**ET3272: Design and Analysis of Algorithms**

**Name of the student: Aditya Jannawar Roll No.: 02**

**Div.: A Batch: 1**

**Date of performance: 13/02/2023**

**Experiment No. 9**

# Title: Program for Fibonacci numbers

**Theory/Description of the Problem Statement:**

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding numbers. The sequence starts with 0 and 1, so the first few numbers in the sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on. The sequence is named after Leonardo Fibonacci,

The program I provided uses a recursive function to generate the n-th Fibonacci number. A recursive function is a function that calls itself within its own code.

**Algorithm :**

* Define an array dp of size 10 to store previously computed Fibonacci numbers.
* Define a function fib that takes an integer n and returns the n-th Fibonacci number.
* If n is less than or equal to 1, return n (since the 0th Fibonacci number is 0 and the 1st Fibonacci number is 1).
* Check if dp[n-1] is already computed (i.e. not equal to -1). If so, store its value in the variable first. Otherwise, recursively call fib with argument n-1 and store its value in first.
* Check if dp[n-2] is already computed (i.e. not equal to -1). If so, store its value in the variable second. Otherwise, recursively call fib with argument n-2 and store its value in second.
* Add first and second together and return the result.
* Store the result of the computation in dp[n] to memoize the value for later use.
* In the main function, initialize dp to -1 using memset (a function that sets a block of memory to a specified value).
* Call fib with argument n (in this case, n = 9) and output the result.
* The program then waits for user input before exiting.

**Pseudo Code :**

* // Define array to store Fibonacci numbers
* dp = array of size 10
* for i = 0 to 9
* dp[i] = -1
* // Define function to compute Fibonacci numbers
* function fib(n)
* // Base cases
* if n <= 1
* return n
* // Check if value is already memoized
* if dp[n-1] != -1
* first = dp[n-1]
* else
* // Recursively compute Fibonacci of n-1
* first = fib(n-1)
* // Memoize the result
* dp[n-1] = first
* // Check if value is already memoized
* if dp[n-2] != -1
* second = dp[n-2]
* else
* // Recursively compute Fibonacci of n-2
* second = fib(n-2)
* // Memoize the result
* dp[n-2] = second
* // Compute the sum of the two previous Fibonacci numbers
* sum = first + second
* // Memoize the result
* dp[n] = sum
* // Return the sum
* return sum
* // Main function
* function main()
* // Call the Fibonacci function with argument 9
* result = fib(9)
* // Output the result
* output result

**Analysis of the Algorithm**

The code implements a dynamic programming approach to compute the n-th Fibonacci number. It uses an array dp of size 10 to store previously computed Fibonacci numbers. When computing the n-th Fibonacci number, the code first checks if the values of fib(n-1) and fib(n-2) are already stored in dp. If so, it retrieves these values from dp instead of recomputing them. If not, it recursively computes fib(n-1) and fib(n-2) and stores the results in dp for later use.

**Time Complexity:**

Initialization of dp array takes constant time, O(1).

Computing fib(n) requires two recursive calls to fib(n-1) and fib(n-2) (if not already memoized) and addition of two integers. Therefore, the time complexity of computing fib(n) is O(2^n), where n is the input argument.

Since dp array stores previously computed Fibonacci numbers, the time complexity of computing fib(n) is reduced to O(n).

Therefore, the overall time complexity of the algorithm is O(n), which is significantly faster than the recursive approach that has a time complexity of O(2^n).

**Space Complexity:**

The space complexity of the algorithm is O(n), due to the need to store dp array of size n.

**Experiment and result:**

Code:

#include <bits/stdc++.h>

using namespace std;

int dp[10];

int fib(int n)

{

    if (n <= 1)

        return n;

    // temporary variables to store

    // values of fib(n-1) & fib(n-2)

    int first, second;

    if (dp[n - 1] != -1)

        first = dp[n - 1];

    else

        first = fib(n - 1);

    if (dp[n - 2] != -1)

        second = dp[n - 2];

    else

        second = fib(n - 2);

    // memoization

    return dp[n] = first + second;

}

// Driver Code

int main()

{

    int n = 9;

    memset(dp, -1, sizeof(dp));

    cout << fib(n);

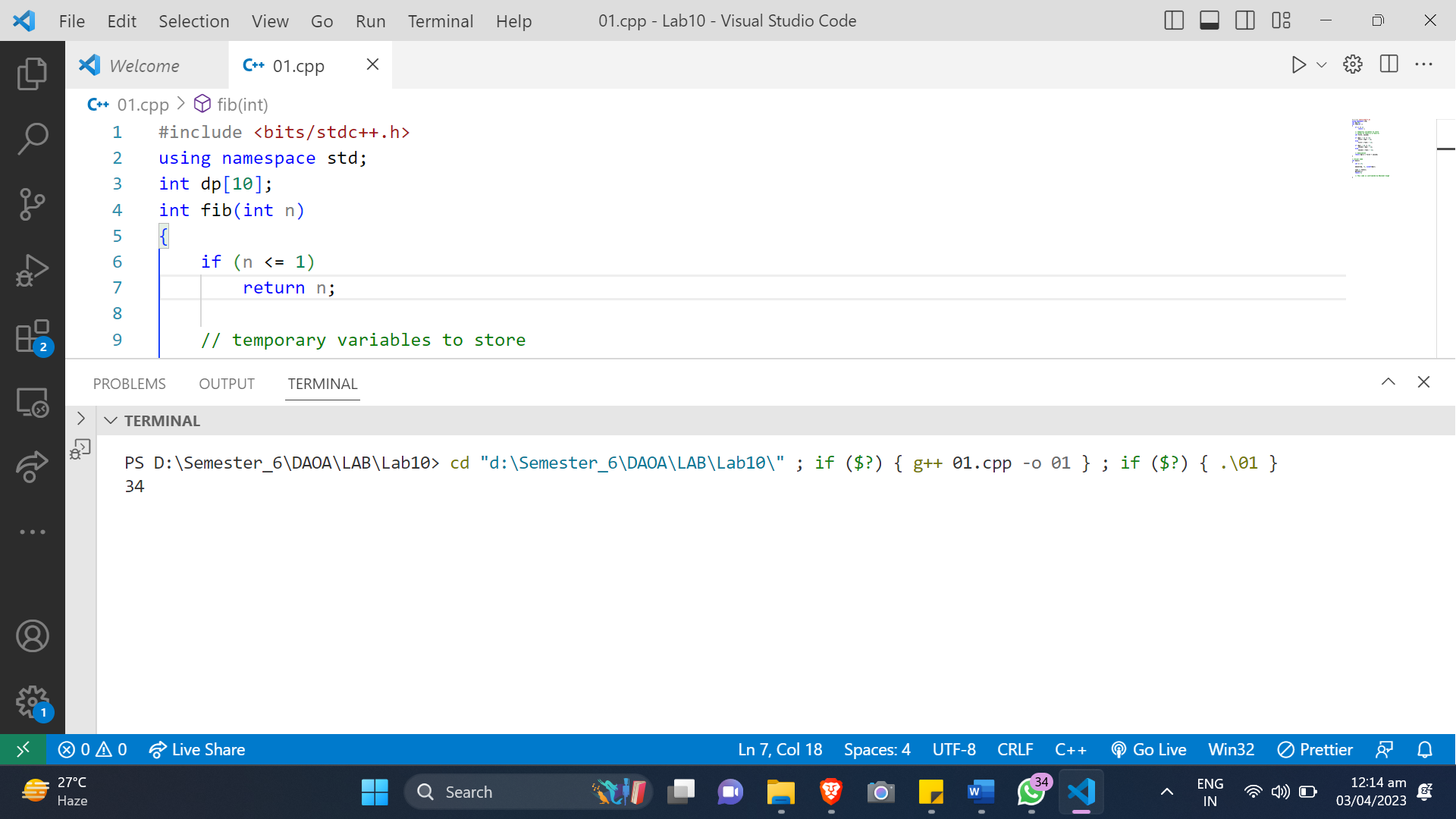
    getchar();

    return 0;

    // This code is contributed by Bhavneet Singh

}

Output:



**Conclusions:**

The code provided implements a dynamic programming approach to compute the n-th Fibonacci number, which significantly reduces the time complexity of the algorithm compared to the recursive approach. The time complexity of the algorithm is O(n), and the space complexity is also O(n). Overall, this is a very efficient and optimized solution to compute Fibonacci numbers.