UNIT 7

Computational Thinking

Student Learning Outcomes

By the end of this chapter, you will be able to:

- Define computational thinking and its key components: decomposition, pattern recognition, abstraction, and algorithms.
- Explain the principles of computational thinking, including problem understanding, problem simplification, and solution selection and design.
- Describe algorithm design methods, specifically flowcharts and pseudocode, and understand the differences between them.
- Create and interpret flowcharts to represent algorithms visually.
- Write pseudocode to outline algorithms in a structured, human-readable format.
- Engage in algorithmic activities, such as design and evaluation techniques.
- Conduct dry runs of flowcharts and pseudocode to manually verify their correctness.
- Understand the concept and importance of LARP (Logic of Algorithms for Resolution of Problems).
- Implement LARP activities to practice writing algorithms and drawing flowcharts.
- Identify different types of errors in algorithms, including syntax errors, logical errors, and runtime errors.
- Apply debugging techniques to find and fix errors in algorithms.
- Recognize common error messages encountered during LARP and learn how to address them.
- Demonstrate problem-solving skills by applying computational thinking principles to real- world scenarios.
- Evaluate the efficiency of different algorithms and improve them based on performance analysis.

Not for Sale PCTB

Introduction

Introduction Computational thinking is an essential skill that enables individuals to solve complex problems using methods that align with processes involved in computer science. This chapter begins by defining computational thinking and breaking it down into its fundamental components: decomposition, pattern recognition, abstraction, and algorithms. These components are essential for simplifying complicated problems, identifying patterns that can lead to solutions, focusing on relevant details while ignoring unnecessary ones, and creating step-by-step procedures for solving problems. Understanding these concepts is not only beneficial for computer scientists but also for anyone looking to improve their problem-solving skills across various fields.

In addition to defining computational thinking, this chapter explores the principles that guide it, such as understanding the problem at hand, simplifying it to make it more manageable, and selecting the best solution design. The chapter introduces different methods for designing algorithms, including the use of flowcharts and pseudocode, and explains how to distinguish between these two approaches. Furthermore, it emphasizes the importance of practicing algorithm design and evaluation through hands-on activities like LARP (Logic of Algorithms for Resolution of Problems). Lastly, the chapter covers essential aspects of error identification and debugging, providing techniques for recognizing and fixing common errors encountered during the implementation of algorithms. By mastering these skills, students will be well-equipped to tackle a wide range of computational problems efficiently and effectively.

Not for Sale PCTB

7.1 Definition of Computational Thinking

Computational Thinking (CT) is a problem-solving process that involves a set of skills and techniques to solve complex problems in a way that a can be executed by a computer. This approach can be used in various fields beyond computer science, such as biology, mathematics, and even daily life

Did You Know

Computational thinking is not limited to computer science. It is used in everyday problem solving, such as planning a trip or organizing tasks.

Let's break down computational thinking into its key components:

7.1.1 Decomposition

Decomposition is the method of breaking down a complicated problem into smaller, more convenient components.

Decomposition is an important step in computational thinking. It involves dividing a complex problem into smaller, manageable tasks. Let's take the example of building a birdhouse. This task might look tough at first, but if we break it down, we can handle each part one at a time.

Here's how we can decompose the task of building a birdhouse. Figure 7.1 shows the decomposed tasks for building a birdhouse.

- **Design the Birdhouse:** Decide on the size, shape, and design. Sketch a plan and gather all necessary measurements.
- **Gather Materials:** List all the materials needed such as wood, nails, paint, and tools like a hammer and saw.
- **Cut the Wood:** Measure and cut the wood into the required pieces according to the design.
- **Assemble the Pieces:** Follow the plan to assemble the pieces of wood together to form the structure of the birdhouse.
- **Paint and Decorate:** Paint the birdhouse and add any decorations to make it attractive for birds.
- **Install the Birdhouse:** Find a suitable location and securely install the birdhouse where birds can easily access it.

Not for Sale PCTB

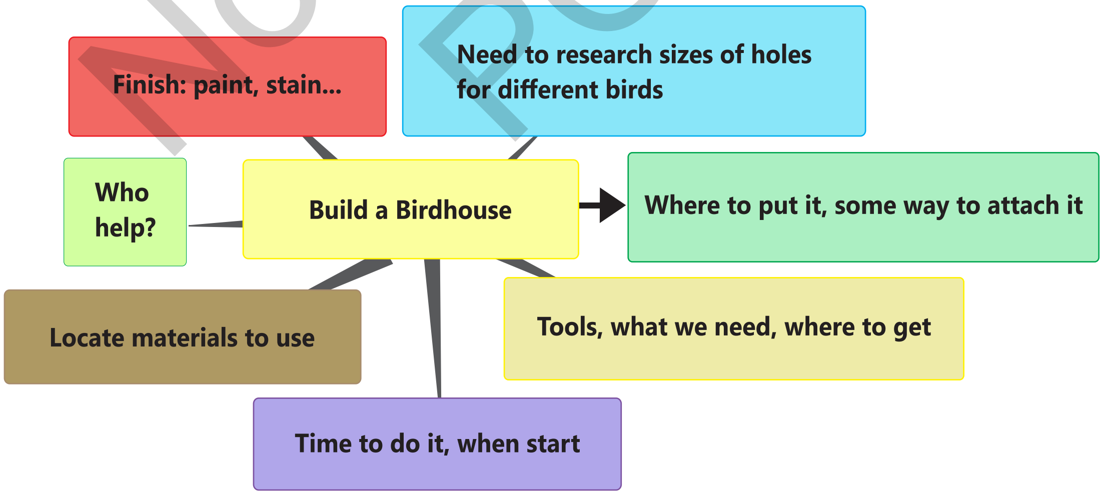

Finish: paint, stain...
Need to research sizes of holes for different birds
Who help?
Build a Birdhouse
Where to put it, some way to attach it
Locate materials to use
Tools, what we need, where to get
Time to do it, when start

Figure 7.1: Building a Birdhouse

Class activity

Decompose a Task

Think of a complex task you do regularly, like organizing a school event or cooking a meal. Break it down into smaller, manageable parts. Write down each step and discuss with your classmates how decomposition makes the task easier to handle.

7.1.2 Pattern Recognition

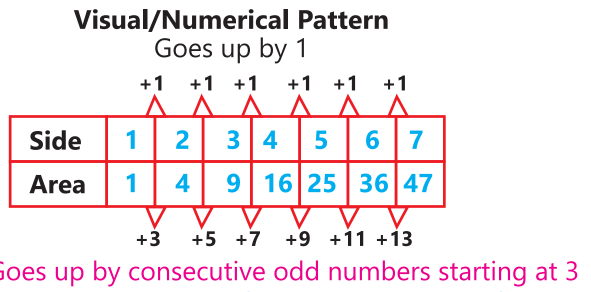
Pattern recognition involves looking for similarities or patterns among and within problems. For instance, if you notice that you always forget your homework on Mondays, you might recognize a pattern and set a reminder specifically for Sundays.

Pattern recognition is an essential aspect of computational thinking. It involves identifying and understanding regularities or patterns within a set of data or problems. Let's consider the example of recognizing patterns in the areas of squares.

The upper row in Figure 7.2 represents the side lengths of squares, ranging from 1 to 7. The lower row shows the corresponding areas of these squares. Here, we can observe a pattern in how the areas increase.

- Side Length 1: Area = 1^2 = 1
- Side Length 2: Area = 2^2 = 4 (1 + 3)
- Side Length 3: Area = 3^2 = 9 (1 + 3 + 5)
- Side Length 4: Area = 4^2= 16 (1 + 3+ 5 + 7)
- Side Length 5: Area = 5^2= 25 (1 + 3 + 5+ 7 + 9)
- Side Length 6: Area = 6^2= 36 (1 + 3 + 5+ 7 + 9 + 11)
- Side Length 7: Area = 7^2= 49 (1 + 3 + 5 + 7 + 9 + 11+ 13)

We can see that the area of each square can be calculated by adding consecutive odd numbers. For example, the area of a square with a side length of 3 can be found by adding the first three odd numbers: 1 + 3 + 5 = 9.



Side	1	2	3	4	5	6	7
Area	1	4	9	16	25	36	47

Figure 7.2: Pattern in areas of squares with sides from 1 to 7

Not for Sale PCTB

Class activity

Create a table with side lengths from 1 to 10. Calculate the areas of the squares using the pattern of adding consecutive odd numbers. Verify your results by squaring the side lengths and see if the pattern holds.

7.1.3 Abstraction

Abstraction is a fundamental concept in problem solving, especially in computer science. It involves simplifying complex problems by breaking them down into smaller, more manageable parts, and focusing only on the essential details while ignoring the unnecessary ones. This helps in understanding, designing, and solving problems more efficiently.

- **Definition:** Abstraction is the process of hiding the complex details while exposing only the necessary parts. It helps reduce complexity by allowing us to focus on the high-level overview without getting lost in the details.
- **Example:** Making a Cup of Tea - **High-level Steps:** 1. Boil water. 2. Add tea leaves or a tea bag. 3. Steep for a few minutes. 4. Pour into a cup and add milk/sugar if desired.

Tidbits

When solving complex problems, try to break them down into smaller parts and focus on the main steps. This will helps you understand the problem better and find a solution more easily. By using abstraction, we can tackle complex problems by dealing with them at a higher level.

Not for Sale PCTB

7.1.3 Algorithms

An algorithm is a step-by-step collection of instructions to solve a problem or complete a task similar to following a recipe to bake a cake..

An **algorithm** is a precise sequence of instructions that can be followed to achieve a specific goal, like a recipe or a set of directions that tells you exactly what to do and in what order.

HOW TO BAKE A CAKE?

1) Preheat the oven
2) Gather the ingredients
3) Measure out the ingredients
4) Mix together the ingredients to make the batter
5) Grease a pan
6) Pour the batter into the pan
7) Put the pan in the oven
8) Set a timer
9) When the timer goes off, take the pan out of the oven
10) Enjoy!

Figure 7.3: Algorithm example: Recipe to bake a cake

- **Example 1:** Baking a Cake: In Figure7.3, we see a recipe for baking a cake. The recipe provides a list of ingredients and step-by-step instructions to mix them and bake the cake. This is an example of an algorithm because it outlines a clear sequence of steps to achieve the goal of baking a cake.
- **Example 2:** Planting a Tree: Here is a simple algorithm to plant a tree, an activity that can be very meaningful and beneficial:
  1. Choose a suitable spot in your garden.
  2. Dig a hole that is twice the width of the tree's root ball.
  3. Place the tree in the hole, making sure it is upright.
  4. Fill the hole with soil, pressing it down gently to remove air pockets.
  5. Water the tree generously to help it settle.
  6. Add mulch around the base of the tree to retain moisture.
  7. Water the tree regularly until it is established.

  This algorithm gives clear instructions on how to plant a tree, making it easy to follow for anyone.

Class activity

Let's create an algorithm! Think of something you do every day, like brushing your teeth or packing your school bag. Write down the steps you follow, one by one. Share your algorithm with your class and see if your friends can follow it!

Did You Know

Did you know that algorithms are not just used in computers? They are everywhere! When you follow directions to your friend's house or play a board game with rules, you are using algorithms. Algorithms help us solve problems logically.

Class activity

- Outline an algorithm for applying to the Board of Intermediate and Secondary Education (BISE) for 9th Grade Examination.

Algorithm Challenge

- Work in pairs to create an algorithm for a common task, such as making a sandwich or getting ready for school. Write down each step clearly, then exchange algorithms with another pair. Follow their algorithm exactly as written and see if you can complete the task.

Not for Sale PCTB

7.2 Principles of Computational Thinking

Computational thinking involves several key principles that guide the process of problem-solving in a structured manner.

7.2.1 Problem Understanding

Understanding a problem involves identifying the core issue, defining the requirements, and setting the objectives. Understanding the problem is the first and most important step in problem-solving, especially in computational thinking. This involves thoroughly analyzing the problem to identify its key components and requirements before attempting to find a solution.

"If I had an hour to solve a problem I'd spend 55 minutes thinking about the problem and 5 minutes thinking about solutions". — **Albert Einstein**

Importance of Problem Understanding:

- **Clarity and Focus:** By fully understanding the problem, you gain clarity on what needs to be solved. This helps you focus on the right aspects without getting distracted by irrelevant details.
- **Defining Goals:** Proper understanding of the problem allows you to define clear and achievable goals. You can determine what the final outcome should look like and set specific objectives to reach that outcome.
- **Efficient Solutions:** When you comprehend the problem well, you can devise more efficient and effective solutions. You can choose the best methods and tools to address the problem, saving time and resources.
- **Avoiding Mistakes:** By thoroughly understanding the problem, you can avoid common pitfalls and mistakes. Misunderstanding the problem can lead to incorrect solutions and wasted effort.

Example: Building a School Website

Imagine you are asked to build a website for your school. Before jumping into coding, you need to understand the problem:

1. **Identify Requirements:** What features does the website need? For example, pages for news, events, class schedules, and contact information.
2. **User Needs:** Who will use the website? Students, teachers, parents? Understanding your audience helps in designing user-friendly interfaces.
3. **Technical Constraints:** What resources and tools are available? Do you have access to a web server and the necessary software?

By understanding these aspects, you can plan and build a website that meets the needs of your school community.

Not for Sale PCTB

Tidbits

Always take time to thoroughly understand a problem before starting to solve it. Ask questions, gather information, and clarify any doubts. This foundational step will lead to better and more effective solutions.

7.2.2 Problem Simplification

Simplifying a problem involves breaking it down into smaller, more manageable sub-problems. Example: To design a website, break down the tasks into designing the layout, creating content, and coding the functionality.

7.2.3 Solution Selection and Design

Choosing the best solution involves evaluating different approaches and selecting the most efficient one. Designing the solution requires creating a detailed plan or algorithm.

7.3 Algorithm Design Methods

Algorithm design methods provide a range of tools and techniques to tackle various computational problems effectively. Each method has its strengths and weaknesses, making it suitable for different types of problems. Understanding different methods allows one to choose the most appropriate approach for a given problem, leading to more efficient and elegant solutions. Let's discuss two of these methods.

7.3.1 Flowcharts

Flowcharts are visual representations of the steps in a process or system, depicted using different symbols connected by arrows. They are widely used in various fields, including computer science, engineering, and business, to model processes, design systems, and communicate complex workflows clearly and effectively.

7.3.1.1 Importance of Flowcharts

- **Clarity**: Flowcharts provide a clear and concise way to represent processes, making them easier to understand at a glance.
- **Communication:** They are excellent tools for communicating complex processes to a wide audience, ensuring everyone has a common understanding.
- **Problem Solving:** Flowcharts help identify bottlenecks and inefficiencies in a process, aiding in problem-solving and optimization.
- **Documentation:** They serve as essential documentation for systems and

Not for Sale PCTB

processes, which is useful for training and reference purposes.

Did You Know

The first standardized flowchart symbols were developed in 1947 by the American National Standards Institute (ANSI).

7.3.1.2 Flowchart Symbols

Flowchart symbols are visual representations used to illustrate the steps and flow of a process or system as shown in Table 7.1.

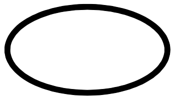
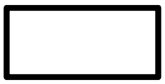
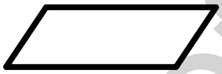
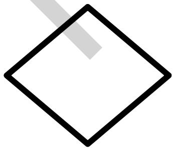
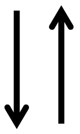
Symbol	Name	Description
	Oval (Terminal)	Represents the start or end of a process. Often labeled as "Start" or "End."
	Rectangle (Process)	Represents a process, task, or operation that needs to be performed.
	Parallelogram (Input/Output)	Represents data input or output (e.g., reading input from a user or displaying output on a screen).
	Diamond (Decision)	Represents a decision point in the process where the flow can branch based on a yes/no question or true/false condition.
	Arrow (Flowline)	Shows the direction of flow within the flowchart, connecting the symbols to indicate the sequence of steps.

Table 7.1: Flowchart symbols

Not for Sale PCTB

Did You Know

Flowcharts were popularized by computer scientists such as John von Neumann and Herman Goldstine in the early days of computing.

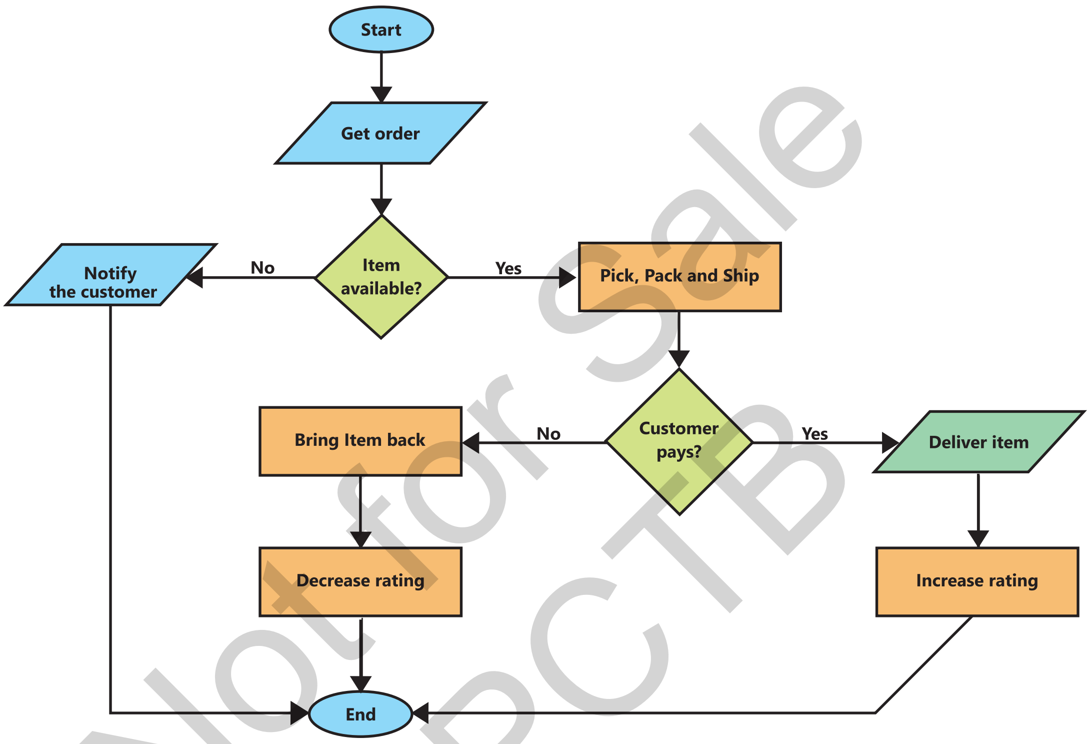


Figure 7.4: Flowchart of the Shop Order Process

Not for Sale PCTB

Class activity

Create a flowchart for a daily routine activity, such as getting ready for school. Include decision points like choosing what to wear based on the weather.

Example: A Shop Near Your House: Suppose a shop takes orders via cell phone messages. The flowchart in Figure 7.4 outlines the order processing steps. The input is the order, and the outputs are item delivery or a notification to the customer if the item is unavailable.

Decisions are made regarding item availability and customer payment. If the customer does not accept the item or make the payment, the item is returned to the shop, and the customer rating is decreased by 1. The customer's rating increases by 1 if they pay for the item. If the item is unavailable, the shop notifies the customer; otherwise, the shop picks, packs, and ships the item.

Enhancing Flowchart by Using Customer Rating

Note that while the customer rating is included in the flowchart shown in Figure 7.4, it is not utilized. Let's revise the flowchart to ensure only customers with a rating greater than 0 are attended to. The updated flowchart is shown in Figure 7.5.

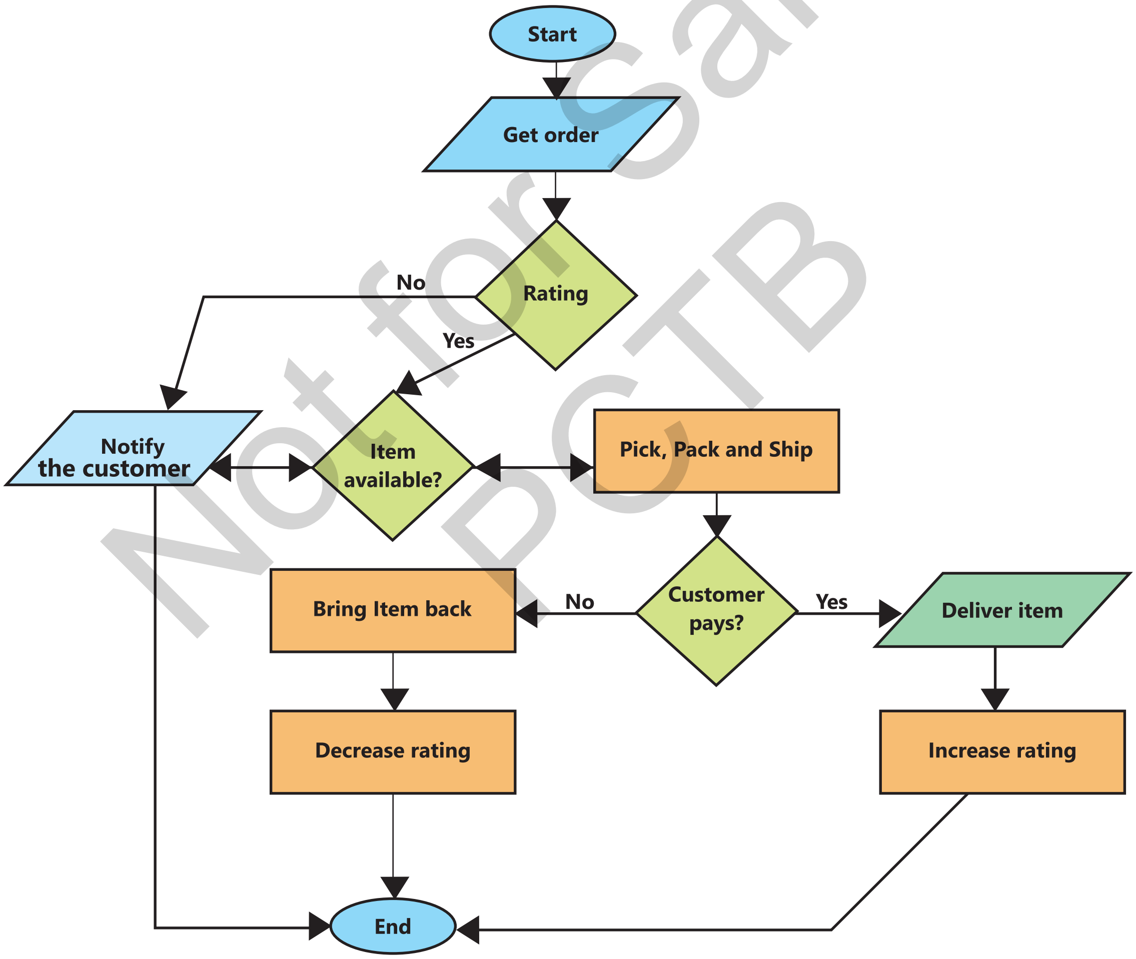


Figure 7.5: Flowchart of the shop using customer's rating

Not for Sale PCTB

Class activity

Modify Figure 7.5 to ensure that customer ratings are within the valid range of 0 to 5, inclusive. Ratings cannot be negative or exceed 5

Example: A flowchart for a login system showing steps such as inputting a username and password, verifying credentials, and granting access shown in Figure 7.6. A user can make a maximum of five attempts.

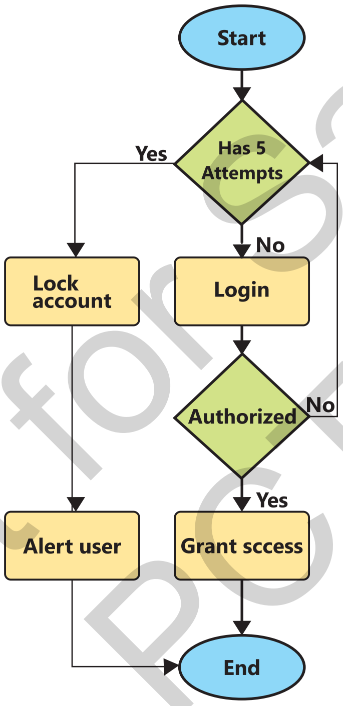


Figure 7.6: Flowchart for a login system

Not for Sale PCTB

Class activity

Draw a flowchart for selecting the school cricket team. The team can have a maximum of 11 players, and each player must have parental permission.

7.3.2 Pseudocode

Pseudocode is a method of representing an algorithm using simple and informal language that is easy to understand. It combines the structure of programming clearity with the readability of plain English, making it a useful tool for planning and explaining algorithms.

What is Pseudocode?

Pseudocode is not actual code that can be run on a computer, but rather a way to describe the steps of an algorithm in a manner that is easy to follow. It helps programmers and students focus on the logic of the algorithm without worrying about the syntax of a specific programming language.

Example-1

Determining whether a number is even or odd is a fundamental task in programming and computer science. An even number is divisible by 2 without any remainder, whereas an odd number has a remainder of 1 when divided by 2. Below is the pseudocode for this process, followed by an explanation.

Algorithm 1 Pseudocode for determining if a number is even or odd.

```
1:   Procedure CheckEvenOdd(number)
2:   Input: number {The number to be checked}
3:   Output: "Even" if number is even, "Odd" if number is odd
4:   Begin
5:       if (number % 2 == 0) then
6:           print "Even"
7:       else
8:           print "Odd"
9:       End if
10:  End
```

Not for Sale PCTB

Explanation

1. **Procedure Declaration:** The pseudocode begins with the declaration of the procedure 'CheckEvenOdd' which takes a single input, 'number'.
2. **Input:** The procedure accepts a variable 'number' which is the integer to be checked.
3. **Output:** The procedure outputs "Even" if the number is even, and "Odd" if the number is odd.
4. **Begin:** Mark the start of the procedure.
5. **Condition Check:** The condition 'if (number % 2 == 0)' checks if the remainder of the number when divided by 2 is zero. The modulo operator '%' is used for this purpose.
6. **Even Case:** If the condition is true, the procedure prints "Even".
7. **Odd Case**: If the condition is false, the procedure prints "Odd".
8. **End:** Marks the end of the procedure.

Example-2

Determining whether a number is prime is a fundamental task in number theory and computer science. A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. Below is the pseudocode for this process, followed by an explanation.

Algorithm 2 Pseudocode for determining if a number is prime.

```
Procedure Is Prime(number)
Input: number {The number to be checked}
Output: True if number is prime, False otherwise
Begin
    if (number <= 1) then
        return False
    end if
    for i from 2 to sqrt(number) do
        if (number % i == 0) then
            return False
        end if
    end for
    return True
End
```

Not for Sale PCTB

Explanation

1. **Procedure Declaration:** The pseudocode begins with the declaration of the procedure 'IsPrime' which takes a single input, 'number'.
2. **Input:** The procedure accepts a variable 'number', the integer to be checked.
3. **Output:** The procedure will output 'True' if the number is prime, and 'False' otherwise.
4. **Begin:** Mark the start of the procedure.
5. **Initial Check:** The condition 'if (number <= 1)' checks if the number is less than or equal to 1. If true, the procedure returns 'False' because numbers less than or equal to 1 are not prime.
6. **Loop Through Possible Divisors:** The 'for' loop iterates from 2 to the square root of the integer. This is because a greater factor of the number is a multiple of a previously tested smaller factor.
7. **Divisibility Check:** Inside the loop, the condition 'if (number % i == 0)'

checks if the number is divisible by 'i' without a remainder. If true, the procedure returns 'False' because the number has a divisor other than 1 and itself.

8. **Prime Confirmation:** If no divisors are found in the loop, the procedure returns 'True', confirming the number is prime.
9. **End:** Marks the end of the procedure.

Class activity

Create Your Own Pseudocode: Divide the students into small groups and assign each group a different simple problem, such as finding the maximum number in a list or calculating the factorial of a number. Ask them to write the pseudocode for their assigned problem and then present it to the class.

Not for Sale PCTB

Did You Know

Pseudocode is often used in software development before writing the actual code to ensure that the logic is sound and to facilitate communication between team members who may be using different programming languages.

Why Use Pseudocode?

Using pseudocode has several benefits:

- **Clarity:** It helps in understanding the logic of the algorithm without worrying about syntax.
- **Planning:** It allows programmers to outline their thoughts and plan the steps of the algorithm.
- **Communication:** It is a universal way to convey the steps of an algorithm, making it easier to discuss with others.
-

7.3.3 Differentiating Flowcharts and Pseudocode

Flowcharts and pseudocode are both tools used to describe algorithms, but they do so in different ways. Understanding their differences can help you decide which method is more suitable to use for your scenario.

Pseudocode	Flowcharts
• Pseudocode uses plain language and structured format to describe the steps of an algorithm. • It is read like a story, with each step is written out sequentially. • Pseudocode communicates the steps in a detailed, narrative -like format. • It is particularly useful for documenting algorithms in a way that can be easily converted into actual code in any programming language.	• Flowcharts use graphical symbols and arrows to represent the flow of an algorithm. • It is like watching a movie, where each symbol (such as rectangles, diamonds, and ovals) represents a different type of action or decision, and arrows indicate the connection and direction of the flow. • Flowchart c ommunicates the process in a visual format, which can be more intuitive for understanding the overall flow and structure. • They are useful for identifying the steps and decisions in an algorithm at a glance.

Table 7.2 Difference between Pseudocode and Flowcharts

Not for Sale PCTB

Example-3

Algorithm 3 presents the pseudocode for checking a valid username and password.

```
1.  Procedure CheckCredentials(username, password)
2.  Input: username, password
3.  Output: Validity message
4.  Begin
5.  validUsername = "user123" {Replace with the actual valid username}
6.  validPassword = "pass123" {Replace with the actual valid password}
7.  if (username == validUsername) then
```

```
8:          if (password == validPassword) then
9:                    print "Login successful"
10:         else
11:                   print "Invalid password"
12:     end if
13:     else
14:         print "Invalid username"
15:     end if
16:     End
```

7.4 Algorithmic Activities

7.4.1 Design and Evaluation Techniques

Techniques to essential algorithms are essential to understand how efficiently they solve problems. In this section, we will explore different techniques for evaluating algorithms, focusing on their time and space complexities.

7.4.1.1 Time Complexity

Time Complexity measures how fast or slow an algorithm performs. It shows how the running time of an algorithm changes as the size of the input increases. Here's an easy way to understand it:

Imagine you have a list of names, and you want to find a specific name. If you have 10 names, it might only take a few seconds to look through the list. But what if you have 100 names? Or 1,000 names? The time it takes to find the name increases as the list gets longer. Time complexity helps us understand this increase.

Not for Sale PCTB

Did You Know

Time complexity is usually expressed using Big *O* notation, like $O(n)$, $O(logn)$, or $O(n^2)$. It helps us compare different algorithms to see which one is faster!

Tidbits

When writing an algorithm, consider how many steps it takes to complete the task. Fewer steps means a faster algorithm!

Class activity

Think of a simple task, like finding the largest number in a list. Write down the steps you would take to complete this task. Now, imagine the list has 10 numbers, then 100 numbers. How do the steps change?

Did You Know

Some algorithms can perform the same task much faster than others. For example, sorting a list of 100 items might take one algorithm 1 second and another algorithm 10 seconds!

7.4.1.2 Space Complexity

Space complexity measures the amount of memory an algorithm uses relative to input size. It is essential to consider both the memory required for the input and any extra memory used by the algorithm.

Designing and evaluating algorithms involves activities like dry runs and simulations to ensure they work as intended.

7.5 Dry Run

A dry run involves manually going through the algorithm with sample data to identify any errors.

7.5.1 Dry Run of a Flowchart

A dry run of a flowchart involves manually walking through the flowchart step-by-step to understand how the algorithm works without using a computer. This helps identify any logical errors and understand the flow of control.

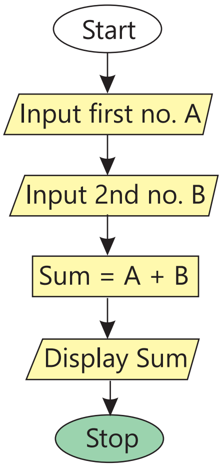


Figure 7.7: Flowchart for adding two numbers

Example: Calculating the Sum of Two Numbers

Consider the flowchart given in figure 7.7 for adding two numbers:

Steps to dry run this flowchart:

1. Start
2. Input the first number (e.g., 3)
3. Input the second number (e.g., 5)
4. Add the two numbers (3 + 5 = 8)

Class activity

Dry Run a Flowchart

Draw a flowchart for finding the largest of two numbers. Perform a dry run for the numbers 7 and 4. Write down each step and the values of variables.

Not for Sale PCTB

5. Output the result (8)
6. Stop

7.5.2 Dry Run of Pseudocode

A dry run of pseudocode involves manually simulating the execution of the pseudocode line-by-line.

This helps in verifying the logic and correctness of the algorithm.

Example: Finding the Maximum of Two Numbers

Consider the pseudocode for finding the maximum of two numbers:

Did You Know

Did you know that different algorithms can solve the same problem more efficiently? For instance, one algorithm might quickly find the highest marks in a list, while another might take much longer. Learning how to evaluate and choose the best algorithm is a key skill in computer science!

Algorithm 4 FindMax

```
1. Input: num1, num2
2. if num1 > num2 then
3.     max = num1
4. else
5.     max = num2
6. end if
7. Output: max
```

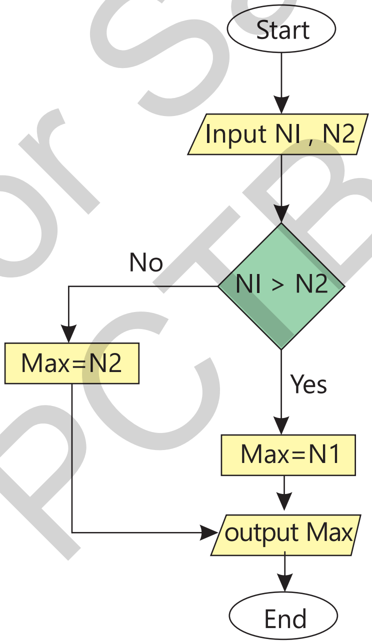
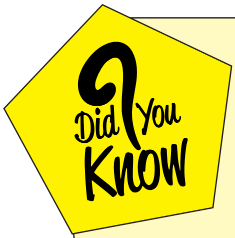


Figure 7.8: Flowchart for finding maximum of two numbers

Steps to dry run this pseudocode:

1. Input num1 and num2 (e.g., 10 and 15)
2. Check if num1 > num2 (10>15: False)
3. Since the condition is False, max = num2 (max = 15)
4. Output max (15)

Not for Sale PCTB



- Dry running your code or algorithm helps catching errors early in the development process, saving time and effort.
- Many professional programmers and computer scientists use dry running as a debugging technique to ensure their algorithms work correctly!

7.5.3 Simulation

Simulation is we use of computer programs to create a model of a real-world process or system. This helps us understand how things work by testing different ideas or algorithms without needing to try them out in real life.

Why Use Simulation?

1. **Testing Algorithms:** We can use simulation to see how well an algorithm works with different types of data. For example, if we want to test a new way to sort numbers, we can simulate it with different sets of numbers to see how fast it is.
2. **Exploring Scenarios:** Simulation allows us to create many different situations to see what happens. For example, in a science experiment about plant growth, we can simulate different amounts of water or sunlight to find out which conditions help plants grow best.

Benefits of Simulation

- **Cost-Effective:** It is often cheaper and faster to run simulations than to conduct real experiments.
- **Safe:** We can test dangerous situations, like a fire in a building, without putting anyone at risk.
- **Repeatable:** We can run the same simulation multiple times with different settings to observe how things change.

Examples of Simulation

1. **Weather Forecasting:** Meteorologists use simulations to predict the weather. They input data about temperature, humidity, and wind speed into a computer model to see how the weather might change over the next few days.
2. **Traffic Flow:** City planners can simulate traffic to see how changes to roads or traffic lights might affect the flow of cars. This helps them design better roads and reduce traffic jams.

Not for Sale PCTB

7.6 Introduction to LARP (Logic of Algorithms for Resolution of Problems)

LARP stands for Logic of Algorithms for resolution of Problems. It is a fun and interactive way to learn how algorithms work by actually running them and seeing the results. Think of it as a playground where you can experiment with different algorithms and understand how they process data.

Did You Know

For the latest versions and updates of LARP software, check trusted educational and coding platforms, or search for "LARP software download" on your favorite search engine.

Not for Sale PCTB

7.6.1 Why is LARP Important?

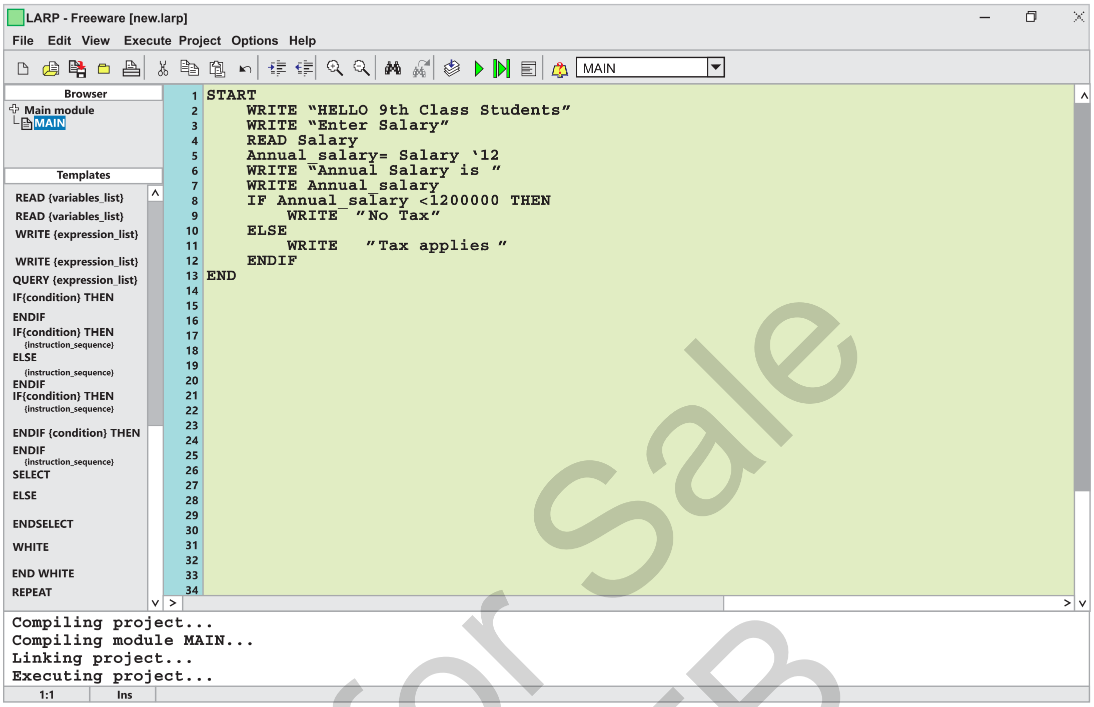
LARP helps you:

- Understand how algorithms work. For instance, refer to Figure 7.9, which illustrates an algorithm designed to determine the applicability of tax on the annual salary of a person.
- See the effect of different inputs on the output.
- Practice writing and improving your own algorithms.

7.6.2 Writing Algorithms

Writing algorithms using LARP involves a structured and simplified approach to developing logical solutions for computational problems. LARP employs a clear syntax that begins with a START command and ends with an END command, ensuring that each step of the algorithm is easy to follow. Within this framework, instructions are provided in a straightforward manner, such as using WRITE to display messages, READ to input values, and conditional statements like IF...THEN...ELSE to handle decision-making processes. By breaking down complex problems into manageable steps, LARP allows learners to focus on the logical flow of the algorithm without getting stuck on complex coding syntax. This method not only aids in understanding the fundamental concepts of algorithm design but also enhances problem-solving skills by encouraging clear and logical thinking.

Here's an example of a simple algorithm to check if a number is even or odd:



```
START
    WRITE "HELLO 9th Class Students"
    WRITE "Enter Salary"
    READ Salary
    Annual_salary= Salary '12
    WRITE "Annual Salary is "
    WRITE Annual_salary
    IF Annual_salary <1200000 THEN
        WRITE "No Tax"
    ELSE
        WRITE "Tax applies "
    ENDIF
END
```

```
Compiling project...
Compiling module MAIN...
Linking project...
Executing project...
```

Figure 7.9: LARP Software

Not for Sale PCTB

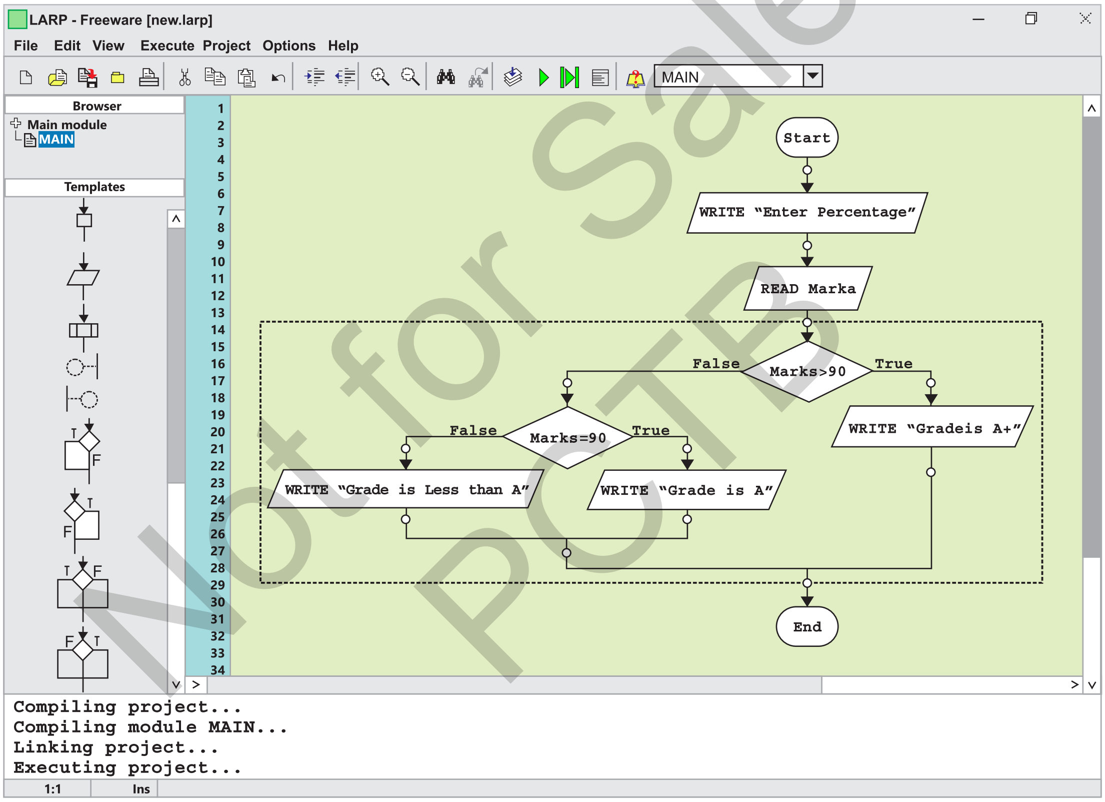
```
START
    WRITE "Enter a number"
    READ number
    IF number % 2 == 0 THEN
        WRITE "The number is even"
    ELSE
        WRITE "The number is odd"
    ENDIF
END
```

7.6.3 Drawing Flowcharts in LARP

Drawing flowcharts in LARP involves visually representing the algorithm's steps using standard flowchart symbols such as rectangles for processes, diamonds for decisions, and parallelograms for input/output operations. Once the flowchart is created, it can be executed in LARP by translating the flowchart into LARP syntax, which uses straightforward commands like START, WRITE, READ, IF...THEN...ELSE, and END. This process allows students to visualize the logic of their algorithm and see its step-by-step execution. For example, Figure 7.9 shows a flowchart for determining whether a student's grade is above 'A' or not. We can execute the flowchart to verify its correctness. This hands-on approach reinforces understanding of how a flowchart works.

7.7 Error Identification and Debugging

When we write algorithms or create flowcharts in LARP, we sometimes make mistakes called errors or bugs. These mistakes can prevent our algorithms from functioning correctly. Error handling and debugging are processes that help us find and fix these errors.



```
Compiling project...
Compiling module MAIN...
Linking project...
Executing project...
```

Not for Sale PCTB

Figure 7.10: Flowchart in LARP

7.7.1 Types of Errors

There are three main types of errors you might encounter:

- **Syntax Errors:** These occur when we write something incorrectly in our algorithm or flowchart. For example, missing a step or using the wrong symbol.
- **Runtime Errors:** These happen when the algorithm or flowchart is being

executed. For example, trying to perform an impossible operation, such as dividing by zero.

- Logical Errors: These are mistakes in the logic of the algorithm that cause it to behave incorrectly. For example, using the wrong condition in a decision step.

Did You Know

Syntax errors are the easiest to find because the LARP tool usually points them out. However, logical errors are the hardest to find because the algorithm still runs but does not procedure correct answers.

7.7.2 Debugging Techniques

Debugging is the process of finding and fixing errors in an algorithm or flowchart. Here are some common debugging techniques:

- **Trace the Steps:** Go through each step of your algorithm or flowchart to see identity where it goes wrong.
- **Use Comments:** Write comments or notes in your algorithm to explain what each part is supposed to do. This can help you spot mistakes.
- **Check Conditions:** Ensure that all conditions in decision steps are correct.
- **Simplify the Problem:** Break down the algorithm into smaller parts and test each part separately.

Tidbits

Always read error messages carefully. They often tell you exactly where the problem is.

Not for Sale PCTB

7.7.3 Common Error Messages in LARP

Here are some common error messages you might see in LARP and what they mean:

- **Missing Step -** You probably forgot to include an important step in your algorithm.
- **Undefined Variable -** You are using a variable that hasn't been defined yet.
- **Invalid Operation -** You are trying to perform an operation that is not allowed, like dividing by zero.

Class activity

Create a simple flowchart in LARP that calculates the average of three numbers. Introduce a syntax error, a runtime error, and a logical error in your flowchart. Then, try to fix them using the debugging techniques we discussed.

Did You Know

The term "debugging" comes from an actual bug—a moth—that was found causing problems in an early computer. The moth was removed, and the process was called "debugging"

Summary

- Computational thinking is important skill that enables individuals to solve complex problems using methods that mirror the processes involved in computer science.
- Decomposition is the process of breaking down a complex problem into smaller, more manageable parts.
- Pattern recognition involves looking for similarities or patterns among and within problems.
- Abstraction involves simplifying complex problems by breaking them down into smaller, more manageable part, and focusing only on the essential details while ignoring the unnecessary ones.
- An algorithm is a step-by-step set of instructions to solve a problem or complete a task.
- Understanding the problem is the first and most important step in problem-solving, especially in computational thinking.
- Simplifying a problem involves breaking it down into smaller, more manageable sub-problems.
- Choosing the best solution involves evaluating different approaches and selecting the most efficient one.
- Flowcharts are visual representations of the steps in a process or system, depicted using different symbols connected by arrows.
- Pseudocode is a way of representing an algorithm using simple and informal language that is easy to understand. It combines the structure of programming languages with the readability of plain English, making it a useful tool for planning and explaining algorithms.
- Time Complexity is a way to measure how fast or slow an algorithm performs. It tells us how the running time of an algorithm changes as the

Not for Sale PCTB

size of the input increases.

- Space complexity measures the amount of memory an algorithm uses in relation to the input size. It is important to consider both the memory needed for the input and any additional memory used by the algorithm.
- A dry run involves manually going through the algorithm with sample data to identify any errors.
- Simulation is when we use computer programs to create a model of a real-world process or system.
- LARP stands for logic of Algorithm for Resolution of Problems. It is a fun and interactive way to learn how algorithms work by actually running them and seeing the results.
- Debugging is the process of finding and fixing errors in an algorithm or flowchart.

EXERCISE

Multiple Choice Questions

1. Which of the following best defines computational thinking?
   (a) A method of solving problems using mathematical calculations only.
   (b) A problem-solving approach that employs systematic, algorithmic, and logical thinking.
   (c) A technique used exclusively in computer programming.
   (d) An approach that ignores real-world applications.
2. Why is problem decomposition important in computational thinking?
   (a) It simplifies problems by breaking them down into smaller, more manageabl parts.
   (b) It complicates problems by adding more details.
   (c) It eliminates the need for solving the problem.
   (d) It is only useful for simple problems.
3. Pattern recognition involves:
   (a) Finding and using similarities within problems
   (b) Ignoring repetitive elements
   (c) Breaking problems into smaller pieces
   (d) Writing detailed algorithms
4. Which term refers to the process of ignoring the details to focus on the main idea?
   (a) Decomposition (b) Pattern recognition
   (c) Abstraction (d) Algorithm design

Not for Sale PCTB

5. Which of the following is a principle of computational thinking?
   (a) Ignoring problem understanding (b) Problem simplification
   (c) Avoiding solution design (d)Implementing random solutions
6. Algorithms are:
   (a) Lists of data
   (b) Graphical representations
   (c) Step-by-step instructions for solving a problem
   (d) Repetitive patterns
7. Which of the following is the first step in problem-solving according to computational thinking?
   (a) Writing the solution (b) Understanding the problem
   (c) Designing a flowchart (d) Selecting a solution
8. Flowcharts are used to:
   (a) Code a program
   (b) Represent algorithms graphically
   (c) Solve mathematical equations
   (d) Identify patterns
9. Pseudocode is:
   (a) A type of flowchart
   (b) A high-level description of an algorithm using plain language
   (c) A programming language
   (d) A debugging tool
10. Dry running a flowchart involves:
   (a) Writing the code in a programming language
   (b) Testing the flowchart with sample data
   (c) Converting the flowchart into pseudocode
   (d) Ignoring the flowchart details

Not for Sale PCTB

Short Questions

1. Define computational thinking.
2. What is decomposition in computational thinking?
3. Explain pattern recognition with an example.
4. Describe abstraction and its importance in problem-solving.
5. What is an algorithm?
6. How does problem understanding help in computational thinking?
7. What are flowcharts and how are they used?
8. Explain the purpose of pseudocode.
9. How do you differentiate between flowcharts and pseudocode?
10. What is a dry run and why is it important?
11. Describe LARP and its significance in learning algorithms.
12. List and explain two debugging techniques.

Long Questions

1. Write an algorithm to assign a grade based on the marks obtained by a student. The grading system follows these criteria:
    - 90 and above: A+
    - 80 to 89: A
    - 70 to 79: B
    - 60 to 69: C
    - Below 60: F
2. Explain how you would use algorithm design methods, such as flowcharts and pseudocode, to solve a complex computational problem. Illustrate your explanation with a detailed example.
3. Define computational thinking and explain its significance in modern problem-solving. Provide examples to illustrate how computational thinking can be applied in different fields.
4. Discuss the concept of decomposition in computational thinking. Why is it important?
5. Explain pattern recognition in the context of computational thinking. How does identifying patterns help in problem-solving?
6. What is an abstraction in computational thinking? Discuss its importance and provide examples of how abstraction can be used to simplify complex problems.
7. Describe what an algorithm is and explain its role in computational thinking. Provide a detailed example of an algorithm for solving a specific problem, and draw the corresponding flowchart.
8. Compare and contrast flowcharts and pseudocode as methods for algorithm design. Discuss the advantages and disadvantages of each method, and provide examples where one might be preferred over the other.
9. Explain the concept of a dry run in the context of both flowcharts and pseudocode. How does performing a dry run help in validating the correctness of an algorithm?
10. What is LARP? Discuss its importance in learning and practicing algorithms.
11. How does LARP enhance the understanding and application of computational thinking principles? Provide a scenario where LARP can be used to improve an algorithm.

Not for Sale PCTB