Assignment 1.1	
Assignment 1.1 Using C++ for Recursion	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 08 - 07 - 25
Section: CPE21S4	**Date Submitted:** 08 - 07 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado

6. Output

Task 1: Summing a List of Numbers

```
#include <iostream>

// Sum of elements in list using Recursion
int sumRecur(int list[], int n);
// Sum of elements in list using Iteration
int sumIterate(int list[], int n) ;

int main() {
   int n;
   std::cout << "Enter number of elements: ";
   std::cin >> n;
   int list[n];
   for (int i = 0; i < n; i++) {
       std::cout<<"Enter element "<<i+1<<": ";
       std::cin >> list[i];
   }
   std::cout << "Sum of Numbers (Iteration): "<<sumIterate(list, n)<<'\n';;
   std::cout << "Sum of Numbers (Recur): "<<sumRecur(list, n)<<'\n';

   return 0;

}
int sumRecur(int list[], int n) {
   if (n == 0) {
       return 0;
   }
   return list[n - 1] + sumRecur(list, n - 1);
}
int sumIterate(int list[], int n) {
   int sum = 0;
   for (int i = 0; i < n; i++)
       sum += list[i];
   return sum;
}
```

Output:

```
"D:\College\Year 2 CPE\Sem 1\untitled\cmake-build-debug\untitled.exe"
Enter number of elements:3
 Enter element 1:5
 Enter element 2:10
 Enter element 3:15
 Sum of Numbers (Iteration): 30
Sum of Numbers (Recur): 30

Process finished with exit code 0
```

Task 2: Fibonacci

```cpp
#include <iostream>

// Fibonacci Recursion
int fibonacciRecur(int x);
// Fibonacci Iteration
int fibonacciIterate(int x);
// Printing
void printFibonacciRecur(int terms);
void printFibonacciIterate(int terms);

int main() {
   int terms;
   std::cout << "Enter terms for Fibonacci: ";
   std::cin >> terms;

   std::cout << "Fibonacci Series using recursion and if-else: ";
   printFibonacciRecur(terms);

   std::cout << "Fibonacci Series iteration: ";
   printFibonacciIterate(terms);

}
// Print Recursion
void printFibonacciRecur(int terms) {
   for (int i = 0; i < terms; i++) {
       std::cout << fibonacciRecur(i) << " ";
   } std::cout << std::endl;
}
// Print Iteration
void printFibonacciIterate(int terms) {
   for (int i = 0; i < terms; i++) {
       std::cout << fibonacciIterate(i) << " ";
   } std::cout << std::endl;
}
int fibonacciRecur(int x) {
   if (x <= 1) {
       return x;
   } else {
       return fibonacciRecur(x - 1) + fibonacciRecur(x - 2);
   }
}
int fibonacciIterate(int n) {
   // For 0th and 1st term
   if (n <= 1)
       return n;
   // Variable to store the last two terms
   int prev1 = 1, prev2 = 0;
```

```
    // Variable that stores the current fibonacci term
    int curr;
    // Calculating the next fibonacci  number by using the previous two number
    for (int i = 2; i <= n; i++) {
        curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return curr;
}
```

Output:

```
"D:\College\Year 2 CPE\Sem 1\untitled\cmake-build-debug\untitled.exe"
Enter terms for Fibonacci:10
 Fibonacci Series using recursion and if-else: 0 1 1 2 3 5 8 13 21 34
Fibonacci Series iteration: 0 1 1 2 3 5 8 13 21 34

Process finished with exit code 0
```

7. Explanations

I. Task 1: Summing a List of Numbers

The program asks the user for the number of items in the list (n). Once n is entered, it will create an array with size (n) then ask the user for the elements in the array using a for loop. After the list is created, two functions will be called:

- int sumRecur(int list[], int n);
  This function will continuously call itself and execute until the number of elements in the list has been passed through. When called in the main function, it will return the sum of all elements in the list.
  Its time complexity is O(n) because there is only one recursive call.

- int sumIterate(int list[], int n)
  This function will first initialize the variable sum that begins at 0. It will then run a for-loop that iterates through each element and then updates the sum value by adding the current element to it. Once finished, the final value sum will be the sum of all elements in the list.
  The time complexity is O(n) because there are no nested loops within the function.

II. Task 2: Fibonacci

The program asks the user for the number of terms in the Fibonacci sequence to be displayed. Once entered, the program will call two void functions that utilize the two functions that calculate and provide the fibonacci numbers in a series (For example: 10 terms = first 10 terms in the fibonacci sequence):

- fibonacciRecur(int n)
- This function computes the x-th Fibonacci number using recursion. It defines two base cases: if x is 0 or 1, it returns x directly. Otherwise, it makes two recursive calls: one to compute the (x - 1)-th number and another for the (x - 2)-th, then adds them together. This uses the mathematical definition of the Fibonacci sequence but is inefficient for large inputs because it recomputes values many times.
  Thus, it has exponential time complexity $O(2^n)$ and can cause performance issues or stack overflows for large x.

- fibonacciIterate(int n)
- The funcition calculates the n-th Fibonacci number using a loop. It handles the base cases (when n is 0 or 1) by returning n directly. For all other values, it uses a for loop starting from index 2 up to n, building the sequence by repeatedly adding the two previous values (prev1 and prev2). These variables are updated in each iteration: curr stores the current Fibonacci number, prev2 becomes the old prev1, and prev1 becomes curr. It is printed through iteration using the corresponding void function.
  The time complexity for this function is O(n) as it only uses one for-loop for the iteration.

8. Conclusion

In conclusion, recursion can be used to simplify complex functions. Instead of using loops or nested loops, we can reduce code complexity by making the function call itself. However, there are also downsides in using recursion. For instance, each recursion uses high memory and may cause stack overflow if not optimized. In the code I wrote, the recursion for fibonacci has a time complexity of $O(2^n)$, which is exponential and can cause stack overflow and performance issues for larger values. Hence, for general code readability and intuitiveness, recursion is useful. However, iteration is generally safer and easier to visualize, thus, easier to fix if errors occur.

9. Assessment Rubric