Assignment 1.1 Using C++ for Recursion

Follow the instructions:

1. Create C++ code to implement a recursive and non- recursive solution for the following tasks:

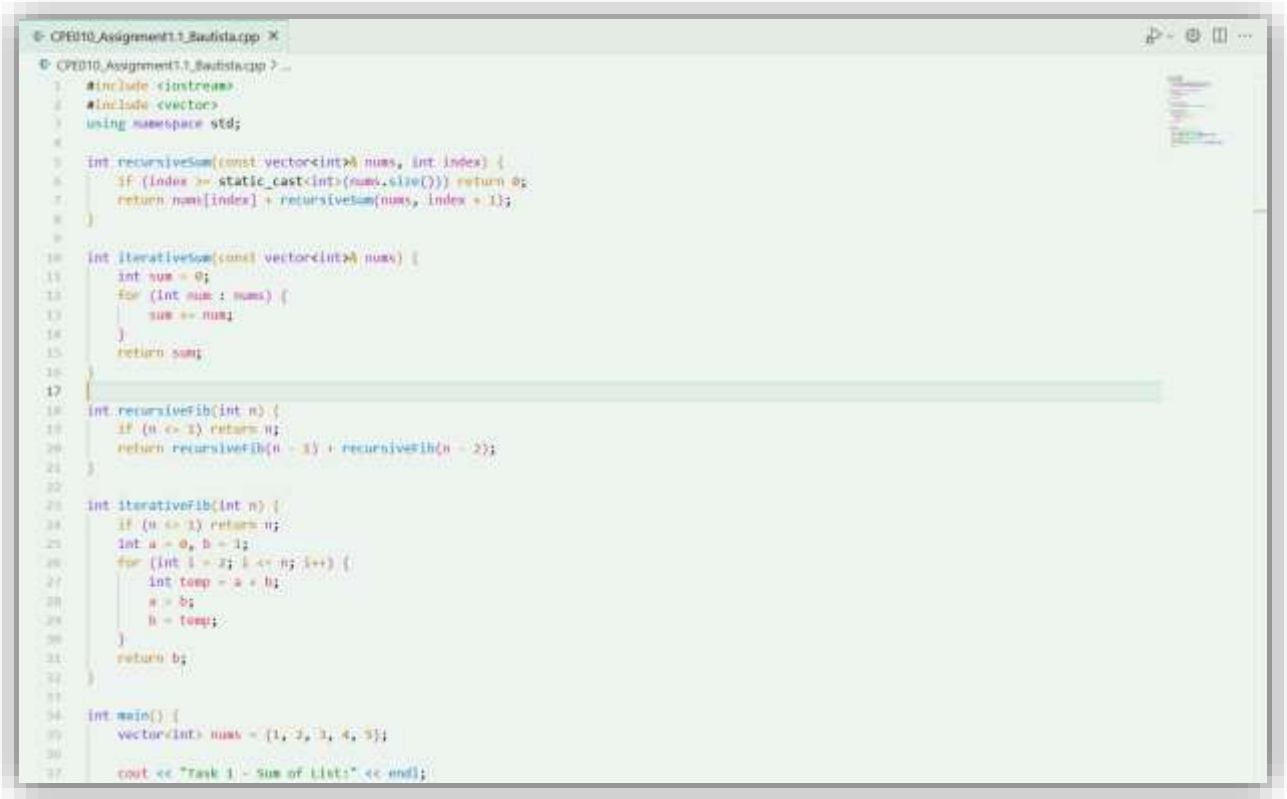
- Task 1: Summing a List of Numbers
- Task 2: Fibonacci

2. Analyze the above algorithms using Big-O Notation.

Your submission must contain the following:

- CPP files
- PDF file explaining your solutions to task 1 and task 2.

Code:



```
    cout << "Recursive Sum: " << recursiveSum(nums, 0) << endl;
    cout << "Iterative Sum: " << iterativeSum(nums) << endl;

    cout << "\nTask 2 - Fibonacci:" << endl;
    int n = 10;
    cout << "Recursive Fib(" << n << "): " << recursiveFib(n) << endl;
    cout << "Iterative Fib(" << n << "): " << iterativeFib(n) << endl;

    return 0;
}
```

Output:

```
C:\Users\[redacted]\Codes and Programs>main.exe
Task 1 - Sum of List:
Recursive Sum: 15
Iterative Sum: 15

Task 2 - Fibonacci:
Recursive Fib(10): 55
Iterative Fib(10): 55
```

Explanation:

Task 1 – Summing a List of Numbers

The recursive approach works by starting at a given index, adding the current element to the sum of the remaining elements, and continuing until the end of the list is reached. Once the index equals the size of the list, the recursion stops and returns 0. This approach breaks the problem into smaller subproblems until the base case is met.

The iterative approach uses a loop to accumulate the total sum of all elements in the list. It begins with a sum of 0, then sequentially adds each element to this total until all items have been processed. This method avoids the overhead of recursive calls and is generally more efficient in practice.

Task 2 – Fibonacci Sequence

The recursive Fibonacci function directly implements the mathematical definition of the sequence: Fib(n) = Fib(n – 1) + Fib(n – 2), with base cases when n equals 0 or 1. While conceptually simple, it is computationally expensive because it recalculates many values multiple times.

iterative Fibonacci method starts with the first two numbers in the sequence and uses a loop to build up to the desired term. Each new term is calculated as the sum of the previous two, and the values are updated iteratively until reaching the nth term. This eliminates redundant calculations and greatly improves performance.