Advanced Algorithms — Quiz 1

Problem 1: Fibonacci Sequence — Two Approaches

Background

The Fibonacci Sequence is a classic mathematical sequence defined as follows:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2}, \quad \text{for all } n > 1. \end{aligned}$$

The first few terms are: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Your Task

Given a non-negative integer n, compute the n-th term of the Fibonacci sequence (F_n). Provide **two different** approaches with **pseudocode** and analyze their performance.

Solution

1. **Recursive Approach (10pts)**

   This method directly translates the mathematical definition of the Fibonacci sequence, $F_n = F_{n-1} + F_{n-2}$, into code. The function recursively calls itself until it reaches the base cases of $n = 0$ or $n = 1$, which prevents infinite recursion.

   ```
   fib_recursive(n):

       if (n == 0) return 0
       if (n == 1) return 1
       return fib_recursive(n-1) + fib_recursive(n-2)
   ```

2. **Iterative Approach (10pts)**

   This method, known as a "bottom-up" approach, avoids recursion. It calculates the sequence starting from the beginning (F_0, F_1) and iteratively builds up to F_n. This is far more efficient as it avoids the re-computation of values.

   ```
   fib_iterative(n):

       if (n == 0) return 0
       if (n == 1) return 1

       prev = 0  // Represents F_{i-2}
       curr = 1  // Represents F_{i-1}

       for i = 2 to n:
   ```

```
        next = prev + curr // Calculate F_i
        prev = curr
        curr = next

    return curr
```

3. **Performance Analysis (10pts)**

   A comparison of the performance of both functions:

   - **Recursive Approach**: **Time Complexity**: $O(2^n)$ (Exponential). This is due to a massive number of **overlapping subproblems**. For example, to compute 'fib(5)', 'fib(3)' is computed twice and 'fib(2)' is computed three times. This redundant work leads to an exponential number of function calls. **Space Complexity**: $O(n)$, which comes from the maximum depth of the recursion call stack.
   - **Iterative Approach**: **Time Complexity**: $O(n)$ (Linear). The algorithm consists of a single loop that runs from 2 to n. **Space Complexity**: $O(1)$ (Constant). A fixed number of variables ('prev', 'curr', 'next') are used, regardless of the size of n.
   - **Conclusion**: Clearly, the **iterative approach is far more efficient**. It avoids redundant calculations and uses minimal space, making it the superior solution for this problem.

Problem 2: Recursion Trees — Solve the Recurrences

Your Task

Use recursion trees to solve each of the following recurrences and give tight asymptotic bounds.

$$A(n) = A(n/2) + A(n/3) + A(n/6) + n, \qquad B(n) = 2B(n/4) + n$$

Solution

1. **Recurrence $A(n) = A(n/2) + A(n/3) + A(n/6) + n$ (15pts)**

   Core Idea: Observe the total cost at each level of the recursion tree.

   The key insight for this recurrence is that the sum of the subproblem sizes is exactly equal to the original problem size:

   $$\frac{n}{2} + \frac{n}{3} + \frac{n}{6} = \frac{3n + 2n + n}{6} = n$$

   This means that the total non-recursive work *at every level* of the tree remains constant at $\Theta(n)$.

   Next, we determine the depth of the tree. The depth is determined by the longest path, which is the one that divides by the smallest factor (2). This path has a length of $\log_2 n$, so the tree's depth is $O(\log n)$.

The total cost is the cost per level multiplied by the number of levels: $\Theta(n) \times O(\log n)$.

Therefore, the final time complexity is:

$$\boxed{A(n) = \Theta(n \log n)}$$

2. **Recurrence** $B(n) = 2B(n/4) + n$ **(15pts)**

   Core Idea: Analyze the pattern of the per-level cost and sum it up.

   Let's analyze the cost at each level of the recursion tree:

   - Level 0 (root): n
   - Level 1: $2 \times (n/4) = n/2$
   - Level 2: $2^2 \times (n/4^2) = 4 \times (n/16) = n/4$
   - Level i: $2^i \times \frac{n}{4^i} = n \cdot \left(\frac{2}{4}\right)^i = n \cdot \left(\frac{1}{2}\right)^i$

   The total cost is the sum of all level costs, which forms a **geometric series**:

$$B(n) = n \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right)$$

   This is a decreasing geometric series whose sum converges to a constant (specifically, 2). The total work is therefore dominated by the cost at the root node. (This result also aligns with Case 3 of the Master Theorem).

   Therefore, the final time complexity is:

$$\boxed{B(n) = \Theta(n)}$$

Problem 3: N-Queens — Backtracking

Your Task

Design a backtracking algorithm to **count** the number of valid solutions for n-queens. Your algorithm should return only the **total number of solutions**.

Solution

1. **Pseudocode (20pts)**

   Core Idea: Use recursion to perform a depth-first search. We place queens row by row, trying all possible columns in each. If a position is valid, we recurse to the next row. If all rows are successfully filled, a solution is found. We use a 1D array 'Q[1..n]' to represent the board, where 'Q[r]' stores the column index of the queen in row 'r'.

```
procedure PlaceQueens(Q[1..n], r):

    // Base Case: If r > n, we have successfully placed
    // queens in all n rows.
    if r = n + 1 then
        count := count + 1
    else
        // Recursive Step: Try placing a queen in each column 'j'
        // of the current row 'r'.
        for j = 1 to n do
            legal := true
            // Check for conflicts with previously placed queens.
            for i = 1 to r-1 do
                // Check for column and diagonal conflicts.
                // The check abs(Q[i]-j) == abs(r-i) is a
                // concise way to check both diagonals at once.
                if (Q[i] = j) or (abs(Q[i] - j) = abs(r - i)) then
                    legal := false

            if legal then
                Q[r] := j                  // Place the queen.
                PlaceQueens(Q, r + 1) // Recurse.
                // Backtracking happens implicitly when this
                // function call returns.
            end if
        end for
    end if


function countNQueens(n):

    initialize empty array Q[1..n]
    // count should be a global variable or passed by reference.
    count := 0
    PlaceQueens(Q, 1) // Start from the first row.
    return count
```

2. **Complexity (20pts)**

   Time Complexity: $O(n \cdot n!)$ **(Loose Upper Bound)**

   - $O(n!)$ **Factor (Search Space)**: In the worst case, the algorithm explores a search tree. There are n choices for the first queen, at most $n - 1$ for the second, and so on. This leads to a search space size related to the number of permutations, $n!$.
   - $O(n)$ **Factor (Work per Node)**: At each node in the tree (i.e., for each placement attempt), a safety check is performed. This check loops through all previously placed queens (up to $n - 1$), costing $O(n)$ time.

- **Total**: The overall complexity is the product of these factors. It's a loose upper bound because backtracking "prunes" the search tree, making the actual performance much better.

Space Complexity: $O(n)$

- The space is determined by two components: the storage for the board state (the 'Q' array, which is $O(n)$), and the maximum depth of the recursion stack (which is also n). Therefore, the total space complexity is $O(n)$.