

Date: - 10th April 2024 Day 2

Topic:

1. Pseudocode for Bubble and Insertion sort.
2. Pseudocode for Linear search and Binary search.
3. AND, OR, NOT Logical Operators.
4. Function.
5. Recursions.
6. Flowcharts.
7. IF-ELSE, WHILE and FOR.
8. Functions and Threads.
9. Concurrency.
10. Modular Design.
11. Algorithm of add 2 number.
12. Write an algorithm for the generating Fibonacci series.
13. Flowchart Creation - Design a flowchart that outlines the logic for a user login process. It should include conditional paths for successful and unsuccessful login attempts and a loop that allows a user three attempts before locking the account.
14. Function Design and Modularization - Create a document that describes the design of two modular functions: one that returns the factorial of a number, and another that calculates the nth Fibonacci number. Include pseudocode and a brief explanation of how modularity in programming helps with code reuse and organization.

Answer:

1. Bubble sort.

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

```
1  Start BubbleSort(arr)
2      n = length of arr
3      for i from 0 to n-1 do
4          for j from 0 to n-1-i do
5              if arr[j] > arr[j+1] then
6                  swap arr[j] and arr[j+1]
7              end if
8          end for
9      end for
10 End BubbleSort
```

Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array one element at a time. It iterates through the input list and repeatedly selects the next element to be inserted into the correct position in the already sorted part of the list.

```
1 Start InsertionSort(arr)
2   n = length of arr
3   for i from 1 to n-1 do
4       key = arr[i]
5       j = i - 1
6       while j >= 0 and arr[j] > key do
7           arr[j + 1] = arr[j]
8           j = j - 1
9       end while
10      arr[j + 1] = key
11  end for
12 End InsertionSort
```

2. Linear search

- Linear search, also known as sequential search, is a straightforward search algorithm that checks each element of the list sequentially until the target element is found or the end of the list is reached.
- It works well for small lists or unsorted lists.
- The time complexity of linear search is $O(n)$, where n is the number of elements in the list.
- Linear search can be implemented using a loop to iterate through each element in the list and compare it with the target value.

```
1 Start LinearSearch(arr, target)
2   for each element in arr do
3       if element equals target then
4           return the index of the element
5   end for
6   return -1
7 End LinearSearch
```

Binary search

- Binary search is a more efficient search algorithm that works on sorted lists.
- It divides the list into halves and repeatedly narrows down the search space by comparing the target value with the middle element of the list.
- If the middle element is equal to the target value, the search is successful. Otherwise, it determines whether the target value is in the left or right half of the list and continues searching in that half.
- Binary search has a time complexity of $O(\log n)$, where n is the number of elements in the list.
- It can be implemented recursively or iteratively to achieve optimal performance.

```
1  Start BinarySearch(arr, target)
2      left = 0
3      right = length of arr - 1
4      while left <= right do
5          mid = (left + right) // 2
6          if arr[mid] equals target then
7              return mid
8          else if arr[mid] < target then
9              left = mid + 1
10         else
11             right = mid - 1
12     end while
13     return -1
14 End BinarySearch
15
```

3.

AND Operator (&&):

- The AND operator returns True if both operands are True, otherwise, it returns False.
- It is represented by the symbol && in some programming languages.
- In Python and many other programming languages, the AND operator is represented by the keyword and.
- Example: True and True returns True, True and False returns False, False and False returns False.

OR Operator (||):

- The OR operator returns True if at least one of the operands is True, otherwise, it returns False.
- It is represented by the symbol || in some programming languages.
- In Python and many other programming languages, the OR operator is represented by the keyword or.
- Example: True or True returns True, True or False returns True, False or False returns False.

NOT Operator (!):

- The NOT operator returns the negation of the operand. If the operand is True, NOT returns False, and if the operand is False, NOT returns True.
- It is represented by the symbol ! in some programming languages.
- In Python and many other programming languages, the NOT operator is represented by the keyword not.
- Example: not True returns False, not False returns True.

A	B	A AND B	A OR B	NOT A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

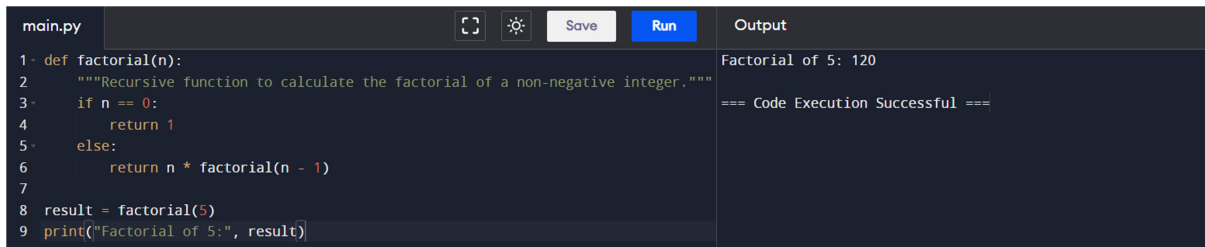
4. A function is a block of reusable code that performs a specific task or set of tasks. It is a self-contained unit of code that can be called or invoked from other parts of a program. Functions allow programmers to organize their code into modular and reusable components, making it easier to manage and maintain large and complex programs.

```
main.py  [Icons] Save Run Output
1 def add_numbers(a, b):
2     """Function to add two numbers."""
3     return a + b
4
5 result = add_numbers(5, 3)
6 print("Result:", result)
7
```

```
Result: 8
=== Code Execution Successful ===
```

5. Recursion is a programming technique where a function calls itself in order to solve a problem. In recursive solutions, the problem is divided into smaller, similar sub problems, which are then solved recursively. These sub problems eventually reach a base case, where they are solved directly without further recursion, thus ending the recursion and returning control back to the original caller.

- **Base Case:** This is the condition where the recursion stops. It serves as the exit condition for the recursive calls and prevents infinite recursion.
- **Recursive Case:** This is where the function calls itself with modified arguments to solve a smaller sub problem. Each recursive call typically moves closer to the base case.
- **Divide and Conquer:** Recursion often involves breaking down a problem into smaller, more manageable sub problems. The results of these subproblems are combined to solve the original problem.
- **Function Call Stack:** Each recursive call is added to the call stack, which keeps track of the state of the function calls. The stack is popped as the recursion unwinds back to the base case.



The screenshot shows a code editor with a file named 'main.py'. The code defines a recursive function 'factorial(n)' that calculates the factorial of a non-negative integer. The function has a base case where 'n' is 0, returning 1, and a recursive case where it returns 'n * factorial(n - 1)'. The code then calls 'factorial(5)' and prints the result. The output pane on the right shows 'Factorial of 5: 120' and '=== Code Execution Successful ==='.

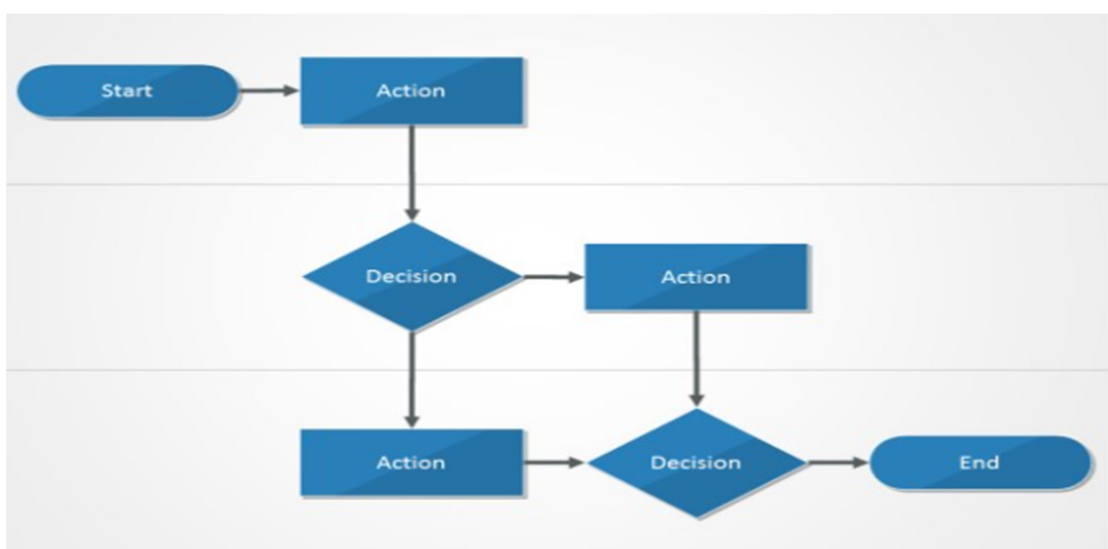
```
main.py 1 def factorial(n):  
2     """Recursive function to calculate the factorial of a non-negative integer."""  
3     if n == 0:  
4         return 1  
5     else:  
6         return n * factorial(n - 1)  
7  
8 result = factorial(5)  
9 print("Factorial of 5:", result)
```

Output
Factorial of 5: 120
=== Code Execution Successful ===

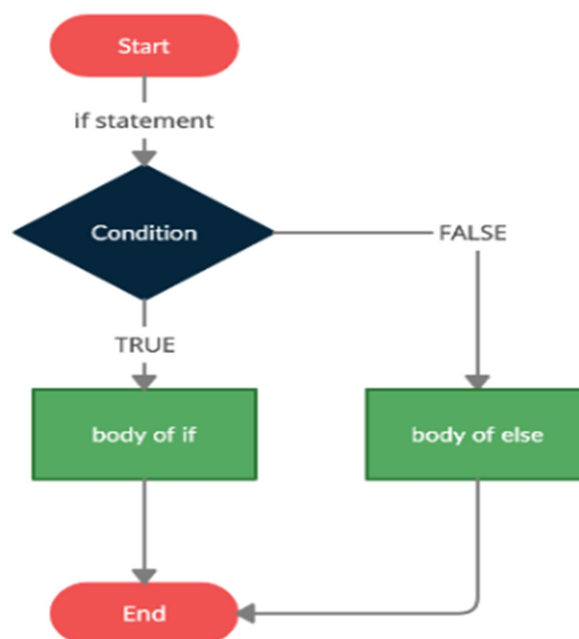
6. A flowchart is a graphical representation of a process, algorithm, or workflow. It uses various symbols and shapes to depict the steps involved in completing a task or solving a problem. Flowcharts are widely used in programming, business processes, engineering, and other fields to visualize and communicate complex processes in a clear and structured manner.

- **Symbols and Shapes:** Flowcharts use standardized symbols and shapes to represent different elements of a process. Common symbols include rectangles (to represent process steps), diamonds (to represent decisions or branching), ovals (to represent start/end points), and arrows (to represent flow direction).
- **Flow of Control:** Arrows or lines connect the symbols to show the flow of control or sequence of steps in the process. Arrows indicate the direction in which the process progresses from one step to another.

- **Decision Points:** Decision points in a flowchart are represented by diamonds or lozenges. These decision points indicate where the process flow diverges based on certain conditions or criteria.
- **Start and End Points:** Flowcharts typically begin with a start point (usually an oval) and end with an end point (also an oval). The start point indicates where the process begins, while the end point marks the completion of the process.
- **Modularity and Hierarchy:** Flowcharts can be structured hierarchically, with subprocesses represented as separate flowcharts or modules. This allows complex processes to be broken down into smaller, more manageable components.

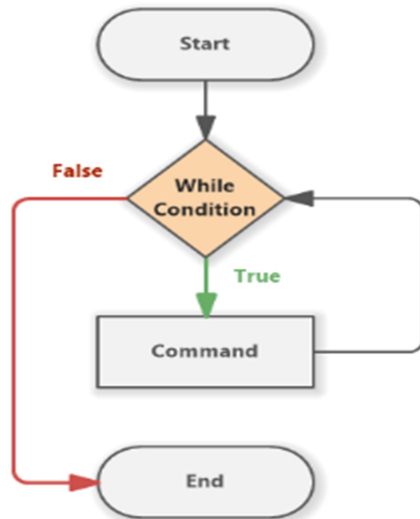


7. IF-ELSE flowchart

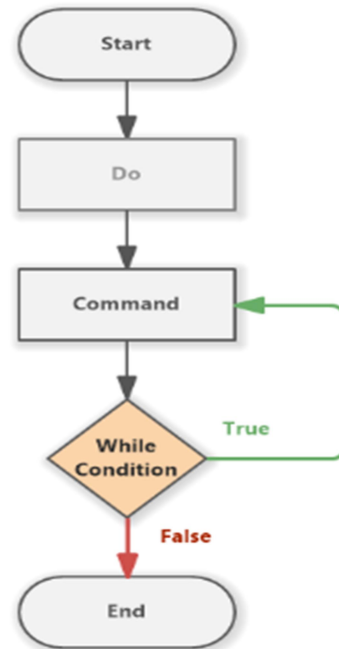


WHILE LOOP and DO-WHILE LOOP

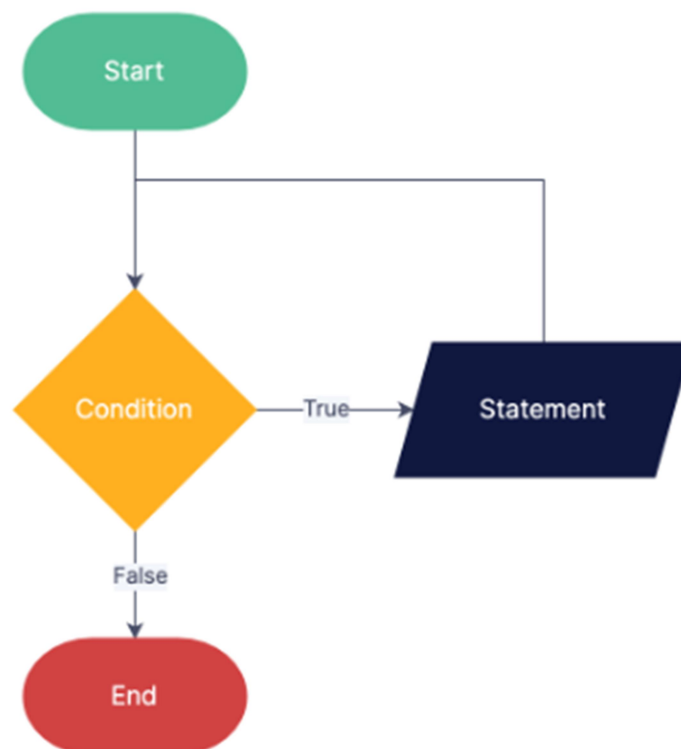
WHILE



DO-WHILE



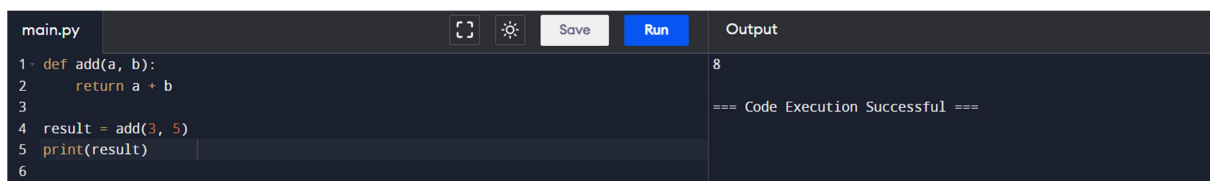
For Loop



8.

Functions:

- A function is a block of code that performs a specific task or computation.
- Functions allow you to break down your program into smaller, reusable pieces, making your code more modular, readable, and maintainable.
- In most programming languages, functions can take parameters (inputs), perform computations, and return results (outputs).
- Functions can be called or invoked from other parts of the program to perform their defined tasks.



The screenshot shows a code editor with a file named 'main.py'. The code defines a function 'add(a, b)' that returns 'a + b'. It then calls this function with 'add(3, 5)' and prints the result. The output pane on the right shows the number '8' and a message '=== Code Execution Successful ==='. The editor has a dark theme and includes icons for file operations and a 'Run' button.

```
main.py
1 def add(a, b):
2     return a + b
3
4 result = add(3, 5)
5 print(result)
6
```

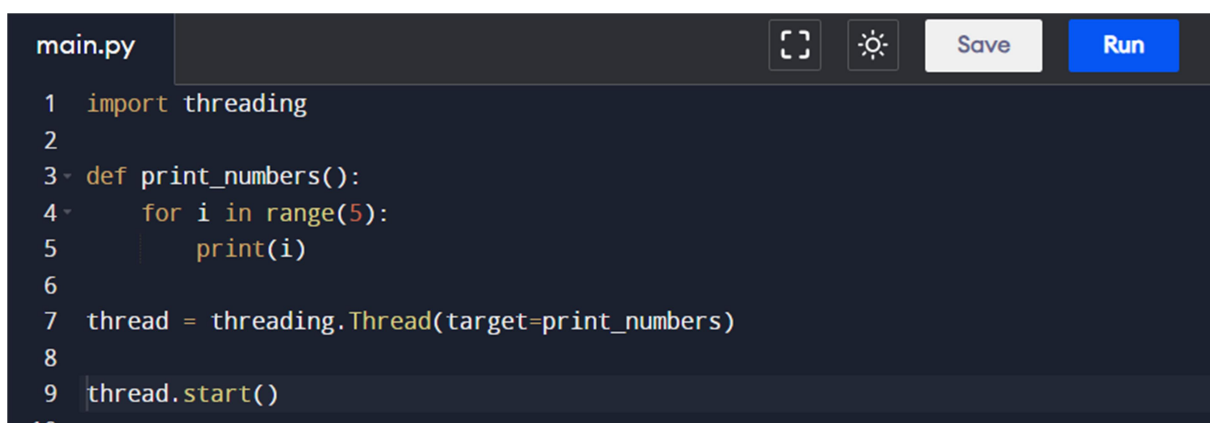
Output

8

=== Code Execution Successful ===

Threads:

- A thread is the smallest unit of execution within a process. It represents a separate flow of control or execution path that can run concurrently with other threads.
- Threads allow programs to perform multiple tasks concurrently, enabling parallelism and multitasking.
- Multithreading is particularly useful for tasks such as handling I/O operations, performing background tasks, and improving responsiveness in user interfaces.
- However, multithreading introduces challenges such as race conditions, synchronization, and resource sharing, which must be carefully managed to ensure correct behavior.



The screenshot shows a code editor with a file named 'main.py'. The code imports the 'threading' module, defines a function 'print_numbers()' that prints numbers from 0 to 4, creates a 'Thread' object with 'print_numbers' as the target, and starts the thread. The editor has a dark theme and includes icons for file operations, a 'Save' button, and a 'Run' button.

```
main.py
1 import threading
2
3 def print_numbers():
4     for i in range(5):
5         print(i)
6
7 thread = threading.Thread(target=print_numbers)
8
9 thread.start()
10
```

9. Concurrency refers to the ability of a computer system to handle multiple tasks or processes simultaneously. In a concurrent system, different tasks or processes make progress independently and possibly concurrently, leading to improved performance, responsiveness, and resource utilization.

10. Modular design, also known as modular programming, is an approach to software design that emphasizes breaking down a system into smaller, more manageable, and reusable modules or components. Each module is responsible for a specific function or feature of the system and can be developed, tested, and maintained independently. These modules are then combined to create the overall software system.

- **Encapsulation:** Modules encapsulate related functionality and data, hiding the internal details and implementation from other modules. This helps to reduce complexity and dependencies between different parts of the system.
- **Abstraction:** Modules provide a high-level abstraction of functionality, allowing developers to focus on the interface and behavior of the module without worrying about the implementation details. This promotes code reuse and modifiability.
- **Decomposition:** Decomposition involves breaking down a complex system into smaller, more manageable modules. Each module should have a clear and well-defined purpose, making it easier to understand, develop, and maintain.
- **Modularity:** Modularity refers to the degree to which a system is composed of independent modules. A highly modular system consists of loosely coupled modules that can be easily replaced, modified, or extended without affecting other parts of the system.
- **Reusability:** Modular design promotes code reuse by allowing modules to be reused in different contexts or projects. Reusable modules can save time and effort by avoiding the need to implement functionality that already exists.

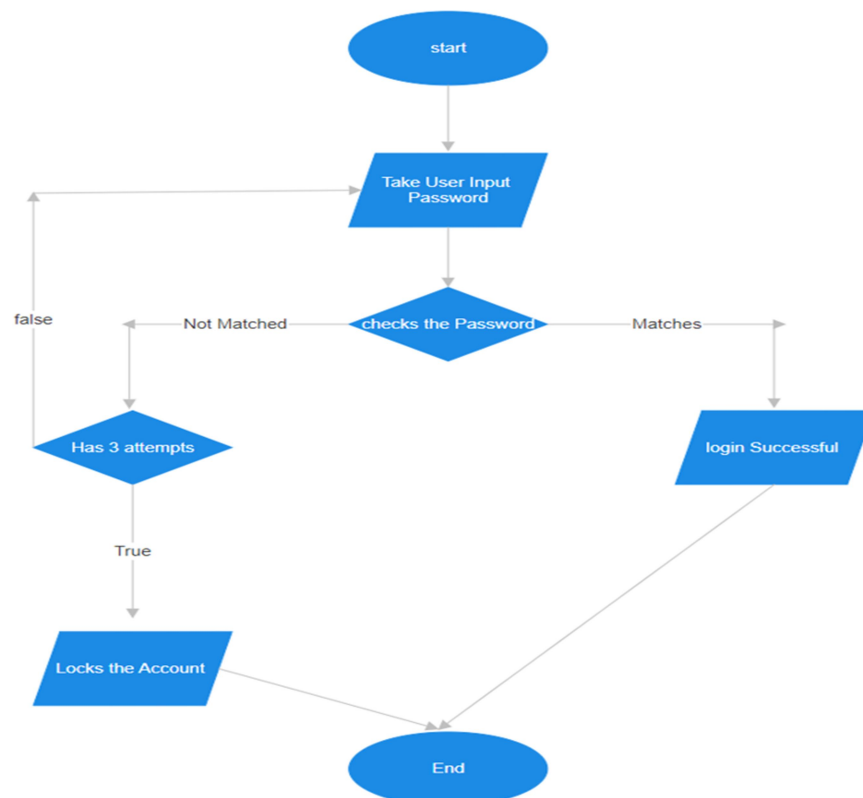
11. Algorithm of add 2 number.

```
1 Steps:
2
3 Start
4 Read the values of num1 and num2 from input
5 Add num1 and num2 together
6 Store the result of the addition in a variable, let's call it sum
7 Output or return the value of sum
8 End
```

12. Write an algorithm for the generating Fibonacci series.

```
1 Steps:
2
3 Start
4 Read the value of n from input
5 Initialize variables first and second to 0 and 1 respectively (these are the
  first two numbers of the Fibonacci series)
6 Initialize a loop counter count to 2 (since we already have the first two numbers
  )
7 Print or output the values of first and second (these are the first two terms of
  the Fibonacci series)
8 While count is less than n:
9   a. Compute the next term of the Fibonacci series by adding first and second,
     store it in a variable next_term
10  b. Print or output the value of next_term
11  c. Update first to the value of second
12  d. Update second to the value of next_term
13  e. Increment count by 1
14 End
```

13. Login process flowchart



14.