Computational Thinking & Algorithm

S.L.O. # 2
Sub Topics: 2 Total SLO: 4
MCQ: (8) 8 Marks CRQ: (1) 3 Marks ERQ: (0) 0 Marks

Prepared By: Naseer Ahmad (Nusrat Jahan Boys College) 0333-9790903

2.1 Computational Thinking and Artefacts

SLO	Students should be able to	Cognitive Level
2.1.1	define the term computational thinking;	R
2.1.2	describe different steps of computational thinking;	U
2.1.3	explain the following computational artefacts of software development process: a. computational solution design, b. planning and development of computational artefacts, c. testing computational artefacts;	U

SLO 2.1.1 R

define the term computational thinking;

Prepared By: Naseer Ahmad (Nusrat Jahan Boys College) 0333-9790903

Computational Thinking

SLO 2.1.1 R

- Computational thinking is a problem-solving process that involves understanding a problem and expressing its solutions in a way that a computer or a human can effectively execute.
- It includes skills such as:
  - Decomposition: Breaking down a complex problem into smaller, manageable parts.
  - Pattern recognition: Identifying similarities or patterns to make solving easier.
  - Abstraction: Focusing on the important information only, ignoring irrelevant details.
  - Algorithm design: Creating a step-by-step solution or rules to solve the problem.
- In short, computational thinking is a method of thinking logically and systematically to solve problems, often using concepts from computer science.

Prepared By: Naseer Ahmad (Nusrat Jahan Boys College) 0333-9790903

SLO 2.1.2 U

describe different steps of computational thinking;

Prepared By: Naseer Ahmad (Nusrat Jahan Boys College) 0333-9790903

Different Steps of Computational Thinking

SLO 2.1.2 U

Computational thinking is a structured approach to problem-solving, widely applied in computer science. It consists of four key steps: decomposition, pattern recognition, abstraction, and algorithm design.

Decomposition

- Decomposition involves breaking a complex computing problem into smaller, manageable sub-components that can be solved independently.
- Example: Developing a website. Instead of tackling the entire project at once, you decompose it into:
  - Designing the user interface (e.g., HTML/CSS for layout and styling).
  - Creating the backend logic (e.g., setting up a database and server with Python or Node.js).
  - Implementing user authentication (e.g., login/signup functionality).
  - Testing and debugging (e.g., ensuring cross-browser compatibility).
  - For instance, you might focus on coding the navigation bar first, then move to database integration, addressing each piece systematically.

Different Steps of Computational Thinking

SLO 2.1.2 U

Pattern Recognition

- This step involves identifying similarities or trends in data or processes within a computing problem, allowing you to apply solutions efficiently across similar cases.
- Example: Optimizing a search function in a database.
  - Suppose you're building a search feature for an e-commerce app.
  - You notice that users frequently search for products by category (e.g., "laptops" or "phones") and price range.
  - By recognizing this pattern, you can index the database by these fields to speed up queries.
  - This reduces redundant work and improves performance for repeated search patterns.

Different Steps of Computational Thinking

SLO 2.1.2 U

Abstraction

- Abstraction focuses on extracting essential details while ignoring irrelevant ones, creating simplified models or representations of a computing problem.
- Example: Creating a file compression algorithm.
  - When designing a tool like ZIP, you abstract the problem to focus on key elements like file size and data redundancy, ignoring specifics like file names or creation dates unless needed.
  - For instance, you might use Huffman coding to assign shorter binary codes to frequent data patterns, abstracting away complex file content into a manageable encoding scheme, making compression efficient.

Different Steps of Computational Thinking

SLO 2.1.2 U

Algorithm Design

- Algorithm design involves creating a step-by-step set of instructions (an algorithm) to solve a computing problem, ensuring clarity and efficiency.
- Example: Writing a program to find the shortest path in a navigation app (e.g., like Google Maps). You could use Dijkstra's algorithm:
  - Start with a graph of roads (nodes as intersections, edges as roads with weights as distances).
  - Initialize the starting node's distance as 0, others as infinity.
  - Repeatedly select the node with the smallest distance, update distances to its neighbors, and mark it as visited.
  - Continue until you reach the destination, tracing the shortest path (e.g., output: "Turn left at X, then right at Y, total 5.2 km").
  - This algorithm can be coded in Python or C++ and ensures consistent, optimal route calculations.

SLO 2.1.3 U

explain the following computational artefacts of software development process:

a. computational solution design,
b. planning and development of computational artefacts,
c. testing computational artefacts;

Computational Artefacts of Software Development

SLO 2.1.3 U

a. Computational Solution Design

- This is the phase where the problem is analyzed and a detailed plan or blueprint for the software solution is created. It involves:
  - Understanding the requirements and constraints of the problem.
  - Designing algorithms, data structures, and overall system architecture.
  - Creating models such as flowcharts, pseudocode, or UML diagrams to represent how the software will solve the problem. The goal is to provide a clear, logical outline of how the software will function before actual coding begins.

Computational Artefacts of Software Development

SLO 2.1.3 U

b. Planning and Development of Computational Artefacts

- This phase focuses on the actual creation of the software components (computational artefacts) based on the solution design. It includes:
  - Writing source code or scripts in a programming language.
  - Developing modules, functions, or classes that implement parts of the solution.
  - Organizing code and resources effectively.
  - Managing version control and collaboration if in a team.
  - Planning here also involves deciding on timelines, resources, tools, and technologies to be used for development.

Computational Artefacts of Software Development

SLO 2.1.3 U

c. Testing Computational Artefacts

- Testing ensures that the developed software works correctly and meets the specified requirements. It involves:
  - Running the software with various inputs to check for bugs, errors, or unexpected behavior.
  - Performing different types of testing such as unit testing (testing individual components), integration testing (testing combined parts), and system testing (testing the complete system).
  - Verifying that the software handles edge cases and error conditions properly.
  - Refining and fixing issues identified during testing to improve reliability and performance.

2.2 Algorithm

SLO	Students should be able to	Cognitive Level
2.2.1	apply basic computational algorithms to perform the following operations in programming: a. arithmetic, relational, and logical computations, b. conditional (selection) statements, c. iterative (loop-based) processes, d. sorting techniques (insertion sort, bubble sort), e. searching techniques (binary search, linear search).	A

SLO 2.2.1 A

apply basic computational algorithms to perform the following operations in programming:
a.arithmetic, relational, and logical computations,
b.conditional (selection) statements,
c.iterative (loop-based) processes,
d.sorting techniques (insertion sort, bubble sort),
e.searching techniques (binary search, linear search).

Basic Computational Algorithms

SLO 2.2.1 A

a. Arithmetic, Relational, and Logical Computations

- Arithmetic computations involve basic mathematical operations like addition (+), subtraction (−), multiplication (*), division (/), and modulus (%).
- Relational computations compare two values and return true or false, using operators like ==, !=, <, >, <=, >=.
- Logical computations combine relational expressions using logical operators such as AND (&& or and), OR (| | or or), and NOT (! or not).

Basic Computational Algorithms

SLO 2.2.1 A

a. Arithmetic, Relational, and Logical Computations

```
a = 10
b = 5

# Arithmetic
sum = a + b       # 15
diff = a - b      # 5

# Relational
is_greater = a > b  # True

# Logical
result = (a > b) and (b > 0)  # True
```

Basic Computational Algorithms

SLO 2.2.1 A

b. Conditional (Selection) Statements

- Used to execute different blocks of code based on a condition.

```
if a > b:
    print("a is greater")
elif a == b:
    print("a and b are equal")
else:
    print("b is greater")
```

Basic Computational Algorithms

SLO 2.2.1 A

c. Iterative (Loop-based) Processes

- Loops repeat a block of code while a condition holds true.
- For loop: Runs a set number of times.
- While loop: Runs while a condition remains true.

```
# For loop
for i in range(5):
    print(i)

# While loop
count = 0
while count < 5:
    print(count)
    count += 1
```

Basic Computational Algorithms

SLO 2.2.1 A

d. Sorting Techniques

- Insertion Sort Algorithm
- Goal: Sort an array by repeatedly inserting elements into their correct position.
- Algorithm:
  1. For each element from the second element (index 1) to the last:
     a. Store the current element as key.
     b. Compare key with elements to its left.
     c. Shift all elements greater than key one position to the right.
     d. Insert key into the correct position.
  2. Repeat until the entire array is sorted.

Basic Computational Algorithms

SLO 2.2.1 A

d. Sorting Techniques

- Insertion Sort Pseudocode

```
procedure insertionSort(array)
    n = length of array
    for i from 1 to n-1
        key = array[i]
        j = i - 1
        while j >= 0 and array[j] > key
            array[j + 1] = array[j]  // Shift larger elements right
            j = j - 1
        array[j + 1] = key  // Insert key in correct position
    end for
end procedure
```

Basic Computational Algorithms

SLO 2.2.1 A

d. Sorting Techniques

- Insertion Sort Example
- Initial array: [64, 34, 25, 12, 22]
- Sorted portion: Initially, just [64]. Unsorted: [34, 25, 12, 22].
- Step 1: Take 34
  - Compare 34 with 64 (sorted portion). Since 34 < 64, shift 64 right: [_, 64, 25, 12, 22]
  - Insert 34 in position 0: [34, 64, 25, 12, 22]
  - Sorted portion: [34, 64]

Basic Computational Algorithms

SLO 2.2.1 A

d. Sorting Techniques

- Insertion Sort Example
- Step 2: Take 25
  - Compare 25 with 64. Since 25 < 64, shift 64 right: [34, _, 64, 12, 22]
  - Compare 25 with 34. Since 25 < 34, shift 34 right: [_, 34, 64, 12, 22]
  - Insert 25 in position 0: [25, 34, 64, 12, 22]
  - Sorted portion: [25, 34, 64]

Basic Computational Algorithms

SLO 2.2.1 A

d. Sorting Techniques

- Insertion Sort Example
- Step 3: Take 12
  - Compare 12 with 64, 34, 25 (all larger), shift them right: [_, 25, 34, 64, 22]
  - Insert 12 in position 0: [12, 25, 34, 64, 22]
  - Sorted portion: [12, 25, 34, 64]

Basic Computational Algorithms

SLO 2.2.1 A

d. Sorting Techniques

- Insertion Sort Example
- Step 4: Take 22
  - Compare 22 with 64, 34, 25 (all larger), shift them right: [12, _, 25, 34, 64]
  - Compare 22 with 12. Since 22 > 12, insert 22 in position 1: [12, 22, 25, 34, 64]
  - Sorted portion: [12, 22, 25, 34, 64]
- Final sorted array: [12, 22, 25, 34, 64]

Basic Computational Algorithms

SLO 2.2.1 A

d. Sorting Techniques

- Bubble Sort Algorithm
- Goal: Sort an array by repeatedly swapping adjacent elements if they are in the wrong order.
- Algorithm:
  1. Repeat for n-1 passes (where n is the length of the array):
     a. For each pair of adjacent elements:
        - If the left element is greater than the right element, swap them.
  2. After each pass, the largest element bubbles up to its correct position at the end.
  3. Repeat until no swaps are needed (array is sorted).

Basic Computational Algorithms

SLO 2.2.1 A

d. Sorting Techniques

- Bubble Sort Pseudocode

```
procedure bubbleSort(array)
    n = length of array
    for i from 0 to n-1
        swapped = false  // Optimization flag
        for j from 0 to n-i-2
            if array[j] > array[j+1]
                swap array[j] and array[j+1]
                swapped = true
        if not swapped
            break  // Array is sorted
    end for
end procedure
```

Basic Computational Algorithms

SLO 2.2.1 A

d. Sorting Techniques

- Bubble Sort Example
- Consider sorting the array [64, 34, 25, 12, 22] in ascending order
- Initial array: [64, 34, 25, 12, 22]
- Pass 1:
  - Compare 64 > 34? Yes → Swap: [34, 64, 25, 12, 22]
  - Compare 64 > 25? Yes → Swap: [34, 25, 64, 12, 22]
  - Compare 64 > 12? Yes → Swap: [34, 25, 12, 64, 22]
  - Compare 64 > 22? Yes → Swap: [34, 25, 12, 22, 64]
  - End of pass: Largest element (64) is at the end. Unsorted portion now: first 4 elements.

Basic Computational Algorithms

SLO 2.2.1 A

d. Sorting Techniques

- Bubble Sort Example
- Pass 2:
- Compare 34 > 25? Yes → Swap: [25, 34, 12, 22, 64]
- Compare 34 > 12? Yes → Swap: [25, 12, 34, 22, 64]
- Compare 34 > 22? Yes → Swap: [25, 12, 22, 34, 64]
- End of pass: Second largest (34) is in place. Unsorted portion: first 3 elements.

Basic Computational Algorithms

SLO 2.2.1 A

d. Sorting Techniques

- Bubble Sort Example
- Pass 3:
  - Compare 25 > 12? Yes → Swap: [12, 25, 22, 34, 64]
  - Compare 25 > 22? Yes → Swap: [12, 22, 25, 34, 64]
  - End of pass: Third largest (25) is in place. Unsorted portion: first 2 elements.
- Pass 4:
  - Compare 12 > 22? No → No swap.
  - End of pass: Already sorted.
- Final sorted array: [12, 22, 25, 34, 64]elements.

Basic Computational Algorithms

SLO 2.2.1 A

e. Searching Techniques

- Linear Search Algorithm
- Goal: Find a target element by checking each element sequentially.
- Algorithm:
  1. Start at the first element of the array.
  2. Compare the current element to the target.
  3. If they match, return the index of the current element.
  4. If not, move to the next element.
  5. Repeat steps 2–4 until the target is found or the end of the array is reached.
  6. If the target is not found, return -1.

Basic Computational Algorithms

SLO 2.2.1 A

e. Searching Techniques

- Linear Search Pseudocode

```
procedure linearSearch(array, target)
    n = length of array
    for i from 0 to n-1
        if array[i] = target
            return i  // Return index of target
    return -1  // Target not found
end procedure
```

Basic Computational Algorithms

SLO 2.2.1 A

e. Searching Techniques

- Linear Search Example
- Initial array: [64, 34, 25, 12, 22]
- Target: 25
- Step 1: Check index 0: 64 ≠ 25 → Move to next element.
- Step 2: Check index 1: 34 ≠ 25 → Move to next element.
- Step 3: Check index 2: 25 = 25 → Target found! Return index 2.
- Result: Target 25 is at index 2.
- If the target was 99:
  - Check all elements: 64, 34, 25, 12, 22 (none match 99).
  - Reach the end of the array → Return -1 (not found).

Basic Computational Algorithms

SLO 2.2.1 A

e. Searching Techniques

- Binary Search Algorithm
- Goal: Efficiently find a target element in a sorted array by dividing the search interval in half repeatedly.
- Algorithm:
  1. Initialize two pointers: low = 0, high = n - 1 (where n is the array length).
  2. While low is less than or equal to high:
     a. Calculate mid = (low + high) // 2.
     b. If the element at mid equals the target, return mid.
     c. If the element at mid is less than the target, set low = mid + 1.
     d. Else, set high = mid - 1.If the target is not found, return -1.

Basic Computational Algorithms

SLO 2.2.1 A

e. Searching Techniques

- Binary Search Pseudocode

```
procedure binarySearch(array, target)
    low = 0
    high = length of array - 1
    while low <= high
        mid = (low + high) // 2
        if array[mid] = target
            return mid  // Target found
        else if array[mid] > target
            high = mid - 1  // Search lower half
        else
            low = mid + 1  // Search upper half
    return -1  // Target not found
end procedure
```

Basic Computational Algorithms

SLO 2.2.1 A

e. Searching Techniques

- Binary Search Example
- Initial array: [12, 22, 25, 34, 64] (already sorted)
- Target: 25
- Initial pointers: low = 0, high = 4 (array length - 1)
- Step 1:
  - mid = (0 + 4) // 2 = 2
  - Check array[2] = 25.
  - Matches target → Return index 2.
  - Result: Target 25 is at index 2.

Basic Computational Algorithms

SLO 2.2.1 A

e. Searching Techniques

- Binary Search Example
- Now, consider searching for 50:
- Step 1:
  - mid = (0 + 4) // 2 = 2
  - array[2] = 25 < 50 → Search upper half: low = 2 + 1 = 3, high = 4
- Step 2:
  - mid = (3 + 4) // 2 = 3
  - array[3] = 34 < 50 → Search upper half: low = 3 + 1 = 4, high = 4
- Step 3:
  - mid = (4 + 4) // 2 = 4
  - array[4] = 64 > 50 → Search lower half: low = 4, high = 4 - 1 = 3
- Step 4:
  - low = 4 > high = 3 → Target not found. Return -1.

Prepared By: Naseer Ahmad (Nusrat Jahan Boys College) 0333-9790903

Prepared By: Naseer Ahmad (Nusrat Jahan Boys College) 0333-9790903