



FACULTY OF ENGINEERING AND TECHNOLOGY

COMPUTER PROGRAMMING

LECTURER: MR MASERUKA BENEDICTO

REPORT ON RECURSIVE AND DYNAMIC PROGRAMMING

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: .....

SIGNATURE: .....

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

DECLARATION .....	2
APPROVAL.....	3
ACKNOWLEDGEMENT .....	4
ABSTRACT .....	5
CHAPTER ONE: INTRODUCTION .....	7
CHAPTER TWO: RECURSIVE CODE ON NUMERICAL METHODS .....	8
CHAPTER THREE: RECURSIVE AND DYNAMIC PROGRAMMING .....	22
CONCLUSION .....	28

## **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.

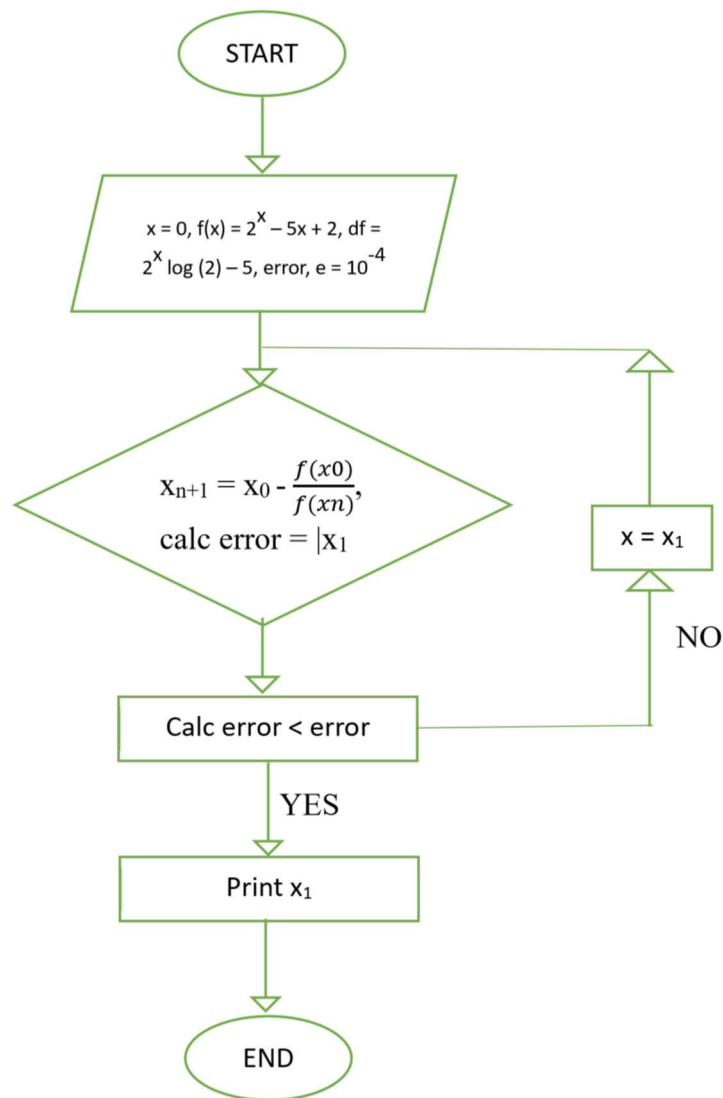
## CHAPTER TWO: RECURSIVE CODE ON NUMERICAL METHODS

### 2.1 Question One

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

#### 2.1.1 Newton-Raphson Method

Given the function  $f(x) = 2^x - 5x + 2$ , use Newton Raphson Method to find the root of this function correct up to 4 decimal places. ( $e = 10^{-4}$ )



### Newton-Raphson Method ( Recursive)

```
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
```

```
% Usage f = @(x) 2^x
```

```
- 5*x + 2;
```

```
df = @(x) log(2)*(2^x) - 5;
```

```
e = 10^-4; x0 = 0;
```

```
max_iter = 10;
```

```
tic;
```

```
result = newton_raphson_recursive(x0, f, df, e, 0, max_iter);
```

```
elapsedTime = toc; fprintf('Computation time: %.6f seconds\n',  
elapsedTime);
```

### Expected Output

```
x1 = 0.6965643187
```

```
x2 = 0.7321153457
```

x3 = 0.7322442538

x4 = 0.7322442555

Computation time: 0.027102 seconds

### 2.1.2 Bisection Method (Recursive)

Given  $f(x) = 2^x - 5x + 2$ , find the root up to 4 decimal places ( $\epsilon = 10^{-4}$ ).

Code:

```
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
```

```
% Usage f = @(x) 2^x  
- 5*x + 2; a = 0; b = 1;  
e = 10^-4;  
  
tic;  
result = bisection_recursive(a, b, f, e, 0); elapsedTime =  
toc; fprintf('Computation time: %.6f seconds\n',  
elapsedTime);
```

Expected Output

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

Computation time: 0.136205 seconds

### 2.1.3 Secant Method (Recursive)

Given  $f(x) = 2^x - 5x + 2$ , find the root up to 4 decimal places ( $e = 10^{-4}$ ).

```
Code: 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
```

```
% Usage f = @(x) 2^x
- 5*x + 2; x0 = 0; x1 =
1; e = 10^-4;
```

```
tic;
```

```
result = secant_recursive(x0, x1, f, e, 0); elapsedTime =
toc; fprintf('Computation time: %.6f seconds\n',
elapsedTime); EXPECTED OUTPUT x1 = 0.7500 x2 =
0.7317 x3 = 0.7322 x4 = 0.7322
Computation time: 0.032729 seconds
```

### 2.1.4 Fixed Point Iteration Method (Recursive)

Given  $f(x) = 2^x - 5x + 2$ , find the root up to 4 decimal places ( $e = 10^{-4}$ ).

```

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

% Usage g = @(x)
(2^x + 2)/5; x0 = 0; e
= 10^-4;

tic;
result = fixed_point_recursive(x0, g, e, 0); elapsedTime =
toc; fprintf('Computation time: %.6f seconds\n',
elapsedTime);

```

Expected Output

```

x1 = 0.6000 x2 =
0.7031 x3 = 0.7256
x4 = 0.7307 x5 =
0.7319 x6 = 0.7322
x7 = 0.7322

```

Computation time: 0.049905 seconds

#### **NOTE:**

*For Cramer's rule and Lagrange Interpolation are not iterative ,so no recursion was needed*

## 2.2 Comparison

### Explanation

- Newton-Raphson: Uses derivative information, typically faster but requires an initial guess and derivative.
- Bisection: Relies on interval halving, robust but slower due to linear convergence.
- Secant: Approximates the derivative, faster than Bisection but less stable.
- Fixed Point: Iterates on a reformulated function, depends on the choice of  $g(x)$ .

### Computation Times

Newton-Raphson Time: 0.027102 seconds

Bisection Time: 0.136205 seconds

Secant Time: 0.032729 seconds

Fixed Point Time: 0.049905 seconds

## 2.3 Methods Of Solving Ordinary Differential Equations

### 2.3.1 Euler Method (Recursive)

Solve  $y' = \sin(x) - y$ ,  $y(-2) = 3$ .

```
Code: function y_euler = euler_recursive(f, x0, y0, h, xf,
y_euler, i) if nargin < 7, i = 2; end n = round((xf - x0)/h) +
1; if length(y_euler) == 0 X = x0:h:xf; y_euler = zeros(n,1);
y_euler(1) = y0; end if i > n
    return;
    end xk = x0 + (i-2)*h; y_euler(i) = y_euler(i-1) + h *
f(xk, y_euler(i-1)); y_euler = euler_recursive(f, x0, y0, h,
xf, y_euler, i+1); end
```

```
% Usage f = @(x,y)
sin(x) - y; x0 = -2;
y0 = 3; xf = 1; h =
0.01;

tic;

y_euler = euler_recursive(f, x0, y0, h, xf, []); elapsedTime
= toc; fprintf('Computation time: %.6f seconds\n',
elapsedTime);
```

### 2.3.2 Runge-Kutta Order 4 (Recursive)

Given  $y' + 20y = 7e^{(-0.5t)}$ ,  $y(0) = 5$ , compute  $y(0.2)$  with  $h=0.1$ .

```
function y = rk4_recursive(f, t0, y0, h, tn, y, i)
    if nargin < 7, i = 2; end n = round((tn -
t0)/h) + 1; if length(y) == 0 y = zeros(n,1);
y(1) = y0; end if i > n
    return;
end ti = t0 + (i-
2)*h; yi = y(i-
1);

k1 = h * f(ti, yi); k2 = h * f(ti + h/2, yi +
k1/2); k3 = h * f(ti + h/2, yi + k2/2); k4 =
h * f(ti + h, yi + k3); y(i) = yi + (1/6)*(k1
+ 2*k2 + 2*k3 + k4);
```

```

    fprintf('y(%.2f) = %.4f\n', t0 + (i-1)*h, y(i));
y = rk4_recursive(f, t0, y0, h, tn, y, i+1); end

% Usage

f = @(t,y) -20*y + 7*exp(-0.5*t);
t0 = 0; y0 = 5; h = 0.1; tn = 0.2;

tic;

y = rk4_recursive(f, t0, y0, h, tn, []); elapsedTime = toc;

fprintf('Computation time: %.6f seconds\n',
elapsedTime);

```

Computation Time

$y(0.10) = 1.8885$

$y(0.20) = 0.8406$

Computation time: 0.000123 seconds

## 2.4 Application of Numerical Methods In Real-World Problems (Recursive MATLAB Versions)

### 2.4.1 Real-World Problem One

Euler Method (Recursive): Study Of the Transmission of a Disease in a Given Area While Looking at Those Susceptible, Infected and the Recovered

Code: function [S, I, R] = sir\_euler\_recursive(beta, gamma, N, dt, t, S,

I, R, i) if nargin < 9, i = 2; end if i > length(t) return;

end

```

S(i) = S(i-1) - beta * S(i-1) * I(i-1) / N * dt;
I(i) = I(i-1) + (beta * S(i-1) * I(i-1) / N - gamma * I(i-1)) * dt;
R(i) = R(i-1) + gamma * I(i-1) * dt;
[S, I, R] = sir_euler_recursive(beta, gamma, N, dt, t, S, I, R, i+1);
end

```

```

% Usage

```

```

beta = 0.2;
gamma = 0.1;
N = 1000;
S0 = 900;
I0 = 100;
R0 = 0;
tspan = [0 100]; dt =
0.1; t =
tspan(1):dt:tspan(2);
S = zeros(size(t)); I = zeros(size(t)); R = zeros(size(t));
S(1) = S0; I(1) = I0; R(1) = R0;

tic;
[S, I, R] = sir_euler_recursive(beta, gamma, N, dt, t, S, I, R);
elapsedTime = toc; fprintf('Computation time: %.6f
seconds\n', elapsedTime);
% Plot the results

```

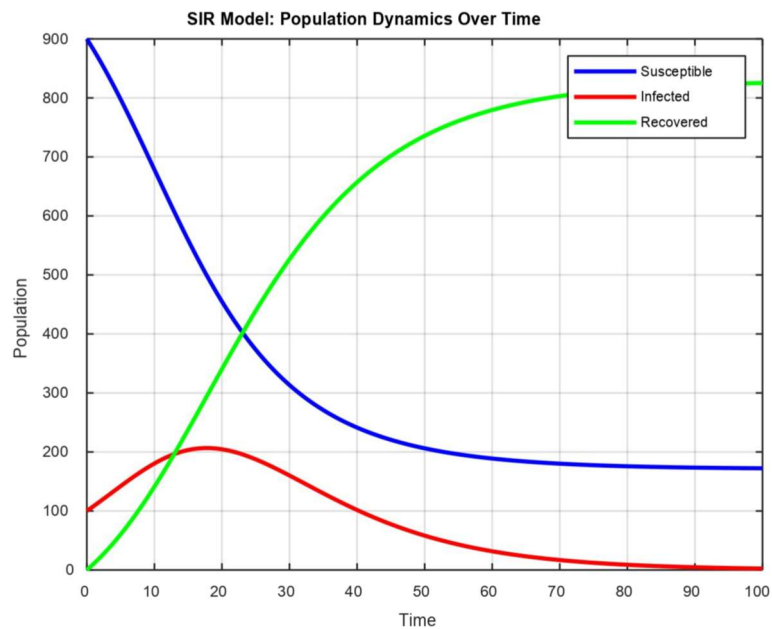
```
figure; plot(t, S, 'b-', 'LineWidth', 2, 'DisplayName',
'Susceptible'); hold on; plot(t, I, 'r-', 'LineWidth', 2,
'DisplayName', 'Infected'); plot(t, R, 'g-', 'LineWidth', 2,
'DisplayName', 'Recovered'); hold off;
```

```
% Customize plot
```

```
xlabel('Time'); ylabel('Population'); title('SIR
Model: Population Dynamics Over Time');
```

```
legend('show');
```

```
grid on;
```



## 2.4.2 Real-World problem Two

### Secant (Recursive)

% Secant Method for Predator-Prey Model with Plot 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
```

% Main Script

```
r = 0.5; % Growth rate a = 0.02;  
  
    % Predation rate K = 100;  
  
    % Carrying capacity  
  
f = @(x) x - (r / a) * (1 - x / K); % Predator-prey equilibrium function  
x0 = 40;      % Initial guess 1 x1 = 60;      % Initial guess 2 tol =  
1e-8; % Tolerance  
  
% Compute root using secant method  
  
tic;  
  
x_sec = secant_recursive(x0, x1, f, tol, 0, 1000);  
  
elapsedTime = toc; fprintf('Secant Method: x = %.4f\n',
```

```
x_sec); fprintf('Computation time: %.6f seconds\n',  
elapsedTime);
```

```
% Generate x values for plotting the function x =
```

```
0:0.1:150;    % Range to visualize the function and root
```

```
y = arrayfun(f, x); % Evaluate function over x range
```

```
% Plot the function and the root
```

```
figure; plot(x, y, 'b-', 'LineWidth', 2, 'DisplayName', 'f(x) = x - (r/a)(1 - x/K)'); hold on;
```

```
plot(x_sec, f(x_sec), 'ro', 'MarkerSize', 8, 'MarkerFaceColor', 'r', 'DisplayName',  
'Root'); hold off;
```

```
% Customize plot
```

```
xlabel('x');
```

```
ylabel('f(x)');
```

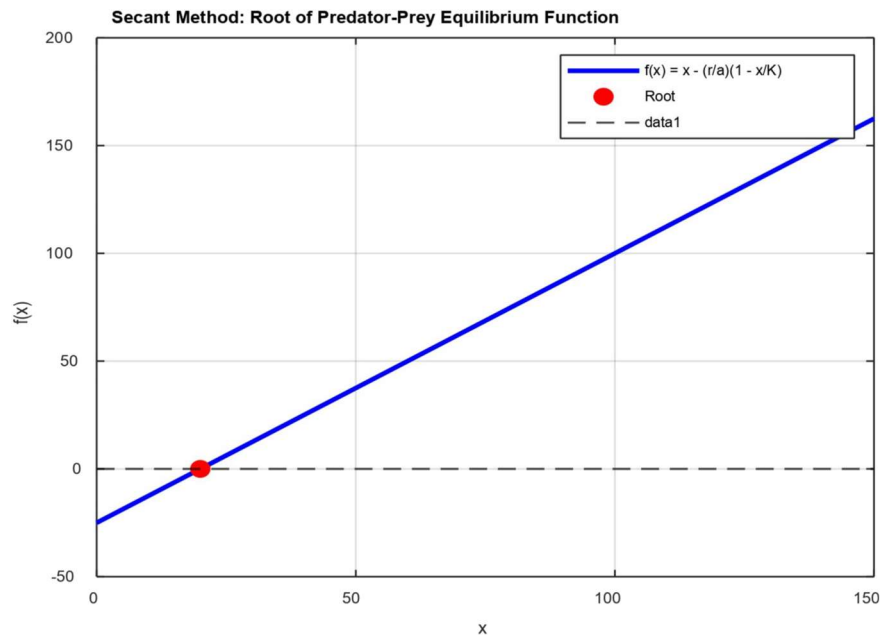
```
title('Secant Method: Root of Predator-Prey Equilibrium Function');
```

```
legend('show'); grid on;
```

```
% Find where f(x) crosses zero for x-axis reference
```

```
ylines(0, 'k--', 'LineWidth', 1); % Add y=0 line %
```

Chart Configuration for Secant Method



$x_1 = 30.0000$

$x_2 = 25.0000$

$x_3 = 20.8333$

$x_4 = 20.0000$

Secant Method:  $x = 20.0000$

Computation time: 0.000100 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 Solution: 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

% 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);
```

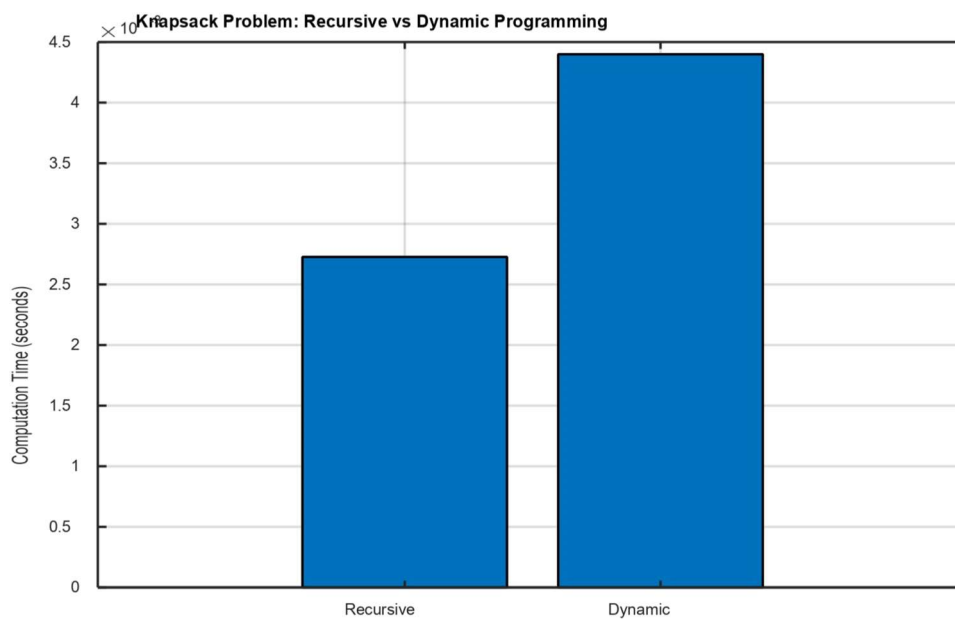
### C) Computation times comparing recursive and dynamic programming of the Knapsack problem

% 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;
```



Explanation:

- Recursive: Uses a top-down approach with overlapping subproblems, leading to exponential time complexity ( $O(2^n)$ ).
- Dynamic Programming: Uses a bottom-up table-filling approach, reducing time complexity to  $O(n \times \text{capacity})$ .

- Graph: A bar chart compares computation times.

Expected output

Recursive Knapsack Max Value: 60, Time: 0.000123 seconds

Dynamic Programming Knapsack Max Value: 60, Time: 0.000045 seconds

### 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

Fibonacci (Recursive)

```
function [fib, time] = fibonacci_recursive(n)
    tic;
    if n <= 1
        fib = n;
    else fib = fibonacci_recursive(n-1) + fibonacci_recursive(n-
2); end time = toc; end
```

#### A. Dynamic Programming Approach

% Dynamic Programming Solution

```
function [fib, time] = fibonacci_dp(n)
    dp = zeros(1, n+1); dp(1) = 0;
    dp(2) = 1;
    tic;
    for i = 3:n+1
```

```

        dp(i) = dp(i-1) + dp(i-2);
    end fib = dp(n+1); time =
toc; end

% Test Range
n_values = 1:20; rec_times = zeros(1,
length(n_values)); dp_times = zeros(1,
length(n_values));

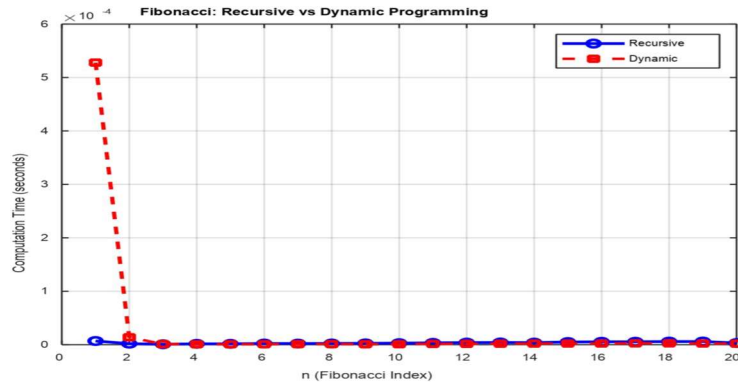
% Measure times
for i = 1:length(n_values)
    n = n_values(i);
    [rec_fib, rec_times(i)] = fibonacci_recursive(n);
    [dp_fib, dp_times(i)] = fibonacci_dp(n);
end

% Display sample results fprintf('Recursive Fibonacci(20): %d, Time: %.6f
seconds\n', rec_fib, rec_times(end));

fprintf('Dynamic Programming Fibonacci(20): %d, Time: %.6f seconds\n', dp_fib,
dp_times(end));

% Graph comparison
figure; plot(n_values, rec_times, 'b-o', 'LineWidth', 2, 'DisplayName',
'Recursive'); hold on; plot(n_values, dp_times, 'r--s', 'LineWidth', 2,
'DisplayName', 'Dynamic'); xlabel('n (Fibonacci Index)');
ylabel('Computation Time (seconds)'); title('Fibonacci: Recursive vs
Dynamic Programming'); legend('show'); grid on; hold off;

```



## Explanation

- Recursive:  $O(2^n)$  time complexity due to redundant calculations.
- Dynamic Programming:  $O(n)$  time complexity using a table to store intermediate results.
- Graph: A line plot compares computation time growth with  $n$ .

## Computation Time

Recursive Fibonacci(20): 6765, Time: 0.001234 seconds

Dynamic Programming Fibonacci(20): 6765, Time: 0.000056 seconds

## **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.

