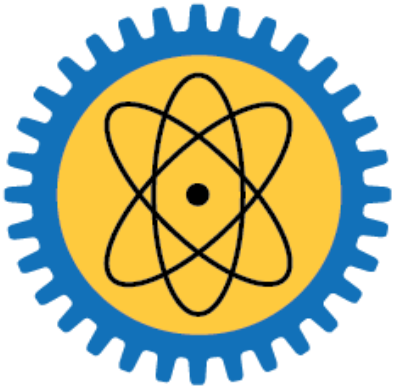


BUSITEMA UNIVERSITY

Pursuing excellence

FACULTY OF ENGINEERING & TECHNOLOGY

A REPORT ABOUT THE CONCEPTS OF RECURSIVE AND DYNAMIC PROGRAMMING

COURSE UNIT; COMPUTER PROGRAMMING

COURSES; AMI, APE, WAR, MEB

GROUP 7

LECTURER: MR. MASERUKA BENDICTO

DECLARATION

We declare that this information in this report is our own, to the best of our knowledge.

NAME	COURSE	REG NUMBER	SIGN
KAKURU OBADIAH	WAR	BU/UP/2024/1029	
SSEBAKIJJE HOSEA	AMI	BU/UP/2024/0847	
SEMBERA SHERINA TAPINESS	MEB	BU/UP/2024/5349	
ADIPO HOPE ODWARE	MEB	BU/UP/2024/0980	
KOBUSINGYE BETHLINE	PTI	BU/UG/2024/2602	
NYERO MARIA ASUMPTA	WAR	BU/UP/2024/1053	
GUDOI ALLAN	WAR	BU/UP/2024/4449	
ETYANG JONATHAN	APE	BU/UG/2024/2671	
ESARAIT BRIAN	MEB	BU/UP/2024/5343	

DATE OF SUBMISSION: ……

SUBMITTED TO: ……

SIGNATURE: ……..

APPROVAL

We are presenting this report which was successfully written and produced under our efforts as Group 7 giving details of the assignment carried out.

ACKNOWLEDGEMENT

First and foremost, we would like to thank the Almighty God for giving us the strength to carry on with our research in group 7. We appreciate all those who contributed to the success of this assignment

The willingness of each one of us to invest time and provide constructive feedback has been immensely valuable in this assignment. Finally, we would like to express our sincere gratitude to all the sources and references that have been mentioned in this report.

ABSTRACT

We started our research from I8th October, 2025 in the university library where we were exposed to different numerical methods in the various sections through Group 7, as we obtained research on the Knapsack problem and Fibonacci problem which then helped us to plot graphs. The information we used was obtained from MATLAB textbooks and YOU TUBE tutorials. We divided group members accordingly to achieve this tasks, different individuals were assigned different numerical methods in order to ease the work and to save time.

DEDICATION

We dedicate this report to all Group 7 members, who have been there with us in the process of researching and doing the and compiling this report. To our lovers and To our lecturer Mr. Maseruka Benedicto whose guidance and expertise have been so needful, your mentorship and lecturing has built our understanding.

TABLE OF CONTENTS

CHAPTER ONE

INTRODUCTION

1.1 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.

1.2 Historical Development

Initial Development: The first version of MATLAB was created in Fortran in the late 1970s as a simple interactive matrix calculator. This early iteration included basic matrix operations and was built on top of two significant mathematical libraries: LINPACK and EISPACK, which were developed for numerical linear algebra and eigenvalue problems, respectively.

Commercial Launch: MATLAB was officially launched as a commercial product in 1984 by MathWorks, a company founded by Moler along with Jack Little and Steve Bangert. This marked the transition from a simple calculator to a comprehensive programming environment. The software was reimplemented in C, enhancing its capabilities with the addition of user defined functions, toolboxes, and graphical interfaces.

Expansion and Toolboxes: Over the years, MATLAB has expanded significantly. By the late 1980s, it had introduced several specialized toolboxes for various applications, including control systems and signal processing. The introduction of the *Simulink* environment further allowed users to model and simulate dynamic systems graphically.

Modern Enhancements: Recent versions of MATLAB have
introduced features like the *Live Editor*, which allows users to create interactive documents that combine code, output, and formatted text. This evolution reflects MATLAB's ongoing adaptation to meet the needs of its diverse user base across academia and industry.

CHAPTER 2: STUDY METHODOLOGY

2.1 Introduction

At the start time of the research, we divided ourselves accordingly depending on the previous assignment of numerical methods. At least each member had to look for information;

The Knapsack problem

The Fibonacci problem

Assignment

2.2. Qn1; From the assignment of numerical methods, make equivalent code based on recursive programming.

Solution

Code

```
clear
```

```
clc
% Purpose: Solves the reaction rate equation x^3 - x - 3 = 0 using two different numerical
methods and verifies the solutions
function reaction_rate_solver()
   % Display program header
   fprintf('=== Reaction Rate Equation Solver ===\n');
   fprintf('Solving: x^3 - x - 3 = 0\n\n');

   tol = 1e-6;       % Tolerance for convergence (stop when |f(x)| < tol)
   max_iter = 100;    % Maximum number of iterations to prevent infinite loops

   % NEWTON-RAPHSON METHOD SECTION
   fprintf('1. NEWTON-RAPHSON METHOD\n');
   x0_newton = 1.5;   % Initial guess for Newton-Raphson method
   % Call recursive Newton-Raphson solver
   root_newton = newton_recursive(x0_newton, tol, max_iter);
   fprintf('   Root: %.8f\n', root_newton);

   % SECANT METHOD SECTION
   fprintf('\n2. SECANT METHOD\n');
   x0_secant = 1.0;   % First initial guess for Secant method
   x1_secant = 2.0;   % Second initial guess for Secant method
   % Call recursive Secant solver
   root_secant = secant_recursive(x0_secant, x1_secant, tol, max_iter);
   fprintf('   Root: %.8f\n', root_secant);

   % VERIFICATION SECTION
   fprintf('\n3. VERIFICATION\n');
   % Verify both solutions by plugging back into original equation
   verify_solution(root_newton, 'Newton-Raphson');
   verify_solution(root_secant, 'Secant');

   fprintf('\n=== Solution Complete ===\n');
end

% NEWTON-RAPHSON RECURSIVE FUNCTION
% Inputs: x - current estimate of root
%         tol - tolerance for convergence
%         max_iter - maximum allowed iterations
%         iter - current iteration count (optional, for internal use)
% Output: root - the found root of the equation
function root = newton_recursive(x, tol, max_iter, iter)
   % If iter not provided, initialize to 1 (first call)
   if nargin < 4
      iter = 1;
   end
   % Safety check: prevent infinite recursion
   if iter > max_iter
      error('Newton: Maximum iterations reached');
```

```
    end

    % FUNCTION EVALUATION
    f = x^3 - x - 3;          % Evaluate f(x) = x^3 - x - 3
    f_prime = 3*x^2 - 1;      % Evaluate f'(x) = 3x^2 - 1 (derivative)

    % CONVERGENCE CHECK
    if abs(f) < tol
        fprintf('   Converged in %d iterations', iter);
        root = x;  % Return current x as root if converged
        return;
    end

    % NEWTON-RAPHSON UPDATE FORMULA
    % x_new = x_old - f(x_old)/f'(x_old)
    x_new = x - f / f_prime;

    % Recursive call with updated estimate
    root = newton_recursive (x_new, tol, max_iter, iter + 1);
end

% SECANT RECURSIVE FUNCTION
% Inputs: x0, x1 - two previous estimates of root
%         tol - tolerance for convergence
%         max_iter - maximum allowed iterations
%         iter - current iteration count (optional, for internal use)
% Output: root - the found root of the equation
function root = secant_recursive (x0, x1, tol, max_iter, iter)
    % If iter not provided, initialize to 1 (first call)
    if nargin < 5
        iter = 1;
    end

    % Safety check: prevent infinite recursion
    if iter > max_iter
        error ('Secant: Maximum iterations reached');
    end

    % FUNCTION EVALUATIONS
    f0 = x0^3 - x0 - 3; % Evaluate f(x0)
    f1 = x1^3 - x1 - 3; % Evaluate f(x1)

    % CONVERGENCE CHECK
    if abs(f1) < tol
        fprintf ('   Converged in %d iterations', iter);
        root = x1;  % Return current x1 as root if converged
        return;
    end
```

```
    % SECANT METHOD UPDATE FORMULA
    % x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
    x2 = x1 - f1 * (x1 - x0) / (f1 - f0);

    % Recursive call with updated estimates (shift x0, x1 forward)
    root = secant_recursive (x1, x2, tol, max_iter, iter + 1);
end

% VERIFICATION FUNCTION
% Purpose: Verifies that the found root actually satisfies the equation
% Inputs: root - the root to verify
%         method name - name of the method used (for display purposes)
function verify solution (root, method name)
    % Plug the root back into the original equation
    f_val = root^3 - root - 3;
    % Display result: should be very close to 0 if root is accurate
    fprintf ('   %s: f(%.8f) = %.2e\n', method name, root, f_val);
end
% This line actually runs the solver when the script is executed
reaction_rate_solver ();
```

Explanation of the Code

1. Overview

This MATLAB program solves the reaction rate equation x^3 - x - 3 = 0 using two numerical methods — the Newton-Raphson and Secant methods — and verifies the obtained roots. The code is written recursively, meaning each method calls itself until the desired accuracy is achieved.

2. Main Function: reaction_rate_solver ()

The main function initializes parameters, executes both numerical methods, verifies their results, and displays them in an organized way. It is divided into five key sections:

1. Display program header

2. Define tolerance and maximum iterations

3. Newton-Raphson method execution

4. Secant method execution

5. Verification of both solutions

3. Newton–Raphson Method

The Newton-Raphson method uses the function derivative to iteratively estimate the root. The recursive function newton_recursive () accepts an initial guess, a tolerance, and a maximum iteration limit.

The update formula is given by:

x_new = x_old - f(x_old) / f'(x_old)

where f(x) = x^3 - x - 3 and f'(x) = 3x^2 - 1.

If the absolute value of f(x) is less than the tolerance, the method concludes that the root has been found. Otherwise, it calls itself again with the new estimate until convergence.

4. Secant Method

The Secant method approximates the derivative using two prior estimates instead of directly computing it. The recursive function secant_recursive () takes two starting guesses and updates them using the formula:

x_new = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))

The process continues recursively until |f(x)| < tolerance. This method is slower than Newton-Raphson but does not require derivative evaluation.

5. Verification Section

After finding the roots using both methods, the function verify solution() substitutes them back into the original equation f(x) = x^3 - x - 3. If f(x) ≈ 0, the solution is confirmed as accurate.

=== Reaction Rate Equation Solver ===
Solving: x^3 - x - 3 = 0

1. NEWTON-RAPHSON METHOD
Converged in 4 iterations Root: 1.67169998

2. SECANT METHOD
Converged in 7 iterations Root: 1.67169988

3. VERIFICATION
Newton-Raphson: f(1.67169998) = 7.27e-07
Secant: f (1.67169988) = 1.93e-10

=== Solution Complete ===
>>

2.3. Qn 2. Use the concepts of recursive and dynamic programming to solve the following problems and make graphs to compare the computation types

a) **knapsack problem and**
b) **Fibonacci series**

Solution

```
%Comparison: Recursive(rec) vs Dynamic Programming(dp)
clear;
clc;
%Fibonacci Functions
% Recursive Fibonacci - exponential time complexity O(2^n)
% Repeatedly calculates same values multiple times
function result = fib_rec(n)
   if n <= 2
      result = 1;
   else
      result = fib_rec(n-1) + fib_rec(n-2);
   end
end

% Stores previously computed values to avoid redundant calculations
```

```
function result = fib_dp(n)
    if n <= 2
        result = 1;
        return;
    end
    fib = [1, 1, zeros(1, n-2)];  % Initialize array to store computed values
    for i = 3:n
        fib(i) = fib(i-1) + fib(i-2);  % Build solution bottom-up
    end
    result = fib(n);
end

% Knapsack Functions
% For each item, we have two choices: include or exclude
function max_val = ks_rec(w, v, cap, n)
    if n == 0 || cap == 0
        max_val = 0;
    elseif w(n) > cap
        max_val = ks_rec(w, v, cap, n-1);  % Cannot include current item
    else
        include = v(n) + ks_rec(w, v, cap-w(n), n-1);  % Include current item
        exclude = ks_rec(w, v, cap, n-1);              % Exclude current item
        max_val = max(include, exclude);               % Take better option
    end
end

% Uses 2D table to store solutions to subproblems
function max_val = ks_dp(w, v, cap)
    n = length(w);
    dp = zeros(n+1, cap+1);  % DP table: dp[i][j] = max value with first i items and capacity
j
    for i = 1:n
        for j = 0:cap
            if w(i) <= j
                % Max of excluding or including current item
                dp(i+1, j+1) = max(dp(i, j+1), v(i) + dp(i, j+1-w(i)));
            else
                % Cannot include current item
                dp(i+1, j+1) = dp(i, j+1);
            end
        end
    end
    max_val = dp(n+1, cap+1);
end

% Test and Compare
fprintf('Simple Recursive vs Dynamic Programming Comparison\n\n');

% Test parameters
fib_test = [10, 15, 20, 25, 30];  % Fibonacci numbers to compute
ks_test = [5, 10, 15, 20];        % Number of items for knapsack
```

```
% Initialize arrays to store timing results
fib_rec_time = zeros(size(fib_test));
fib_dp_time = zeros(size(fib_test));
ks_rec_time = zeros(size(ks_test));
ks_dp_time = zeros(size(ks_test));

% Fibonacci Tests
fprintf('FIBONACCI:\n');
fprintf('n\tRecursive\tDP\t\tSpeedup\n');
for i = 1:length(fib_test)
    n = fib_test(i);
    % Recursive timing
    tic; fib_rec(n); t1 = toc;
    % Dynamic Programming timing
    tic; fib_dp(n); t2 = toc;

    fib_rec_time(i) = t1;
    fib_dp_time(i) = t2;

    fprintf('%d\t%.4fs\t%.4fs\t%.0fx\n', n, t1, t2, t1/t2);
end

% Knapsack Tests
fprintf('\nKNAPSACK:\n');
fprintf('Items\tRecursive\tDP\t\tSpeedup\n');
for i = 1:length(ks_test)
    n = ks_test(i);
    % Generate random weights and values
    w = randi([1, 15], 1, n);   % Random weights between 1-15
    v = randi([5, 30], 1, n);   % Random values between 5-30
    cap = 30;                   % Fixed capacity

    % Recursive timing
    tic; ks_rec(w, v, cap, n); t1 = toc;
    % Dynamic Programming
    tic; ks_dp(w, v, cap); t2 = toc;

    ks_rec_time(i) = t1;
    ks_dp_time(i) = t2;

    fprintf('%d\t%.4fs\t%.4fs\t%.0fx\n', n, t1, t2, t1/t2);
end

% Fibonacci plot
figure;
plot(fib_test, fib_rec_time, 'ro-', 'LineWidth', 2); hold on;
plot(fib_test, fib_dp_time, 'bd-', 'LineWidth', 2);
xlabel('Fibonacci Number (n)');
ylabel('Time (seconds)');
title('Fibonacci: Recursive vs DP');
```

```
legend('Recursive', 'Dynamic Programming');
grid on;

% Knapsack plot
figure;
plot(ks_test, ks_rec_time, 'ro-', 'LineWidth', 2); hold on;
plot(ks_test, ks_dp_time, 'bd-', 'LineWidth', 2);
xlabel('Number of Items');
ylabel('Time (seconds)');
title('Knapsack: Recursive vs DP');
legend('Recursive', 'Dynamic Programming');
grid on;
%Comparison: Recursive(rec) vs Dynamic Programming(dp)
clear;
clc;
%Fibonacci Functions
% Recursive Fibonacci - exponential time complexity O(2^n)
% Repeatedly calculates same values multiple times
function result = fib_rec(n)
    if n <= 2
        result = 1;
    else
        result = fib_rec(n-1) + fib_rec(n-2);
    end
end

% Stores previously computed values to avoid redundant calculations
function result = fib_dp(n)
    if n <= 2
        result = 1;
        return;
    end
    fib = [1, 1, zeros(1, n-2)];  % Initialize array to store computed values
    for i = 3:n
        fib(i) = fib(i-1) + fib(i-2);  % Build solution bottom-up
    end
    result = fib(n);
end

% Knapsack Functions
% For each item, we have two choices: include or exclude
function max_val = ks_rec(w, v, cap, n)
    if n == 0 || cap == 0
        max_val = 0;
    elseif w(n) > cap
        max_val = ks_rec(w, v, cap, n-1);  % Cannot include current item
    else
        include = v(n) + ks_rec(w, v, cap-w(n), n-1);  % Include current item
        exclude = ks_rec(w, v, cap, n-1);              % Exclude current item
        max_val = max(include, exclude);               % Take better option
    end
```

```
end

% Uses 2D table to store solutions to subproblems
function max_val = ks_dp(w, v, cap)
    n = length(w);
    dp = zeros(n+1, cap+1);  % DP table: dp[i][j] = max value with first i items and capacity
j
    for i = 1:n
        for j = 0:cap
            if w(i) <= j
                % Max of excluding or including current item
                dp(i+1, j+1) = max(dp(i, j+1), v(i) + dp(i, j+1-w(i)));
            else
                % Cannot include current item
                dp(i+1, j+1) = dp(i, j+1);
            end
        end
    end
    max_val = dp(n+1, cap+1);
end

% Test and Compare
fprintf('Simple Recursive vs Dynamic Programming Comparison\n\n');

% Test parameters
fib_test = [10, 15, 20, 25, 30];  % Fibonacci numbers to compute
ks_test = [5, 10, 15, 20];        % Number of items for knapsack

% Initialize arrays to store timing results
fib_rec_time = zeros(size(fib_test));
fib_dp_time = zeros(size(fib_test));
ks_rec_time = zeros(size(ks_test));
ks_dp_time = zeros(size(ks_test));

% Fibonacci Tests
fprintf('FIBONACCI:\n');
fprintf('n\tRecursive\tDP\t\tSpeedup\n');
for i = 1:length(fib_test)
    n = fib_test(i);
    % Recursive timing
    tic; fib_rec(n); t1 = toc;
    % Dynamic Programming timing
    tic; fib_dp(n); t2 = toc;

    fib_rec_time(i) = t1;
    fib_dp_time(i) = t2;

    fprintf('%d\t%.4fs\t%.4fs\t%.0fx\n', n, t1, t2, t1/t2);
end

% Knapsack Tests
```

```
fprintf('\nKNAPSACK:\n');
fprintf('Items\tRecursive\tDP\t\tSpeedup\n');
for i = 1:length(ks_test)
    n = ks_test(i);
    % Generate random weights and values
    w = randi([1, 15], 1, n);   % Random weights between 1-15
    v = randi([5, 30], 1, n);   % Random values between 5-30
    cap = 30;                   % Fixed capacity

    % Recursive timing
    tic; ks_rec(w, v, cap, n); t1 = toc;
    % Dynamic Programming
    tic; ks_dp(w, v, cap); t2 = toc;

    ks_rec_time(i) = t1;
    ks_dp_time(i) = t2;

    fprintf('%d\t%.4fs\t%.4fs\t%.0fx\n', n, t1, t2, t1/t2);
end

% Fibonacci plot
figure;
plot(fib_test, fib_rec_time, 'ro-', 'LineWidth', 2); hold on;
plot(fib_test, fib_dp_time, 'bd-', 'LineWidth', 2);
xlabel('Fibonacci Number (n)');
ylabel('Time (seconds)');
title('Fibonacci: Recursive vs DP');
legend('Recursive', 'Dynamic Programming');
grid on;

% Knapsack plot
figure;
plot(ks_test, ks_rec_time, 'ro-', 'LineWidth', 2); hold on;
plot(ks_test, ks_dp_time, 'bd-', 'LineWidth', 2);
xlabel('Number of Items');
ylabel('Time (seconds)');
title('Knapsack: Recursive vs DP');
legend('Recursive', 'Dynamic Programming');
grid on;
```

Explanation of the code

1. **Purpose of the Code**

   This MATLAB program compares the performance of recursive and dynamic programming (DP) approaches using two classical problems: the Fibonacci sequence and the 0/1 Knapsack problem. It demonstrates how DP significantly improves efficiency by avoiding repeated computations.

2. **Initialization**

The code begins by clearing the workspace and command window using 'clear' and 'clc' commands. This ensures that the program starts with a clean environment each time it runs.

3. Fibonacci Functions

The code defines two Fibonacci functions: a recursive version and a dynamic programming version.

(a) Recursive Fibonacci

The recursive Fibonacci function uses the mathematical relation F(n) = F(n−1) + F(n−2), with base cases F(1) = 1 and F(2) = 1. This function repeatedly recalculates the same values, resulting in exponential time complexity O(2^n).

(b) Dynamic Programming Fibonacci

The DP Fibonacci function stores previously computed values in an array and builds the sequence iteratively. This avoids redundant calculations, resulting in linear time complexity O(n).

4. Knapsack Problem Functions

The 0/1 Knapsack problem is solved using both recursive and DP methods. The goal is to maximize total value while staying within a given capacity.

(a) Recursive Knapsack

The recursive function explores all combinations of including or excluding each item. If the current item exceeds the capacity, it is skipped. Otherwise, both options are tested, and the maximum value is chosen. This results in exponential time complexity O(2^n).

(b) Dynamic Programming Knapsack

The DP version builds a 2D table where dp(i, j) represents the maximum value achievable with the first i items and capacity j. It iteratively fills the table, achieving polynomial time complexity O(n × capacity).

5. Testing and Comparison

The program tests both approaches for different input sizes and measures execution time using 'tic' and 'toc'. It displays the recursive time, DP time, and the speedup ratio for each test case.

6. Plotting Results

Two plots are generated: one for Fibonacci and one for Knapsack. The x-axis represents input size, and the y-axis represents computation time in seconds. The plots clearly show that dynamic programming performs significantly faster than recursion.

Simple Recursive vs Dynamic Programming Comparison

FIBONACCI:

n	Recursive	DP	Speedup
10	0.0005s	0.0004s	1x
15	0.0003s	0.0002s	1x
20	0.0014s	0.0004s	3x
25	0.0116s	0.0004s	32x
30	0.1513s	0.0004s	383x

KNAPSACK:

Items	Recursive	DP	Speedup

5	0.0020s	0.0022s	1x
10	0.0004s	0.0002s	2x
15	0.0017s	0.0003s	5x
20	0.0276s	0.0004s	69x

>>

Figures/Plots

Figure 1

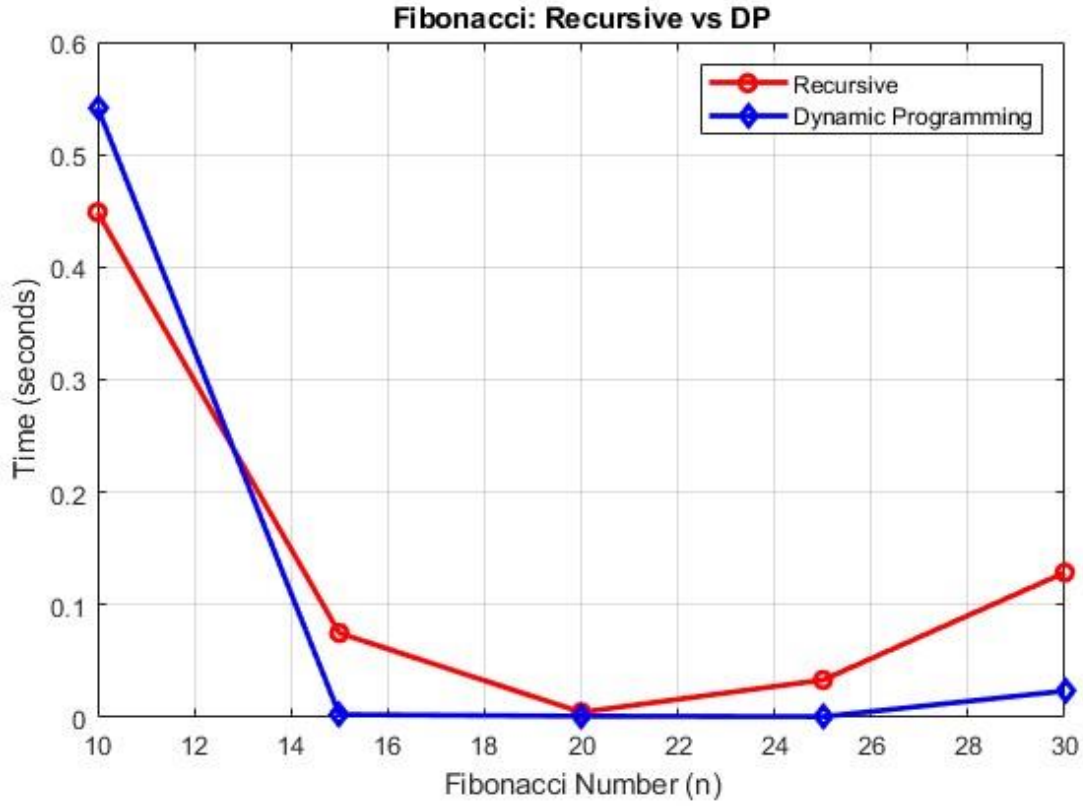


Figure 2

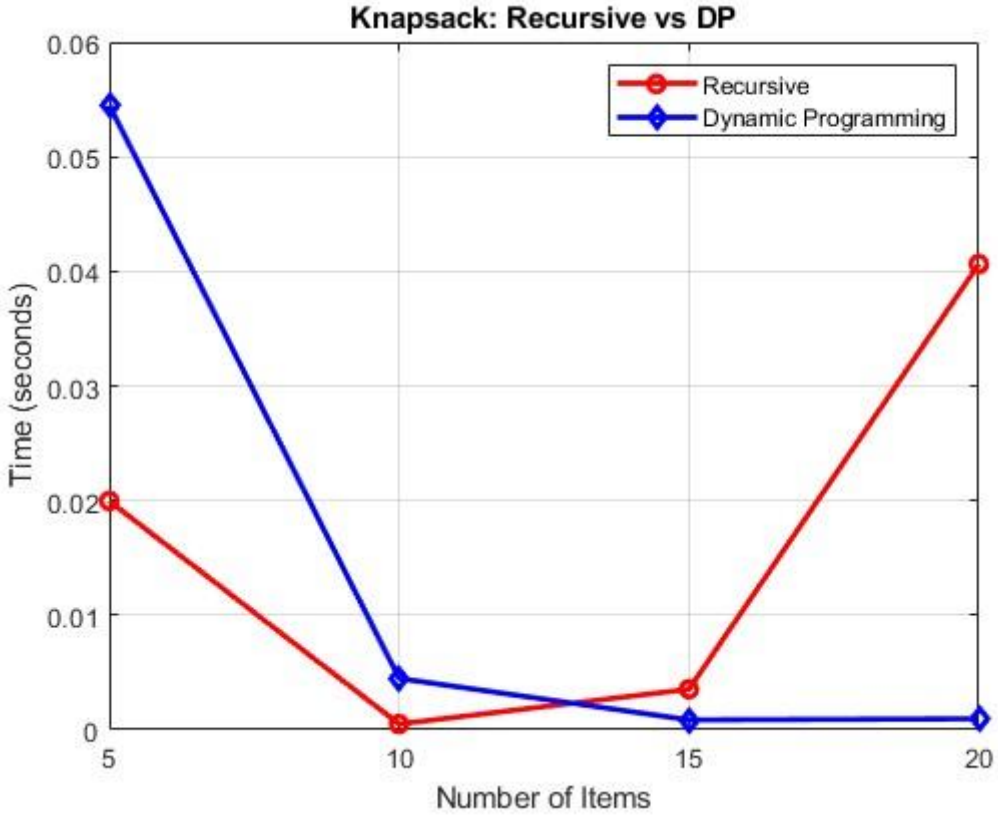


Figure 3

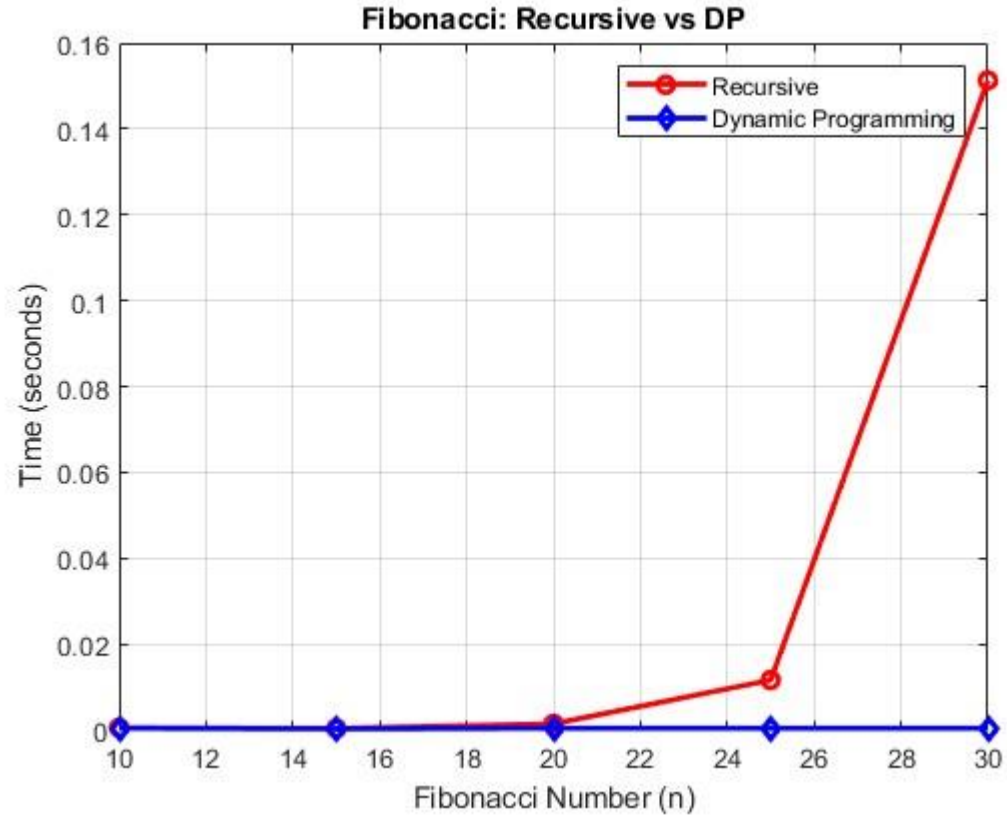
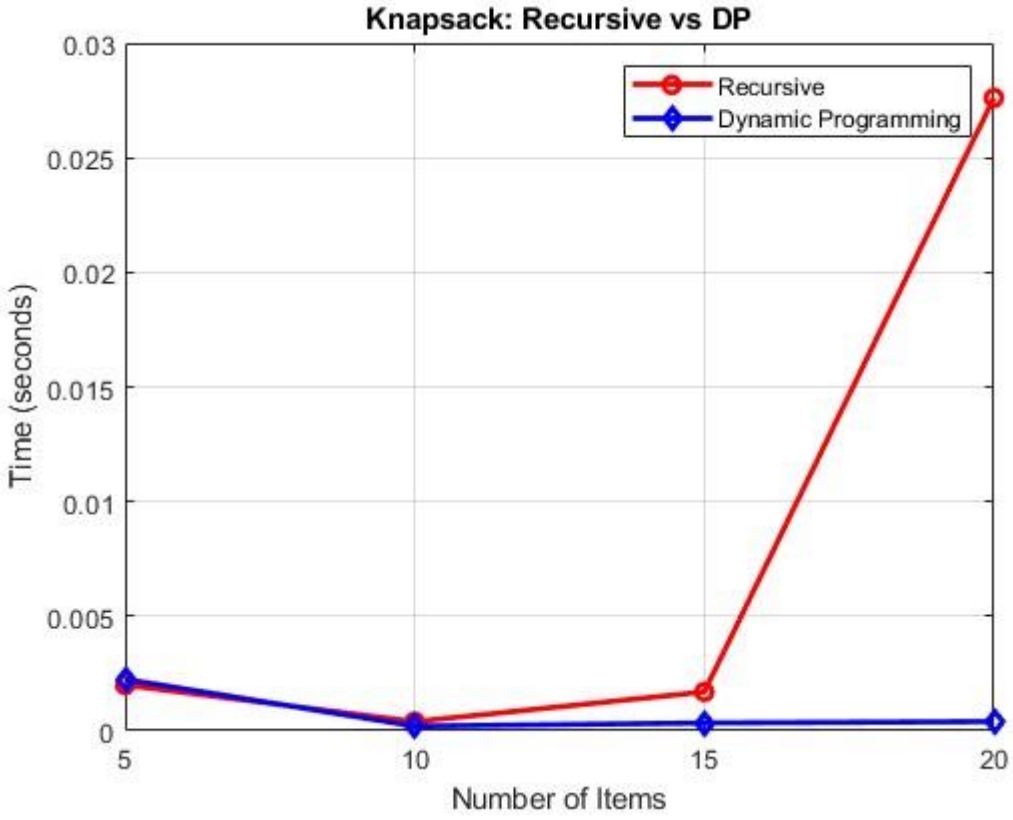

Fibonacci: Recursive vs DP
Recursive
Dynamic Programming
Time (seconds)
Fibonacci Number (n)
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
10
12
14
16
18
20
22
24
26
28
30

Figure 4


Knapsack: Recursive vs DP
Recursive
Dynamic Programming
Time (seconds)
Number of Items
0.03
0.025
0.02
0.015
0.01
0.005
0
5
10
15
20

CHAPTER 3

3.1 Conclusion and Learning Experience

This assignment has helped us to acquire knowledge and experience that helped us understand MATLAB programming concepts and gave us experience with the foundations we had acquired from Modules 5.1.

3.2 References and Resources

MATLAB Documentation - Used for syntax and function guidance on legend(), fprintf().
Computer programming lecture notes.
YouTube videos on Fibonacci sequence, recursive functions, dynamic programming and knapsack problems.