Practical 4: Fibonacci Series

Student: Dhruv Sachdeva, `sachdevadhruv023@ce.du.ac.in`

Introduction

The Fibonacci sequence is a classic problem in computer science and mathematics:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2$$

We implement and compare four approaches to generate the first n Fibonacci numbers:

1. Recursive approach
2. Iterative approach
3. Dynamic Programming – Top-Down (Memoization)
4. Dynamic Programming – Bottom-Up (Tabulation)

Problem 1. *4(a) Recursive Fibonacci Generation*

Algorithm 1 Recursive Fibonacci

```
function FibRecursive(n)
    if n ≤ 1 then
        return n
    else
        return FibRecursive(n − 1) + FibRecursive(n − 2)
    end if
end function
```

C Code:

```c
#include <stdio.h>
#include <time.h>

int fib(int n) {
    if(n <= 1) return n;
    return fib(n-1) + fib(n-2);
}

int main() {
    int n;
    printf("Enter number: ");
    scanf("%d", &n);
    clock_t start = clock();
    for(int i=0;i<n;i++){
        printf("%d ", fib(i));
    }
    clock_t end = clock();
    double time_taken = (double)(end-start)/CLOCKS_PER_SEC;
    printf("\nTime taken: %f seconds\n", time_taken);
    return 0;
}
```

Problem 2. *4(b) Iterative Fibonacci Generation*

Algorithm 2 Iterative Fibonacci

function FIBITERATIVE(n)
    $t0 \leftarrow 0, t1 \leftarrow 1$
    for $i \leftarrow 0$ to $n - 1$ **do**
        Print $t0$
        $next \leftarrow t0 + t1$
        $t0 \leftarrow t1, t1 \leftarrow next$
    end for
end function

C Code:

```c
#include <stdio.h>
#include <time.h>

void fib(int n) {
    int t0=0, t1=1, next;
    for(int i=0;i<n;i++){
        printf("%d ", t0);
        next = t0 + t1;
        t0 = t1;
        t1 = next;
    }
    printf("\n");
}

int main() {
    int n;
    printf("Enter number: ");
    scanf("%d", &n);
    clock_t start = clock();
    fib(n);
    clock_t end = clock();
    double time_taken = (double)(end-start)/CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n", time_taken);
    return 0;
}
```

Problem 3. *4(c) Dynamic Programming – Top-Down (Memoization)*

Algorithm 3 Top-Down DP Fibonacci

Initialize memo array with -1
function FibTopDown(n)
    if $n \leq 1$ **then return** n
    end if
    if memo[n] $\neq -1$ **then return** memo[n]
        memo[n] $\leftarrow$ FibTopDown($n-1$) + FibTopDown($n-2$)
        return memo[n]

C Code:

```c
#include <stdio.h>
#include <time.h>
#define MAX 1000
int memo[MAX];

int fib(int n) {
    if(n <= 1) return n;
    if(memo[n] != -1) return memo[n];
    memo[n] = fib(n-1) + fib(n-2);
    return memo[n];
}

int main() {
    int n;
    printf("Enter number: ");
    scanf("%d", &n);
    for(int i=0;i<MAX;i++) memo[i] = -1;

    clock_t start = clock();
    for(int i=0;i<n;i++) printf("%d ", fib(i));
    clock_t end = clock();
    printf("\nTime taken: %f seconds\n", (double)(end-start)/CLOCKS_PER_SEC)
    return 0;
}
```

Problem 4. *4(d) Dynamic Programming – Bottom-Up (Tabulation)*

Algorithm 4 Bottom-Up DP Fibonacci

function FIBBOTTOMUP(n)
  dp[0] $\leftarrow$ 0, dp[1] $\leftarrow$ 1
  for $i \leftarrow 2$ to $n - 1$ **do**
    dp[i] $\leftarrow$ dp[$i - 1$] + dp[$i - 2$]
  end for
  Print dp array
end function=0

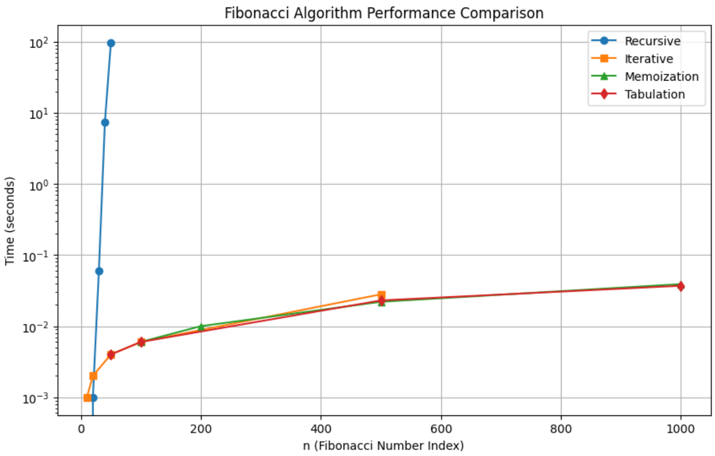
C Code:

```c
#include <stdio.h>
#include <time.h>
#define MAX 1000
int dp[MAX];

void fib(int n){
    dp[0] = 0;
    dp[1] = 1;
    for(int i=2;i<n;i++) dp[i] = dp[i-1] + dp[i-2];
    for(int i=0;i<n;i++) printf("%d ", dp[i]);
    printf("\n");
}

int main() {
    int n;
    printf("Enter number: ");
    scanf("%d",&n);
    clock_t start=clock();
    fib(n);
    clock_t end=clock();
    printf("Time taken: %f seconds\n", (double)(end-start)/CLOCKS_PER_SEC);
    return 0;
}
```

Comparison and Analysis

- ***Recursive:*** *Simple, but exponential time $O(2^n)$ and high stack usage.*
- ***Iterative:*** *Linear time $O(n)$ and $O(1)$ space (ignoring output), efficient for large n.*
- ***Top-Down DP:*** *Linear time $O(n)$ with $O(n)$ space due to memoization, avoids redundant recursion.*
- ***Bottom-Up DP:*** *Linear time $O(n)$ with $O(n)$ space; can be optimized to $O(1)$ space using two variables.*

Graph:



Conclusion

- *Recursive method is only suitable for small n due to exponential growth.*
- *Iterative and DP approaches scale well for large n.*
- *Bottom-up DP is usually fastest and most memory-efficient in practice.*