MATLAB ASSIGNMENT FOUR

GROUP ONE

FACULTY OF ENGINEERING AND TECHNOLOGY

BUSITEMA UNIVERSITY

BY ; ENG. BENEDICTO MASERUKA

NUMBER ONE

FROM THE PREVIOUS ASSIGNEMENT OF NUMERICAL METHODS, MAKE EQUIVALENT CODE E

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

continuation

```
% 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 SECTION

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
```

continuation

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 RECURSIVE PROGRAMMING WORKS

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)

NEWTON –RAPHSON METHOD USING RECURSIVE PROGRAMMING

```
% Newton-Raphson Method using Recursive Programming
% function
f = @(x) 2.^x - 5.*x + 2;
% derivative
df = @(x) log(2).*2.^x - 5;
% Main recursive function
function [root, converged] = newton_raphson_recursive(x, tol, maxIter, currentIter, f, df)
   if currentIter > maxIter
      fprintf('Maximum iterations reached without convergence.\n');
      root = x;
      converged = false;
      return;
   end
```

CONTINUATION OF CODE

```
% 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));

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

CONTINUATION OF CODE

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

% 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);
```

PLOTTING THE FUNCTION AND ROOT

```
% 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');
```

Key changes made for recursive implementation:

1. **Recursive Function**: Created newton_raphson_recursive that calls itself until convergence or max iterations
2. **Parameters**: The recursive function takes:
   - x: Current approximation

CONTINUATION

- tol: Tolerance for convergence
- maxIter: Maximum allowed iterations
- currentIter: Current iteration counter
- f, df: Function and its derivative

3. **Base Cases**:
   - **Convergence**: When abs(x_new - x) < tol
   - **Max Iterations**: When currentIter > maxIter
4. **Recursive Call**: If not converged, calls itself with:
   - Updated x_new
   - Incremented currentIter
   - Same tol, maxIter, f, df
5. **Return Values**: Returns both the root and a converged flag

Advantages of this recursive approach:

CONTINUATION

- More elegant mathematical representation
- Clear separation of base cases and recursive step
- No explicit loop variables to manage
- Easier to extend with additional termination conditions

The output and results will be identical to the iterative version, but the implementation follows recursive programming principles.

RECURSIVE AND DYNAMIC PROGRAMMING COMPARISON

Comparing Recursive and Dynamic Programming Approaches for Fibonacci Computation

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).

Defining the Fibonacci function

```
% 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.

CONTINUATION

```
% 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.

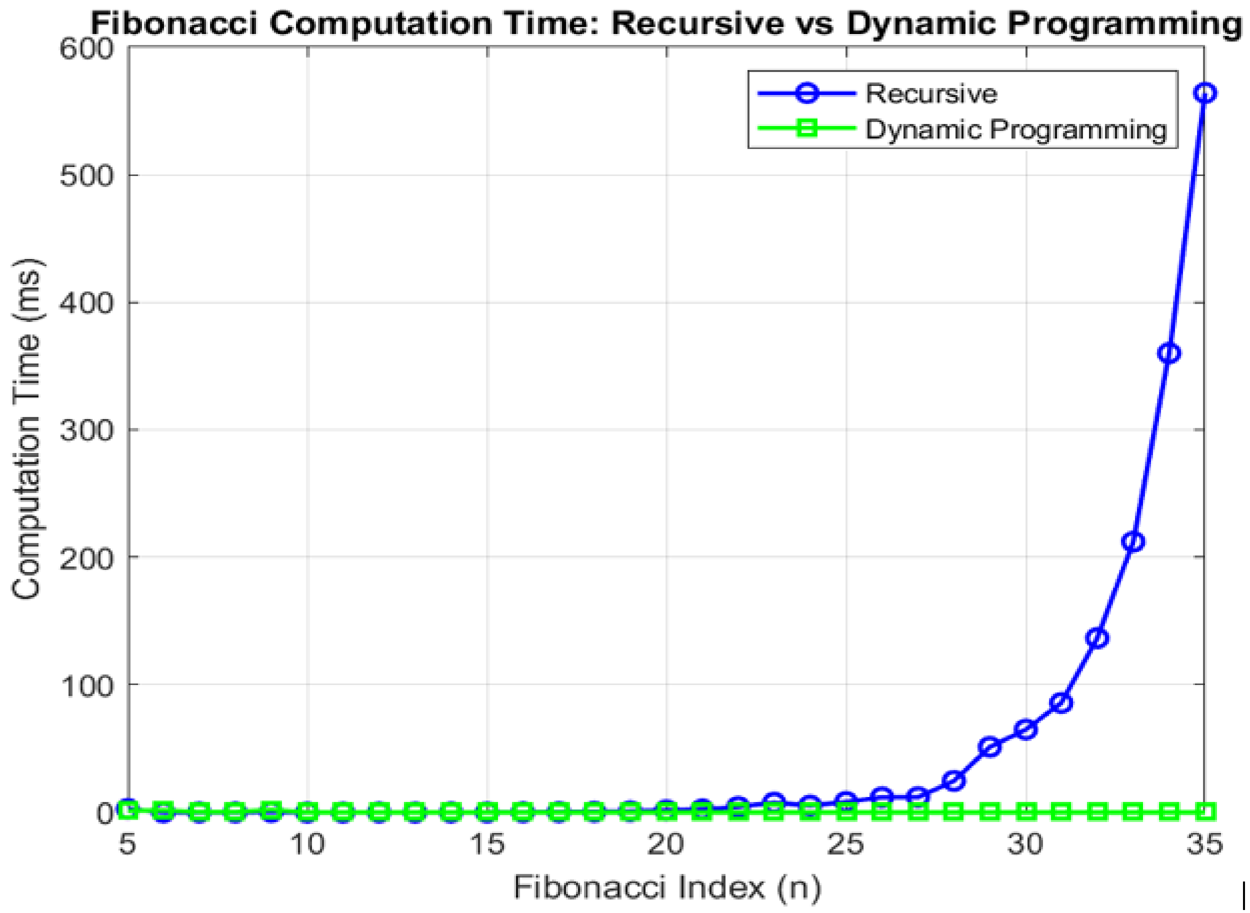
PLOTTING THE RESULTS FOR COMPARISON

```
% 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.

A GRAPH FOR COMPARISON OF COMPUTATION TIME



KNAPSACK PROBLEM

Recursive Concept of solving a knapsack problem.

```
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;
```

Line by line explanation.

function max_value = solveSimpleKnapsack(W, wt, val)

Defines function name and input/output parameters

n = length(wt);

Gets number of items from weight array length

memo = -ones(n + 1, W + 1);

Creates a matrix filled with -1 values for memorization

function result = knapsack(i, w)

Defines nested recursive function

if i == 0 || w == 0

Checks if no items left or no capacity remaining

result = 0;

Returns zero value for base case

return;

Exits function early

CONTINUATION OF CODE

```
% 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);
```

Checks if solution already computed

result = memo(i, w);

Returns stored result from memo table

current_weight = wt(i);

Gets weight of current item

current_value = val(i);

Gets value of current item

if current_weight > w

Checks if item exceeds remaining capacity

result = knapsack(i - 1, w);

Recurses without current item

val_exclude = knapsack(i - 1, w);

Computes value excluding current item

val_include = current_value + knapsack(i - 1, w - current_weight);

Computes value including current item

result = max(val_exclude, val_include);

CONTINUATION OF CODE

```
        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)
```

memo(i, w) = result;
Stores computed result in memo table
max_value = knapsack(n, W);
Starts recursion with all items and full capacity
W = 5;
Sets knapsack capacity to 5
wt = [2, 3, 4, 5];
Defines item weights array
val = [3, 4, 5, 6];
Defines item values array
result = solveSimpleKnapsack(W, wt, val);
Calls the knapsack function
disp(['Maximum Value: ', num2str(result)]);
Displays the computed result

DYNAMIC PROGRAMMING

Dynamic Programming

```
    % Dynamic Programming solution for 0/1 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);
```

Function Definition

function[max_value, selected_items] = solveSimpleKnapsack-WithItems(W, wt, val)

Defines function that returns both max value and selected items

Initial Setup

n = length(wt);

Gets number of items

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

Creates DP table initialized with zeros

DP Table Filling

for i = 1:n

Loops through each item

for current_capacity = 0:W

Loops through all possible capacities

if wt(i) <= current_capacity

CONTINUATION

```
                dp(i + 1, current_capacity + 1) = max(in-
clude_value, exclude_value);
            else
                % Item too heavy - must exclude
                dp(i + 1, current_capacity + 1) = dp(i, cur-
rent_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)
```

Checks if current item fits in current capacity

include_value = val(i) + dp(i, current_capacity - wt(i) + 1);

Calculates value if item is included

exclude_value = dp(i, current_capacity + 1);

Calculates value if item is excluded

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

Stores better choice in DP table

else

Item doesn't fit case

dp(i + 1, current_capacity + 1) = dp(i, current_capacity + 1);

Carry forward previous value

Get Result

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

Final answer is in bottom-right cell

Backtracking

selected_items = [];

CONTINUATION

```
                % 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);
```

Initialize empty list for selected items

remaining_capacity = W;

Start with full capacity

for item_index = n:-1:1

Loop backwards through items

if dp(item_index + 1, remaining_capacity + 1) ~= dp(item_index, remaining_capacity + 1)

Check if item was included

selected_items = [item_index, selected_items];

Add item to selected list

remaining_capacity = remaining_capacity - wt(item_index);

Reduce remaining capacity

Testing Code

fprintf('Knapsack Problem Solution\n');

Print header

W = 5; wt = [2, 3, 4, 5]; val = [3, 4, 5, 6];

CONTINUATION

```
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(se-
lected_items));
fprintf('Selected weights: [%s]\n', num2str(wt(se-
lected_items)));
fprintf('Selected values: [%s]\n', num2str(val(se-
lected_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(se-
lected_items)));
end
```

Define test inputs

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

Call function

fprintf('Maximum value: %d\n', max_val);

Display results

fprintf('Selected items (indices): [%s]\n', num2str(selected_items));

Show which items were chosen

fprintf('Total weight used: %d\n', sum(wt(selected_items)));

Show total weight used

if ~isempty(selected_items)

Verification check

fprintf('Verification - Total weight: %d (<= capacity: %d)\n', ...

Confirm weight doesn't exceed capacity

Graphical comparison between recursion and dynamic programming.

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

GRAPHICAL REPRESENTATION