NAME: ODEYEMI TEMITOPE BENJAMIN

COURSE: ADVANCE ALGORITHM

DEPARTMENT: COMPUTER SCIENCE

REG. NO: 24000058

MATRIC NO: 24PGCD00499

PROGRAM: MPHIL/PhD

C IMPLEMENTATION OF FIBONACCI USING DYNAMIC PROGRAMMING

```
#include <stdio.h>
Void fibonacci (int n) {
// Array to store Fibonacci numbers
Int fib[n + 2];
//First base case
Fib[0] = 0;
//Second base case
Fib[1] = 1;
For (int i = 2; i <=n; i++) {
Fib[i] = fib[i – 1] + fib[i – 2];
}
Printf(“Fibonacci series: “);
For (int i = 0; i < n; i++) {
Printf(“%d,  “, fib[i];
}
}
Int main() {
Int n;
Printf(“Enter the number of Fibonacci numbers to generate: “);
Scanf(“%d”, &n);
Fibonacci(n);
Return 0;
}
```

Explanation:

In the code above, we create an array **Fib** to store Fibonacci values. We iterate from 2 to n and calculate the Fibonacci value at each index by summing the previous two values. By using dynamic programming, we avoid redundant calculations and improve the efficiency of the algorithm.

The **base case** for fibonacci is **0** and **1.**

Time and space complexity for the worse case: Linear **i.e.** O (n), **Time and space complexity for the best case:** Constant **i.e.** O (1) and **Time and space complexity for the average case:** Linear **i.e.** O (n)

C IMPLEMENTATION OF THE FIBONACCI SEQUENCE USING DYNAMIC PROGRAMMING WITH MEMOIZATION

```
#include <stdio.h>
#include <stdlib.h>
// Function to compute the n-th Fibonacci number using memoization
int fibonacci(int n, int *memo) {
   // Base cases
   if (n <= 1) {
      return n;
   }
   // Check if the value is already computed
   if (memo[n] != -1) {
      return memo[n];
   }
   // Compute and store the value in the memo array
   memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
   return memo[n];
}
int main() {
   int n;
   printf("Enter the value of n: ");
   scanf("%d", &n); // Get input from the user
   // Allocate memory for the memoization array
   int *memo = (int *)malloc((n + 1) * sizeof(int));
   if (memo == NULL) {
      printf("Memory allocation failed!\n");
      return 1;
   }
   // Initialize the memo array with -1 (indicating uncomputed values)
   for (int i = 0; i <= n; i++) {
      memo[i] = -1;
   }
   // Compute and print the n-th Fibonacci number
   printf("Fibonacci number at position %d is %d\n", n, fibonacci(n, memo));
   // Free the allocated memory
   free(memo);
   return 0;
}
```

Explanation:

- The “fibonacci” function takes an integer n and an optional dictionary “memo” to store previously computed Fibonacci numbers.
- The base cases handle the first two Fibonacci numbers.
- If the Fibonacci number for “n” is not already in “memo”, it is computed recursively and stored in “memo”.
- Finally, the function returns the Fibonacci number for “n”.

This approach ensures that each Fibonacci number is computed only once, significantly improving efficiency in term of time complexity compared to a naive recursive solution. **Time and space complexity for the worse case:** Linear **i.e.** O (n), **Time and space complexity for the best case:** Constant **i.e.** O (1), and **Time and space complexity for the average case:** Linear **i.e.** O (n).