Code explanation:

Absolutely, Vishwas. Let's break down the **entire second code (class-based approach)** line-by-line and explain **everything in depth**, assuming you're familiar with basic C++ and STL like vector, string, loops, and functions.

**🔧 Code: Full Deep Explanation**

#include <bits/stdc++.h>

using namespace std;

* #include <bits/stdc++.h>: This includes **all standard C++ libraries**. It's commonly used in competitive programming to save time.
* using namespace std;: This allows you to use vector, cout, etc., without std:: prefix.

**✅ Class Declaration**

class Solution {

We are using **Object-Oriented Programming**. All logic is encapsulated inside the Solution class.

**🔐 Function 1: isSafe1()**

bool isSafe1(int row, int col, vector<string> board, int n)

This function **checks if it's safe to place a queen** at position (row, col) on the current board. The function looks **only on the left side**, because we're placing queens **from left to right**, column-by-column.

**Let's break it down further:**

**🟠 Upper-left diagonal ↖**

int duprow = row;

int dupcol = col;

while (row >= 0 && col >= 0) {

if (board[row][col] == 'Q') return false;

row--;

col--;

}

* Moves diagonally **up and to the left**.
* If it finds a 'Q', returns false – it’s **not safe**.

**🟡 Left row ←**

row = duprow;

col = dupcol;

while (col >= 0) {

if (board[row][col] == 'Q') return false;

col--;

}

* Moves **straight left** in the same row.
* Checks if there’s already a queen.

**🔵 Lower-left diagonal ↙**

row = duprow;

col = dupcol;

while (row < n && col >= 0) {

if (board[row][col] == 'Q') return false;

row++;

col--;

}

* Moves diagonally **down and to the left**.
* If it finds a 'Q', placement is not safe.

✅ If all three checks pass, returns true.

return true;

**🔄 Function 2: solve()**

void solve(int col, vector<string> &board, vector<vector<string>> &ans, int n)

This is the **core backtracking function**.

* col: Current column to place a queen.
* board: The current board.
* ans: A 2D vector storing all valid board configurations.
* n: Board size.

**🔁 For each row in the current column:**

for (int row = 0; row < n; row++) {

if (isSafe1(row, col, board, n)) {

board[row][col] = 'Q'; // place queen

solve(col + 1, board, ans, n); // recursively move to next column

board[row][col] = '.'; // backtrack if needed

}

}

* Try every row in the current column.
* If isSafe1 says it’s okay:
  + Place a queen.
  + Move to the **next column** recursively (col + 1).
  + If placing a queen doesn’t work later, remove the queen and try next row (backtracking).

**✅ Base Case: All queens placed**

if (col == n) {

ans.push\_back(board);

return;

}

* When col == n, it means we placed n queens (one per column).
* So, we **store this board in the answer**.

**🧠 Function 3: solveNQueens()**

vector<vector<string>> solveNQueens(int n)

This function initializes variables and **starts the solving process**.

**Initialization:**

vector<vector<string>> ans;

vector<string> board(n);

string s(n, '.');

* ans: stores all final solutions.
* board: an N×N board made of strings like ["....", "....", ...].
* s: a string of n dots like "...." (empty row).

for (int i = 0; i < n; i++) {

board[i] = s;

}

* Fills all n rows with "...." (i.e., no queens placed yet).

**Start the backtracking:**

solve(0, board, ans, n);

return ans;

* Starts solving from column 0.

**🧾 Function: main()**

int main() {

int n = 8; // or 4, 5, etc.

Solution obj;

vector<vector<string>> ans = obj.solveNQueens(n);

* Creates an object of class Solution.
* Calls the solveNQueens(n) function to get all solutions.

**Prints all solutions:**

for (int i = 0; i < ans.size(); i++) {

cout << "Arrangement " << i + 1 << "\n";

for (int j = 0; j < ans[0].size(); j++) {

cout << ans[i][j];

cout << endl;

}

cout << endl;

}

* Outer loop: each board (solution).
* Inner loop: each row in the board.
* ans[i][j] is a string like "..Q.".

**💡 How It Works – Step by Step (n = 4 Example)**

1. Start with an empty board.
2. Place queen in first column (try rows 0–3).
3. For each successful placement:
   * Move to next column.
   * Repeat safety check.
4. If stuck → backtrack (remove last queen).
5. If all columns are filled → store the board.

**🧠 What You’ve Learned (Key Concepts)**

| **Concept** | **Explanation** |
| --- | --- |
| **Backtracking** | Try, fail, backtrack, try another |
| **Recursion** | Solve smaller sub-problems (col + 1) |
| **Safety Check** | Avoid placing queen where it can be attacked |
| **2D Board** | Represented using vector of strings |
| **Base Case** | All queens placed = valid solution |

**🔁 Final Note**

This approach is readable and useful for:

* Visual understanding (as the board is string-based)
* Generating all solutions
* Learning backtracking

But it can be **optimized** using additional arrays (cols, diag1, diag2) like your first version for speed.

Would you like me to show:

* How to **optimize this with hash arrays** (like first code)?
* A **visual demo with N = 4** step-by-step?
* Or help you **modify the code** to find only the first solution instead of all?

Let me know!

Great, Vishwas! Since your problem statement is to **implement the N-Queens problem using Backtracking and optionally Branch and Bound**, and the implementation is in C++, I’ll give you:

**✅ Expected Viva Questions with Answers**

**→ Based on: Problem Statement, Code, Logic, Algorithms**

**🔷 PART 1: Problem & Algorithm Understanding**

**1. ❓ What is the N-Queens problem?**

**Answer:**  
The N-Queens problem is a classic **constraint satisfaction problem** where the goal is to place **N queens on an N×N chessboard** such that **no two queens attack each other**. That means:

* No two queens can be in the same row
* No two queens can be in the same column
* No two queens can be on the same diagonal

**2. ❓ Which algorithm is used to solve the N-Queens problem in your code?**

**Answer:**  
The algorithm used is **Backtracking**. We try to place queens column-by-column and **backtrack** when we find a conflict. The code explores all valid configurations recursively and stores the valid ones.

**3. ❓ What is backtracking?**

**Answer:**  
Backtracking is a **recursive algorithmic technique** for solving problems incrementally. It builds the solution piece by piece and removes the last added piece (backtracks) when it determines that the current solution doesn't work.

**4. ❓ What is Branch and Bound?**

**Answer:**  
Branch and Bound is an optimization technique where we **prune branches** of the search tree that are guaranteed not to lead to a valid or optimal solution. In the N-Queens context, this can be done by keeping arrays to quickly check whether a queen can be placed (e.g., cols[], diag1[], diag2[]) — which helps reduce the number of recursive calls.

**5. ❓ How is safety of a queen’s position ensured in your code?**

**Answer:**  
In the isSafe1() function, we check:

* The left side of the row
* The upper-left diagonal
* The lower-left diagonal  
  Because queens are being placed column-by-column from left to right, we don’t need to check the right side.

**6. ❓ What is the time complexity of the N-Queens backtracking solution?**

**Answer:**  
The worst-case time complexity is **O(N!)** because for each column, we try placing a queen in N rows, and for each successful placement, we move to the next column. But pruning (like with isSafe1() or hash arrays) helps reduce actual operations.

**7. ❓ What is the base condition in your recursive function?**

**Answer:**  
In the solve() function, the base case is:

if (col == n)

This means that we have successfully placed all N queens and can add the current board configuration to the answer.

**🔷 PART 2: Code-Specific Questions**

**8. ❓ Why do you use vector<string> board(n, string(n, '.'))?**

**Answer:**  
This creates a board of size N×N filled with '.' characters, which represent **empty squares**. We use 'Q' to represent placed queens later in the code.

**9. ❓ What is the use of the solve() function?**

**Answer:**  
The solve() function is the **recursive backtracking function**. It tries to place a queen in each row of the current column, checks if the position is safe, places the queen, and then recursively calls itself for the next column.

**10. ❓ Why do we backtrack using board[row][col] = '.'?**

**Answer:**  
This line **removes a previously placed queen** if the next recursive call fails (i.e., no valid placement in further columns). It's essential for trying different possibilities — the heart of backtracking.

**11. ❓ How many solutions are there for N = 4?**

**Answer:**  
There are **2 valid solutions** for N = 4. The code returns all such configurations.

**12. ❓ Can we use Branch and Bound here? How?**

**Answer:**  
Yes. We can use Branch and Bound by keeping three arrays:

* cols[n]: Tracks which columns are occupied
* diag1[2n-1]: Tracks '/' diagonals (row + col)
* diag2[2n-1]: Tracks '' diagonals (row - col + n - 1)  
  This improves time efficiency by avoiding multiple checks in isSafe().

**🔷 PART 3: Theory / Related Concepts**

**13. ❓ Is N-Queens an NP problem?**

**Answer:**  
Yes, the N-Queens problem is a **constraint satisfaction problem (CSP)** and belongs to the **NP (nondeterministic polynomial time)** class. Finding one solution is not too hard, but checking all solutions requires backtracking unless optimized.

**14. ❓ How many solutions exist for different N?**

**Answer:**  
Examples:

* N = 1 → 1
* N = 2 or 3 → 0 (no solution)
* N = 4 → 2
* N = 8 → 92

**15. ❓ What data structures are used in your code?**

**Answer:**

* vector<string> → to represent the board
* vector<vector<string>> → to store all valid boards
* string s(n, '.') → single row initialization  
  These are used for efficient storage and manipulation.

**16. ❓ Can this problem be solved using graphs?**

**Answer:**  
Yes, it can be modeled as a **graph coloring problem**, where:

* Each column can be a node.
* A queen placement is a color assignment.
* Constraints prevent adjacent nodes (columns) from having conflicting queen positions.

**🔷 PART 4: Simple Basics (to test your fundamentals)**

**17. ❓ What is recursion?**

**Answer:**  
Recursion is a process in which a function **calls itself** directly or indirectly. It is used to solve problems by breaking them into smaller subproblems.

**18. ❓ What is the difference between iterative and recursive approaches?**

**Answer:**

* **Iterative**: Uses loops to solve problems.
* **Recursive**: Solves problems using self-calls and base conditions.  
  Recursive methods are often more elegant and easier to write for tree and backtracking problems.

**19. ❓ Why is backtracking better than brute force here?**

**Answer:**  
Backtracking **eliminates** invalid configurations early and avoids trying all possibilities, making it much more efficient than brute force which blindly checks all combinations.

**20. ❓ What are constraints in CSP?**

**Answer:**  
In CSPs, constraints are **rules** that must be followed. For N-Queens:

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

**✅ Summary Table**

| **Category** | **Questions** |
| --- | --- |
| Core Problem | What is N-Queens, algorithm, CSP, constraints |
| Algorithm | Backtracking, Branch and Bound, recursion |
| Code Specific | solve(), isSafe1(), backtracking logic |
| Theory | Time complexity, data structures, NP problem |
| Basics | Recursion, iterative vs recursive, CSP modeling |

Would you like a **PDF version** of these viva questions and answers to revise offline?