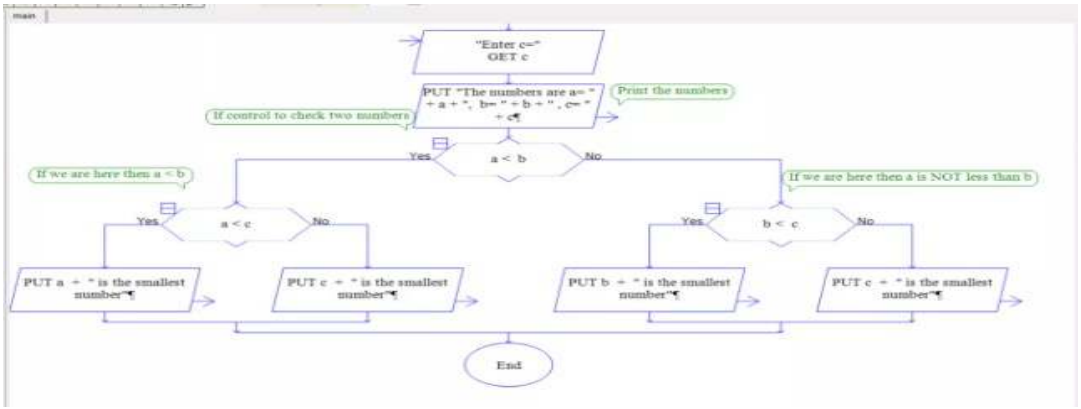
Answer KEY - PPS CT1 SET 5 AND 6 OCT 2023

Programming For Problem Solving (SRM Institute of Science and Technology)

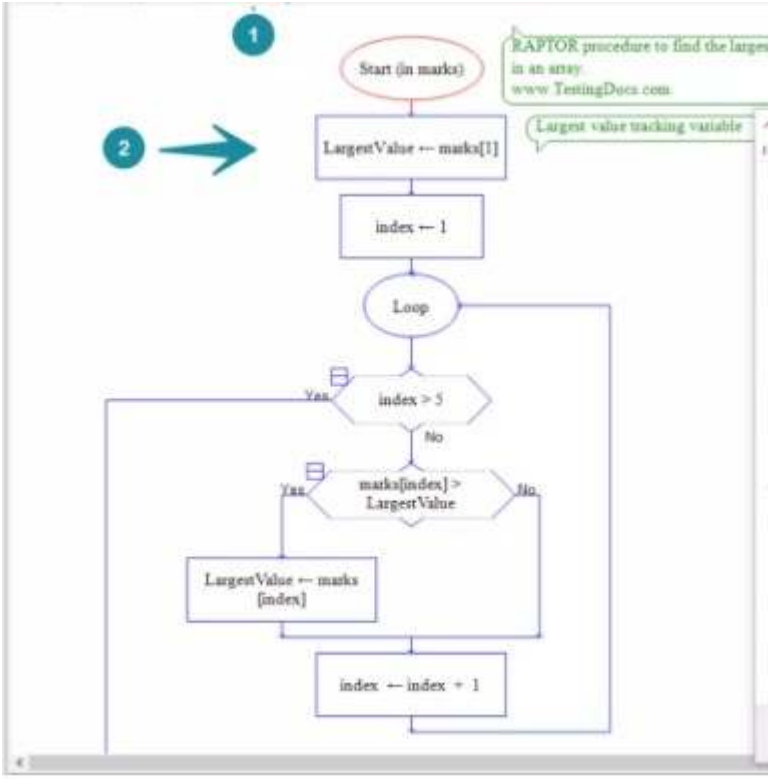
SET 5

PART A

1. Flow chart to find the minimum of 3 numbers



Flow chart to find the maximum of 3 numbers



2.

Assignment Operators:

Assignment operators are used to assign values to variables. The basic assignment operator in C is the equal sign =. There are also compound assignment operators that combine arithmetic or bitwise operations with assignment.

Examples:

```
int x = 10;   // Assigns the value 10 to the variable x.
int y = 5;

x = y;        // Assigns the value of y (5) to x. Now, x = 5.
x += 3;       // Adds 3 to x and assigns the result to x. Now, x = 8.
x -= 2;       // Subtracts 2 from x and assigns the result to x. Now, x = 6.
x *= 4;       // Multiplies x by 4 and assigns the result to x. Now, x = 24.
x /= 3;       // Divides x by 3 and assigns the result to x. Now, x = 8.
```

Compound assignment operators combine an operation with assignment in a single step, making code more concise.

Relational Operators: Relational operators are used to compare values and determine the relationship between them. They return a Boolean (true or false) result. Here are the common relational operators in C:

- == (Equal to): Checks if two values are equal.
- != (Not equal to): Checks if two values are not equal.
- < (Less than): Checks if the left operand is less than the right operand.
- > (Greater than): Checks if the left operand is greater than the right operand.
- <= (Less than or equal to): Checks if the left operand is less than or equal to the right operand.
- >= (Greater than or equal to): Checks if the left operand is greater than or equal to the right operand.

Examples:

```
int a = 5;
int b = 10;

// Relational operators
bool isEqual = (a == b); // false, a is not equal to b
bool isNotEqual = (a != b); // true, a is not equal to b
bool isLessThan = (a < b); // true, a is less than b
bool isGreaterThan = (a > b); // false, a is not greater than b
bool isLessOrEqual = (a <= b); // true, a is less than or equal to b
bool isGreaterOrEqual = (a >= b); // false, a is not greater than or equal to b
```

These relational operators are often used in conditional statements (e.g., if statements) to make decisions based on the comparison results. Remember that relational operators return either true (1) or false (0) as their result, which can be stored in a Boolean variable or used directly in conditional expressions.

3.

```
#include <stdio.h>

int main() {
    char operator;
    double num1, num2, result;

    // Input
    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operator);

    printf("Enter two numbers: ");
    scanf("%lf %lf", &num1, &num2);

    // Perform calculation based on the operator
    switch (operator) {
        case '+':
            result = num1 + num2;
            printf("Result: %.2lf + %.2lf = %.2lf\n", num1, num2, result);
            break;
        case '-':
            result = num1 - num2;
            printf("Result: %.2lf - %.2lf = %.2lf\n", num1, num2, result);
            break;
        case '*':
            result = num1 * num2;
            printf("Result: %.2lf * %.2lf = %.2lf\n", num1, num2, result);
            break;
        case '/':
            if (num2 != 0) {
                result = num1 / num2;
                printf("Result: %.2lf / %.2lf = %.2lf\n", num1, num2, result);
            } else {
                printf("Error: Division by zero is not allowed.\n");
```

```
            }
            break;
        default:
            printf("Error: Invalid operator\n");
    }

    return 0;
}
```

4.

Condition Checking:

- While Loop: In a "while" loop, the condition is checked before the loop body is executed. If the condition is false from the beginning, the loop body may never execute.
- Do...While Loop: In a "do...while" loop, the loop body is executed at least once, regardless of whether the condition is true or false. The condition is checked after the loop body has executed.

Entry Control vs. Exit Control:

- While Loop: "while" loops are entry-controlled loops because the condition is checked before entering the loop. If the condition is false initially, the loop won't run at all.
- Do...While Loop: "do...while" loops are exit-controlled loops because they execute the loop body first and then check the condition to decide whether to continue looping.

WHILE

```
int i = 0;
while (i < 5) {
    cout << i << " ";
    i++;
}
// Output: 0 1 2 3 4
```

DO WHILE

```
int i = 0;
do {
    cout << i << " ";
    i++;
} while (i < 5);
// Output: 0 1 2 3 4
```

PART B

5(I)

An algorithm is a step-by-step procedure or set of instructions that is designed to solve a specific problem or perform a particular task. Algorithms are used in computer programming and various other fields to define a clear and unambiguous way of solving problems. Here are some general rules for writing an algorithm:

Understand the Problem: Before writing an algorithm, make sure you fully understand the problem you're trying to solve. Define the problem statement and identify the input and desired output.

Break It Down: Divide the problem into smaller subproblems or steps. This helps in organizing your algorithm and makes it more manageable.

Use Clear and Precise Language: Write your algorithm in a clear and precise manner, using simple and unambiguous language. Avoid ambiguity or confusion.

Specify Inputs and Outputs: Clearly define what the algorithm expects as input and what it should produce as output. This helps users understand how to use the algorithm.

Consider Efficiency: Try to optimize your algorithm for efficiency. Consider time and space complexity to ensure it runs efficiently for large inputs.

Use Flowcharts or Pseudocode: Before writing code in a specific programming language, consider using flowcharts or pseudocode to outline the logic and structure of your algorithm.

Test and Validate: Test your algorithm with various inputs, including edge cases, to ensure it produces correct results.

Documentation: Provide comments and documentation to explain the purpose of each step and any relevant details.

PART B

5(II)

```
#include <stdio.h>

int main() {
   int num, i, flag = 0;

   // Input a number from the user
   printf("Enter a positive integer: ");
   scanf("%d", &num);
```

```
    // Check if the number is less than 2
    if (num <= 1) {
        printf("Not a prime number.\n");
        return 0;
    }

    // Check for prime by testing divisibility from 2 to the square root of the number
    for (i = 2; i * i <= num; i++) {
        if (num % i == 0) {
            flag = 1; // Set flag to 1 if the number is divisible by any integer from 2 to
          sqrt(num)
            break;
        }
    }

    // Print the result based on the flag
    if (flag == 0) {
        printf("%d is a prime number.\n", num);
    } else {
        printf("%d is not a prime number.\n", num);
    }

    return 0;
}
```

6.

Arithmetic Operators: Arithmetic operators are used to perform basic mathematical operations on numeric values.

```
#include <stdio.h>

int main() {
    int a = 10, b = 5;
    int sum = a + b;
    int difference = a - b;
    int product = a * b;
    int quotient = a / b;

    printf("Sum: %d\n", sum);
    printf("Difference: %d\n", difference);
    printf("Product: %d\n", product);
```

```
    printf("Quotient: %d\n", quotient);

    return 0;
}
```

Relational Operators: Relational operators are used to compare values and return a Boolean result (true or false).

```
#include <stdio.h>

int main() {
    int x = 10, y = 20;

    printf("x < y: %d\n", x < y);
    printf("x > y: %d\n", x > y);
    printf("x <= y: %d\n", x <= y);
    printf("x >= y: %d\n", x >= y);
    printf("x == y: %d\n", x == y);
    printf("x != y: %d\n", x != y);

    return 0;
}
```

Logical Operators: Logical operators are used to perform logical operations on Boolean values (true or false).

```
#include <stdio.h>

int main() {
    int a = 1, b = 0;

    printf("a && b: %d\n", a && b);
    printf("a || b: %d\n", a || b);
    printf("!a: %d\n", !a);
    printf("!b: %d\n", !b);

    return 0;
}
```

Assignment Operators: Assignment operators are used to assign values to variables. They can also perform an operation while assigning the value.

```
#include <stdio.h>
```

```
int main() {
    int x = 5;
    x += 3; // Equivalent to x = x + 3
    printf("x = %d\n", x);

    int y = 10;
    y *= 2; // Equivalent to y = y * 2
    printf("y = %d\n", y);

    return 0;
}
```

SET 6

PART A

1.

In C, the = operator and == operator serve very different purposes:

= Operator (Assignment Operator):

- The = operator is used for assignment in C. It assigns the value on the right-hand side of the operator to the variable on the left-hand side.
- It is used to store a value in a variable.

Example:

```
int x = 5;  // Assigns the value 5 to the variable x.
```

== Operator (Equality Operator):

In C The == operator is used for comparison in C. It checks if the values on both sides of the operator are equal.

- It is used to compare two values for equality and returns a Boolean result (true or false).

Example:

```
int a = 5;
int b = 7;
if (a == b) {
   // This condition is false because 5 is not equal to 7.
   // The code inside the if block will not be executed.
} else {
   // This block will be executed because the condition is false.
   // a is not equal to b.
}
```

2.

In C programming, bitwise operations are used to manipulate individual bits of an integer value. There are several bitwise operators available in C for this purpose:

Bitwise AND (&):

- The & operator performs a bitwise AND operation on each pair of corresponding bits of two integers. The result will be 1 if both bits are 1; otherwise, it will be 0.

```
#include <stdio.h>
```

```
int main() {
    int a = 12; // binary: 1100
    int b = 25; // binary: 11001

    int result = a & b; // bitwise AND

    printf("a & b = %d\n", result); // Output: 8 (binary: 1000)

    return 0;
}
```

Bitwise OR (|):

- The | operator performs a bitwise OR operation on each pair of corresponding bits of two integers. The result will be 1 if at least one of the bits is 1.

```
#include <stdio.h>

int main() {
    int a = 12; // binary: 1100
    int b = 25; // binary: 11001

    int result = a | b; // bitwise OR

    printf("a | b = %d\n", result); // Output: 29 (binary: 11101)

    return 0;
}
```

Bitwise XOR (^):

- The ^ operator performs a bitwise XOR (exclusive OR) operation on each pair of corresponding bits of two integers. The result will be 1 if the bits are different; otherwise, it will be 0.

```
#include <stdio.h>

int main() {
    int a = 12; // binary: 1100
    int b = 25; // binary: 11001

    int result = a ^ b; // bitwise XOR
```

```
    printf("a ^ b = %d\n", result); // Output: 21 (binary: 10101)

    return 0;
}
```

Bitwise NOT (~):

- The ~ operator performs a bitwise NOT operation on each bit of an integer. It inverts each bit, turning 1s into 0s and 0s into 1s.

```
#include <stdio.h>

int main() {
    int a = 12; // binary: 1100

    int result = ~a; // bitwise NOT

    printf("~a = %d\n", result); // Output: -13 (binary: 11110100)

    return 0;
}
```

3.

```
// Include directives (header files)
#include <stdio.h>  // This includes the standard input/output library

// Function prototypes (optional)
// You can declare functions here before defining them
// For example: void myFunction();

// Main function
int main() {
    // Declarations and variable initialization
    // Declare and initialize variables here

    // Statements and code logic
    // Write the main logic of your program here

    // Output
    // Print results or messages to the console
    printf("Hello, World!\n");
```

```
    // Return statement
    // Return an integer to the operating system (0 for success, non-zero for error)
    return 0;
}

// Function definitions (optional)
// You can define functions outside the main function
// For example:
// void myFunction() {
//     // Function logic here
// }
```

Include Directives (Header Files):

- #include <stdio.h>: This line includes the standard input/output library, which provides functions like printf and scanf for input and output operations.

Function Prototypes (Optional):

- You can declare functions here before defining them. This is optional but can be useful for organizing your code.

Main Function:

- Every C program must have a main function. It serves as the entry point of the program.
- The main function returns an integer value (int) to the operating system. A return value of 0 typically indicates a successful execution, while non-zero values indicate errors.

Declarations and Variable Initialization:

- Declare and initialize variables that you will use in your program. This section is where you define your data types and initial values.

Statements and Code Logic:

- This is where you write the main logic of your program, which includes conditional statements (if, else, switch), loops (for, while, do-while), and other operations.

Output:

- Use the printf function (from the stdio.h library) to display output on the console. In the example, "Hello, World!" is printed to the console.

Return Statement:

- The return statement is used to return an integer value from the main function. As mentioned earlier, a return value of 0 usually indicates a successful execution.

Function Definitions (Optional):

- You can define functions outside the main function. These functions can be called from within the main function or other functions.

Here's a simple C program that prints "Hello, World!" to the console:

```
#include <stdio.h>

int main() {
   printf("Hello, World!\n");
   return 0;
}
```

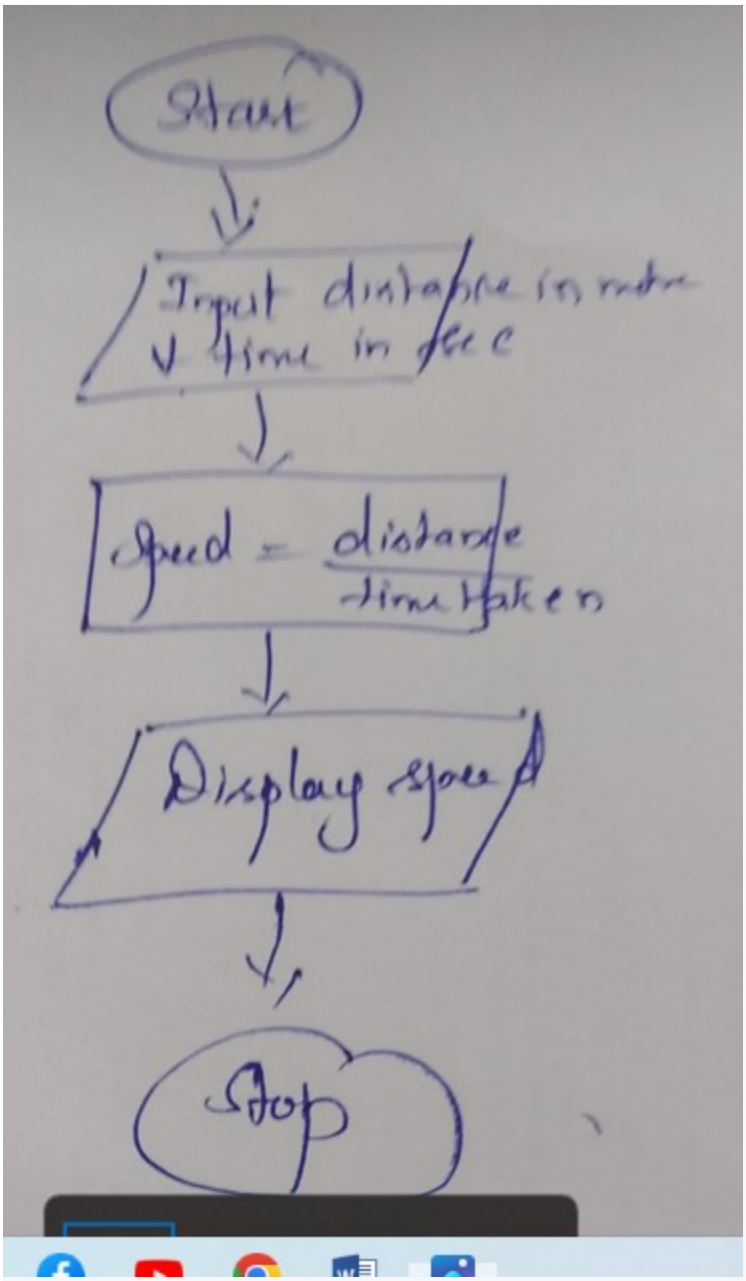
4.

```
#include <stdio.h>

int main() {
   for (int i = 1; i <= 100; i++) {
      if (i % 3 == 0) {
         printf("%d ", i);
      }
   }

   return 0;
}
```

PART B

5.

FLOW CHART



PSEUDO CODE

1. Start
2. Input distance (d) in meters
3. Input time (t) in seconds
4. Calculate speed (s) as d / t
5. Display speed
6. End

PROGRAM

```
#include <stdio.h>

int main() {
   double distance, time, speed;

   // Step 2: Input distance in meters
   printf("Enter the distance (in meters): ");
   scanf("%lf", &distance);

   // Step 3: Input time in seconds
   printf("Enter the time (in seconds): ");
   scanf("%lf", &time);

   // Step 4: Calculate speed (s) as d / t
   speed = distance / time;

   // Step 5: Display speed
   printf("The speed of the car is %.2lf meters per second.\n", speed);

   return 0;
}
```

OUTPUT

```
Enter the distance (in meters): 80
Enter the time (in seconds): 16
The speed of the car is 5.00 meters per second.
```

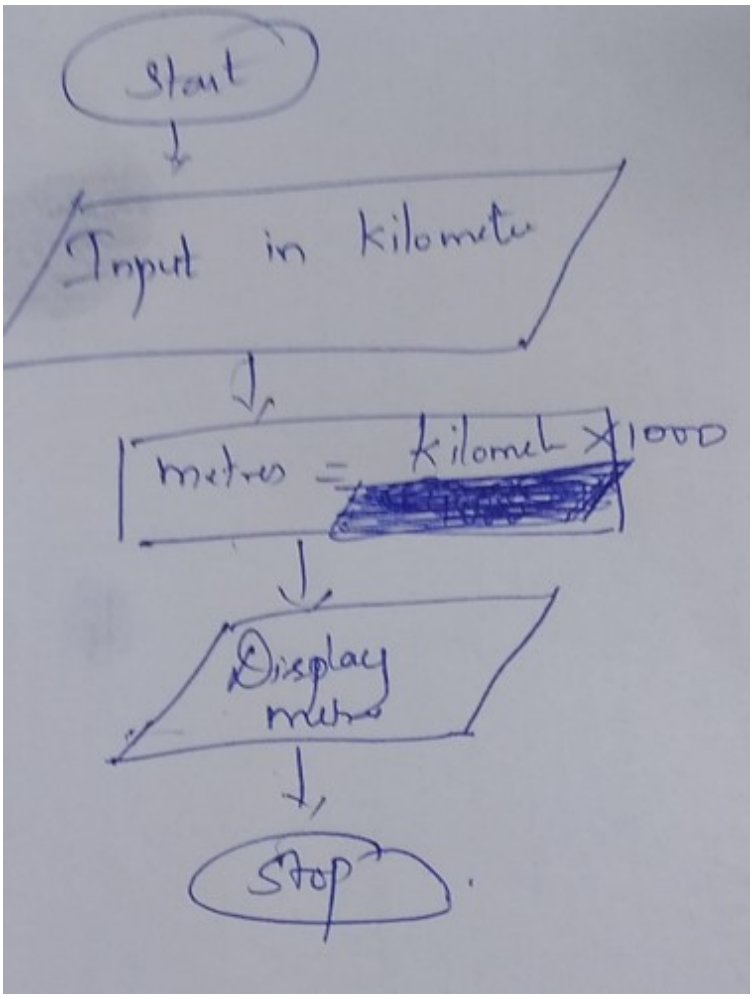
5(II)

Agorithm, flowchart, and a simple C program to convert kilometers to meters:

Algorithm:

Start
Initialize variables: kilometer, meter
Input the value of kilometer
Convert kilometer to meter by multiplying it by 1000 (1 kilometer = 1000 meters)
Display the result (meter)
End

FLOW CHART



PROGRAM

```
#include <stdio.h>

int main() {
    double kilometer, meter;

    // Input kilometer
    printf("Enter the distance in kilometers: ");
    scanf("%lf", &kilometer);

    // Convert kilometer to meter
    meter = kilometer * 1000;

    // Display the result
    printf("%.2lf kilometers is equal to %.2lf meters.\n", kilometer, meter);
```

```
    return 0;
}
```

6.(I)

In C programming, the continue statement is used to control the flow of a loop (usually a for or while loop) by skipping the current iteration and moving on to the next iteration of the loop. Its primary purpose is to allow you to skip a portion of the loop's code for a specific condition while continuing the loop's execution. The continue statement is typically used when you want to bypass the remaining code in the current iteration but continue looping with the next iteration.

Here's how the continue statement works:

- When the continue statement is encountered inside a loop, it immediately transfers control to the loop's condition-checking expression.
- If the condition is still true, the loop continues with the next iteration, effectively skipping the code that follows the continue statement within the current iteration.
- If the condition is false, the loop terminates, and the program continues executing the code after the loop.

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            printf("Skipping iteration %d\n", i);
            continue; // Skip the rest of the loop's code for i == 3
        }
        printf("Processing iteration %d\n", i);
    }
    return 0;
}
```

OUTPUT

```
Processing iteration 1
Processing iteration 2
Skipping iteration 3
Processing iteration 4
Processing iteration 5
```

continue is used to skip the current iteration and continue with the next iteration of the loop, while break is used to exit the loop prematurely, regardless of the loop's condition.
continue allows you to skip part of the loop's code but keeps the loop structure intact, while break terminates the entire loop.

6(II)

```
#include <stdio.h>

int main() {
    int height;
        // Ask the user for the height of the triangle
    printf("Enter the height of the right-angled triangle: ");
    scanf("%d", &height);

    // Check if the entered height is greater than 0
    if (height <= 0) {
        printf("Height must be greater than 0.\n");
        return 1; // Exit the program with an error code
    }
    // Loop to print the right-angled triangle
    for (int i = 1; i <= height; i++) {
        // Print spaces before the asterisks
        for (int j = 1; j <= height - i; j++) {
            printf(" ");
        }

        // Print asterisks for the current row
        for (int k = 1; k <= i; k++) {
            printf("*");
        }

        // Move to the next line after each row
        printf("\n");
    }

    return 0;
}
```