1. Algorithm Completion (Personalized):

Student Number: 123456789 (Example)

Personalized Algorithm 1 (Fibonacci):

```
int fibonacci(int n) {
    if (n <= 1) return n; // Base case (digits 8, 9)
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
}
```

Worst-Case Operation Count:

- O(2^n) time complexity (exponential growth).
- O(n) space complexity (due to recursive calls).

Best-Case: 1 operation (when n is 0 or 1).Average-Case: Similar to worst-case, slightly better due to some early returns.

Personalized Algorithm 2 (Maze Solving):

```
bool solveMaze(int x, int y) {
    if (x == N - 1 && y == N - 1) { // Base case (reached goal) (digits 6, 7)
        solution[x][y] = 0;
        printSolution();
        return true; // Found solution
    }

    if (isSafe(x, y)) {
        solution[x][y] = 1; // Mark as part of the solution path

        if (solveMaze(x + 1, y) || solveMaze(x, y + 1))
            return true;

        solution[x][y] = 0; // Backtrack (unmark)
    }
    return false; // No solution in this path
}
```

Worst-Case Operation Count:

- Exponential in the worst case (O(4^n)), as it potentially explores every path in a maze with many branching points.

2. Recursive Function Analysis:

Factorial:

- Base Case: n = 0 (Factorial of 0 is 1).
- Recursive Case: n > 0. factorial(n) = n * factorial(n - 1)

Fibonacci:

- Base Cases: n = 0 or n = 1 (First two Fibonacci numbers are 0 and 1).
- Recursive Case: n > 1. fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)

Maze Solving:

- Base Case: Reaching the goal coordinates (N-1, N-1).
- Recursive Case: Explore neighboring cells (if safe) until the goal is reached or a dead end is encountered (then backtrack).

3. Backtracking Problem (N-Queens):

Refer to the code file attached

4. Complexity Analysis:

- Factorial:
  - Recursive: O(n) time and space.
  - Iterative: O(n) time, O(1) space.
- Fibonacci:
  - Recursive: O(2^n) time, O(n) space.
  - Iterative: O(n) time, O(1) space.
- N-Queens:
  - Recursive: O(N!) time (worst case), O(N) space.

5. Recursion vs. Iteration:

- Recursion:
  - Elegant for tree traversal and divide-and-conquer algorithms.
  - Can lead to stack overflow if not used carefully.
- Iteration:
  - Generally more efficient for simple loops and tasks with limited stack space.
  - Can be less intuitive for recursive problems.