Algorithms

Homework 3: 2023-12753 EunSu Yeo

1. Environment & Setting

Environment	Setting
OS	Window 11
Code Editor	Visual Studio Code (Linux VM)
Language	Python
Version	3.11.5

2. Program description

This assignment presents two methods to solve the N-Queens problem on a board containing holes. Queens cannot be placed on blocked cells, and blocked cells also obstruct attack paths.

I made the two functions as the below explanation.

solve_iterative_nqueens: Iterative backtracking using an explicit stack.

solve_recursive_nqueens: Classical recursive backtracking.

Each function is implemented with 6 steps.

Shared steps:

① **Hole Board Initialization**: Create a 2D boolean array hole_board and mark blocked cells as True. The initial empty cells are marked as False.

② **Column-to-Holes Mapping**: Build column_to_holes, a dictionary mapping each column to the list of blocked rows. This is used to generate the segments.

③ **Segment Construction**: Split each column's rows into contiguous segments without holes, storing tuples (col, row_start, row_end) in segments.

④ **Hole-Blocking Helpers**: Functions to check if any hole lies strictly between two cells on the same row, column, or diagonal.

⑥ **Backtracking**: Place one queen per segment, ensuring no two queens attack each other unless a blocking hole lies between them. This step works differently in 2 functions.

A more detailed explanation of step 6 is like below.

a) solve_iterative_nqueens

```
# Stack Frame: [segment_index, placed_queens_count, part, next_row_to_try]
    stack: List[List[int]] = [[0, 0, 0, 0]]
    while stack:
        frame = stack[-1]
        segment_index, placed_count, part, next_row = frame

        # All queens placed: count a solution
        if placed_count == n:
            solution_count += 1
            # Exit the current stack
            stack.pop()
            continue

        # Out of segments or not enough segments left: break
        # Do not go deeper, some kind of pruning added for performance
        if segment_index == total_segments or total_segments - segment_index < n - placed_count:
            stack.pop()
            continue

        # Part 1: skip current segment
        if part == 0:
            # change to Part 2
            frame[2] = 1
            stack.append([segment_index + 1, placed_count, 0, 0])
            continue

        # Part 2-1: initialize row scanning for placement
        if part == 1:
            col, row_start, row_end = segments[segment_index]
            # Set the next row to try search for queen placement to row_start
            frame[3] = row_start
            # change part not to go back to Part 1 or 2-1
            frame[2] = 2

        # Part 2-2: scan rows in the segment to place queen
        col, row_start, row_end = segments[segment_index]
        placed = False
        while frame[3] <= row_end:
            # Get the current row to try
            row = frame[3]
            # Increse the row to try for the next iteration
            frame[3] += 1
            # Check if the current position is blocked by a hole
            if hole_board[row][col]:
                continue
```

```python
            # Check if the current position is safe to place a queen
            safe = True
            for i in range(placed_count):
                qr, qc = queen_rows[i], queen_cols[i]
                # Check if the current position is blocked by a hole or attaked by another queen
                if qr == row and not is_hole_between_in_row(row, qc, col):
                    safe = False
                    break
                if qc == col and not is_hole_between_in_col(col, qr, row):
                    safe = False
                    break
                if abs(qr - row) == abs(qc - col) and not
                    is_hole_between_in_diag(qr, qc, row, col):
                    safe = False
                    break
            if not safe:
                continue

            # Place the queen at (r, col)
            queen_rows[placed_count] = row
            queen_cols[placed_count] = col
            # Update the stack frame to reflect the placement
            stack.append([segment_index + 1, placed_count + 1, 0, 0])
            placed = True
            break

        # If no queen was placed in the current segment, pop the stack
        if not placed:
            stack.pop()
```

The iteration with backtracking looks like the code above. This consists of 3 steps.

① **Solution Counting**: Increment solution count when placed_count == n

② **Pruning**: Immediately backtrack if the remaining segments cannot accommodate the remaining queens.

③ **Brach with placing Queen**: This step can be finished in 2 parts. Because we can put more than 1 queen in the row where there is a hole, there might be a row that has no queen. To find all possible situations we can think of 2 possible situations of each segment.

Part 1: Skip the current segment. (which means this segment is empty)

Part 2: Initialize row scanning within the current segment. Attempt queen placement row by row. (place the queen in this segment)

With this process we can find all possible cases.

b) solve_recursive_nqueens

```python
# Backtracking function
    def backtrack(seg_idx: int, placed: int):
        """
        Recursive backtracking function to place queens.

        Args:
            seg_idx (int): Segment index to consider for placement.
            placed (int): Number of queens already placed.
        """
        nonlocal solution_count
        # All queens placed: count a solution
        if placed == n:
            solution_count += 1
            return
        # Out of segments or not enough segments left: break
        # Do not go deeper, some kind of pruning added for performance
        if seg_idx == total_segments or total_segments - seg_idx < n - placed:
            return

        # Part 1: Skip segment
        backtrack(seg_idx + 1, placed)

        # Part 2: Try placing queen in segment
        col, row_start, row_end = segments[seg_idx]
        for row in range(row_start, row_end + 1):
            # Check if the current position is blocked by a hole
            if hole_board[row][col]:
                continue
            safe = True
            for i in range(placed):
                # Check if the current position is blocked by a hole or attacked by another queen
                qr, qc = queen_rows[i], queen_cols[i]
                if qr == row and not is_hole_between_in_row(row, qc, col):
                    safe = False
                    break
                if qc == col and not is_hole_between_in_col(col, qr, row):
                    safe = False
                    break
                if abs(qr - row) == abs(qc - col) and not
                    is_hole_between_in_diag(qr, qc, row, col):
                    safe = False
                    break
            if not safe:
                continue
            # Place the queen at (row, col)
            queen_rows[placed] = row
            queen_cols[placed] = col
            backtrack(seg_idx + 1, placed + 1)
```

This recursive function with backtracking looks like the code above.

This also consists of 3 steps.

① **Solution Counting**: Increment solution count when placed == n

② **Pruning**: Immediately stop if the remaining segments cannot accommodate the remaining queens.

③ **Brach with placing Queen**: This step can be finished in 2 parts. Because we can put more than 1 queen in the row where there is a hole, there might be a row that has no queen. To find all possible situations we can think of 2 possible situations of each segment.

Part 1: Skip the current segment. (which means this segment is empty) This is simply implemented by calling backtrack(seg_idx + 1, placed).

Part 2: Initialize row scanning within the current segment. Attempt queen placement row by row. (place the queen in this segment) if placed call the next process.

With this process we can find all possible cases.

As the explanation of the 2 functions, I optimized both functions for maximum speed in the following way.

Segment-based Placement: By splitting each column into hole-free segments, we avoid iterating over all cells in blocks and only attempt placements in valid intervals.

Early Pruning: Before descending into deeper recursion or stack frames, we check if the number of remaining segments is sufficient for the remaining queens, cutting off unproductive branches.

Unified Hole Test: We consolidated row, column, and diagonal hole checks into a single helper (is_hole_between), minimizing duplicate code and inlining overhead.

3. Result Analysis

To compare the running time of two functions, I ran the code with different n and different number of holes. The running time of each functions looks like the below.

① Iterative nqueens

The unit of the running time is milli second.

n hole	5	6	7	8	9	10
1	0.215381	0.611286	2.720083	13.818505	70.000980	402.379844
2	0.354235	1.142960	7.071381	34.769333	194.428268	1253.863809
3	0.249805	1.059964	6.943373	79.996963	479.538017	3554.985622

② Recursive nqueens

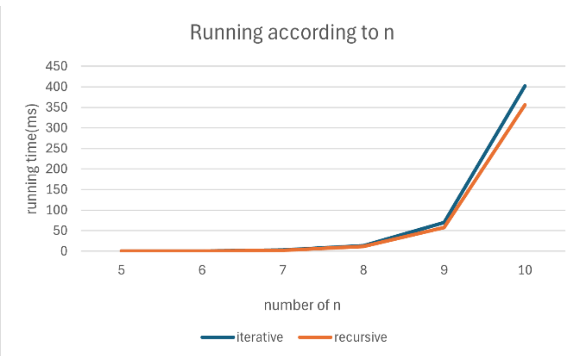
n hole	5	6	7	8	9	10
1	0.126734	0.484077	2.301283	11.965761	58.088262	355.790514
2	0.285872	0.991806	5.853652	29.031568	168.348853	1120.658214
3	0.141657	0.784948	5.602550	71.945306	407.898721	3173.842826

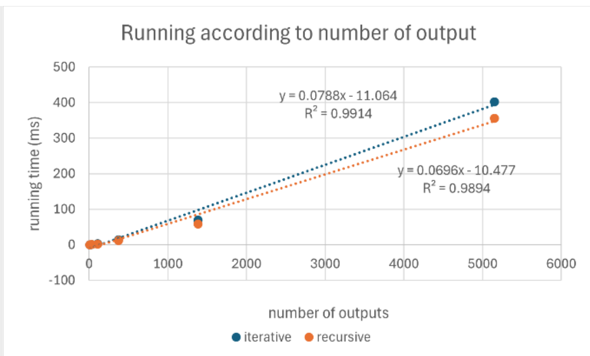
③ Number of outputs for each case I put.

n hole	5	6	7	8	9	10
1	10	32	110	371	1386	5151
2	9	50	294	1561	8115	37235
3	16	111	275	3774	25678	217844

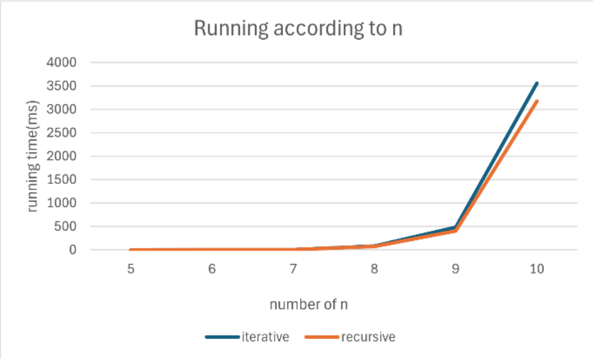
Now let's analysis the results.

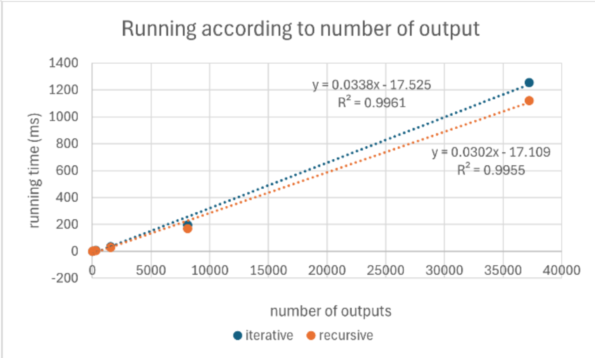
① Different n with 1 hole



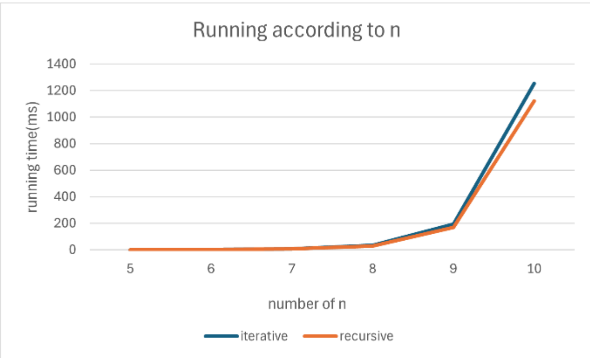


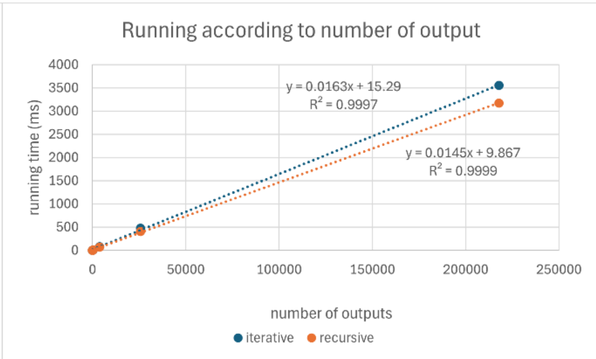
② Different n with 2 holes





③ Different n with 3 holes





With the results above, we can see that the running time of functions are linear to the total number of outputs. Also, the recursive version is slightly faster than the iterative version.

This difference may be from the structure of stack frame management. In the iterative version we track the state by using 'part' flag, also including segment_index, placed_count, next_row, adding extra condition checks. But the recursive version does not have those extra condition checks.

Also, the difference may be from the local-variable access cost. In the iterative version, the function frequently performs nested list indexing (stack[-1], then frame[...]). But in the recursive version, accesses go directly on the call stack, which is much faster than Python's sequence indexing.

Due to these reasons recursive version of backtracking nqueen problems may work faster than the iterative version.