Explanation of the Practical Steps with Code:

This practical involves calculating Fibonacci numbers both using a **recursive** approach and an **iterative** approach, and then analyzing their **time** and **space complexity**.

1. **Fibonacci Series Overview:**

The Fibonacci series is an integer sequence where each number is the sum of the two preceding ones, usually starting with 0 and 1. The first few Fibonacci numbers are:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

In mathematical terms, the Fibonacci sequence is defined as:

- F(0) = 0
- F(1) = 1
- F(n) = F(n-1) + F(n-2), for n > 1

2. **Recursion vs. Iteration:**

- **Recursive Approach**: The function `fibonacciRecursive(int n)` calls itself to calculate the Fibonacci number for smaller values of `n` until it reaches the base case (`n = 0` or `n = 1`).
- **Iterative Approach**: The function `fibonacciIterative(int n)` calculates the Fibonacci number by iterating from 2 up to `n`, storing the last two numbers in the sequence and adding them to get the next number.

3. **Code Walkthrough:**

Recursive Fibonacci Function (`fibonacciRecursive`):

```cpp
int fibonacciRecursive(int n) {
```

```
    if (n <= 1) return n;  // Base case: if n is 0 or 1, return n.
    return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);  // Recursive case: F(n) = F(n-1) + F(n-2)
}
```

- For any `n`, if `n` is 0 or 1, the function simply returns `n`.
- Otherwise, it calls itself twice (once for `n-1` and once for `n-2`) and adds the results to calculate `F(n)`.

Iterative Fibonacci Function (`fibonacciIterative`):

```cpp
int fibonacciIterative(int n) {
    if (n <= 1) return n;  // Base case: if n is 0 or 1, return n.

    int prev1 = 0, prev2 = 1, result;
    for (int i = 2; i <= n; i++) {
        result = prev1 + prev2;  // Calculate the next Fibonacci number.
        prev1 = prev2;          // Update prev1 to be the previous Fibonacci number.
        prev2 = result;         // Update prev2 to be the current Fibonacci number.
    }
    return result;  // Return the nth Fibonacci number.
}
```

- The iterative approach starts with `prev1 = 0` and `prev2 = 1`, then iteratively calculates the next Fibonacci number by summing the two previous values until the `n`th number is reached.

Main Function (`main`):

```cpp
int main() {
    int n;
    cout << "Enter the position of the Fibonacci number: ";
```

```
    cin >> n;

    // Time measurement for recursive approach
    auto start_recursive = high_resolution_clock::now();
    int result_recursive = fibonacciRecursive(n);
    auto end_recursive = high_resolution_clock::now();
    auto duration_recursive = duration_cast<nanoseconds>(end_recursive - start_recursive);

    cout << "Recursive Fibonacci of " << n << " is " << result_recursive << endl;
    cout << "Time taken by recursive approach: " << duration_recursive.count() << " nanoseconds" << endl;
    cout << "Space complexity of recursive approach: O(n) (due to recursion stack)" << endl;

    // Time measurement for iterative approach
    auto start_iterative = high_resolution_clock::now();
    int result_iterative = fibonacciIterative(n);
    auto end_iterative = high_resolution_clock::now();
    auto duration_iterative = duration_cast<nanoseconds>(end_iterative - start_iterative);

    cout << "Iterative Fibonacci of " << n << " is " << result_iterative << endl;
    cout << "Time taken by iterative approach: " << duration_iterative.count() << " nanoseconds" << endl;
    cout << "Space complexity of iterative approach: O(1) (constant space)" << endl;

    return 0;
}
```

- The user is prompted to enter a number `n` for which the Fibonacci number is to be calculated.
- Time is measured for both recursive and iterative approaches using the `chrono` library.
- Results and their respective time and space complexities are displayed.

4. **Time and Space Complexity Analysis:**

Recursive Approach:

- **Time Complexity**:

  The time complexity of the recursive Fibonacci function is **O(2^n)**. This is because each call to `fibonacciRecursive(n)` results in two more calls, leading to an exponential growth in the number of calls.

- **Space Complexity**:

  The space complexity is **O(n)**. This is because of the recursion stack. At most, there will be `n` function calls on the stack.

Iterative Approach:

- **Time Complexity**:

  The time complexity of the iterative approach is **O(n)**. This is because the loop runs `n-1` times to calculate the Fibonacci number.

- **Space Complexity**:

  The space complexity is **O(1)** because the iterative approach only requires a constant amount of space (for `prev1`, `prev2`, and `result`), regardless of the value of `n`.

5. **Conclusion:**

- The Fibonacci series can be computed both recursively and iteratively.

- The recursive approach is less efficient due to its exponential time complexity and higher space complexity due to the recursion stack.

- The iterative approach is more efficient with linear time complexity and constant space complexity.

6. **Review Questions and Answers:**

1. **What is the Fibonacci series?**

The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, starting from 0 and 1.

2. **What is recursion?**

Recursion is a programming technique where a function calls itself in order to solve smaller instances of the same problem.

3. **Analyze time and space complexity of the Fibonacci series?**

- Recursive Approach:
  - Time complexity: **O(2^n)** (exponential)
  - Space complexity: **O(n)** (due to the recursion stack)
- Iterative Approach:
  - Time complexity: **O(n)** (linear)
  - Space complexity: **O(1)** (constant)