Answers to Questions for Review:

1) What is backtracking? What are the peculiar characteristics & applications of this approach?

Backtracking is a general algorithmic technique used for finding solutions to problems incrementally by trying partial solutions and eliminating those that fail to satisfy the problem's constraints. It is essentially a refined brute-force approach that builds solutions step-by-step, abandoning (backtracking) solutions as soon as it determines that they cannot lead to a valid solution.

Peculiar Characteristics of Backtracking:

- **Incremental**: The algorithm builds solutions one step at a time, making decisions based on previous steps.

- **Recursive**: Backtracking is often implemented using recursion to explore all possible configurations.

- **Constraint Checking**: At each step, the algorithm checks if the current solution is valid according to the constraints. If a constraint is violated, it backtracks to the previous step.

- **Pruning**: The main strength of backtracking is its ability to prune the search space by eliminating invalid configurations early.

- **Exhaustive Search**: Although backtracking eliminates bad paths early, it still explores all possible solutions.

Applications of Backtracking:

- **N-Queens Problem**: Placing N queens on an NxN chessboard such that no two queens threaten each other (the problem you're working on).

- **Sudoku Solver**: Filling a Sudoku puzzle while ensuring the constraints (numbers 1–9 in each row, column, and subgrid) are satisfied.

- **Crossword Puzzles**: Generating valid crossword puzzles by placing words in a grid while following the rules of word placement.

- **Graph Coloring**: Assigning colors to the vertices of a graph such that no two adjacent vertices have the same color.

- **Knapsack Problem**: Solving variations of the knapsack problem by trying combinations of items and backtracking when a solution cannot be achieved.

2) Explain Explicit & Implicit constraints with respect to 8 Queen's Problem?

Explicit Constraints are constraints that are clearly defined by the problem itself, and are part of the problem's definition. In the case of the 8-Queens problem, these constraints are:

- **Each queen must be placed on a separate row**: This is an explicit constraint because we know from the problem's description that there are exactly 8 rows, and each row must have exactly one queen.

- **Each queen must be placed on a separate column**: The queens cannot share columns, so this is another explicit constraint.

- **No two queens can be placed on the same diagonal**: Diagonal threats are an inherent part of the chessboard rules, so this constraint is also explicit.

Implicit Constraints are constraints that arise during the process of solving the problem, and are not part of the problem's initial description, but become clear as we explore potential solutions.

- **Backtracking Limitations**: When we place a queen in a column, we implicitly limit the placement of other queens in the same column, and diagonals are implicitly constrained.

- **Symmetry Breaking**: As the algorithm progresses, we can also implicitly prune certain configurations (like those that are symmetrically equivalent to already explored configurations), without explicitly specifying this in the problem's definition.

3) Compare the space & time complexity of Recursive & Non-Recursive techniques of Backtracking.

Recursive Backtracking:

- **Time Complexity**: The time complexity of recursive backtracking depends on the number of possible configurations. For the N-Queens problem, it's generally $O(N!)$, since for each queen, there are up to N positions to place it, and there are N queens to place. This gives an upper bound of $O(N!)$ for the search space.

- **Space Complexity**: The space complexity is $O(N)$ since we are using recursion and storing a path of length N at any given time (the current state of the solution).

Non-Recursive Backtracking (Iterative Backtracking):

- **Time Complexity**: The time complexity for non-recursive backtracking is also $O(N!)$, as it explores the same number of configurations as recursive backtracking. The algorithm may use an explicit stack to manage the exploration of states, but the overall number of configurations explored remains the same.

- **Space Complexity**: The space complexity can be higher than the recursive approach since it involves explicitly maintaining a stack or queue of states, typically $O(N)$ for storing the current state of exploration, with an added space for the stack, leading to a space complexity of $O(N)$, which is similar to recursive backtracking, but might involve more overhead due to stack management.

In summary, **recursive backtracking** tends to be more elegant and easier to implement, while **non-recursive backtracking** may require additional overhead for managing the state and stack, but can be more flexible in certain contexts (such as controlling the order of exploration or handling very deep recursion).

4) Write realistic applications of this experiment in brief (at least two applications).

1. **N-Queens Problem in Chessboard Arrangement**:

   - The N-Queens problem itself has applications in optimizing solutions for placing non-interfering pieces on a chessboard. Solving this problem can be used to develop algorithms for positioning items or resources in constrained spaces (like scheduling or seating arrangements), ensuring that certain conditions are met (e.g., no overlapping or interference between items).

2. **Sudoku Puzzle Solver**:

   - The 8-Queens problem is closely related to solving constraint satisfaction problems, like Sudoku. By using backtracking, you can generate solutions for puzzles that involve placing numbers on a grid such that no row, column, or subgrid contains duplicate numbers. Backtracking enables an efficient search for solutions by pruning invalid configurations early, making it a powerful tool for solving such puzzles.

Both of these applications demonstrate how backtracking can be used to systematically explore all possible configurations while adhering to specific constraints, making it a powerful tool in problems that require optimization or constraint satisfaction.

……code explanation……..

Let's go through the code step by step to understand how it solves the N-Queens problem, where queens are placed on an \(N \times N\) chessboard such that no two queens threaten each other.

Step-by-Step Explanation:

1. `print_sol(board)` Function

```python
def print_sol(board):
    for row in board:
        print(' '.join(map(str, row)))
```

This function prints the current configuration of the chessboard (the `board` array). It iterates over each row of the board and prints the elements of the row, joining them with a space. The `map(str, row)` ensures that the elements (which are integers) are converted to strings for printing.

2. `is_safe(row, col, rows, left_diagonals, right_diagonals)` Function

```python
def is_safe(row, col, rows, left_diagonals, right_diagonals):
    if rows[row] or left_diagonals[row + col] or right_diagonals[col - row + N - 1]:
        return False
    return True
```

This function checks whether it is safe to place a queen at position `(row, col)` on the chessboard. The safety check is performed based on the following conditions:

- `rows[row]`: A queen is already placed in this row.

- `left_diagonals[row + col]`: A queen is already placed on the left diagonal.

- `right_diagonals[col - row + N - 1]`: A queen is already placed on the right diagonal.

If any of these conditions are `True`, the function returns `False`, indicating that placing a queen at `(row, col)` would result in a conflict. Otherwise, it returns `True`.

The two diagonal checks are based on the formula for diagonals:

- **Left diagonal**: The sum of the row and column indices is constant for all elements on the left diagonal (`row + col`).

- **Right diagonal**: The difference of the row and column indices is constant for all elements on the right diagonal (`col - row`).

3. `solve(board, col, rows, left_diagonals, right_diagonals)` Function

```python
def solve(board, col, rows, left_diagonals, right_diagonals):
    if col >= N:
        return True
    for i in range(N):
        if is_safe(i, col, rows, left_diagonals, right_diagonals):
            rows[i] = True
            left_diagonals[i + col] = True
```

```
            right_diagonals[col - i + N - 1] = True
            board[i][col] = 1

            if solve(board, col + 1, rows, left_diagonals, right_diagonals):
                return True

            rows[i] = False
            left_diagonals[i + col] = False
            right_diagonals[col - i + N - 1] = False
            board[i][col] = 0

    return False
```

This function attempts to solve the N-Queens problem using backtracking.

- **Base Case**:
  ```python
  if col >= N:
      return True
  ```
  If `col` exceeds or equals `N`, it means all queens have been successfully placed in all columns, so it returns `True`, indicating that a solution is found.

- **Iterating Through Rows**:

The function loops through each row (`for i in range(N)`) to check if it is safe to place a queen in the current column (`col`) at row `i`.

- **Safety Check**:

```python
if is_safe(i, col, rows, left_diagonals, right_diagonals):
```

Before placing a queen in row `i`, it checks whether placing the queen in this position is safe using the `is_safe` function. If it is safe:

- Mark the current row as occupied (`rows[i] = True`).

- Mark the diagonals as occupied (`left_diagonals[i + col] = True` and `right_diagonals[col - i + N - 1] = True`).

- Place the queen on the board (`board[i][col] = 1`).

- **Recursion**:

The function then recursively calls itself to place queens in the next column (`solve(board, col + 1, rows, left_diagonals, right_diagonals)`).

If the recursive call returns `True`, it means a valid solution has been found, and the function immediately returns `True`.

- **Backtracking**:

If placing the queen in row `i` does not lead to a valid solution, the function backtracks:

- It removes the queen from the current position (`board[i][col] = 0`).

- It resets the tracking arrays for the row and diagonals (`rows[i] = False`, `left_diagonals[i + col] = False`, `right_diagonals[col - i + N - 1] = False`).

- **Return**: If no valid solution is found after trying all rows in the current column, the function returns `False` to backtrack and try different configurations.

4. `main()` Function

```python
if __name__ == "__main__":
    N = int(input("Enter the number of rows for the square board: "))

    # Take the position for the first queen as input
    first_row = int(input("Enter the row position of the first queen (0-based index): "))
    first_col = int(input("Enter the column position of the first queen (0-based index): "))

    board = [[0] * N for _ in range(N)]
    rows = [False] * N
    left_diagonals = [False] * (2 * N - 1)
    right_diagonals = [False] * (2 * N - 1)

    # Place the first queen and update the tracking arrays
    board[first_row][first_col] = 1
    rows[first_row] = True
    left_diagonals[first_row + first_col] = True
    right_diagonals[first_col - first_row + N - 1] = True
```

```
    # Start solving from the next column after the initial position of the first queen
    ans = solve(board, first_col + 1, rows, left_diagonals, right_diagonals)

    if ans:
        print_sol(board)
    else:
        print("Solution does not exist")
```

- **Input**: The program first takes inputs for `N` (size of the board), and the position of the first queen (`first_row` and `first_col`).

- **Initial Setup**:

  The `board` is initialized as an $N \times N$ grid with all values set to `0`. The arrays `rows`, `left_diagonals`, and `right_diagonals` are also initialized to track which rows and diagonals are blocked by placed queens.

- **Placing the First Queen**: The first queen is placed at the user-specified position (`first_row`, `first_col`), and the tracking arrays are updated accordingly.

- **Recursive Solving**: The `solve` function is called to attempt placing the remaining queens, starting from the next column (`first_col + 1`).

- **Solution Output**: If a solution is found, the `print_sol` function is called to display the board. If no solution is found, it prints "Solution does not exist."

Summary

- This program solves the N-Queens problem using backtracking. It places queens on an \(N \times N\) board such that no two queens threaten each other. The user can specify the initial position of the first queen, and the algorithm will find the placement for the remaining queens while ensuring that no two queens share the same row, column, or diagonal.