Basic Algorithm Concepts

UNIT 1: BASIC CONCEPTS AND PRIMITIVES

1.1 What are Basic Concepts?

When we start learning about algorithms, we need to understand some fundamental ideas that form the foundation of everything we will study.

Key Terms You Must Know:

Data

Data refers to raw facts and figures that have no meaning by themselves. For example, the number "75" is just a number. It becomes meaningful only when we know it represents a student's exam score.

Information

When data is processed and organized, it becomes information. For example, knowing that "75 is the highest score in the class" gives meaning to the raw data.

Instructions

These are individual commands that tell the computer what to do. Think of them like steps in a recipe. Each instruction must be clear and do one specific thing.

Operations

These are the basic actions that a computer can perform. Just like humans can add numbers or compare two things, computers have built-in operations they can execute.

1.2 Understanding Primitives

Primitives are the simplest operations that cannot be broken down further. They are the basic tools we use to build algorithms.

Types of Primitive Operations:

Input Operations

These operations bring data into the computer. For example, reading numbers typed by the user or getting data from a file.

Output Operations

These operations show results to the user. For example, displaying answers on the screen or printing results.

Arithmetic Operations

These are mathematical operations we all know:

- Addition (combining two numbers)

- Subtraction (finding the difference)

- Multiplication (repeated addition)

- Division (splitting into parts)

Comparison Operations

These operations compare two values to determine:

- If they are equal

- If one is greater than the other

- If one is less than the other

Logical Operations

These deal with true/false values:

- AND (both conditions must be true)

- OR (at least one condition must be true)

- NOT (reverses true to false and false to true)

Assignment Operations

This is simply storing a value in memory and giving it a name so we can use it later.

UNIT 2: ALGORITHM AND ITS CHARACTERISTICS

2.1 What is an Algorithm?

An algorithm is a step-by-step procedure to solve a problem. Think of it like a cooking recipe:

- The recipe tells you what ingredients you need (input)

- It gives you clear steps to follow (process)

- At the end, you get a dish (output)

In computer science, an algorithm is the logic behind a computer program. It is the thinking part before we start writing actual code.

2.2 Real-Life Examples of Algorithms

Example 1: Making Tea

1. Boil water in a pot

2. Put tea leaves in a cup

3. Pour hot water into the cup

4. Add sugar and milk

5. Stir well

6. Serve hot

Example 2: Finding a Book in Library

1. Go to the computer catalog

2. Search for the book title

3. Note down the shelf number

4. Go to that shelf

5. Look for the book

6. Take the book

2.3 Important Characteristics of Algorithms

For something to be called a good algorithm, it must have these properties:

1. Finiteness

The algorithm must end after some steps. It cannot run forever. Just like a recipe has a last step, an algorithm must have an end.

2. Definiteness

Each step must be clear and unambiguous. There should be no confusion about what to do. For example, "add a pinch of salt" is not definite because "pinch" means different things to different people. "Add 2 grams of salt" is definite.

3. Input

An algorithm may have zero or more inputs. These are the data it needs to work with. For example, a sorting algorithm needs the list of numbers to sort.

4. Output

An algorithm must produce at least one output. This is the result we want. If there is no output, there is no point in running the algorithm.

5. Effectiveness

Each step must be simple enough that it can be done. You cannot have a step that says "solve all world problems" because that is too complex.

2.4 Why Do We Need Good Algorithms?

- They save time by solving problems efficiently

- They save computer memory and resources

- They are reusable for similar problems

- They help us think logically

- They make programs easier to understand and fix

UNIT 3: PSEUDO LANGUAGE AND ITS CONVENTIONS

3.1 What is Pseudo Language?

Pseudo language, commonly called pseudo-code, is a way to write algorithms using simple English-like statements. It is not a real programming language but a bridge between the problem and the actual code.

Think of it as writing instructions in plain English but in a structured way that programmers can easily understand.

3.2 Why Use Pseudo-code?

It is Language Independent

You can show your algorithm to someone who knows Java, C++, or Python, and they will all understand it.

It Focuses on Logic

You don't have to worry about syntax errors, semicolons, or brackets. You only think about the solution.

It is Easy to Read

Anyone with basic computer knowledge can understand pseudo-code, even if they don't know programming.

It is Quick to Write

When designing a solution, you can quickly write pseudo-code without getting stuck in programming language details.

3.3 Common Conventions in Pseudo-code

Naming Things

Data Storage + Function Definition

- Use meaningful names that tell what the variable stores

- Examples: `studentName` is better than `sn`

- Use `totalMarks` instead of `tm`

- For counters, use `i`, `j`, `k` (this is a tradition)

Showing Steps

- Write one statement per line

- Number steps if needed (Step 1, Step 2, etc.)

- Use indentation to show which statements belong together

Keywords

We use certain words that have special meaning:

For Input:

- INPUT, READ, GET

For Output:

- OUTPUT, PRINT, DISPLAY, SHOW

For Assignment:

- SET...TO..., LET...BE..., or simply =

For Decisions:

- IF, THEN, ELSE, END IF

For Loops:

- WHILE, DO, END WHILE

- FOR, TO, END FOR

- REPEAT, UNTIL

Comments

Comments are notes to explain what the code does. They are not part of the algorithm's logic. We usually start them with //

3.4 A Simple Example

Without pseudo-code, explaining an algorithm might be messy:

"First we take the number, then we check if it is more than zero, if yes we do something..."

With pseudo-code, it becomes clean:

```
INPUT number
IF number > 0 THEN
  PRINT "Positive"
ELSE
  PRINT "Not positive"
END IF
```

4.1 What are Algorithm Notations?

Algorithm notations are different ways to represent or write down algorithms. Just like we can tell a story through writing, pictures, or video, algorithms can be shown in different formats.

4.2 Types of Algorithm Notations

1. Natural Language Notation

This means writing the algorithm in plain English, just like you would explain it to a friend.

Example - Finding if a number is even:

Step 1: Ask the user for a number

Step 2: Divide the number by 2

Step 3: If there is no remainder, the number is even

Step 4: If there is a remainder, the number is odd

Step 5: Show the result to the user

When to use: When explaining to non-technical people or for very simple algorithms.

Advantages: Easy to understand, no special knowledge needed

Disadvantages: Can be long and imprecise, may have ambiguity

2. Pseudo-code Notation

This is what we learned in the previous section. It is structured English with some formal rules.

Example - Finding if a number is even:

```
BEGIN
  INPUT num
  IF num % 2 = 0 THEN
    OUTPUT "Even"
  ELSE
    OUTPUT "Odd"
  END IF
END
```

When to use: For designing algorithms, communicating with programmers, before coding

3. Flowchart Notation

This uses symbols and arrows to show the flow of an algorithm visually.

Common Flowchart Symbols:

Oval (Terminal)

Shape like a stretched circle

Used for: Start and End of algorithm

Parallelogram (Input/Output)

Shape like a slanted rectangle

Used for: Taking input or showing output

Rectangle (Process)

Regular rectangle shape

Used for: Calculations and assignments

Diamond (Decision)

Diamond shape

Used for: Yes/No questions or decisions

Arrows (Flow Lines)

Lines with arrows

Used for: Showing the direction of flow

When to use: When you want to see the overall structure, teaching beginners, documenting complex logic

Advantages: Visual and easy to follow, shows all paths clearly

Disadvantages: Can become messy for large algorithms, hard to modify

4. Decision Tables

This is a table format that shows what actions to take based on different conditions.

Simple Example - Library Fine Decision:

| Condition | Case 1 | Case 2 | Case 3 | Case 4 |

|---|---|---|---|---|

| Book returned late? | No | Yes | Yes | Yes |

| Is it a holiday? | - | No | Yes | No |

| Days late ≤ 7? | - | Yes | - | No |

| Fine amount | 0 | 10 | 5 | 20 |

When to use: When there are many conditions to check, business rules, complex decision logic

4.3 Which Notation Should You Use?

| Situation | Best Notation |

|-----------|--------------|

| Explaining to non-programmers | Natural Language or Flowchart |

| Designing a solution | Pseudo-code |

| Team discussion | Pseudo-code or Flowchart |

| Complex logic with many conditions | Decision Table |

| Before writing actual code | Pseudo-code |

| Documentation | Flowchart or Pseudo-code |

| Teaching beginners | Flowchart first, then Pseudo-code |

UNIT 5: PRACTICE ON WRITING SIMPLE ALGORITHMS

5.1 How to Approach Algorithm Writing

Writing algorithms is a skill that improves with practice. Here is a step-by-step approach:

Step 1: Understand the Problem

- Read the problem carefully

- Identify what is given (inputs)

- Identify what is needed (outputs)

- Ask questions if anything is unclear

Step 2: Think of the Solution

- How would you solve it manually?

- What steps do you follow in your mind?

- Write down these steps in any order

Step 3: Organize the Steps

- Put steps in logical order

- Group related steps together

- Identify where decisions are needed

- Identify where repetition is needed

Step 4: Write the Algorithm

- Use pseudo-code conventions

- Give meaningful names

- Show the structure with indentation

- Add comments for complex parts

Step 5: Test Your Algorithm

- Take sample inputs

- Follow your steps manually

- Check if you get the right output

- Fix any problems you find

5.2 Simple Algorithm Examples for Practice

Example 1: Find the Sum of Two Numbers

Problem: Write an algorithm to add two numbers and show the result.

Thinking Process:

- We need two numbers as input

- We add them together

- We show the answer

Algorithm:

```
Algorithm AddTwoNumbers
  // This algorithm adds two numbers and displays the sum
  BEGIN
    // Get the numbers from user
    INPUT firstNumber
    INPUT secondNumber
    // Calculate the sum
    sum = firstNumber + secondNumber

    // Show the result
    OUTPUT sum
```

```
END
```

Example 2: Find the Larger of Two Numbers

Problem: Write an algorithm to find which number is bigger between two numbers.

Thinking Process:

- Get two numbers
- Compare them
- Decide which is larger
- Show the answer

Algorithm:

```
Algorithm FindLarger
  // This algorithm finds the larger of two numbers
  BEGIN
    // Get the numbers
    INPUT num1
    INPUT num2

    // Compare and find larger
    IF num1 > num2 THEN
      OUTPUT num1
    ELSE
      IF num2 > num1 THEN
        OUTPUT num2
      ELSE
        OUTPUT "Both numbers are equal"
      END IF
    END IF
  END
```

Example 3: Check if a Number is Positive, Negative, or Zero

Problem: Write an algorithm that tells whether a number is positive, negative, or zero.

Thinking Process:

- Get one number

- Check if it is greater than zero

- If not, check if it is less than zero

- If neither, it must be zero

Algorithm:

```
Algorithm CheckNumber
   // This algorithm checks if a number is positive, negative, or zero
   BEGIN
      // Get the number
      INPUT number

      // Check and classify
      IF number > 0 THEN
         OUTPUT "Positive"
      ELSE
         IF number < 0 THEN
            OUTPUT "Negative"
         ELSE
            OUTPUT "Zero"
         END IF
      END IF
   END
```

Example 4: Print Numbers from 1 to 10

Problem: Write an algorithm to display numbers 1 through 10.

Thinking Process:

- We need to repeat an action 10 times

- Start from 1, go up to 10

- Show each number

Algorithm:

```
Algorithm PrintOneToTen
  // This algorithm prints numbers from 1 to 10
  BEGIN
    // Initialize counter
    counter = 1

    // Repeat until counter reaches 10
    WHILE counter <= 10 DO
      OUTPUT counter
      counter = counter + 1
    END WHILE
  END
```

Example 5: Calculate Average of Three Numbers

Problem: Write an algorithm to find the average of three test scores.

Thinking Process:

- Get three scores
- Add them together
- Divide by 3
- Show the average

Algorithm:

```
Algorithm CalculateAverage
  // This algorithm calculates average of three numbers
  BEGIN
    // Get the three scores
    INPUT score1
    INPUT score2
    INPUT score3
```

```
    // Calculate sum and average
    sum = score1 + score2 + score3
    average = sum / 3

    // Show the result
    OUTPUT average
  END
```

Example 6: Determine if a Student Passed or Failed

Problem: A student passes if they score 50 or more. Write an algorithm to check pass/fail.

Thinking Process:

- Get the student's score
- Compare with passing marks (50)
- If score is 50 or more, they pass
- Otherwise, they fail

Algorithm:

```
Algorithm CheckPassFail
  // This algorithm determines if a student passed or failed
  BEGIN
    // Get student's score
    INPUT score

    // Check against passing marks
    IF score >= 50 THEN
      OUTPUT "Pass"
    ELSE
      OUTPUT "Fail"
    END IF
  END
```

Example 7: Find the Sum of First N Natural Numbers

Problem: Write an algorithm to add numbers from 1 to N, where N is given by the user.

Thinking Process:

- Get N from user

- Start sum at 0

- Add each number from 1 to N to the sum

- Show the total

Algorithm:

```
Algorithm SumNaturalNumbers
  // This algorithm finds sum of first N natural numbers
  BEGIN
    // Get N from user
    INPUT N

    // Initialize sum and counter
    sum = 0
    counter = 1

    // Add numbers from 1 to N
    WHILE counter <= N DO
      sum = sum + counter
      counter = counter + 1
    END WHILE

    // Show the result
    OUTPUT sum
  END
```

5.3 Common Mistakes to Avoid

Mistake 1: Missing Steps

- Always ensure every logical step is included

- Don't assume the computer knows what you mean

Mistake 2: Unclear Instructions

- "Process the data" is too vague

- "Add all numbers and divide by count" is clear

Mistake 3: No Termination Condition

- Loops must have a way to end

- Always update counters in loops

Mistake 4: Wrong Order

- Steps must be in correct sequence

- You cannot calculate average before finding sum

Mistake 5: Forgetting Input/Output

- Every algorithm needs input (maybe none) and output

- Show results to the user

5.4 Practice Exercises

Try writing algorithms for these problems:

1. Convert temperature from Celsius to Fahrenheit

2. Find if a number is even or odd

3. Calculate the area of a rectangle

4. Find the smallest of three numbers

5. Print the multiplication table of a given number

6. Calculate the factorial of a number

7. Count how many vowels are in a word

8. Check if a year is a leap year

9. Find the reverse of a number

10. Determine if a person is eligible to vote (age 18 or above)

SUMMARY

Key Points to Remember

Algorithms are step-by-step solutions to problems. They are like recipes for computers.

Good algorithms must:

- End after finite steps

- Have clear instructions

- Produce correct output

- Be efficient

Pseudo-code is our friend:

- It is English-like and easy to read

- It works for any programming language

- It focuses on logic, not syntax

We can write algorithms in different ways:

- Natural language (plain English)

- Pseudo-code (structured English)

- Flowcharts (pictures with symbols)

- Decision tables (for complex rules)

When writing algorithms:

1. Understand the problem first

2. Think about inputs and outputs

3. Break into small steps

4. Write clearly with proper structure

5. Test with examples