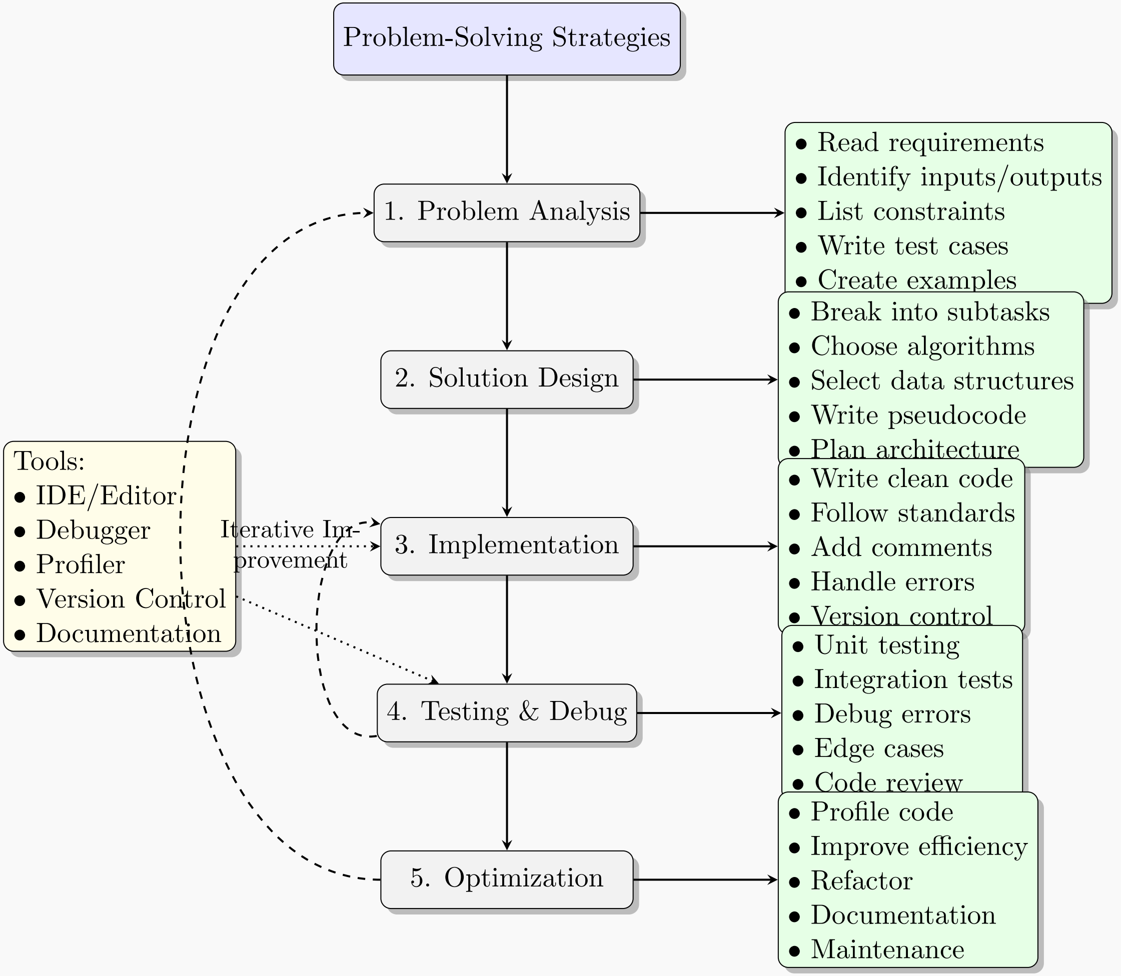
Computing and Software Engineering

GET211

September 23, 2025

Problem Solving Strategies

Problem-solving in programming is a structured approach that involves breaking down complex challenges into manageable parts, creating logical sequences to solve these parts, and integrating them into a cohesive solution. This approach generally involves several key steps, each contributing to the effectiveness, efficiency, and accuracy of the final program.
1. Understanding the Problem This initial phase is crucial for setting a solid foundation. A person must interpret the requirements, clarify ambiguities, and comprehend what the final outcome should achieve.
2. Planning the Approach: Once the problem is well-understood, creating a structured plan is essential. This involves breaking down the problem into smaller, more manageable parts and deciding on the overall flow and structure.
3. **Divide and Conquer:** Complex problems can be daunting if tackled as a whole. Breaking the problem down into functions, modules, or classes simplifies each part and makes it easier to focus on individual tasks.
4. **Writing and Testing Incrementally** Writing code in small sections and testing each one before proceeding is crucial to catch errors early.
5. **Debugging and Iteration:** Errors and unexpected behaviors are natural in programming. Debugging involves systematically locating and correcting these issues.
6. **Optimization:** After achieving a working solution, efficiency improvements are often beneficial. Optimization helps to make the code faster, use less memory, or perform better under certain conditions.
7. **Reflecting and Documenting:** Reflection involves reviewing what worked well, identifying areas for improvement, and considering alternate approaches. Write clear and detailed documentation for other developers or for future reference.
requirements.



Problem-Solving Strategies

1. Problem Analysis

- Read requirements
- Identify inputs/outputs
- List constraints
- Write test cases
- Create examples

2. Solution Design

- Break into subtasks
- Choose algorithms
- Select data structures
- Write pseudocode
- Plan architecture

3. Implementation

- Write clean code
- Follow standards
- Add comments
- Handle errors
- Version control

4. Testing & Debug

- Unit testing
- Integration tests
- Debug errors
- Edge cases
- Code review

5. Optimization

- Profile code
- Improve efficiency
- Refactor
- Documentation
- Maintenance

Tools:

- IDE/Editor
- Debugger
- Profiler
- Version Control
- Documentation

Iterative Improvement

Problem solving strategy

Example of problem solving in MATLAB
Try solving a problem to find the area and perimeter of a circle given the radius. Using these strategies:
1. Understand the Problem:
- Input: ‘radius‘
- Output: ‘area‘ and ‘perimeter‘
- Requirement: Use MATLAB formulas.
2. Plan Your Approach:
- Use the formulas area $= \pi \times \text{radius}^2$ and perimeter $= 2 \times \pi \times \text{radius}$.
3. Divide and Conquer:
- Create two separate calculations for area and perimeter.
4. Write and Test Incrementally:
- Start by testing with a sample radius to check calculations.
5. Debugging:
- Ensure that ‘pi‘ is used correctly in calculations.
6. Optimize and Reflect:
- Reflect on whether the code is efficient and consider adding comments for clarity.

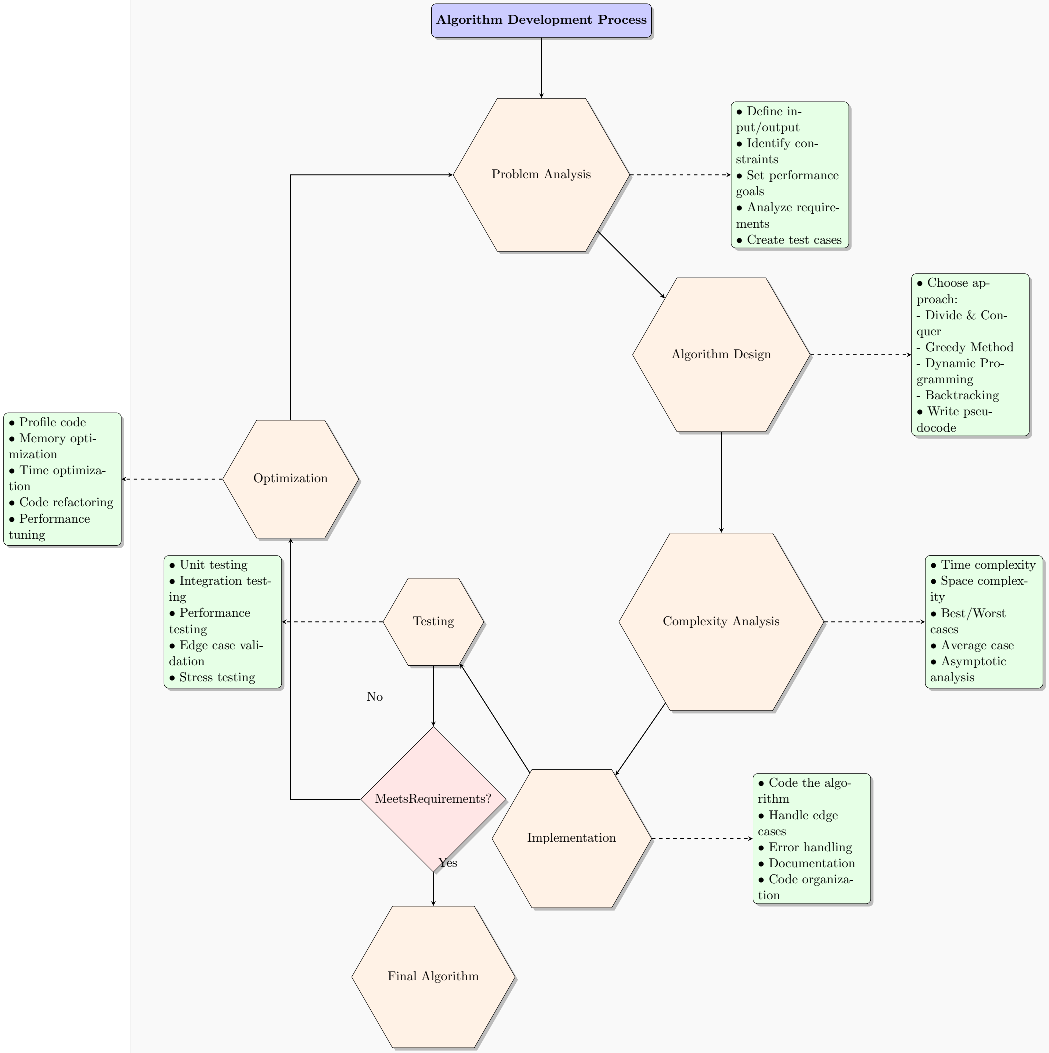
Algorithm Development

Algorithm development is the systematic process of creating a step-by-step procedure to solve a specific problem or perform a particular task. It involves analyzing the problem, defining the objectives and constraints, designing an efficient approach, implementing the solution in code, testing it for accuracy and efficiency, and refining it to ensure optimal performance.
Stages in algorithm development:
1. Problem Analysis: Carefully understanding the problem requirements, constraints, and expected outcomes. This step involves identifying inputs, defining the expected outputs, and understanding any special conditions or edge cases that could affect the algorithm's logic.
2. Objective Definition: Setting goals for the algorithm, such as speed, memory efficiency, precision, or scalability. Depending on the requirements, the algorithm may need to prioritize one objective (like low memory usage) over others (such as processing speed).
3. Algorithm Design: Crafting the sequence of operations to reach the solution. The design phase often includes:
- Choosing the strategy: Common strategies include brute force, divide and conquer, dynamic programming, and greedy approaches.
- Pseudocode/Flowchart: Writing pseudocode or creating a flowchart to outline the sequence of steps visually.
- Data Structures: Selecting the data structures that best suit the problem and improve efficiency, such as arrays, linked lists, or trees.

Algorithm Development

4. Implementation: Writing the code based on the algorithm design, following programming best practices for readability and modularity.
5. Testing and Validation: Running the algorithm with various inputs, especially edge cases, to confirm it produces correct and reliable results. This stage ensures the algorithm behaves as expected and meets the initial requirements.
6. Optimization: Improving the algorithm to make it faster or more memory-efficient, often by analyzing and reducing its time and space complexity.
7. Documentation and Review: Documenting the code, including assumptions and edge cases, for maintainability, and often having peers review it for clarity and correctness.
8. Deployment and Iterative Refinement: After deployment, gathering feedback and data on the algorithm's real-world performance can reveal new ways to optimize or refine it for further improvements.

Algorithm development is both a creative and structured approach to problem-solving. It combines logical reasoning and technical expertise to create solutions that are both correct and efficient in real-world applications.



Algorithm Development Process

Problem Analysis

- Define input/output
- Identify constraints
- Set performance goals
- Analyze requirements
- Create test cases

Algorithm Design

- Choose approach:
  - Divide & Conquer
  - Greedy Method
  - Dynamic Programming
  - Backtracking
- Write pseudocode

Complexity Analysis

- Time complexity
- Space complexity
- Best/Worst cases
- Average case
- Asymptotic analysis

Implementation

- Code the algorithm
- Handle edge cases
- Error handling
- Documentation
- Code organization

Testing

- Unit testing
- Integration testing
- Performance testing
- Edge case validation
- Stress testing

MeetsRequirements?

No

Yes

Optimization

- Profile code
- Memory optimization
- Time optimization
- Code refactoring
- Performance tuning

Final Algorithm

Pseudocode

Pseudocode is an informal, human-readable description of the steps and logic needed to solve a problem or perform a task. It uses plain language mixed with some programming-like structure without strict syntax rules, making it easy to understand and convert into actual code.

Advantages of Pseudocode

- Readability: It's easy for anyone with a basic understanding of programming to read and understand.
- Algorithm Focused: Allows you to concentrate on the logic without worrying about syntax.
- Flexible: Pseudocode can be adapted to any programming language once the logic is solid.

Pseudocode provides a bridge between the algorithm design and actual implementation, making it easier to plan, communicate, and refine an algorithm before coding.

Category	Convention	Example
Input/Output	INPUT OUTPUT DISPLAY	INPUT number OUTPUT ”Hello World” DISPLAY result
Variables	Assignment Declaration Increment Decrement	SET x = 5 DECLARE integer x INCREMENT counter DECREMENT counter

Conditional	If statement	IF condition THEN
		statements
		END IF

	If-Else	IF condition THEN
		statements
		ELSE
		statements
		END IF

Loops	For loop	FOR i = 1 TO n DO statements END FOR
	While loop	WHILE condition DO statements END WHILE
	Repeat-Until	REPEAT statements UNTIL condition

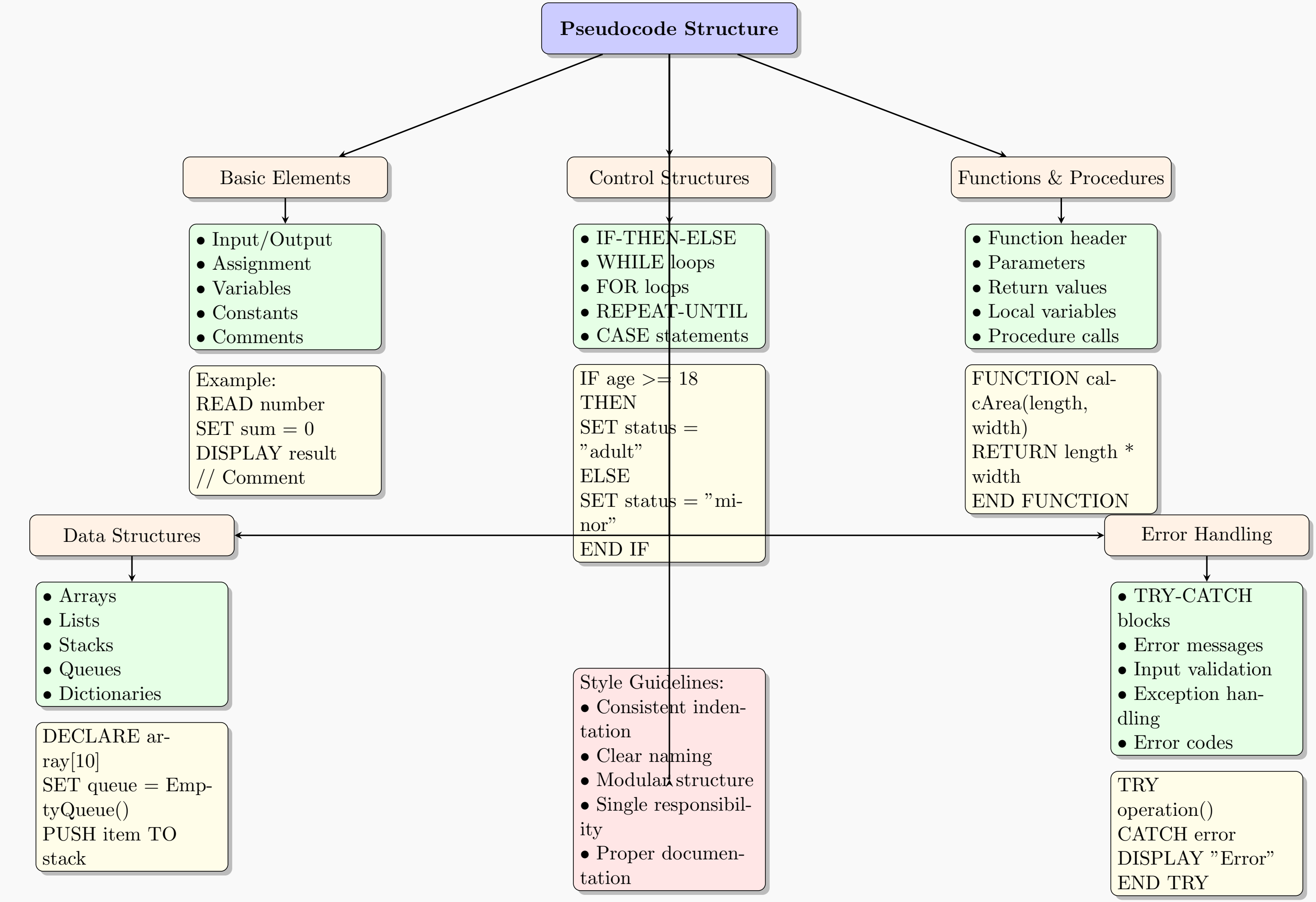
Functions	Function declaration	FUNCTION name(parameters)
		statements
		RETURN value
		END FUNCTION

	Procedure declaration	PROCEDURE name(parameters) statements END PROCEDURE

Arrays	Array declaration
	Array access
	Array assignment
		DECLARE ARRAY numbers[size]
		SET x = array[index]
		SET array[index] = value

Comments	Single line Multiple lines	// This is a comment /* This is a multiple line comment */
Boolean	Logical operators Comparison operators	AND, OR, NOT =, ∼=, <, >, <=, >=

Table 1: Common Pseudocode Conventions



Pseudocode Structure

Basic Elements

- Input/Output
- Assignment
- Variables
- Constants
- Comments

```
Example:
READ number
SET sum = 0
DISPLAY result
// Comment
```

Control Structures

- IF-THEN-ELSE
- WHILE loops
- FOR loops
- REPEAT-UNTIL
- CASE statements

```
IF age >= 18 THEN
SET status = ”adult”
ELSE
SET status = ”minor”
END IF
```

Functions & Procedures

- Function header
- Parameters
- Return values
- Local variables
- Procedure calls

```
FUNCTION calcArea(length, width)
RETURN length * width
END FUNCTION
```

Data Structures

- Arrays
- Lists
- Stacks
- Queues
- Dictionaries

```
DECLARE array[10]
SET queue = EmptyQueue()
PUSH item TO stack
```

Error Handling

- TRY-CATCH blocks
- Error messages
- Input validation
- Exception handling
- Error codes

```
TRY
operation()
CATCH error
DISPLAY ”Error”
END TRY
```

Style Guidelines:

- Consistent indentation
- Clear naming
- Modular structure
- Single responsibility
- Proper documentation

Pseudocode Example

Provide the pseudocode to find Maximum Value in a List

1. Input: A list of numbers ‘nums‘.
2. Output: The maximum value in the list.

Pseudocode:

```
PROCEDURE FindMaximum(numbers)
    IF length of numbers = 0 THEN
        RETURN null
    END IF

    SET max = numbers[0]

    FOR i = 1 TO length of numbers - 1 DO
        IF numbers[i] > max THEN
            SET max = numbers[i]
        END IF
    END FOR

    RETURN max
END PROCEDURE
```

Pseudocode Example

Provide the pseudocode to check if a Number is Prime
1. Input: A positive integer ‘n‘.
2. Output: Boolean value (‘true‘ or ‘false‘) indicating whether ‘n‘ is prime.
Pseudocode:

```
FUNCTION is_prime(n)
    IF n <= 1 THEN
        RETURN false
    END IF

    FOR i FROM 2 TO sqrt(n)  // Loop through possible divisors up t
        IF n MOD i == 0 THEN  // If n is divisible by i
            RETURN false
        END IF
    END FOR

    RETURN true
END FUNCTION
```

Flowchart

Flowcharts are a visual representation of an algorithm or process. They use standardized symbols to represent different types of operations or steps, helping to understand the flow of the program or process.

Flowchart Symbols

1. Oval (Start/End):

- Represents the start or end of the process.
- Example: "Start" or "End".

2. Rectangle (Process):

- Represents a process or operation, such as calculations or variable assignments.
- Example: "a = b + c".

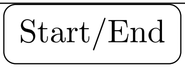
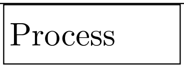
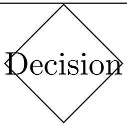
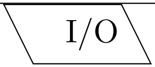
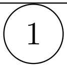
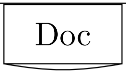
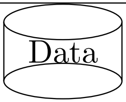
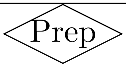
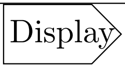
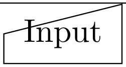
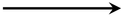
3. Parallelogram (Input/Output):

- Represents input or output operations.
- Example: "Input x" or "Display result".

4. Diamond (Decision):

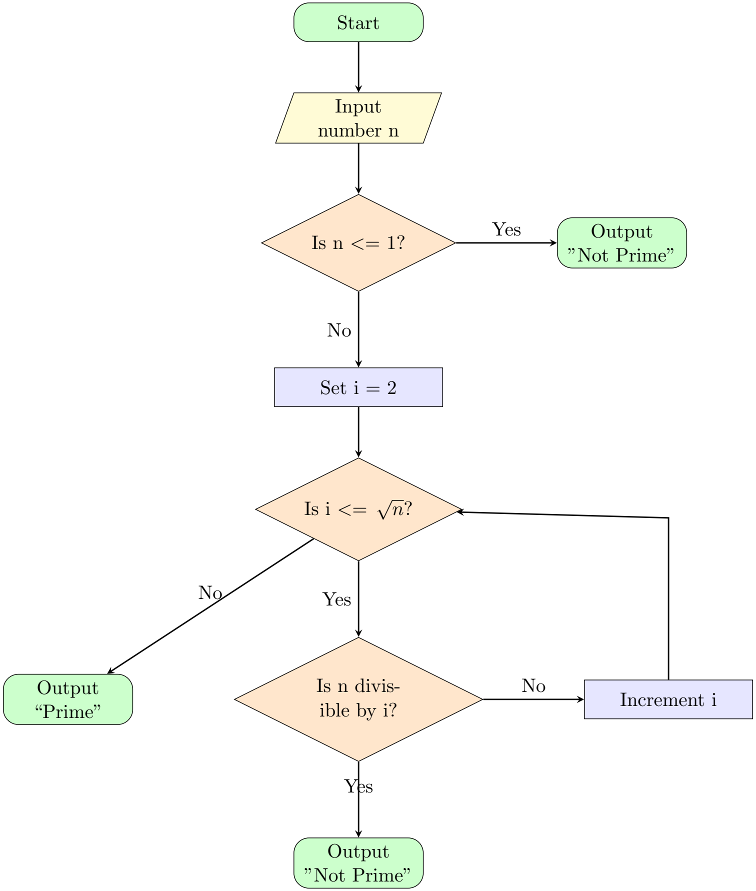
- Represents a decision point where the flow branches based on a condition.

Standard Flowchart Symbols

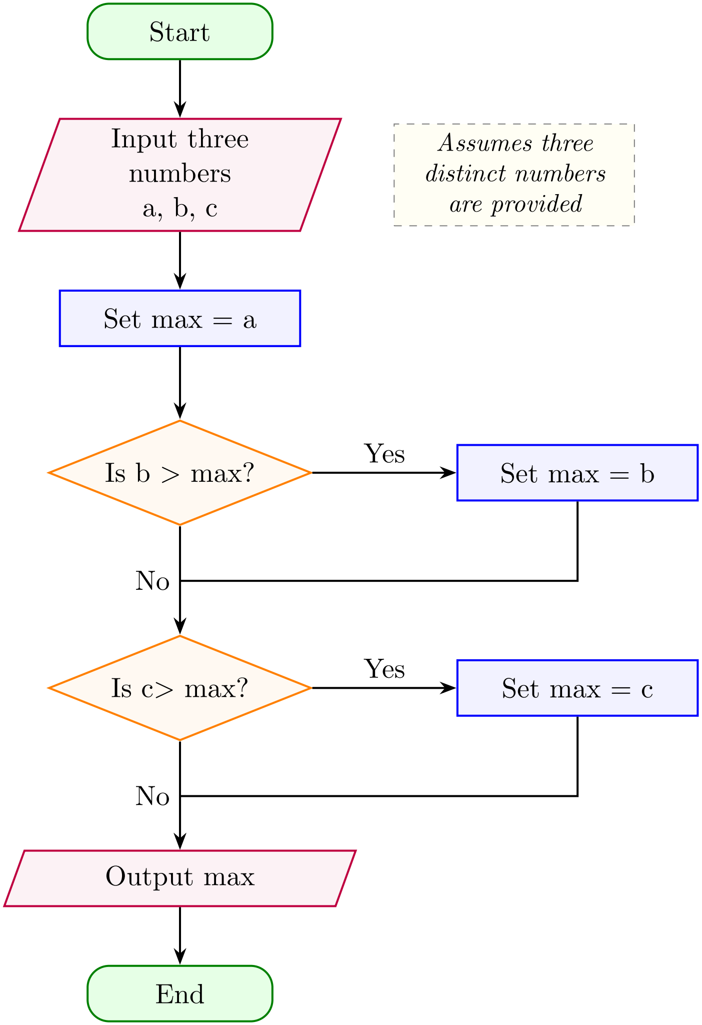
Symbol Name	**Visual Result**
Terminal	
Process	
Decision	
Input/Output	
Connector	
Document	
Database	
Preparation	
Display	
Manual Input	
Flow Line	

Draw a flowchart for the algorithm that checks if a number is prime. The pseudocode used is the one discussed earlier.

Flowchart: Determine if a Number is Prime



Flowchart: Finding the Largest of Three Numbers



Control Structures in MATLAB

Control structures in MATLAB are essential for managing the flow of execution in a program. The most common control structures include:

- Conditional Statements (if, else, elseif, switch)
- Loops (for, while)
- Break and Continue

These structures enable decision-making, and iteration within the program.

Conditional Statements

Conditional statements allow the program to execute specific blocks of code based on logical conditions. MATLAB provides the following conditional statements:

- **if statement**: Executes code if a condition is true.
- **else statement**: Executes code if the `if` condition is false.
- **elseif statement**: Checks multiple conditions in a sequence.
- **switch statement**: Executes one block of code based on the value of a variable.

Conditional Statements

if Statement

The ‘if‘ statement is used to execute a block of code if a condition is true.

Syntax:

```
if condition
    % Code to execute if condition is true
end
```

Example:

```
x = 5;
if x > 3
    disp('x is greater than 3');
end
```

Conditional Statements

if - else Statement

The ‘else‘ statement is executed when the ‘if‘ condition is false.

Syntax:

```
if condition
    % Code if true
else
    % Code if false
end
```

Example:

```
x = 2;
if x > 3
    disp('x is greater than 3');
else
    disp('x is less than or equal to 3');
end
```

elseif Statement

The ‘elseif‘ statement is used when there are multiple conditions to check.

Syntax:

```
if condition1
    % Code for condition1
elseif condition2
    % Code for condition2
else
    % Code if all conditions are false
end
```

Example:

```
x = 5;
if x > 10
    disp('x is greater than 10');
elseif x > 3
    disp('x is greater than 3 but less than or equal to 10');
else
    disp('x is less than or equal to 3');
end
```

Conditional Statements

switch Statement
The ‘switch‘ statement is useful for checking one variable against multiple possible values.
Syntax:

```
switch variable
    case value1
        % Code for value1
    case value2
        % Code for value2
    otherwise
        % Code if no match
end
```

Conditional Statements

switch Statement

Example:

```
day = 'Monday';
switch day
    case 'Monday'
        disp('Start of the work week');
    case 'Friday'
        disp('End of the work week');
    otherwise
        disp('Mid-week');
end
```

Loops in MATLAB

Loops allow repeated execution of a block of code based on a condition. MATLAB provides two types of loops:

- **for loop**: Executes a fixed number of iterations.
- **while loop**: Executes until a specified condition becomes false.

for Loop

The ‘for‘ loop is used for a fixed number of iterations. It is ideal when you know in advance how many times you want to execute a statement or a block of statements.

Syntax:

```
for index = startValue:endValue
    % Code to execute for each value of index
end
```

Example:

```
        for i = 1:5
            disp(i);
        end
```

Loops in MATLAB

while Loop

The ‘while‘ loop repeats a block of code as long as a specified condition remains true. The condition is checked before each iteration.

Syntax:

```
while condition
    % Code to execute as long as condition is true
end
```

Example:

```
        x = 1;
        while x <= 5
            disp(x);
            x = x + 1;
        end
```

Branching Statements

The `break` statements is used to control the flow of loops.

- **break**: Exits the loop immediately.

Example of `break`:

```
        for i = 1:10
            if i == 6
                break; % Exit the loop
            end
            disp(i);
        end
```

Branching Statements

The `continue` statement is used to control the flow of loops.

- **continue**: Skips the current iteration and proceeds to the next iteration of the loop.

Example of `continue`:

```
        for i = 1:5
            if mod(i, 2) == 0
                continue; % Skip even numbers
            end
            disp(i); % Only odd numbers will be displayed
        end
```

Exercise

Simple Login System
Write pseudocode for a simple login system that checks a username and password.

- Set a predefined `username` and `password`.
- Ask the user to input their `username` and `password`.
- If both values match the predefined ones, display a success message.
- If they do not match, display an error message.

Exercise

Sum of Even Numbers
Write pseudocode to calculate the sum of all even numbers up to a given integer, n.

- Ask the user to input a positive integer n.
- Initialize a variable `sum` to zero.
- Use a loop to iterate from 1 to n.
- If the current number is even, add it to `sum`.
- Display the result.

Then, write the MATLAB script.

Exercise

Simple ATM system

Create a flowchart for a simple ATM system that:

- Starts with an initial balance
- Allows user to:
  - Check balance
  - Deposit money
  - Withdraw money (if sufficient balance)
- Asks if user wants to perform another transaction

Exercise

A grading system

Design a flowchart for a grading system that:

- Takes marks as input (0-100)
- Assigns grades: A (90-100), B (80-89), C (70-79), D (60-69), F (below 60)
- Determines if student passed (grades A-D) or failed (grade F)
- Calculates and displays grade point (A=4, B=3, C=2, D=1, F=0)

Then write the MATLAB script.

Exercise

Sum of Odd Numbers

Write a MATLAB script to calculate the sum of all odd numbers up to a given number.

- Ask the user to input a positive integer, ‘n‘.
- Use a ‘while‘ loop to add each odd number from 1 up to ‘n‘.
- Display the result after the loop finishes.

Exercise

Menu Selection

Write a MATLAB script to display a menu and execute different tasks based on user input.

- Display a menu with the following options:
  - 1. Convert Celsius to Fahrenheit
  - 2. Calculate the square of a number
  - 3. Check if a number is even or odd
- Use a ‘switch‘ statement to handle each choice:
  - Option 1: Prompt for Celsius, convert, and display Fahrenheit.
  - Option 2: Prompt for a number, calculate its square, and display.
  - Option 3: Prompt for a number, check even/odd, and display result.

Exercise

Countdown Timer

Write a MATLAB script that creates a countdown timer.

- Ask the user to enter a starting number.
- Use a ‘while‘ loop to count down to zero.
- Display each number in the countdown, then display ”Blast off!” at the end.

Exercise

ATM Simulation
Write a MATLAB script to simulate a simple ATM withdrawal process.

- Set a balance, e.g., ‘balance = 500‘.
- Ask the user to enter the withdrawal amount.
- Use nested ‘if‘ statements to check:
  - If the amount is less than or equal to the balance, deduct it and display the new balance.
  - If the amount exceeds the balance, display ”Insufficient funds.”