Which gives exact optimal solution for 0/1 case?	Dynamic Programming.

Would you like me to make a **PDF viva sheet** (formatted Q&A list for print or submission)?

##

Let’s go through your **N-Queen Python program** carefully — first with a **clear explanation**, then **time & space complexity**, and finally a list of **possible viva questions with answers** 💡

🧩 Program Explanation (N-Queen Problem)

🎯 Goal:

Place **N queens** on an **N×N chessboard** such that **no two queens attack each other** — meaning:

- No two queens are in the same **row**, **column**, or **diagonal**.

🧠 Code Breakdown:

1. placeQueens() — Recursive Backtracking Function

```
def placeQueens(i, cols, leftDiagonal, rightDiagonal, cur):
  n = len(cols)

  if i == n:
    return True
```

- i = current row index where we want to place a queen.
- Base case: If i == n, all queens are successfully placed → return True.

---

2. Trying Each Column in Row i

```
for j in range(n):
    if cols[j] or rightDiagonal[i + j] or leftDiagonal[i - j + n - 1]:
        continue
```

- Loop through each column j.
- Skip if that column or either diagonal is already **attacked**.

---

3. Place the Queen Temporarily

```
cols[j] = 1
rightDiagonal[i + j] = 1
leftDiagonal[i - j + n - 1] = 1
cur.append(j)
```

- Mark column and diagonals as occupied.
- Add the column position j to the cur list.

---

4. Recursive Call to Next Row

```
if placeQueens(i + 1, cols, leftDiagonal, rightDiagonal, cur):
    return True
```

- Try to place the next queen (i+1).
- If successful → return True.

---

5. Backtrack if Not Successful

```
cur.pop()
cols[j] = 0
rightDiagonal[i + j] = 0
```

```
leftDiagonal[i - j + n - 1] = 0
```

- Remove the queen and **unmark** the column and diagonals.
- Try next possible column.

6. No Solution for This Row

```
return False
```

If all columns in the current row fail, backtrack to the previous row.

7. nQueen() — Initializes the Board and Starts the Process

```
cols = [0] * n

leftDiagonal = [0] * (n * 2)

rightDiagonal = [0] * (n * 2)

cur = []

board = [['.' for _ in range(n)] for _ in range(n)]
```

- cols, leftDiagonal, and rightDiagonal keep track of **attacked positions**.
- board initializes a blank chessboard (. means empty).

8. Call the Recursive Function

```
if placeQueens(0, cols, leftDiagonal, rightDiagonal, cur):
  for i in range(n):
    board[i][cur[i]] = 'Q'
  return board
```

- Start placing queens from row 0.
- If successful, mark 'Q' in each row’s respective column.

9. Print the Board

```
def printBoard(board):
    if board:
        for row in board:
            print(" ".join(row))
    else:
        print("No solution exists.")
```

- Displays the chessboard with queens placed.

🧾 Example Output:

```
Enter the number of queens: 4
. Q . .
. . . Q
Q . . .
. . Q .
```

⚙️ Time and Space Complexity

Aspect	Explanation	Complexity
Time Complexity	In the worst case, the algorithm explores almost all configurations. It's **O(N!)** (factorial growth) because for each row, we try all possible column placements.	**O(N!)**
Space Complexity	Uses 3 arrays (cols, leftDiagonal, rightDiagonal) of size O(N) and recursion stack of depth O(N)	**O(N)**

🎓 Viva Questions and Answers

🧠 Basic Conceptual Questions

1. **Q:** What is the N-Queen problem?
   A: It is the problem of placing N queens on an N×N chessboard such that no two queens attack each other horizontally, vertically, or diagonally.

2. **Q:** Which algorithm is used to solve the N-Queen problem?
   A: Backtracking algorithm.

3. **Q:** What is backtracking?
   A: It’s a method of solving problems incrementally by trying partial solutions and abandoning them (“backtracking”) if they don’t lead to a valid full solution.

⚙️ Code-Related Questions

4. **Q:** What do the cols, leftDiagonal, and rightDiagonal arrays represent?
   A:
   - cols[j] → Whether column j is occupied.
   - leftDiagonal[i - j + n - 1] → Marks / diagonal occupancy.
   - rightDiagonal[i + j] → Marks \ diagonal occupancy.

5. **Q:** Why do we use i - j + n - 1 for the left diagonal?
   A: Because diagonal indices can be negative (i - j), so we offset by (n - 1) to make them positive.

6. **Q:** What does backtracking mean in this program?
   A: If placing a queen in a position leads to no solution, we remove (backtrack) the queen and try the next possible column.

7. **Q:** What happens when placeQueens() returns True?
   A: It means a valid arrangement of N queens has been found, and recursion stops.

8. **Q:** Can there be multiple solutions?
   A: Yes, but this code returns only the **first valid solution**. To get all solutions, we’d need to store each valid configuration instead of stopping after the first one.

📊 Complexity & Theory Questions

9. **Q:** What is the time complexity of the N-Queen problem using backtracking?
   A: O(N!), as we try placing queens in every possible column for each row.

10. **Q:** What is the space complexity?
    A: O(N) — for tracking arrays and recursion stack.
11. **Q:** Can the N-Queen problem be solved using dynamic programming?
    A: No, because it’s a **constraint satisfaction** problem, not an optimization problem.

Would you like me to also show how to modify this same code to **print *all possible solutions*** (not just one)? That’s a common follow-up viva question.