N Queens Problem Explanation

```
def solveNQueens(n: int, first_queen_col: int):
    col = set()
    posDiag = set()
    negDiag = set()

    res = []
    board = [["."] * n for _ in range(n)]

    def backtrack(r):
        if r == n:
            res.append(["".join(row) for row in board])
            return

        for c in range(n):
            if c in col or (r + c) in posDiag or (r - c) in negDiag:
                continue

            col.add(c)
            posDiag.add(r + c)
            negDiag.add(r - c)
            board[r][c] = "Q"

            backtrack(r + 1)

            col.remove(c)
            posDiag.remove(r + c)
            negDiag.remove(r - c)
            board[r][c] = "."

    col.add(first_queen_col)
    posDiag.add(0 + first_queen_col)
    negDiag.add(0 - first_queen_col)
    board[0][first_queen_col] = "Q"

    backtrack(1)  # Start with the second row
    return res

if __name__ == "__main__":
    n = 8
    first_queen_col = 1
    board = solveNQueens(n, first_queen_col)[0]
    for row in board:
        print(" ".join(row))
```

EXPLAIN THIS LINE BY LINE CODE BY CODE

Here's a breakdown of the `solveNQueens` function, line by line:

```python
def solveNQueens(n: int, first_queen_col: int):
```

- **Purpose**: Define a function called `solveNQueens` that solves the N-Queens problem for an $n \times n$ chessboard, with the first queen placed in a specified column (`first_queen_col`) of the first row.
- **Parameters**: `n` is the size of the board (number of rows/columns), and `first_queen_col` is the column index for the queen in the first row.

```python
    col = set()
    posDiag = set()
    negDiag = set()
```

- **Purpose**: Initialize three sets to track which columns and diagonals are occupied by queens, preventing other queens from being placed in attacking positions.
- `col`: Tracks columns where queens are already placed.
- `posDiag`: Tracks positive diagonals (calculated as $\text{row} + \text{col}$), where queens are already placed.
- `negDiag`: Tracks negative diagonals (calculated as $\text{row} - \text{col}$), where queens are already placed.

```python
    res = []
    board = [["."] * n for _ in range(n)]
```

- **Purpose**: Initialize `res`, a list to store solutions, and `board`, a 2D list representing the chessboard.
- `res`: Stores each solution, where each solution is a list of strings representing the board.
- `board`: The chessboard initialized with `"."` in each cell, representing empty spaces. `n` rows are created, each with `n` columns.

```python
    def backtrack(r):
```

- **Purpose**: Define an inner helper function `backtrack` that uses recursion to place queens row by row, starting from the row `r`.
- **Parameter**: `r` is the current row being processed.

```python
        if r == n:
            res.append(["".join(row) for row in board])
            return
```

- **Base Case**: If `r == n`, it means all queens are placed successfully on the board.
- `res.append(...)`: Convert each row of the board from a list of characters to a string and add it as a solution to `res`.

```python
        for c in range(n):
```

- **Purpose**: Loop through each column `c` in the current row `r` to find a valid position for the queen.

```python
            if c in col or (r + c) in posDiag or (r - c) in negDiag:
                continue
```

- **Skip Invalid Positions**: If column `c`, positive diagonal `(r + c)`, or negative diagonal `(r - c)` is already occupied by a queen, skip this column.

```python
            col.add(c)
            posDiag.add(r + c)
            negDiag.add(r - c)
            board[r][c] = "Q"
```

- **Place Queen**: Mark column `c`, positive diagonal `(r + c)`, and negative diagonal `(r - c)` as occupied.
- `board[r][c] = "Q"`: Place a queen at row `r`, column `c` on the board.

```python
            backtrack(r + 1)
```

- **Recursive Call**: Call `backtrack` for the next row, `r + 1`, to continue placing queens.

```python
            col.remove(c)
            posDiag.remove(r + c)
            negDiag.remove(r - c)
            board[r][c] = "."
```

- **Backtrack (Undo)**: Remove the queen from row `r`, column `c` and update the `col`, `posDiag`, and `negDiag` sets to allow placing queens in other positions.

```python
    col.add(first_queen_col)
    posDiag.add(0 + first_queen_col)
    negDiag.add(0 - first_queen_col)
    board[0][first_queen_col] = "Q"
```

- **Initial Placement**: Place the first queen in the specified `first_queen_col` in the first row.
- Add `first_queen_col` to `col`, `0 + first_queen_col` to `posDiag`, and `0 - first_queen_col` to `negDiag`.
- Set `board[0][first_queen_col] = "Q"` to represent the queen's position on the board.

```python
    backtrack(1)
```

- **Start Recursion**: Begin the recursive backtracking process from the second row (`r = 1`) since the first row already has a queen.

```python
    return res
```

- **Return Results**: Return the list of solutions (`res`), where each solution is a unique board configuration.

The main function call and printing the solution:

```python
if __name__ == "__main__":
    n = 8
    first_queen_col = 1
    board = solveNQueens(n, first_queen_col)[0]
    for row in board:
        print(" ".join(row))
```

- **Execution**: If the script is run directly, `solveNQueens` is called with an 8x8 board and the first queen in column 1.
- **Output**: It prints the first solution in `res`, formatting each row with a space between columns for readability.

EXPLAIN THE TIME AND SPACE COMPLEXITY

Time Complexity

The time complexity of the `solveNQueens` function is generally analyzed as $O(N!)$ due to the nature of backtracking. Here's a step-by-step breakdown:

1. **Placing Queens in Each Row**:
   - We place queens row by row, starting from the first row to the n-th row.
   - For each row, we try every column to see if placing a queen there is valid.

2. **Pruning with Sets**:
   - By using `col`, `posDiag`, and `negDiag` sets, we eliminate positions that would lead to attacks, reducing the number of recursive calls.
   - Even with pruning, in the worst case, we still explore many possible configurations.

3. **Worst-Case Complexity**:
   - With each row, there are approximately N choices (columns) for placing a queen.
   - For each placement in a row, the function goes deeper, trying to place the queen in each column of the next row. In the worst case, this yields approximately $N!$ combinations (for large N).
   - Hence, the **time complexity is** $O(N!)$.

   For example, for an 8×8 board (the classic problem), we explore up to $8!$ (40,320) configurations, but pruning reduces the practical number of configurations explored.

Space Complexity

The space complexity is determined by the storage required for the board and auxiliary data structures (`col`, `posDiag`, `negDiag`), as well as the recursive call stack.

1. **Board Representation**:
   - We create a 2D list `board` of size $N \times N$, which requires $O(N^2)$ space.

2. **Auxiliary Data Structures**:
   - We use three sets (`col`, `posDiag`, `negDiag`), each of which stores up to N values, contributing $O(N)$ space.

3. **Recursive Call Stack**:
   - The depth of recursion is N because we place one queen per row, so the recursion stack uses $O(N)$ space.

4. **Solution Storage (`res`)**:

    - If there are S solutions, each solution requires $O(N^2)$ space to store. Thus, the total space required for storing all solutions is $O(S \times N^2)$.

Overall Complexity

- **Time Complexity**: $O(N!)$
- **Space Complexity**: $O(N^2 + N + S \times N^2) \approx O(S \times N^2)$

```
import random
import timeit

def deterministic_partition(arr, low, high):
    pivot = arr[high]
    i = low - 1

    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def randomized_partition(arr, low, high):
    pivot_index = random.randint(low, high)
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    return deterministic_partition(arr, low, high)

def quick_sort(arr, low, high, pivot_selector):
    if low < high:
        pivot_index = pivot_selector(arr, low, high)
        quick_sort(arr, low, pivot_index - 1, pivot_selector)
        quick_sort(arr, pivot_index + 1, high, pivot_selector)

if __name__ == "__main__":
    arr_sizes = [100, 1000, 10000, 100000]
    for size in arr_sizes:
        arr = [random.randint(1, 1000) for _ in range(size)]
        arr.sort(reverse=True)

        deterministic_time = timeit.timeit("quick_sort(arr.copy(), 0, len(arr) - 1,
deterministic_partition)",
                                        globals=globals(),
                                        number=10)

        randomized_time = timeit.timeit("quick_sort(arr.copy(), 0, len(arr) - 1, randomized_partition)",
                                        globals=globals(),
                                        number=10)
        print(f"Array size: {size}")
        print(f"Deterministic Quick Sort time: {deterministic_time:.6f} seconds")
        print(f"Randomized Quick Sort time: {randomized_time:.6f} seconds")
        print("-" * 40)
```

EXPLAIN THIS LINE BY LINE CODE BY CODE

Here's a line-by-line explanation of the code provided:

python

```python
import random
import timeit
```

- **Imports**: Import the `random` module to select random pivot indices and `timeit` to measure the execution time of sorting algorithms.

Deterministic Partition Function

python

```python
def deterministic_partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
```

- **Function Purpose**: Defines `deterministic_partition`, which partitions the array around a chosen pivot element (the last element in the subarray) and returns the pivot's final position after partitioning.
- `pivot = arr[high]`: Selects the pivot element as the last element in the range `[low, high]`.
- `i = low - 1`: Initializes `i` to one position before the low index to help track elements smaller than the pivot.

python

```python
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
```

- **Loop Through Elements**: Loops from `low` to `high - 1` to compare each element with the pivot.
- **Comparison and Swap**:
    - If `arr[j] < pivot`, it increments `i` and swaps `arr[i]` with `arr[j]`, effectively moving smaller elements to the left side of the pivot.

python

```
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

- **Final Swap**: After the loop, it swaps the pivot (at `high`) with the element at `i + 1`, placing the pivot in its correct, sorted position.
- **Return Pivot Position**: Returns `i + 1`, which is the index of the pivot after partitioning.

Randomized Partition Function

python

```
def randomized_partition(arr, low, high):
    pivot_index = random.randint(low, high)
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    return deterministic_partition(arr, low, high)
```

- **Function Purpose**: Defines `randomized_partition`, which selects a random pivot index within `[low, high]`, swaps it with the last element, and calls `deterministic_partition` to partition around the randomized pivot.
- `pivot_index = random.randint(low, high)`: Selects a random pivot index.
- `arr[pivot_index], arr[high] = arr[high], arr[pivot_index]`: Swaps the pivot with the last element to prepare for partitioning.
- `return deterministic_partition(arr, low, high)`: Calls `deterministic_partition` on the array to partition around this randomized pivot.

Quick Sort Function

python

```
def quick_sort(arr, low, high, pivot_selector):
    if low < high:
        pivot_index = pivot_selector(arr, low, high)
        quick_sort(arr, low, pivot_index - 1, pivot_selector)
        quick_sort(arr, pivot_index + 1, high, pivot_selector)
```

- **Function Purpose**: Defines `quick_sort`, a recursive implementation of the Quick Sort algorithm.
- **Parameters**:
  - `arr`: The array to sort.
  - `low` and `high`: The current subarray boundaries.
  - `pivot_selector`: A function (either `deterministic_partition` or `randomized_partition`) to determine the pivot.
- **Base Condition**: If `low < high`, it proceeds with sorting; otherwise, it ends the recursion.
- **Recursive Partitioning**:
  - Calls `pivot_selector(arr, low, high)` to partition the array and obtain the pivot's sorted position (`pivot_index`).

  - Recursively calls `quick_sort` on the left subarray (`low` to `pivot_index - 1`) and right subarray (`pivot_index + 1` to `high`).

Main Execution Block

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

- **Purpose**: Checks if the script is being run directly. If true, the following code block runs, benchmarking Quick Sort with different array sizes.

```python
    arr_sizes = [100, 1000, 10000, 100000]
    for size in arr_sizes:
        arr = [random.randint(1, 1000) for _ in range(size)]
        arr.sort(reverse=True)
```

- `arr_sizes`: A list of array sizes used for testing.
- **Loop Through Array Sizes**: For each size in `arr_sizes`, it creates a random array of integers.
- `arr.sort(reverse=True)`: Sorts `arr` in descending order to test Quick Sort on a reverse-sorted array, often a worst-case scenario for Quick Sort without randomization.

```python
        deterministic_time = timeit.timeit("quick_sort(arr.copy(), 0, len(arr) - 1, deterministic_partition)",
                                        globals=globals(),
                                        number=10)

        randomized_time = timeit.timeit("quick_sort(arr.copy(), 0, len(arr) - 1, randomized_partition)",
                                        globals=globals(),
                                        number=10)
```

- **Measure Execution Time**: Uses `timeit.timeit` to measure the execution time of `quick_sort` with both `deterministic_partition` and `randomized_partition`.
  - `quick_sort(arr.copy(), 0, len(arr) - 1, deterministic_partition)`: Sorts a copy of `arr` using deterministic Quick Sort.
  - `quick_sort(arr.copy(), 0, len(arr) - 1, randomized_partition)`: Sorts a copy of `arr` using randomized Quick Sort.
  - `globals=globals()`: Allows `timeit` to access all global variables (like `quick_sort`).
  - `number=10`: Repeats the sorting 10 times for each algorithm, returning the total time.

```python
        print(f"Array size: {size}")
        print(f"Deterministic Quick Sort time: {deterministic_time:.6f} seconds")
        print(f"Randomized Quick Sort time: {randomized_time:.6f} seconds")
        print("-" * 40)
```

- **Print Results**: Displays the array size, execution time for deterministic Quick Sort, and execution time for randomized Quick Sort.
- ``` `print("-" * 40)` ```: Prints a separator line for clarity between results.

Summary

- The code benchmarks deterministic and randomized Quick Sort on arrays of increasing sizes and prints out the execution time for each

TIME AND SPACE COMPLEXITY

Time Complexity

1. **Deterministic Quick Sort**:
    - **Best and Average Case**: When the pivot divides the array into two nearly equal parts in each recursive call, the time complexity is $O(N \log N)$, where N is the number of elements in the array. This happens on average if the array is random or when the pivot tends to be near the median.
    - **Worst Case**: If the array is already sorted (like when it's in reverse order and the last element is always chosen as the pivot), the complexity degrades to $O(N^2)$ due to uneven partitioning. This is because, in each recursive call, only one element is partitioned on one side of the pivot, leading to N recursive calls, each requiring $O(N)$ operations.
2. **Randomized Quick Sort**:
    - **Best and Average Case**: With random pivot selection, the chances of consistently bad splits reduce, making the average-case complexity $O(N \log N)$. Randomization reduces the likelihood of the worst-case scenario by ensuring the pivot is selected unpredictably.
    - **Worst Case**: In extremely rare cases, random selection could still lead to $O(N^2)$, but this is highly unlikely in practice. Randomization effectively makes the algorithm behave like $O(N \log N)$ most of the time.
3. **Overall Complexity for Benchmarking**:
    - Since each sorting function is called `` `10` `` times in the benchmark code, the time complexity for each function call is effectively multiplied by `` `10` ``. However, this doesn't change the asymptotic complexity of the function, as $10 \cdot O(N \log N)$ remains $O(N \log N)$.

Space Complexity

1. **Recursive Call Stack**:

- Quick Sort is an in-place algorithm, meaning it doesn’t require additional space for sorting itself, aside from the recursive stack.
- The recursive depth of the call stack can go up to N in the worst case (when each partition has only one element on one side), which takes $O(N)$ space.
- On average, with balanced splits, the recursive depth is $O(\log N)$, giving an average space complexity of $O(\log N)$.

2. **Auxiliary Space for the Array Copies**:
    - In the benchmarking code, `` `arr.copy()` `` is used to ensure the original array remains unsorted for both sorting functions. This takes $O(N)$ space for each copy, but it’s independent of the sorting algorithm's core complexity.

Summary

- **Deterministic Quick Sort Time Complexity**: $O(N^2)$ worst case, $O(N \log N)$ average and best cases.
- **Randomized Quick Sort Time Complexity**: $O(N^2)$ rare worst case, $O(N \log N)$ average and best cases.
- **Space Complexity**: $O(N)$ worst-case call stack, $O(\log N)$ average call stack, plus $O(N)$ for array copies in benchmarking.

ChatGPT can make mistakes. Check important info.