**Interview Questions by mam Day Wise**

**Day1 :**

* **Algorithms**

**1. What is an algorithm?**

An **algorithm** is a well-defined, step-by-step procedure designed to perform a specific task or solve a problem. It consists of a finite set of instructions that take an input, process it, and produce an output.

**Example: Finding the Largest Number in an Array**

1. Take an array of numbers as input.
2. Assume the first element is the largest.
3. Iterate through the array and compare each element with the assumed largest.
4. If a larger element is found, update the assumed largest.
5. Return the largest number.

**Characteristics of an Algorithm:**

* **Well-defined inputs and outputs:** Every algorithm takes input and produces output.
* **Definiteness:** Every step is precisely defined.
* **Finiteness:** The algorithm must terminate after a finite number of steps.
* **Effectiveness:** Each step must be simple enough to be executed in a finite amount of time.

**2. Why is algorithm analysis important?**

Algorithm analysis helps in **evaluating** and **comparing** algorithms to determine the most efficient one. It allows us to understand how an algorithm will perform as input size grows.

**Key Reasons:**

* **Efficiency:** Determines how fast or slow an algorithm executes.
* **Optimization:** Helps in choosing the best approach for large-scale problems.
* **Scalability:** Ensures the algorithm can handle large inputs effectively.

**Example: Searching an Element in an Array**

* **Linear Search:** O(n) time complexity (searching element one by one).
* **Binary Search:** O(log n) time complexity (divides the array in half each step). For large inputs, binary search is significantly faster.

**3. What are the key criteria for analyzing an algorithm?**

The **two primary criteria** are:

1. **Time Complexity:** How execution time increases as input size grows.
   * **Best Case, Average Case, Worst Case** complexities.
   * Measured using **Big-O Notation** (e.g., O(1), O(n), O(log n), O(n²)).
2. **Space Complexity:** The amount of memory required.
   * Includes **input space**, **auxiliary space**, and **stack space** (for recursion).

Example:

* **Bubble Sort:** O(n²) (inefficient for large inputs).
* **Merge Sort:** O(n log n) (better efficiency).

**4. What are the different approaches to developing algorithms?**

1. **Brute Force:** Try all possible solutions (e.g., checking all pairs in a list).
2. **Divide and Conquer:** Break problems into smaller subproblems (e.g., Merge Sort).
3. **Greedy Algorithm:** Make the best choice at each step (e.g., Kruskal’s Algorithm).
4. **Dynamic Programming:** Solve overlapping subproblems efficiently (e.g., Fibonacci).
5. **Backtracking:** Explore all possibilities but discard non-promising ones (e.g., N-Queens).

**5. What are the characteristics of a good algorithm?**

1. **Correctness:** Produces correct output for every valid input.
2. **Efficiency:** Uses minimal time and space.
3. **Finiteness:** Must complete execution in a finite number of steps.
4. **Generality:** Works for different inputs, not just one case.
5. **Simplicity & Clarity:** Should be easy to understand and implement.

* **Data Structures**

**1. What are different types of data structures?**

* **Linear Data Structures:**
  + **Arrays:** Fixed-size contiguous memory storage.
  + **Linked Lists:** Dynamic memory with nodes connected via pointers.
  + **Stacks:** LIFO structure (Last In, First Out).
  + **Queues:** FIFO structure (First In, First Out).
* **Non-Linear Data Structures:**
  + **Trees:** Hierarchical structure with nodes (e.g., Binary Tree, BST).
  + **Graphs:** Nodes connected by edges (e.g., Social Networks).
* **Hashing:** Hash tables, hash maps (fast lookups).
* **Specialized Structures:** Tries, Heaps
* **2. What is the difference between an array and a linked list?**

| **Feature** | **Array** | **Linked List** |
| --- | --- | --- |
| **Memory Allocation** | Contiguous memory (fixed size) | Dynamic memory (flexible size) |
| **Insertion/Deletion** | Expensive (O(n) for shifting) | Efficient (O(1) at head) |
| **Access Time** | O(1) (direct access via index) | O(n) (traversal required) |
| **Extra Memory** | No extra space required | Requires additional space for pointers |

**3. How does a stack work? Provide a real-time example.**

A **stack** follows **LIFO (Last In, First Out)** principle.

**Operations:**

* **Push:** Insert an element.
* **Pop:** Remove the top element.
* **Peek:** Get the top element.

**Real-world Example:** **Undo feature in text editors**

* Each action (typing, deleting) is pushed onto a stack.
* When "Undo" is pressed, the last action is popped.

**4. What are the operations on a queue? Explain different types of queues.**

**Operations:**

* **Enqueue:** Insert element at the rear.
* **Dequeue:** Remove element from the front.
* **Peek:** Get the front element.

**Types of Queues:**

1. **Simple Queue:** Follows FIFO.
2. **Circular Queue:** Connects rear to front, avoiding wasted space.
3. **Priority Queue:** Dequeues elements based on priority.
4. **Deque (Double-Ended Queue):** Insertions & deletions allowed from both ends.

**5. What is a graph? Explain different types of graphs.**

A **graph** is a set of nodes (**vertices**) connected by **edges**.

**Types of Graphs:**

1. **Directed Graph:** Edges have directions.
2. **Undirected Graph:** Edges don’t have direction.
3. **Weighted Graph:** Each edge has a weight/cost.
4. **Unweighted Graph:** All edges have the same cost.
5. **Cyclic Graph:** Contains cycles.
6. **Acyclic Graph:** Does not contain cycles.

* **Recursion in Java**

**1. What is Recursion, and How Does it Work in Java?**

Recursion is a technique in programming where a method calls itself to solve a problem. It continues to call itself with modified parameters until a stopping condition, called the **base case**, is met. Each recursive call creates a new stack frame in memory, and once the base case is reached, the function returns values back through the call stack.

**2. Why is Recursion Used in Java Programming?**

Recursion is used because it:

* **Simplifies complex problems** that can be broken down into smaller subproblems.
* **Is useful in divide-and-conquer algorithms**, such as sorting and searching techniques.
* **Is essential for data structures** like trees and graphs, which are naturally recursive in structure.
* **Helps in backtracking problems**, such as generating permutations or solving mazes.

**3. Advantages and Disadvantages of Recursion in Java**

**Advantages:**

* **Improves code readability** by reducing repetitive logic.
* **Best suited for hierarchical and tree-based structures.**
* **Reduces code complexity** for problems that naturally fit recursive patterns.

**Disadvantages:**

* **Consumes more memory** due to multiple function calls stored in the call stack.
* **Can cause StackOverflowError** if recursion depth is too high.
* **Slower execution** compared to iteration due to the overhead of function calls.

**4. Difference Between Recursion and Iteration in Java**

| **Feature** | **Recursion** | **Iteration** |
| --- | --- | --- |
| **Definition** | A function calls itself repeatedly | Uses loops like for, while |
| **Memory Usage** | Uses stack memory for each function call | Uses constant memory (loop variables) |
| **Performance** | Slower due to function call overhead | Faster as no function calls are required |
| **Best Used For** | Problems that are naturally recursive (tree traversal, Fibonacci, backtracking) | Problems with known and fixed iterations (loop-based calculations) |
| **Risk** | StackOverflowError if no base case is defined | Infinite loop if condition is not correctly set |

**5. What Are Base Cases in Recursion, and Why Are They Important?**

A **base case** is a condition in a recursive function that stops further recursive calls. It prevents infinite recursion and ensures that the function eventually terminates.

Without a base case, the function would keep calling itself indefinitely, leading to a **StackOverflowError**. The base case should be chosen carefully to correctly solve the problem and return results back up the call stack.

**6. Problems That Can Be Solved Using Recursion in Java**

**1. Mathematical Computations**

* Factorial Calculation
* Fibonacci Series
* Greatest Common Divisor (GCD)
* Power of a Number

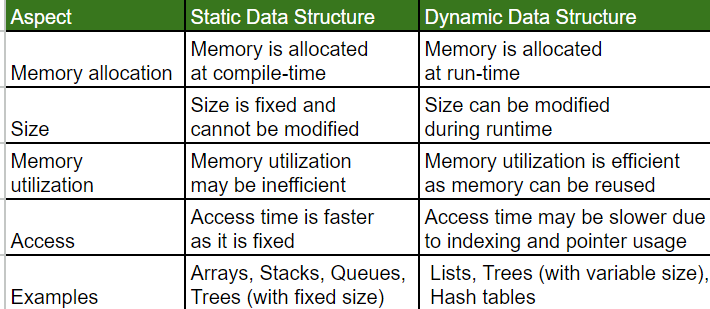
**2. Data Structure Operations**

* Tree Traversals (Preorder, Inorder, Postorder)
* Graph Traversal (Depth-First Search - DFS)
* Linked List Operations (Reverse, Merge)

**3. Algorithmic Problems**

* Merge Sort
* Quick Sort
* Binary Search
* Tower of Hanoi
* Backtracking problems (Sudoku Solver, N-Queens, Maze Solver)

**Difference between static Data Structure and Dynam**ic D**ata Structure**



**Day 4 :**

* **Recursion**

**1. What is the difference between direct and indirect recursion?**

Recursion is a programming technique where a function calls itself to solve a problem. However, recursion can be categorized into two types: **direct recursion** and **indirect recursion**.

**Direct Recursion:**

In **direct recursion**, a function calls itself directly within its own body. This is the most common type of recursion, where the function continues to call itself until a base condition is met, stopping further recursive calls.

For example, consider a simple function that prints numbers from n to 1:

java

CopyEdit

void directRecursion(int n) {

if (n > 0) { // Base condition to stop recursion

System.out.println(n);

directRecursion(n - 1); // Directly calling itself

}

}

When directRecursion(5) is called, it repeatedly calls itself with n-1 until n becomes 0, at which point recursion stops.

**Indirect Recursion:**

In **indirect recursion**, a function does not call itself directly but instead calls another function, which in turn calls the original function. This creates a loop of function calls between multiple functions.

For example:

java

CopyEdit

void functionA(int n) {

if (n > 0) {

System.out.println("A: " + n);

functionB(n - 1); // Calls functionB

}

}

void functionB(int n) {

if (n > 1) {

System.out.println("B: " + n);

functionA(n / 2); // Calls functionA

}

}

If functionA(5) is called, it prints 5, then calls functionB(4), which prints 4 and calls functionA(2), and so on.

The primary difference between direct and indirect recursion is that in direct recursion, a function calls itself, whereas in indirect recursion, multiple functions call each other in a cycle.

**2. What is tail recursion? How is it optimized?**

**Tail recursion** is a special type of recursion where the recursive call is the **last operation** performed in the function before returning a value. This means there is **no further computation** needed after the recursive call.

**Example of Tail Recursion:**

Consider a function that prints numbers from n to 1 using tail recursion:

java

CopyEdit

void tailRecursion(int n) {

if (n == 0) return; // Base case

System.out.println(n);

tailRecursion(n - 1); // Last operation before returning

}

Here, tailRecursion(n - 1) is the last operation before the function returns.

**Tail Recursion Optimization (TCO):**

Many programming languages and compilers optimize tail-recursive functions by converting them into **iterative loops** internally, which reduces **stack usage**. Instead of creating a new stack frame for each function call, the compiler **reuses the current function's stack frame**, making it more memory-efficient.

However, **Java does not support automatic Tail Call Optimization (TCO)**, unlike functional languages like Scala or Lisp. In Java, you can manually optimize tail-recursive functions using iteration.

For example, a **non-tail-recursive factorial function**:

java

CopyEdit

int factorial(int n) {

if (n == 0) return 1;

return n \* factorial(n - 1); // Multiplication happens after the recursive call

}

To convert this into **tail recursion**, we introduce an **accumulator** parameter:

java

CopyEdit

int factorialTail(int n, int accumulator) {

if (n == 0) return accumulator;

return factorialTail(n - 1, n \* accumulator); // Last operation is recursive call

}

Now the recursive call is the last operation, making it a **tail-recursive function**.

**3. How does recursion use the call stack? Explain with an example.**

Recursion relies on the **call stack** to keep track of function calls. Each time a function is called, a new **stack frame** is pushed onto the stack, storing variables, return addresses, and execution states. When a function returns, its stack frame is **popped** off the stack.

**Example: Factorial Calculation**

Let's take a recursive function to calculate factorial(5):

java

CopyEdit

int factorial(int n) {

if (n == 1) return 1; // Base case

return n \* factorial(n - 1); // Recursive call

}

When factorial(5) is executed, it follows this **stack behavior**:

factorial(5) → Calls factorial(4)

factorial(4) → Calls factorial(3)

factorial(3) → Calls factorial(2)

factorial(2) → Calls factorial(1)

factorial(1) → Returns 1 (Base case reached)

Now, the stack **unwinds**:

matlab

CopyEdit

factorial(2) = 2 \* 1 = 2

factorial(3) = 3 \* 2 = 6

factorial(4) = 4 \* 6 = 24

factorial(5) = 5 \* 24 = 120

Each function call **waits** for the next recursive call to return before it can proceed. This is why deep recursion can lead to **StackOverflowError** if the recursion depth is too large.

**4. How do you convert a recursive function to an iterative function?**

A recursive function can be converted to an **iterative function** using loops and, in some cases, a **stack data structure**. This eliminates **extra memory usage** caused by recursive calls.

**Example: Factorial Calculation**

**Recursive Approach:**

java

CopyEdit

int factorial(int n) {

if (n == 0) return 1;

return n \* factorial(n - 1);

}

**Iterative Approach:**

int factorialIterative(int n) {

int result = 1;

for (int i = 1; i <= n; i++) {

result \*= i; // Multiplying iteratively

}

return result;

}

This iterative version does not use additional stack space, making it more efficient in terms of memory.

For more complex recursive functions, such as **DFS (Depth First Search)**, an **explicit stack** can be used instead of recursion.

**5. Write a recursive function to find the greatest common divisor (GCD) of two numbers.**

The **Greatest Common Divisor (GCD)** of two numbers is the largest number that divides both numbers without leaving a remainder. The **Euclidean algorithm** efficiently calculates GCD using recursion.

**Recursive GCD Function:**

java

CopyEdit

int gcd(int a, int b) {

if (b == 0) return a; // Base case: When second number becomes 0

return gcd(b, a % b); // Recursive call with remainder

}

**Example Execution for gcd(48, 18):**

gcd(48, 18) → gcd(18, 48 % 18) → gcd(18, 12)

gcd(12, 6) → gcd(6, 12 % 6) → gcd(6, 0)

Since b = 0, return a = 6

So, gcd(48, 18) = 6.

* Iterative Version of GCD:

**int gcdIterative(int a, int b) {**

**while (b != 0) {**

**int temp = b;**

**b = a % b;**

**a = temp;**

**}**

**return a;**

**}**

Both versions work efficiently, but the **iterative version avoids stack overhead**.

* **Searching Techniques:**

**1. What are the different searching algorithms?**

Searching algorithms are fundamental in computer science and are used to locate an element within a data structure like an array, list, or database. Broadly, there are two types of searching algorithms: **Sequential Search** and **Interval Search**.

**Sequential Search (Brute Force Approach)**

The simplest way to search for an element is to scan through the entire dataset **one element at a time** until the desired element is found. This method works even when the data is unsorted.

For example, **Linear Search** follows this approach. You start at the first element and keep moving until you find the target or reach the end of the list.

**Interval Search (Divide and Conquer Approach)**

When the dataset is **sorted**, we can use more efficient searching techniques that repeatedly divide the search space to reduce the number of comparisons. **Binary Search** is a great example of this, where instead of checking each element one by one, we start from the middle and eliminate half of the search space in each step.

Other searching techniques, such as **Jump Search, Interpolation Search, and Exponential Search**, are also based on dividing the search space intelligently. The choice of algorithm depends on factors like whether the data is sorted, the size of the dataset, and the frequency of searching operations.

**2. Explain linear search and its time complexity.**

**Understanding Linear Search**

Linear search is the simplest searching technique where we traverse through each element in the list one by one, comparing it with the target value. If we find a match, we return the index; otherwise, we continue until we reach the end of the list.

**How Linear Search Works**

Imagine you are looking for a specific book on a **messy bookshelf** where books are not arranged in any order. The only way to find your book is to go through each book one by one until you find the right one. This is exactly how linear search works in programming.

For example, let's say we have an array {3, 8, 15, 22, 35} and we want to find 15. We check:

* First element 3 → No match.
* Second element 8 → No match.
* Third element 15 → Match found!

Thus, the element is found at index 2.

**Time Complexity of Linear Search**

The efficiency of linear search depends on where the target element is located in the list:

* If the element is **at the beginning**, we find it in **1 step** (O(1)).
* If the element is **at the end** or **not present**, we check all elements (O(n)).
* On **average**, we check **half the elements**, making it **O(n)** in most cases.

This means that for large datasets, linear search becomes inefficient, especially when there are better searching methods available.

**3. Explain binary search and its time complexity.**

**Understanding Binary Search**

Binary search is an **optimized search algorithm** that works **only on sorted data**. Instead of checking each element one by one, we start by checking the **middle element** of the dataset.

If the middle element is equal to the target, we have found it.  
If the target is **smaller**, we continue searching in the **left half**.  
If the target is **greater**, we continue searching in the **right half**.

This method reduces the search space by half in every step, making it **much faster** than linear search.

**How Binary Search Works**

Suppose we have a sorted array:

ini

CopyEdit

arr = {10, 20, 30, 40, 50, 60, 70}

Let's say we need to find 40.

1. Find the middle element → mid = (0+6)/2 = 3, arr[3] = 40.
2. Since 40 is exactly what we are searching for, we return the index 3.

If we were searching for 25, we would notice that 25 is **less than 40**, so we would discard the **right half** and continue searching in {10, 20, 30}.

**Time Complexity of Binary Search**

Since we **eliminate half the elements** at each step, the number of steps required is **logarithmic**.

* Best case: If the middle element is the target, we find it in **one step** (O(1)).
* Worst case: If the element is not found, we perform about **log₂(n)** steps (O(log n)).
* Average case: On average, binary search runs in **O(log n)** time.

This makes binary search **much more efficient** than linear search, especially for large datasets.

**4. How does Binary Search work on a Sorted Array?**

Binary search only works on **sorted arrays** because it relies on the ability to divide the dataset into two equal halves. The key idea is that if we know the data is sorted, we can **eliminate large portions** of the search space instead of checking every element.

**Step-by-Step Example**

Consider an array:

ini

CopyEdit

arr = {5, 10, 15, 20, 25, 30, 35}

If we want to find 25:

1. Find the middle element: (0+6)/2 = 3, arr[3] = 20.
2. Since 25 is **greater** than 20, we search the **right half** {25, 30, 35}.
3. New middle: (4+6)/2 = 5, arr[5] = 30.
4. Since 25 is **smaller** than 30, we search the **left half** {25}.
5. Middle element: arr[4] = 25, found at index 4.

This process is **much faster** than linear search because we **skip unnecessary comparisons**.

**5. What are the advantages and disadvantages of Binary Search?**

**Advantages of Binary Search**

1. **Speed:** Binary search is significantly faster than linear search, especially for large datasets. If we have **1,000,000 elements**, binary search will take only **20 steps** instead of checking every element.
2. **Efficient for Large Datasets:** Since it operates in **O(log n) time**, binary search is highly efficient for large **sorted** lists.
3. **Fewer Comparisons:** Because it halves the search space each time, it performs **fewer comparisons** compared to linear search.
4. **Can be Implemented Recursively or Iteratively:** Binary search can be implemented in both iterative and recursive ways, depending on the problem constraints.

**Disadvantages of Binary Search**

1. **Only Works on Sorted Data:** Binary search cannot be used on **unsorted arrays** without sorting them first, which takes **O(n log n)** time.
2. **Does Not Work Well on Dynamic Data:** If elements are frequently added or removed, keeping the array sorted requires extra overhead.
3. **Not Efficient for Small Data Sets:** If the dataset is **small**, linear search may actually be faster because binary search has extra calculations (finding the middle index).
4. **Not Suitable for Linked Lists:** Unlike arrays, linked lists do not support direct access to middle elements, making binary search inefficient on linked lists.

* **Backtracking:**

**1. What is Backtracking, and How Does It Work in Java?**

Backtracking is an **algorithmic technique** used to solve problems incrementally by **trying all possible solutions and undoing choices that lead to failure**. It is a form of **recursion** that systematically explores all possibilities while pruning incorrect or invalid solutions.

**How It Works**

1. **Make a choice** → Try an option.
2. **Check if it leads to a valid solution**.
3. **If not valid, backtrack** (undo the last choice and try another option).
4. **Continue until a solution is found or all possibilities are exhausted**.

**Example in Java**

Imagine solving a **maze** using backtracking:

java

CopyEdit

public boolean solveMaze(int[][] maze, int x, int y) {

if (x == maze.length - 1 && y == maze[0].length - 1) {

return true; // Reached the goal

}

if (isSafe(maze, x, y)) {

maze[x][y] = 1; // Mark as visited

if (solveMaze(maze, x + 1, y) || solveMaze(maze, x, y + 1)) {

return true; // Move right or down

}

maze[x][y] = 0; // Undo move (backtrack)

}

return false;

}

This example shows how **backtracking explores paths recursively** and backtracks if a path leads to failure.

**2. Key Components of a Backtracking Algorithm**

In Java, a **backtracking algorithm** consists of:

1. **Decision Space**: A set of possible moves (e.g., placing numbers in Sudoku).
2. **Constraints**: Rules that must be followed (e.g., no duplicate numbers in Sudoku).
3. **Base Case**: The stopping condition (e.g., board is completely filled).
4. **Recursive Exploration**: Making a choice, moving to the next step, and backtracking if needed.
5. **Backtracking Step**: Undoing a choice if it leads to failure.

**3. Real-World Applications of Backtracking in Java**

Backtracking is useful in various real-world scenarios:

| **Application** | **Description** |
| --- | --- |
| **Sudoku Solver** | Fills an incomplete 9×9 grid following the Sudoku rules. |
| **N-Queens Problem** | Places N queens on an NxN chessboard such that no two attack each other. |
| **Word Search** | Finds words in a matrix of letters by exploring all paths. |
| **Graph Coloring** | Assigns colors to graph vertices so that no two adjacent vertices have the same color. |
| **Puzzles (Knight’s Tour, Crossword, etc.)** | Finds a valid sequence of moves to solve a puzzle. |

**4. Java Code for Sudoku Solver Using Backtracking**

A Sudoku Solver fills a **9x9 board**, ensuring:

1. Each **row** has numbers 1-9 without repetition.
2. Each **column** has numbers 1-9 without repetition.
3. Each **3x3 sub-grid** has numbers 1-9 without repetition.

**Java Code**

java

CopyEdit

public class SudokuSolver {

public static boolean solveSudoku(int[][] board) {

for (int row = 0; row < 9; row++) {

for (int col = 0; col < 9; col++) {

if (board[row][col] == 0) { // Empty cell

for (int num = 1; num <= 9; num++) {

if (isValid(board, row, col, num)) {

board[row][col] = num; // Try placing number

if (solveSudoku(board)) return true; // Recursively solve

board[row][col] = 0; // Undo move (backtrack)

}

}

return false; // No valid number found, backtrack

}

}

}

return true; // Solution found

}

private static boolean isValid(int[][] board, int row, int col, int num) {

for (int i = 0; i < 9; i++) {

if (board[row][i] == num || board[i][col] == num ||

board[row - row % 3 + i / 3][col - col % 3 + i % 3] == num) {

return false; // Check row, column, and 3x3 subgrid

}

}

return true;

}

public static void printBoard(int[][] board) {

for (int[] row : board) {

for (int num : row) {

System.out.print(num + " ");

}

System.out.println();

}

}

public static void main(String[] args) {

int[][] board = {

{5, 3, 0, 0, 7, 0, 0, 0, 0},

{6, 0, 0, 1, 9, 5, 0, 0, 0},

{0, 9, 8, 0, 0, 0, 0, 6, 0},

{8, 0, 0, 0, 6, 0, 0, 0, 3},

{4, 0, 0, 8, 0, 3, 0, 0, 1},

{7, 0, 0, 0, 2, 0, 0, 0, 6},

{0, 6, 0, 0, 0, 0, 2, 8, 0},

{0, 0, 0, 4, 1, 9, 0, 0, 5},

{0, 0, 0, 0, 8, 0, 0, 7, 9}

};

if (solveSudoku(board)) {

printBoard(board);

} else {

System.out.println("No solution exists.");

}

}

}

**Explanation**

1. **Find an empty cell**.
2. **Try numbers 1-9** and check if they are valid.
3. **Recursively fill the next cell**.
4. **Backtrack if needed**.

**5. Java Code for the N-Queens Problem Using Backtracking**

The **N-Queens problem** places N queens on an NxN chessboard so that:

* No two queens attack each other (no same row, column, or diagonal conflicts).

**Java Code**

java

CopyEdit

public class NQueens {

static int N = 8; // Change this for different board sizes

public static boolean solveNQueens(int[][] board, int row) {

if (row == N) {

printBoard(board);

return true;

}

for (int col = 0; col < N; col++) {

if (isSafe(board, row, col)) {

board[row][col] = 1; // Place Queen

if (solveNQueens(board, row + 1)) return true; // Recursive call

board[row][col] = 0; // Undo move (backtrack)

}

}

return false;

}

private static boolean isSafe(int[][] board, int row, int col) {

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

if (board[i][col] == 1) return false; // Check column

if (col - (row - i) >= 0 && board[i][col - (row - i)] == 1) return false; // Left diagonal

if (col + (row - i) < N && board[i][col + (row - i)] == 1) return false; // Right diagonal

}

return true;

}

private static void printBoard(int[][] board) {

for (int[] row : board) {

for (int cell : row) {

System.out.print((cell == 1 ? "Q " : ". "));

}

System.out.println();

}

System.out.println();

}

public static void main(String[] args) {

int[][] board = new int[N][N];

if (!solveNQueens(board, 0)) {

System.out.println("No solution exists.");

}

}

}

**Explanation**

* **Try placing a queen in each row**.
* **Ensure no conflicts** in the column or diagonals.
* **Recursively place the next queen**.
* **Backtrack if needed**.