```python
"""
Implementation of the Fibonacci sequence using memoization.
This approach uses a dictionary to store previously computed Fibonacci numbers,
allowing for efficient recursive computation.
Time Complexity: O(n) - each number is computed once.
Space Complexity: O(n) - for the memo dictionary."""
def fibonacci_memo(n, memo={}):
    # Base cases
    if n <= 0:
        return 0
    if n == 1:
        return 1

    # Check if result is already in memo
    if n in memo:
        return memo[n]

    # Compute and store result in memo
    memo[n] = fibonacci_memo(n-1, memo) + fibonacci_memo(n-2, memo)
    return memo[n]

# Example usage
n = 10
print(f"Fibonacci({n}) using memoization: {fibonacci_memo(n)}")
```

```python
"""
Implementation of the Fibonacci sequence using tabulation dynamic programming.
This approach builds a table to store intermediate results, allowing for efficient
computation of Fibonacci numbers.
Tabulation uses iteration to build a table of solutions from the base cases up to
the desired result, avoiding recursion entirely.
Time Complexity: O(n) - single loop from 2 to n.
Space Complexity: O(n) - for the dp array. (Note: This can be optimized to O(1) by
only storing the last two numbers.)
"""

def fibonacci_tab(n):
    # Handle edge cases
    if n <= 0:
        return 0
    if n == 1:
        return 1

    # Initialize table
    dp = [0] * (n + 1)
    dp[1] = 1

    # Fill table iteratively
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]

# Example usage
n = 10
print(f"Fibonacci({n}) using tabulation: {fibonacci_tab(n)}")
```