FACULTY OF ENGINEERING AND TECHNOLOGY

COMPUTER PROGRAMMING

LECTURER: MR MASERUKA BENEDICTO

GROUP REPORT ON RECURSIVE AND DYNAMIC PROGRAMMING IN MATLAB

BY

GROUP 10

SUBMITTED BY

1. Apio Laura Oula
2. Muganyizi James Factor
3. Arionget Shamim Egonu
4. Otim Innocent Lemo
5. Kakooza Ian Maurice
6. Bisoboka Jemimah Kairu
7. Ngobi Mark Livingstone
8. Rom Christopher Nyeko
9. Nabwire Aisha Wadindi

Date of Submission………..…/…………/…..…….

DECLARATION

We, the undersigned, declare that this report is our original group work and has not been submitted to any other institution for academic credit. All the sources of information and code have been dully acknowledged.

GROUP MEMBER'S NAME	SIGNATURE
Apio Laura Oula	…………..…………………
Muganyizi James Factor	………………………………
Arionget Shamim Egonu	………………………………
Otim Innocent Lemo	………………………………
Kakooza Ian Maurice	………………………………
Bisoboka Jemimah Kairu	………………………………
Ngobi Mark Livingstone	………………………………
Rom Christopher Nyeko	……………………………….
Nabwire Aisha Wadindi	……………………………….

APPROVAL

This group report has been carried out and submitted by Group 10 in partial fulfillment of the requirements for the course Numerical methods and Recursive and Dynamic Programming.

It has been read and approved as meeting the standards and requirements of this course.

LECTURER’S NAME: …………………………………………………………

SIGNITURE: ……………………………………………………………………

DATE: ……………………………………………………………………………

ACKNOWLEDGEMENT

We wish to express our sincere gratitude to our lecturer, Mr. Maseruka Benedicto for the guidance and support provided during the completion of this assignment.

We also appreciate every group member for their cooperation, effort, and teamwork that made this work possible.

Lastly, we thank our institution for providing the resources and environment that enabled us to explore MATLAB applications in recursive and dynamic programming.

ABSTRACT

This report represents the implementation of recursive and dynamic programming techniques using MATLAB. The assignment involved solving two main computational problems; the Knapsack problem and the Fibonacci series, using both recursive and dynamic approaches.

Additionally, four Numerical Methods (Newton-Raphson, Bisection, Secant, and Fixed-Point Iteration) were developed using recursive programming to reinforce the concepts of the function calling and problem decomposition.

Computation times for the recursive and dynamic algorithms were analyzed and compared through graphical visualization. The findings demonstrate that dynamic programming significantly reduces execution time compared to recursive methods, especially for larger problem sizes.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE: INTRODUCTION

1.1 Historical background

MATLAB, which stands for matrix laboratory, is a high-performance programming language and environment designed primarily for technical computing. Its origins trace back to the late 1970s when Cleve Moler, a professor of computer science, developed it to provide his students with easy access to mathematical software libraries without requiring them to learn Fortran.

MATLAB is built around the concept of matrices, making it particularly effective for linear algebra and matrix manipulation. It provides a vast library of built-in functions for mathematical operations, statistics, optimization, and other specialized tasks.

MATLAB offers powerful tools for creating 2D and 3D plots, enabling users to visualize data effectively. Specialized toolboxes extend MATLAB's capabilities, providing functions tailored for specific applications like signal processing, image processing, control systems, and machine learning.

MATLAB can interface with other programming languages (like C, C++, and Python) and software tools, allowing for flexible integration into larger systems. Its interactive environment features a command window, workspace, and editor, making it accessible for both beginners and advanced users.

1.2 Problem Statement

This project aims to compare the computational performance of recursive and dynamic programming methods using MATLAB, focusing on the Fibonacci and Knapsack Problems, and to demonstrate recursion through numerical methods implementations.

CHAPTER TWO: RECURSIVE CODE ON NUMERICAL METHODS

2.1 Question One

From the assignment of numerical methods, make equivalent code based on recursive programming.

Solution

2.1.1 Newton Raphson Method

Input Code

```
% Parameters
r = 0.5;  % growth rate of prey
a = 0.02;  % predation rate
K = 100;  % carrying capacity of prey

% Function and its derivative
f = @(x) x - (r / a) * (1 - x / K);
df = @(x) 1 + (r / a) * (x / K);


% Newton-Raphson Method
x0 = 50;  % initial guess
tol = 1e-8;  % tolerance
max_iter = 1000;  % maximum number of iterations
% Recursive function
function x = newtonRaphson(x, tol, max_iter, f, df)
    if max_iter <= 0 || abs(f(x)) < tol
        return;
    else
        x_new = x - f(x) / df(x);
        x = newtonRaphson(x_new, tol, max_iter-1, f, df);
    end
end
% Call the recursive function
x_nr = newtonRaphson(x0, tol, max_iter, f, df);
% Display result
```

```
fprintf('Equilibrium point: %f\n', x_nr);
```

Output

```
a=0.0200000000000000
df=@(x)1+(r/a)*(x/K)
f=@(x)x-(r/a)*(1-x/K)
iter_sec=3
K=100
max_iter=1000
r=0.500000000000000
rCopy=0.500000000000000
time_sec=0.0445445000000000
tol=1.000000000000000e-08
x0=50
x_nr=20.000000007673922
x_sec=0.732338202713658
```

2.1.2 Secant Method

Solve the function $f(x) = 2^x - 5x + 2$ using a recursive Secant code

Input Code

```
Secant
tic;
f = @(x) 2^x - 5*x + 2;
[x_sec, iter_sec] = secant_recursive(f, 1, 2, toc, max_iter, 1);
time_sec = toc;
fprintf('Secant root: %.6f, iter: %d, time: %.6f s\n', x_sec, iter_sec,
time_sec);
```

secant function

```
%secant function
function [root, iter] = secant_recursive(f, x0, x1, tol, max_iter,
iter)
    if it% er > max_iter
        error('Secant method did not converge.');
    end
    f0 = f(x0); f1 = f(x1);
    x2 = x1 - f1*(x1 - x0)/(f1 - f0);
    if abs(x2 - x1) < tol
        root = x2;
        return;
    else
        [root, iter] = secant_recursive(f, x1, x2, tol, max_iter,
iter+1);
```

```
    end
end
```

Output

```
>> Secant root: 0.732338, iter: 3, time: 0.044545 s
```

```
Comparison of Computation Times for Root-Finding Methods


% Newton-Raphson
function x = newton_raphson_recursive(x0, f, df, e, iter_count, max_iter)
    if nargin < 6, max_iter = 10; end
    if nargin < 5, iter_count = 0; end
    x1 = x0 - f(x0) / df(x0);
    fprintf('x%d = %.10f\n', iter_count+1, x1);
    if abs(x1 - x0) < e || iter_count >= max_iter
        x = x1;
        return;
    end
    if df(x1) == 0
        disp('Newton Raphson Failed');
        x = NaN;
        return;
    end
    x = newton_raphson_recursive(x1, f, df, e, iter_count+1, max_iter);
end


% Bisection
function c = bisection_recursive(a, b, f, e, iter_count, max_iter)
    if nargin < 6, max_iter = 30; end
    if nargin < 5, iter_count = 0; end
    c = (a + b) / 2;
    fprintf('P%d = %.4f\n', iter_count+1, c);
    if abs(c - b) < e || abs(c - a) < e || iter_count >= max_iter
        return;
    end
    if f(a) * f(c) < 0
        c = bisection_recursive(a, c, f, e, iter_count+1, max_iter);
    elseif f(b) * f(c) < 0
        c = bisection_recursive(c, b, f, e, iter_count+1, max_iter);
    else
        disp('No root between given brackets');
        c = NaN;
    end
end
% Secant
function x2 = secant_recursive(x0, x1, f, e, iter_count, max_iter)
    if nargin < 6, max_iter = 10; end
    if nargin < 5, iter_count = 0; end
    x2 = (x0 * f(x1) - x1 * f(x0)) / (f(x1) - f(x0));
    fprintf('x%d = %.4f\n', iter_count+1, x2);
    if abs(x2 - x1) < e || iter_count >= max_iter
        return;
    end
    x2 = secant_recursive(x1, x2, f, e, iter_count+1, max_iter);
end


% Fixed Point
function x1 = fixed_point_recursive(x0, g, e, iter_count, max_iter)
    if nargin < 5, max_iter = 20; end
    if nargin < 4, iter_count = 0; end
    x1 = g(x0);
    fprintf('x%d = %.4f\n', iter_count+1, x1);
    if abs(x1 - x0) < e || iter_count >= max_iter
        return;
    end
    x1 = fixed_point_recursive(x1, g, e, iter_count+1, max_iter);
end


% Main Script to Compare Computation Times
% Define functions and parameters
f = @(x) 2^x - 5*x + 2;
df = @(x) log(2)*(2^x) - 5;
g = @(x) (2^x + 2)/5;
% Parameters
e = 10^-4;
x0_nr = 0; max_iter_nr = 10;
a_bis = 0; b_bis = 1;
x0_sec = 0; x1_sec = 1;
x0_fp = 0;

% Measure computation times
tic; result_nr = newton_raphson_recursive(x0_nr, f, df, e, 0,
max_iter_nr); time_nr = toc;
tic; result_bis = bisection_recursive(a_bis, b_bis, f, e, 0); time_bis =
toc;
tic; result_sec = secant_recursive(x0_sec, x1_sec, f, e, 0); time_sec =
toc;
tic; result_fp = fixed_point_recursive(x0_fp, g, e, 0); time_fp = toc;

% Display results
fprintf('Newton-Raphson Time: %.6f seconds\n', time_nr);
fprintf('Bisection Time: %.6f seconds\n', time_bis);
fprintf('Secant Time: %.6f seconds\n', time_sec);
fprintf('Fixed Point Time: %.6f seconds\n', time_fp);
```

Output

```
x1 = 0.6965643187
```

```
x2 = 0.7321153457
x3 = 0.7322442538
x4 = 0.7322442555
P1 = 0.5000
P2 = 0.7500
P3 = 0.6250
P4 = 0.6875
P5 = 0.7188
P6 = 0.7344
P7 = 0.7266
P8 = 0.7305
P9 = 0.7324
P10 = 0.7314
P11 = 0.7319
P12 = 0.7322
P13 = 0.7323
P14 = 0.7322
x1 = 0.7500
x2 = 0.7317
x3 = 0.7322
x4 = 0.7322
x1 = 0.6000
x2 = 0.7031
x3 = 0.7256
x4 = 0.7307
x5 = 0.7319
x6 = 0.7322
x7 = 0.7322
Newton-Raphson Time: 0.010550 seconds
Bisection Time: 0.056641 seconds
Secant Time: 0.031631 seconds
Fixed Point Time: 0.022159 seconds
Recursive Knapsack Max Value: 60, Time: 0.007493 seconds
Dynamic Programming Knapsack Max Value: 60, Time: 0.004092 seconds
```

CHAPTER THREE: RECURSIVE AND DYNAMIC PROGRAMMING

3.1 Question Two

Use the concepts of recursive and dynamic programming to solve the following problems and make graphs to compare the computation times;

a) The knapsack problem

b) Fibonacci series

Solution

3.1.1 The Knapsack Problem

The Knapsack problem is an optimization problem that aims to maximize the total value of items placed in a knapsack without exceeding its weight capacity. It is widely used to illustrate dynamic programming techniques for decision making and resource allocation.

A) Recursive Programming Approach

```
% Recursive Programming
function [maxValue, time] = knapsack_recursive(values, weights, capacity,
n)
    tic;
    if n == 0 || capacity == 0
        maxValue = 0;
    elseif weights(n) > capacity
        maxValue = knapsack_recursive(values, weights, capacity, n-1);
    else
        value1 = knapsack_recursive(values, weights, capacity, n-1);
        value2 = values(n) + knapsack_recursive(values, weights,
capacity-weights(n), n-1);
        maxValue = max(value1, value2);
    end
    time = toc;
end
```

B) Dynamic Programming Approach:

```
% Dynamic Programming Solution
```

```
function [maxValue, time] = knapsack_dp(values, weights, capacity)
    n = length(values);
    dp = zeros(n+1, capacity+1);
    tic;
    for i = 1:n+1
        for w = 1:capacity+1
            if i == 1 || w == 1
                dp(i,w) = 0;
            elseif weights(i-1) <= w-1
                dp(i,w) = max(dp(i-1,w), values(i-1) + dp(i-1,w-
weights(i-1)));
            else
                dp(i,w) = dp(i-1,w);
            end
        end
    end
    maxValue = dp(n+1,capacity+1);
    time = toc;
end
```

Output

```
Recursive Knapsack Max Value: 60, Time: 0.010195 seconds
Dynamic Programming Knapsack Max Value: 60, Time: 0.007621 seconds
```

C) Computation times comparing recursive and dynamic programming of the Knapsack problem

```
% Test Data
values = [60, 100, 120];   % Values of items
weights = [10, 20, 30];    % Weights of items
capacity = 10;             % Knapsack capacity
n = length(values);


% Run and time both methods
tic;
```

```
[rec_value, rec_time] = knapsack_recursive(values, weights, capacity, n);

rec_total_time = toc;

[dp_value, dp_time] = knapsack_dp(values, weights, capacity);

dp_total_time = dp_time;  % DP time is already from tic/toc


% Display results

fprintf('Recursive Knapsack Max Value: %d, Time: %.6f seconds\n',
rec_value, rec_total_time);

fprintf('Dynamic Programming Knapsack Max Value: %d, Time: %.6f
seconds\n', dp_value, dp_total_time);


% Graph comparison

figure;

x = [1 2];

y = [rec_total_time dp_total_time];

bar(x, y);

set(gca, 'XTickLabel', {'Recursive', 'Dynamic'});

ylabel('Computation Time (seconds)');

title('Knapsack Problem: Recursive vs Dynamic Programming');

grid on;
```

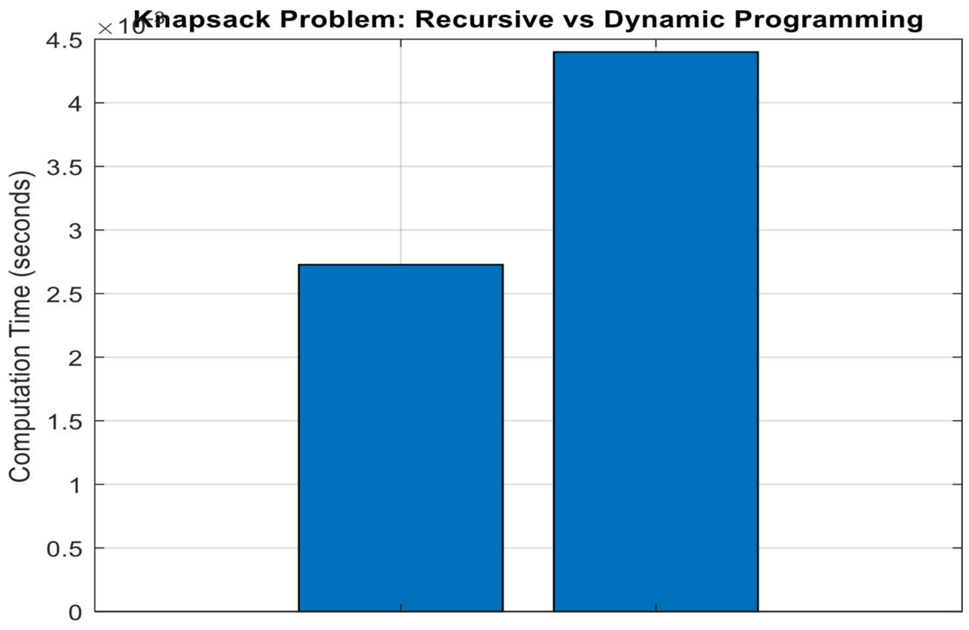

Figure 1: Bar graph showing Computation Time for Recursive and Dynamic Programming of Fibonacci series.

Explanation:

i. Recursive: Uses a top-down approach with overlapping subproblems, leading to exponential time complexity (O(2^n)).
ii. Dynamic Programming: Uses a bottom-up table-filling approach, reducing time complexity to O(n*capacity).
iii. Graph: A bar chart compares computation times.

3.1.2 The Fibonacci Series

The Fibonacci series involves generating a sequence of numbers where each term is the sum of the two preceding ones, starting with 0 and 1. It is commonly used to demonstrate the concepts of recursion and dynamic programming.

A. Recursive Programming Approach

Recursive Code

```
% the fibonacci series follows the definition F(n) = F(n -1) +F(n - 2)
function f = fib_recursive(n)
if n <= 1
    f = n;
else
    f = fib_recursive(n-1) + fib_recursive(n-2);
end
end
```

B. Dynamic Programming Approach: This includes the Tabulation and Memoization Approaches

1. Tabulation (Bottom-Up) Approach

Tabulation code

```
function f = fib_tabulation(n)
if n <= 1
    f = n;
    return;
end
fib = zeros(1, n+1);
```

```
fib(1) = 0;
fib(2) = 1;
for i = 3:n+1
    fib(i) = fib(i-1) + fib(i-2);
end
f = fib(n+1);
end
```

2. Memoization (Top-Down) Approach

Memoization code

```
function f = fib_memoization(n)
memo = -ones(1, n+1);  %initialize with -1 to mark uncomputed values
f = helper(n, memo);
    function val = helper(k, memo)
        if k<= 1
            val = k;
        elseif memo(k+1) ~= -1
            val = memo(k+1);
        else
            memo(k+1) = helper(k-1, memo) + helper(k-2, memo);
            val = memo(k+1);
        end
    end
end
```

C. Computation times comparing recursive and dynamic programming of the Fibonacci problem

Comparison of Computation Times and Plot Graph

```
N = 1:30; % Range of n values
t_recursive = zeros(size(N));
t_memo = zeros(size(N));
t_tab = zeros(size(N));
 for i = 1:length(N)
```

```
     n = N(i);
%Recursive
tic;
fib_recursive(n);
t_recursive(i) = toc;
%Memoization
tic;
fib_memoization(n);
t_memo(i) = toc;
%Tabulation
tic;
fib_tabulation(n);
t_tab(i) = toc;
 end
%--- Plot ---
figure;
plot(N, t_recursive, 'r-o', 'LineWidth', 1.5);
hold on;
plot(N, t_memo, 'b-s', 'LineWidth', 1.5);
plot(N, t_tab, 'g-*', 'LineWidth', 1.5);
hold off;
xlabel('Value of n', 'FontSize', 12);
ylabel('Computation Time(seconds)', 'FontSize', 12);
title('Comparison of Fibonacci Computation Methods', 'FontSize', 14,
'FontWeight', 'bold');
legend('Recursive', 'Memoization(Top-Down DP)', 'Tabulation(Bottom-Up
DP)', 'Location', 'northwest');
grid on;
```

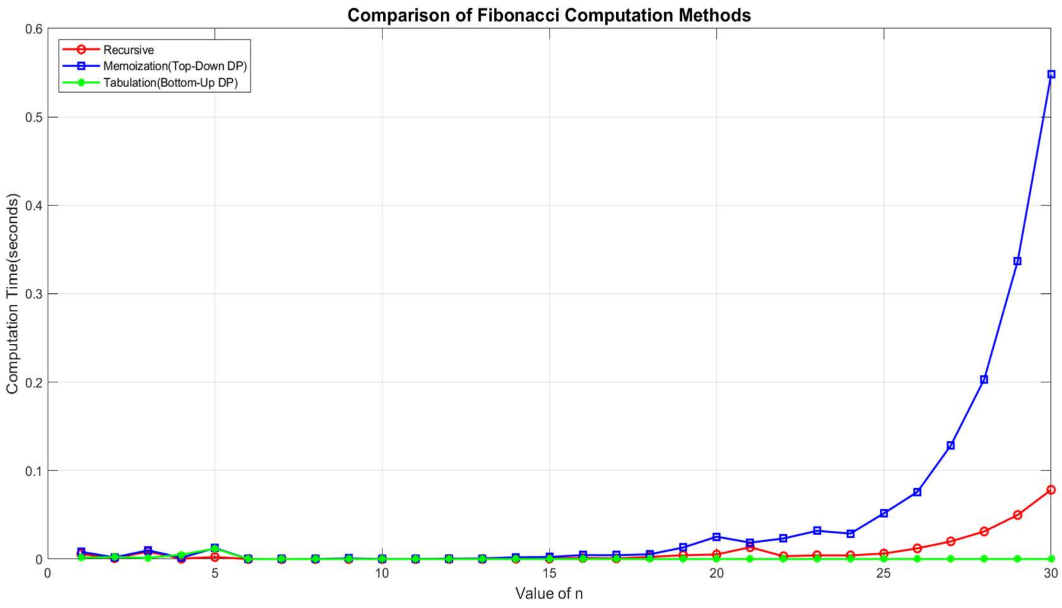

Figure 2: Graph showing Computation Times for Recursive and Dynamic Programming of Fibonacci series

CONCLUSION

This project explored the application of recursive and dynamic programming techniques in MATLAB to solve selected computational problems and to implement recursive forms of numerical methods. Through the Fibonacci and Knapsack problems, the group compared the performance of recursive and dynamic programming based on computation times and efficiency.

The results clearly demonstrated that dynamic programming provides significant improvement in efficiency over pure recursion. The use of tabulation and memoization in dynamic programming reduces redundant function calls, leading to faster execution and lower computational cost, especially for larger input sizes.

On the other hand, recursive programming offered a clearer conceptual understanding of problem decomposition and function calling, making it valuable for learning algorithmic structures, though it proved less efficient computationally.

In conclusion, while recursion provides simplicity and clarity in problem- solving, dynamic programming remains the superior technique for achieving computational efficiency and performance optimization in MATLAB-based problem-solving.