ES-215 Assignment 1

Birudugadda Srivibhav (22110050)
Computer Science And Engineering, Indian Institute Of Technology Gandhinagar

***Note:** All Codes can be found here : https://github.com/Sparky1743/ES-215-Assignments*

Question -1

Implement a program(s) to list the first 50 fibonacci numbers preferably in C/C++

a) Using Recursion:

The program consists of two main functions: fibo_recursive and n_fibo_recursive.

```
long long fibo_recursive(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibo_recursive(n - 1) + fibo_recursive(n - 2);
}

vector <long long> n_fibo_recursive(int n){
    vector <long long> n_fibo_numbers;
    for (int i = 1; i < n + 1; i ++){
        n_fibo_numbers.push_back(fibo_recursive(i));
    }
    return n_fibo_numbers;
}
```

fibo_recursive(int n):

This is a recursive function that takes an integer n as input and returns the n-th Fibonacci number.

The function checks for the base cases:

1) If n is 0, it returns 0.
2) If n is 1, it returns 1.

For all other values of n, it returns the sum of the Fibonacci numbers of the two preceding positions: fibo_recursive(n - 1) + fibo_recursive(n - 2).

n_fibo_recursive(int n):

This function generates the first n Fibonacci numbers. It uses a vector to store the Fibonacci numbers.

A loop iterates from 1 to n, calling fibo_recursive(i) for each iteration and appending the result to the vector.

The vector of Fibonacci numbers is returned.

b) Using Loops

The program consists of two main functions: fibo_iterative and n_fibo_iterative.

```cpp
long long fibo_iterative(long long n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    long long n_minus_one = 1;
    long long n_minus_two = 0;
    long long n_th_term;

    for (int i = 2; i <= n; i++) {
        n_th_term = n_minus_one + n_minus_two;
        n_minus_two = n_minus_one;
        n_minus_one = n_th_term;
    }

    return n_th_term;
}

vector <long long> n_fibo_iterative(int n){
    vector <long long> n_fibo_numbers;
    for (int i = 1; i < n + 1; i ++){
        n_fibo_numbers.push_back(fibo_iterative(i));
    }
    return n_fibo_numbers;
}
```

fibo_iterative(long long n):

This function computes the n-th Fibonacci number using an iterative loop. It handles the base cases for n = 0 and n = 1 by returning 0 and 1, respectively. For n > 1, the function initializes two variables, n_minus_one and n_minus_two, to represent the two preceding Fibonacci numbers. The loop starts from 2 and iterates up to n, updating the variables to calculate the current Fibonacci number, n_th_term, without recalculating any Fibonacci number.

n_fibo_iterative(int n):

This function generates the first n Fibonacci numbers using the fibo_iterative function. A vector is used to store the Fibonacci numbers. A loop runs from 1 to n, calling fibo_iterative(i) for each iteration and appending the result to the vector. The vector containing the Fibonacci sequence is returned.

c) Using Recursion and memoization

The program consists of the following key component:

```
long long fibo_recursive_memoized(int n, vector<long long> &memo) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    if (memo[n] != -1) return memo[n];

    memo[n] = fibo_recursive_memoized(n - 1, memo) + fibo_recursive_memoized(n - 2, memo)
    return memo[n];
}
```

fibo_recursive_memoized(int n, vector<long long> &memo):

This function calculates the n-th Fibonacci number recursively, with memoization to avoid redundant calculations.

Base cases are checked first:

1) If n is 0, it returns 0.
2) If n is 1, it returns 1.

The function checks if the result for the current n is already stored in the memo vector. If so, it returns the stored value to avoid recalculating it. If the result is not memoized, the function computes it recursively and stores the result in the memo vector.

d) Using Loops and memoization

The program consists of the following key component:

```
long long fibo_iterative_memoized(int n, vector<long long> &memo) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    memo[0] = 0;
    memo[1] = 1;

    for (int i = 2; i <= n; i++) {
        memo[i] = memo[i - 1] + memo[i - 2];
    }

    return memo[n];
}
```

fibo_iterative_memoized(int n, vector<long long> &memo):

This function computes the n-th Fibonacci number using an iterative approach while storing each result in a memoization vector to prevent redundant calculations.

Base cases are immediately handled:

1) If n is 0, the function returns 0.
2) If n is 1, the function returns 1.

The memo vector is initialized such that memo[0] equals 0 and memo[1] equals 1. A loop iterates from 2 to n, filling the memo vector with the Fibonacci values, ensuring that each value is computed only once.

Time Computation:

```
struct timespec start, end;

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);

n_fibo_recursive(50);

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end);

// time taken
long long seconds = end.tv_sec - start.tv_sec;
long long nanoseconds = end.tv_nsec - start.tv_nsec;
if (nanoseconds < 0) {
    seconds--;
    nanoseconds += 1000000000L;
}
```

The execution time for each of the four Fibonacci implementations was measured using the clock_gettime function with the CLOCK_PROCESS_CPUTIME_ID setting to get the CPU time. This method provides a high-resolution measure of time, recorded in seconds and nanoseconds. By capturing the time just before and after the function execution, I calculated the elapsed time in nanoseconds to ensure precise measurement. Using this consistent timing method across all implementations allows for a fair comparison of their performance, highlighting differences in speed due to their respective algorithmic efficiency and optimization techniques.

Time taken:

1) recursion: 282.695997200 seconds
2) loops: 0.000008000 seconds
3) recursive memoized: 0.000001000 seconds
4) loop memoized: 0.000000500 seconds

The first 50 fibonacci numbers are:

Speed up = (Execution time of baseline algorithm) / (Execution time of improved algorithm)

S = T_baseline / T_improved

1) loops implementation is 35,336,999.65 times faster than recursive implementation
2) recursive memoized implementation is 282,695,997.2 times faster than recursive implementation
3) loops memoized implementation is 565,391,994.4 times faster than recursive implementation

Question -2

Write a simple Matrix Multiplication program for a given NxN matrix in any two of your preferred Languages from the following listed buckets, where N is iterated through the set of values 64, 128, 256, 512 and 1024. N can either be hardcoded or specified as input.

Language from bucket 1 - c++

Language from bucket 2 - python

a) implementation of integer and double matrix multiplication

Integer implementation for cpp:

```
// Function to multiply integer matrices
void multiplyIntegerMatrices(const vector<vector<int>>& matrix1, const vector<vector<int>>& matrix2, vector<vector<int>>& result, int dimension) {
    for (int i = 0; i < dimension; i++) {
        for (int j = 0; j < dimension; j++) {
            result[i][j] = 0;
            for (int k = 0; k < dimension; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

Double implementation for cpp:

```
// Function to multiply double matrices
void multiplyFloatingPointMatrices(const vector<vector<double>>& matrix1, const vector<vector<double>>& matrix2, vector<vector<double>>& result, int dimension) {
    for (int i = 0; i < dimension; i++) {
        for (int j = 0; j < dimension; j++) {
            result[i][j] = 0.0;
            for (int k = 0; k < dimension; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

Integer implementation for python:

```
def multiply_matrices_int(matrix_a, matrix_b, size):
    result_matrix = [[0] * size for _ in range(size)]
    for row in range(size):
        for col in range(size):
            for k in range(size):
                result_matrix[row][col] += matrix_a[row][k] * matrix_b[k][col]
    return result_matrix
```

Double implementation for python:

```python
def multiply_matrices_double(matrix_a, matrix_b, size):
    result_matrix = [[0.0] * size for _ in range(size)]
    for row in range(size):
        for col in range(size):
            for k in range(size):
                result_matrix[row][col] += matrix_a[row][k] * matrix_b[k][col]
    return result_matrix
```

For CPP:

Dimension	Type	System Time (s)	CPU Time (s)	Multiplication Time (s)
64	Integer	0.002548	0.002518	0.0024871
64	Floating-Point	0.0027406	0.002526	0.0025714
128	Integer	0.0197397	0.015625	0.019626
128	Floating-Point	0.0192835	0.015625	0.0190888
256	Integer	0.1674563	0.140625	0.1670953
256	Floating-Point	0.1617419	0.140625	0.1611423
512	Integer	1.2892412	1.09375	1.2881428
512	Floating-Point	1.3998034	1.234375	1.3974659
1024	Integer	11.3875216	10.5	11.3832001
1024	Floating-Point	22.4779611	19.921875	22.4692746

For Python:

Dimension	Type	System Time (s)	CPU Time (s)	Multiplication Time (s)
64	Integer	0.010989428	0.0099895	0.010989428
64	Double	0.02064085	0.015625	0.02064085
128	Integer	0.13636899	0.09375	0.135789633
128	Double	0.162477493	0.140625	0.162477493
256	Integer	1.125142097	0.9375	1.125142097
256	Double	1.392737627	1.203125	1.392737627
512	Integer	10.35852647	9.484375	10.35852647
512	Double	14.3518312	11.265625	14.3488369
1024	Integer	104.8778133	92.890625	104.8537605
1024	Double	173.6508904	148.125	173.6268759