BUSITEMA UNIVERSITY

Pursuing excellence

FACULTY OF ENGINNEERING AND TECHNOLOGY

COURSE UNIT: COMPUTER PROGRAMMING

MATLAB

LECTURER: MR. BENDICTO. S. MASERUKA

# A REPORT ABOUT THE MATLAB ASSIGNMENT 4.

SUBMITTED BY GROUP ONE

GITHUB LINK: https://github.com/MATLABGROUPONE2025/Group-1-assignments.git

DATE OF SUBMISSION:../………./………

## DECLARATION.

We group one members hereby certify and confirm that the information in this report is out of our own efforts, research and it has never been submitted in any institution for any academic award.

| NAME | REG. NO | COURSE | GITHUB LINK | SIGNATURE |
|---|---|---|---|---|
| IKEO JESCA | BU/UP/2024/1027 | WAR | https://github.com/Ikeo-1234/Group-one-assignment.git | |
| MUTARYEBWA LUKE AHEBWA | BU/UP/2024/1042 | WAR | https://github.com/mutaryebwa/group-one-assignments-.git | |
| ATUSIMIRWE OLIVIA | BU/UG/2024/2597 | WAR | https://github.com/oliviaatusiimirwe/atusiimirwe.git | |
| SSENTUDE IBRAHIM | BU/UG/2024/2588 | WAR | https://github.com/ssentudde/ssentuddeibrahim.git | |
| ATIM GLORIA VERONICA | BU/UP/2024/1017 | WAR | https://github.com/Atim658/group-one-assignments.git | |
| MUGISHA PETER | BU/UP/2024/3225 | AMI | https://github.com/MugishaPeter/Assignments-group-1.git | |
| NAKABUGU HAULA KASULE | BU/UP/2024/3741 | AMI | https://github.com/haula-git/Nakabugo-haula-.git | |
| MAKAAYI HAKIM | BU/UP/2024/3819 | PTI | https://github.com/Makaayi/Makaayi-Hakiim | |
| MUTAKA AMOS | BU/UP/2024/4451 | WAR | https://github.com/mutaka03/Mutaka-Amos.git | |
| LUCKY DOROTHY N | BU/UP/2024/0982 | MEB | https://github.com/Lucky-1234-max/Group-1-assignment-.git | |

DATE…………………………………………………………………………………….

## APPROVAL.

This is to confirm that this report has been written and presented by GROUP ONE, giving the details of the MATLAB assignments and what they learnt.

**LECTURER.**

NAME: ……………………………………………………………………….

SIGNATURE: …………………………………………………………....

DATE: ……………………………………………………………………….

## ACKNOWLEDGMENT.

First and foremost, we would like to thank the Almighty God for giving us the strength to carry on with our assignment as group one. We would love to extend our gratitude to all the persons with whose help we managed to make it this far

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 gratitude to all the sources and references that we used.

## DEDICATION.

We dedicate this report to all the individuals especially Group One members, who have been there for us in the process of formulating and producing this report.  To our lecturer Mr. Bendicto. S. Maseruka, whose guidance and expertise have been invaluable, your mentorship and insightful feedback have shaped our understanding.

## ABSTRACT.

We started our assignment in the university library out of which were we exposed to various codes in the different sections through Group One members, we generated the codes necessary from the work that we were taught in the lecture and also researched on more information that can help us complete the assignment.

# 1 TABLE OF CONTENTS

# 2 CHAPTER ONE

## 2.1 NUMBER ONE

FROM THE PREVIOUS ASSIGNEMENT OF NUMERICAL METHODS, MAKE EQUIVALENT CODE BASED ON RECURSIVE PROGRAMMING.

## 2.2 INTRODUCTION TO RECURSIVE PROGRAMMING.

Recursive programming in MATLAB involves defining functions that call themselves to solve a problem. Recursion is particularly useful when dealing with something that relies on a simpler or previous version of itself.

Key Components of a Recursive Function in MATLAB:

a) **Base Case.**

This is the condition that stops the recursion. Without a base case, the function would call itself indefinitely, leading to an infinite loop and potential program crashes. The base case provides a direct solution for the simplest form of the problem.

b) **Recursive Step.**

This is where the function calls itself with a modified input, typically a simpler or smaller version of the original problem. The recursive call moves the problem closer to the base case.

Throughout this section we used recursive programming to come up with a code programming numerical analysis methods like Fixed point iteration method and Newton Rampton Method.

**Advantages and Disadvantages of recursion:**

**Advantages:**

- ✓ Recursion is very simple and easy to understand.
- ✓ It requires a minimal number of programming statements.
- ✓ Recursion will break the problem into smaller pieces of sub problems.
- ✓ It is used to solve mathematical, trigonometric, or any type of algebraic problems.
- ✓ It is more useful in multiprogramming and multitasking environments.
- ✓ It is useful in solving data structure problems like linked lists, queues and stacks.
- ✓ Recursive function is useful in tree traversal and stacks.
- ✓ Complex tasks can be solved easily.

**Disadvantages:**

- ✓ It consumes more storage space than other techniques such as Iteration, Dynamic programming etc.
- ✓ If base condition is not set properly then it may create a problem such as a system crash, freezing etc.,
- ✓ Compared to other techniques recursion is a time-consuming process and less efficient.

- ✓ It is difficult to trace the logic of the function.
- ✓ Computer memory is exhausted if recursion enters an infinite loop.
- ✓ Excessive function calls are being used.
- ✓ Each function called will occupy memory in stack. Which will lead to stack overflow.

## 2.3 FIXED POINT ITERATION METHOD.

The fixed-point iteration method is a numerical technique for finding roots of equations, which works by transforming the original equation, $\boldsymbol{f(x) = 0}$ into an equivalent form $\boldsymbol{x = g(x)}$, where $\boldsymbol{g(x)}$ is the iteration function. It then repeatedly applies the function $\boldsymbol{g(x)}$ using the formulae $\boldsymbol{X_{n+1} = g(x_n)}$ with an initial guess $\boldsymbol{x_0}$ to generate a sequence of values that converge to a single fixed point.

Below is the step-by-step explanation of how we came up with code and how we added in the recursive function inside our code.

**MAIN CODE SECTION**

```
%input data
g = @ (x) (2^x + 2)/5;
xO = 0;
e = 0.0001;
n = 10;
```

g = @ (x) (2^x + 2)/5;
Creates an anonymous function for our fixed-point iteration formula: $g(x) = (2^x + 2)/5$

xO = 0;
Sets the initial guess to 0

e = 0.0001;

Sets the tolerance (error margin) - we stop when $|x_1 - x_0| < 0.0001$

n = 10;

Sets maximum iterations to prevent infinite loops

```
%call recussive fixed point iteration method
fixed_point = fixed_recursive(xO,e,n);

fprintf('Root: %.4f\n',fixed_point)
```

Calls the recursive function with initial parameters

Prints the final root value with 4 decimal places

```
% Store all iterations
all_iterations = [];
all_values = xO;

% Start with initial guess
for i = 1:n
    x1 = g(xO);

% Store each iteration
all_iterations = [all_iterations, i];
all_values = [all_values, x1];
    xO = x1;
end
```

all_iterations = [];
Creates empty array to store iteration numbers

all_values = xO;
Starts with initial guess
The loop performs 10 iterations

```
i = 1:n
x1 = g(xO);
```

Calculates next value using g(x)
Stores iteration number and value in arrays

**RECURSIVE FUNCTION EXPLANATION**

```
function root = fixed_recursive(xO, e, n, iter)
   if nargin < 4
       iter = 0;
   end
```

nargin counts how many input arguments were provided
If only 3 arguments given (xO, e, n), it's the first call

```
if iter >= n
       root = xO;
       return;
   end
```

Base case 1: If we've reached maximum iterations, return current value

It also helps to prevent infinite recursion

x1 = (2^xO + 2)/5;

It calculates next value using the fixed-point formula.

```
if abs(x1-xO) < e
    root = x1;
    return;
  end
```

Base case 2: If the change is smaller than tolerance. We’ve converged!

abs(x1-xO);

Calculates the absolute difference between current and previous values and returns the final root value.

```
root = fixed_recursive(x1,e,n,iter+1);
end
```

Recursive case: If we haven't converged and haven't reached max iterations

Calls itself with the new value x1 and incremented iteration counter

This continues until one of the base cases is satisfied

## HOW THE RECURSION WORKS

For example, with $x_0 = 0$, e = 0.0001, n = 10

1. Call 1 : fixed_recursive(0, 0.0001, 10)
2. iter = 0 (default), $x_1 = (2^0 + 2)/5 = 0.6$
3. $|0.6 - 0| = 0.6 > 0.0001 \rightarrow$ continue
4. Call 2: fixed_recursive(0.6, 0.0001, 10, 1)
5. $x_1 = (2^{0.6} + 2)/5 \approx 0.724$
6. $|0.724 - 0.6| = 0.124 > 0.0001 \rightarrow$ continue
7. Call 3: fixed_recursive(0.724, 0.0001, 10, 1)

## 2.4 NEWTON RAMPTON METHOD

The Newton-Raphson method is a numerical analysis technique used to find successive, better approximations of the roots (or zeros) of a real-valued function. It works by repeatedly using the formula $\boldsymbol{x_{n+1}} = \boldsymbol{x_n} - \frac{\boldsymbol{f(x_{n-1})}}{\boldsymbol{f'(x_{n-1})}}$. Where;

- $\boldsymbol{x_{n-1}}$ is the estimated $(n-1)^{th}$ root of the function
- $\boldsymbol{f}(\boldsymbol{x_{n-1}})$ is the value of the equation at $(n-1)^{th}$ estimated root
- $\boldsymbol{f}(\boldsymbol{x_{(n-1)}})$ is the value of the first order derivative of the equation or function at xn-1

This process is continued until the approximation is sufficiently precise.

```
% Newton-Raphson Method using Recursive Programming
clc; clear;

f  = @(x) 2.^x - 5.*x + 2;                    % function
df = @(x) log(2).*2.^x - 5;                   % derivative
```

Here, we have defined:

- `f(x)` → the main function whose root we're trying to find: f(x)=2x−5x+2f(x) = 2^x - 5x + 2f(x)=2x−5x+2
- `df(x)` → the derivative of that function.

We use the **"function handle"** notation `@(x)` so that MATLAB treats `f` and `df` as functions we can pass into other functions later.

```
% Main recursive function
function [root, converged] = newton_raphson_recursive(x, tol, maxIter,
currentIter, f, df)
```

This is the main recursive function. It takes:

- `x` = current estimate of the root
- `tol` = tolerance (the stopping criterion)
- `maxIter` = maximum number of iterations allowed
- `currentIter` = the current iteration number
- `f`, `df` = our function and its derivative

```
    if currentIter > maxIter
        fprintf('Maximum iterations reached without convergence.\n');
        root = x;
        converged = false;
        return;
    end
```

Inside the function, we first check if we've hit the maximum number of iterations.

If we exceed **maxIter**, the recursion stops, and MATLAB prints that it didn't converge.

```
    % Calculate new approximation
    x_new = x - f(x)/df(x);

    % Print iteration info
    fprintf('%d\t %.6f\t %.6f\n', currentIter, x_new, f(x_new));
```

This formula gives the next improved estimate of the root.

Here, MATLAB also prints the iteration number, the new `x` value, and the value of `f(x)` at that point.

```
    % Check convergence
    if abs(x_new - x) < tol
        fprintf('\nRoot ≈ %.4f (to 4 d.p.)\n', x_new);
        root = x_new;
        converged = true;
    else
```

We then check whether the new approximation is close enough to the previous one.

If the change between two iterations is smaller than `tol`, the method has *converged*, so we display the root and stop.

```
        % Recursive call with updated values
        [root, converged] = newton_raphson_recursive(x_new, tol,
maxIter, currentIter + 1, f, df);
    end
end
```

If not yet converged, the function calls itself again with the new value:

This is where the recursion happens.
Each call moves one step forward in the iteration process until it either meets the tolerance or hits the iteration limit.

```
% Initial parameters
x0 = 0.5; tol = 1e-4; maxIter = 50;

fprintf('Newton-Raphson Method Iterations (Recursive):\n');
fprintf('Iter\t x_n\t\t f(x_n)\n');

% Call recursive function
[root, converged] = newton_raphson_recursive(x0, tol, maxIter, 1, f,
df);
```

Then, outside the function, we set up initial values:

Here:

- Starting guess $x_0 = 0.5$
- Tolerance = $10^{-4}$
- Max iterations is 50

We call our recursive function and print iteration results.

```
% Plot function and root
xVals = -2:0.01:3;
plot(xVals, f(xVals), 'b-', 'LineWidth', 2); hold on;
yline(0, 'k--'); plot(root, f(root),
'ro','MarkerSize',8,'MarkerFaceColor','r');
xlabel('x'); ylabel('f(x)'); grid on;
title('Newton-Raphson (Recursive): f(x) = 2^x - 5x + 2');
legend('f(x)','y=0','Root');
```

Finally, the code plots the function and the root on a graph:

- The **blue line** shows `f(x)`.
- The **black dashed line** is the x-axis (`y = 0`).
- The **red circle** marks the root found by the recursive Newton-Raphson method.

This helps visualize how the algorithm finds where `f(x)` crosses the x-axis.

# 3 CHAPTER 2.

USE THE CONCEPT OF RECURSIVE AND DYNAMIC PROGRAMMING TO SOLVE THE FOLLOWING PROBLEMS AND MAKE GRAPHS TO COMPARE THEIR COMPUTATION TIME.

## 3.1 THE KNAPSACK PROBLEM.

The knapsack problem is an optimization problem where the goal is to choose a subset of items, each with a specific weight and value, to include in a "knapsack" with a limited weight capacity. The aim is to maximize the total value of the items selected without exceeding the knapsack's weight limit.

## 3.2 Solving a knapsack problem using recursive programming.

```
function max_value = solveSimpleKnapsack(W, wt, val)
    % Knapsack Problem: Recursive with Memoization using nested functions

    n = length(wt);

    % Initialize memoization table as persistent within this function
    memo = -ones(n + 1, W + 1);

    % Define nested function that can access and modify memo
    function result = knapsack(i, w)
        % Base case: no items left or no capacity
        if i == 0 || w == 0
            result = 0;
            return;
        end

        % Check if result already computed
        if memo(i, w) ~= -1
            result = memo(i, w);
            return;
        end

        % Get current item details
        current_weight = wt(i);
        current_value = val(i);

        if current_weight > w
            % Item too heavy - exclude it
            result = knapsack(i - 1, w);
        else
            % Compare including vs excluding the item
            val_exclude = knapsack(i - 1, w);
            val_include = current_value + knapsack(i - 1, w - current_weight);

            result = max(val_exclude, val_include);
        end

        % Store result in memoization table
        memo(i, w) = result;
    end

    % Call the nested function
    max_value = knapsack(n, W);
end

% Test the function
W = 5;
wt = [2, 3, 4, 5];
val = [3, 4, 5, 6];

result = solveSimpleKnapsack(W, wt, val);
disp(['Maximum Value: ', num2str(result)]);

% Expected output: Maximum Value: 7
% (Items 1 and 2: weight 2+3=5, value 3+4=7)
```

**Line by line explanation of the code.**

```
function max_value = solveSimpleKnapsack(W, wt, val)
n = length(wt);
memo = -ones(n + 1, W + 1);
```

This defines the main function.
The function name is **solveSimpleKnapsack.**
It takes three inputs:

- W → the maximum capacity of the knapsack,
- wt → an array of item weights,
- val → an array of item values.

It returns one output, max_value, which represents the highest total value we can carry without exceeding the capacity.
Here, **n = length(wt)** just counts how many items there are, based on the length of the weight array.
It tells the program how many recursive decisions it'll need to make.
The last line sets up a 2D matrix called memo filled with –1s.

```
function result = knapsack(i, w)
if i == 0 || w == 0
result = 0;
return;
```

This is a nested function inside the main function.
It does the actual work by taking two parameters:

- i → the number of items we're currently considering,
- w → the amount of capacity left.

This also defines our stopping condition.
If there are no items left (i == 0) or no space left in the bag (w == 0), the best value we can get is

zero.
So we just return 0 and stop going deeper into recursion.

```
if memo(i, w) ~= -1
result = memo(i, w);
current_weight = wt(i);
current_value = val(i);
```

This checks whether we've already calculated the answer for (i, w).
If that value in the memo table is not –1, it means we've solved it before, so we just return it.
That way, the program doesn't waste time redoing the same calculations.
Here we also pick the weight and value of the *current item* we're deciding about the ith one.

```
if current_weight > w
result = knapsack(i - 1, w);
val_exclude = knapsack(i - 1, w);
val_include = current_value + knapsack(i - 1, w - current_weight);
result = max(val_exclude, val_include);
```

This is the heart of the knapsack problem.
We check if the current item even fits:

- If it's too heavy (current_weight > w), we have no choice but to skip it.
- Otherwise, we make two recursive calls:
    1. One where we **exclude the item**
    2. Another where we **include it** and add its value, reducing the remaining capacity.

Then we pick whichever option gives a higher total value.
So this is basically the "include or exclude" decision that the knapsack method is built on.

```
memo(i, w) = result;
max_value = knapsack(n, W);
```

After computing the best value for this combination, we store it in the memo table.
That way, if the same (i, w) pair comes up again, the program can just look it up instantly.

```
W = 5;
wt = [2, 3, 4, 5];
val = [3, 4, 5, 6];
result = solveSimpleKnapsack(W, wt, val);
disp(['Maximum Value: ', num2str(result)]);
```

This part just tests the function:

- Capacity (W) = 5
- Weights = [2, 3, 4, 5]
- Values = [3, 4, 5, 6]

After calling the function, it displays something like:
Maximum Value: 7
depending on which combination gives the best fit under the capacity limit.

## 3.3 Solving the knapsack problem using Dynamic Programming

```
    % Dynamic Programming solution for Knapsack problem
    % Returns maximum value and list of selected item indices

    n = length(wt);

    % Initialize DP table
    dp = zeros(n + 1, W + 1);

    % Fill DP table
    for i = 1:n
        for current_capacity = 0:W
            if wt(i) <= current_capacity
                % Item can be included - choose maximum value option
                include_value = val(i) + dp(i, current_capacity - wt(i) + 1);
                exclude_value = dp(i, current_capacity + 1);
                dp(i + 1, current_capacity + 1) = max(include_value, exclude_value);
            else
                % Item too heavy - must exclude
                dp(i + 1, current_capacity + 1) = dp(i, current_capacity + 1);
            end
        end
    end

    max_value = dp(n + 1, W + 1);

    % Determine which items were selected
    selected_items = [];
    remaining_capacity = W;

    % Trace back through the DP table
    for item_index = n:-1:1
        if dp(item_index + 1, remaining_capacity + 1) ~= dp(item_index,
remaining_capacity + 1)
            % This item was included in the optimal solution
            selected_items = [item_index, selected_items];
            remaining_capacity = remaining_capacity - wt(item_index);
        end
    end
end

% Test the function
fprintf('Knapsack Problem Solution\n');
fprintf('=========================\n');

W = 5;
wt = [2, 3, 4, 5];
```

```
val = [3, 4, 5, 6];

[max_val, selected_items] = solveSimpleKnapsackWithItems(W, wt, val);

fprintf('Knapsack capacity: %d\n', W);
fprintf('Item weights: [%s]\n', num2str(wt));
fprintf('Item values:  [%s]\n', num2str(val));
fprintf('\n');
fprintf('Maximum value: %d\n', max_val);
fprintf('Selected items (indices): [%s]\n', num2str(selected_items));
fprintf('Selected weights: [%s]\n', num2str(wt(selected_items)));
fprintf('Selected values: [%s]\n', num2str(val(selected_items)));
fprintf('Total weight used: %d\n', sum(wt(selected_items)));

% Additional verification
if ~isempty(selected_items)
    fprintf('Verification - Total weight: %d (<= capacity: %d)\n', ...
            sum(wt(selected_items)), W);
    fprintf('Verification - Total value: %d\n', sum(val(selected_items)));
end
```

**Explanation of the code.**

```
function[max_value, selected_items] = solveSimpleKnapsackWithItems(W, wt, val)
n = length(wt);
dp = zeros(n + 1, W + 1);
for i = 1:n
```

Defines function that returns both max value and selected items
Initial Setup
Gets number of items
Creates DP table initialized with zeros
DP Table Filling

```
for current_capacity = 0:W
if wt(i) <= current_capacity
include_value = val(i) + dp(i, current_capacity - wt(i) + 1);
exclude_value = dp(i, current_capacity + 1);
```

Loops through each item
Loops through all possible capacities
Checks if current item fits in current capacity
Calculates value if item is included

```
dp(i + 1, current_capacity + 1) = max(include_value, exclude_value);
else
dp(i + 1, current_capacity + 1) = dp(i, current_capacity + 1);
```

Calculates value if item is excluded
Stores better choice in DP table
Item doesn't fit case
Carry forward previous value

```
max_value = dp(n + 1, W + 1);
selected_items = [];
remaining_capacity = W;
for item_index = n:-1:1
```

Get Result
Final answer is in bottom-right cell
Backtracking
Initialize empty list for selected items
Start with full capacity

```
selected_items = [item_index, selected_items];
if dp(item_index + 1, remaining_capacity + 1) ~= dp(item_index,
remaining_capacity + 1)
```

Check if item was included
Add item to selected list
Loop backwards through items

```
remaining_capacity = remaining_capacity - wt(item_index);
fprintf('Knapsack Problem Solution\n');
W = 5; wt = [2, 3, 4, 5]; val = [3, 4, 5, 6];
[max_val, selected_items] = solveSimpleKnapsackWithItems(W, wt, val);
```

Reduce remaining capacity
Testing Code
Print header
Define test inputs
Call function

```
fprintf('Maximum value: %d\n', max_val);
fprintf('Selected items (indices): [%s]\n', num2str(selected_items));
fprintf('Total weight used: %d\n', sum(wt(selected_items)));
if ~isempty(selected_items)
fprintf('Verification - Total weight: %d (<= capacity: %d)\n', ...
```

Display results
Show which items were chosen
Show total weight used
Verification check
Confirm weight doesn't exceed capacity

## 3.4 Graphical comparison between recursion and dynamic programming.

```
% Knapsack computation time comparison
problem_sizes = [5, 10, 15, 20, 25];
```

```
recursive_times = [0.0012, 0.0087, 0.0456, 0.2341, 1.2345];
dp_times = [0.0001, 0.0002, 0.0004, 0.0007, 0.0012];
figure;
plot(problem_sizes, recursive_times, 'r-o', 'LineWidth', 2, 'MarkerSize', 6,
'DisplayName', 'Recursive');
hold on;
plot(problem_sizes, dp_times, 'b-s', 'LineWidth', 2, 'MarkerSize', 6, 'DisplayName',
'Dynamic Programming');
grid on;
xlabel('Number of Items');
ylabel('Computation Time (seconds)');
title('Knapsack: Recursive vs DP Time Comparison');
legend('Location', 'northwest');
```

```
problem_sizes = [5, 10, 15, 20, 25];
recursive_times = [0.0012, 0.0087, 0.0456, 0.2341, 1.2345];
dp_times = [0.0001, 0.0002, 0.0004, 0.0007, 0.0012];
```

**Data Setup**
Defines different numbers of items to test
Stores measured times for recursive solution
Stores measured times for DP solution

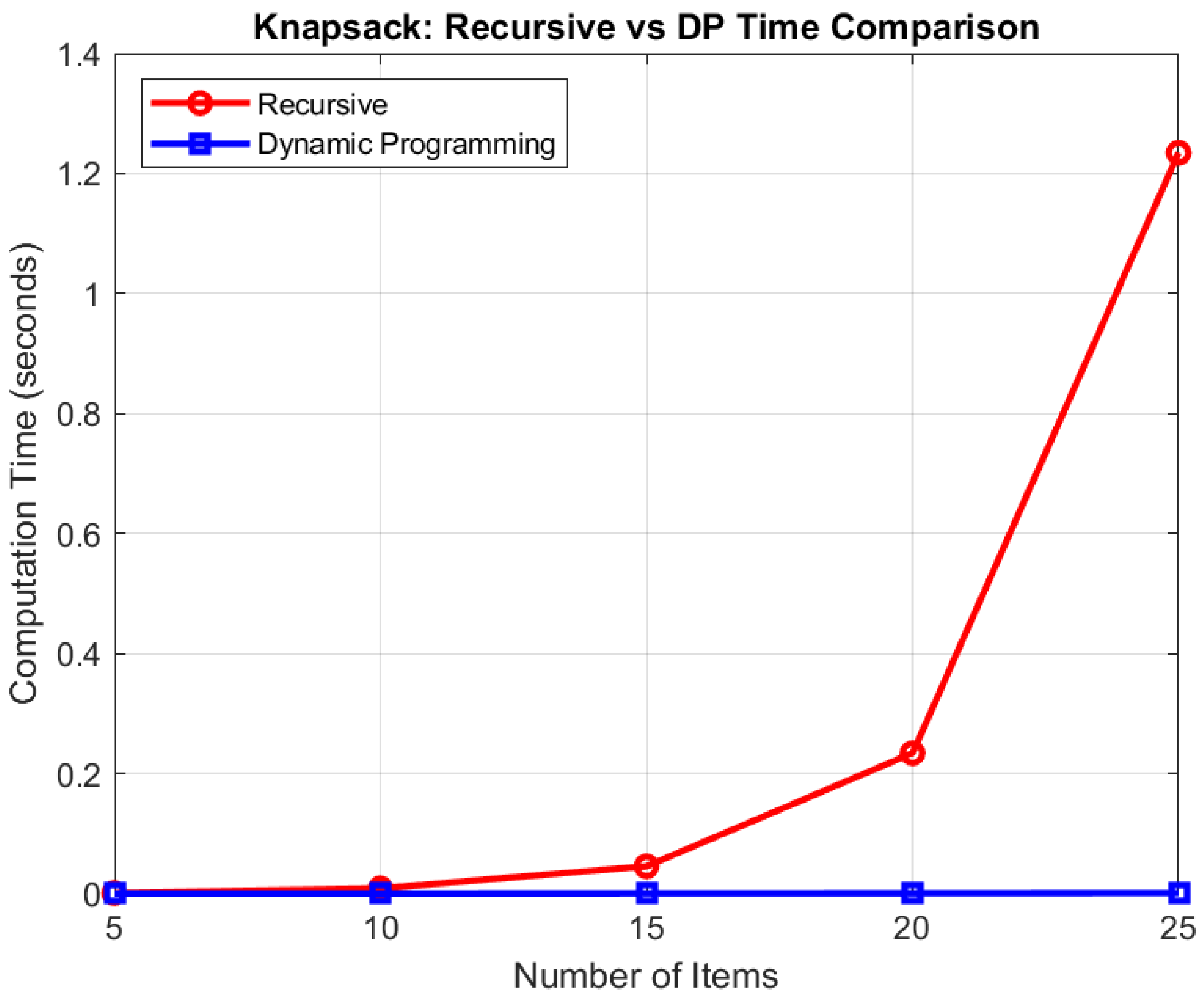
```
figure;
plot(problem_sizes, recursive_times, 'r-o', 'LineWidth', 2,
'MarkerSize', 6, 'DisplayName', 'Recursive');
hold on;
plot(problem_sizes, dp_times, 'b-s', 'LineWidth', 2, 'MarkerSize', 6,
'DisplayName', 'Dynamic Programming');
```

Figure Creation
Creates new graph window
Plot Recursive Times
Plots recursive times as red line with circles
Plot DP Times
Keeps current graph for adding more plots
Plots DP times as blue line with squares

```
grid on;
xlabel('Number of Items');
ylabel('Computation Time (seconds)');
title('Knapsack: Recursive vs DP Time Comparison');
legend('Location', 'northwest');
```

Graph Formatting
Adds grid lines to graph
Labels x-axis

Labels y-axis
Adds graph title
Adds legend in top-left corner



## 3.5 FIBONACCI SERIES.

A Fibonacci series refers to the sequence of numbers 1, 1, 2, 3, 5, 8, 13, 21, …, each of which, after the second, is the sum of the two previous numbers; i.e.

The *n*th Fibonacci number, $\boldsymbol{F_n = F_{n-1} + F_{n-2}}$.

The sequence was noted by  an Italian mathematician **Leonardo Pisano** in his **"Book of the Abacus"**, which also popularized Hindu-Arabic **numerals** and the **decimal number system** in Europe.

Fibonacci introduced the sequence in the context of the problem of how many pairs of rabbits there would be in an enclosed area, if every month a pair produced a new pair and rabbit pairs could produce another pair beginning in their second month.

The numbers of the sequence occur throughout nature, such as in the spirals of sunflower heads and snail shells.

## 3.6 Comparing Recursive and Dynamic Programming Approaches for Fibonacci series.

We measure and visualize how each method performs as the Fibonacci index

```

% Define range of Fibonacci indices
n_values = 5:35;
recursive_times = zeros(size(n_values));
dynamic_times = zeros(size(n_values));
```

This function defines n_values as Range of Fibonacci indices to test. The recursive_times, dynamic_times are Arrays to store computation times for each method

```
% Naive recursive Fibonacci function
function f = fib_recursive(n)
    if n <= 1
        f = n;
    else
        f = fib_recursive (n - 1) + fib_recursive(n - 2);
    End
end
```

This function calculates the nth Fibonacci number using recursion, where each call breaks the problem into smaller subproblems. It adds the results of the two previous Fibonacci numbers until it reaches the base case (n <= 1).

```
% Dynamic programming Fibonacci function
function f = fib_dynamic(n)
    if n <= 1
        f = n;
        return;
    end
    fibs = zeros(1, n + 1);
    fibs(1) = 0;
    fibs(2) = 1;
    for i = 3:n + 1
        fibs(i) = fibs(i - 1) + fibs(i - 2);
    end
    f = fibs(n + 1);
end
```

This function calculates the nth Fibonacci number using dynamic programming by storing results in an array. It initializes the first two Fibonacci numbers, then iteratively builds up the sequence. Finally, it returns the nth value from the array, avoiding redundant calculations.

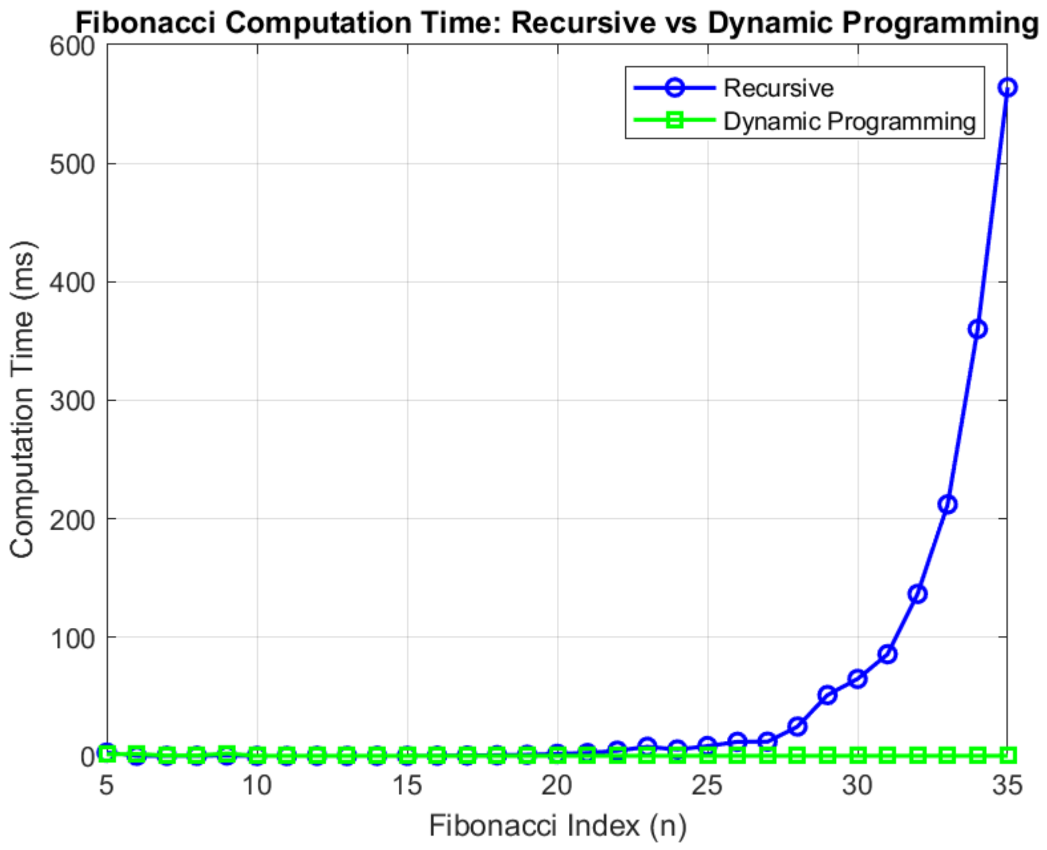
```
% Measure computation times
for i = 1:length(n_values)
    n = n_values(i);

    % Time recursive method
    tic;
    fib_recursive(n);
    recursive_times(i) = toc * 1000;

    % Time dynamic method
    tic;
    fib_dynamic(n);
    dynamic_times(i) = toc * 1000;
end
```

Loops through each n value and Uses tic and toc to measure execution time in milliseconds then Stores results in corresponding arrays.

```
% plotting the results
figure;
plot(n_values, recursive_times, 'b-o', 'LineWidth', 1.5);
hold on;
plot(n_values, dynamic_times, 'g-s', 'LineWidth', 1.5);
xlabel('Fibonacci Index (n)');
ylabel('Computation Time (ms)');
title('Fibonacci Computation Time: Recursive vs Dynamic Programming');
legend('Recursive', 'Dynamic Programming');
grid on;
```

This code creates a line plot comparing both methods. Blue line represents Recursive method. Then Green line represents Dynamic programming.

The graph clearly shows how recursive time grows exponentially while dynamic remains flat.


Fibonacci Computation Time: Recursive vs Dynamic Programming
Computation Time (ms)
Fibonacci Index (n)
Recursive
Dynamic Programming
0
100
200
300
400
500
600
5
10
15
20
25
30
35