Department of Information and Communication Engineering
Pabna University of Science and Technology
Faculty of Engineering and Technology
B.Sc. (Engineering) 4th Year 2nd Semester Exam-2023
Session: 2019-2020

Course Title: System Analysis and Software Testing Sessional.
Course Code: ICE-4204

Practical Lab Report

Submitted By: MD RASHED
Roll No**:** 200613
Dept. of Information and Communication Engineering
Pabna University of Science and Technology
Pabna-6600, Bangladesh

Submitted To: Md. Anwar Hossain
Professor
Dept. of Information and Communication Engineering.
Pabna University of Science and Technology
Pabna-6600, Bangladesh

Date of Submission: 21-05-2025 **Signature**

Index

Problem-01: Write a program in "JAVA" or "C" to develop a simple calculator that would be able to take a number, an operator (addition/ subtraction/ multiplication/ division/ modulo) and another number consecutively as input and the program will display the output after pressing "=" sign.
Sample input: 1+2; 8%4; **Sample output:** 1+2=3; 8%4=0.

Problem-02: Write a program in "JAVA" or "C" that will take two 'n' integers as input until a particular operator and produce 'n' output.
Sample input: 4 5 7 8 20 40 +; **Sample output:** 9 15 60

Problem-03: Write a program in "JAVA" or "C" to check whether a number or string is palindrome or not.
N.B.: Your program must not take any test case number such as 1 or 2 for the desired cases from the user. Program user will insert a number or string as input directly and the program will display the exact result in the output console.

Problem-04: Write down the ATM system specifications and report the various bugs.

Problem-05: Write a program in "JAVA" or "C" to find out the factorial of a number using while or for loop. Also verify the results obtained from each case.

Problem-06: Write a program in "JAVA" or "C" that will find sum and average of array using do while loop and 2 user defined functions.

Problem-07: Write a simple "JAVA" program to explain classNotFound Exception and endOfFile (EOF) exception.

Problem-08: Write a program in "JAVA" or "C" that will read a input.txt file containing n positive integers and calculate addition, subtraction, multiplication and division in separate output.txt file.

Sample input: 5 5 9 8;
Sample output: Case-1: 100 25 1; **Case-2:** 171 72 1

Problem-09: Explain the role of software engineering in Biomedical Engineering and in the field of Artificial Intelligence and Robotics.

Problem-10: Study the various phases of Water-fall model. Which phase is the most dominated one?

Problem-11: Using COCOMO model estimate effort for specific problem in industrial domain.

Problem-12: Identify the reasons behind software crisis and explain the possible solutions for the following scenario:

Case 1: "Air ticket reservation software was delivered to the customer and was installed in an airport at 12.00 AM (midnight) as per the plan. The system worked quite fine till the next day 12.00 PM (noon). The system crashed at 12.00 PM and the airport authorities could not continue using software for ticket reservation till 5.00 PM. It took 5 hours to fix the defect in the software."

Case 2: "Software for financial systems was delivered to the customer. Customer informed the development team about a malfunction in the system. As the software was huge and complex, the development team could not identify the defect in the software."

Problem No: 01
Problem Name: Write a program in "JAVA" or "C" to develop a simple calculator that would be able to take a number, an operator (addition/ subtraction/ multiplication/ division/ modulo) and another number consecutively as input and the program will display the output after pressing "=" sign.
Sample input: 1+2; 8%4; **Sample output:** 1+2=3; 8%4=0

Objectives:

- ❖ To build a simple calculator using Java or the C programming language.
- ❖ To understand how to take arithmetic expressions as input.
- ❖ To perform basic arithmetic operations like addition, subtraction, multiplication, division, and modulo.
- ❖ To learn the parsing of user input and display formatted output.

Theory:
A simple calculator program is one of the foundational projects for beginners in programming languages like Java or C. This type of program helps in understanding the fundamental concepts of input handling, operators, conditional logic (such as if-else or switch statements), and output formatting. The main purpose of this problem is to develop a calculator that can take an arithmetic expression (consisting of two numbers and an operator) as input and return the correct result as output.

Problem Understanding
The problem asks us to write a program that:

- ✓ Takes a number (first operand),
- ✓ Accepts an arithmetic operator (such as +, -, *, /, or %),
- ✓ Takes another number (second operand),
- ✓ Displays the result after the user inputs the = sign.

Sample Input:

- ❖ 1+2
- ❖ 8%4

Sample Output:

- ❖ 1+2=3
- ❖ 8%4=0

This shows that the program must parse the input expression, identify the operator, perform the correct mathematical operation, and display the final result in the required format.

Concepts Involved

1. Input and Output Handling

In both **Java** and **C**, reading input from the user is a basic but crucial operation. The input format in this problem is not separated (like separate inputs for number, operator, number), but rather combined (e.g., "1+2"). This implies we need to **parse** the input string properly.

- In **Java**, Scanner or BufferedReader can be used.
- In **C**, scanf() can be used with specific format specifiers.

2. Parsing the Expression

The core part of the problem is to **parse** or **extract** the numbers and operator from the input string. For example:

- From 1+2, we need to extract 1, +, and 2.

This parsing can be done using different methods:

- Using string functions (like charAt(), substring() in Java).
- Using pattern matching or simple character checking (e.g., checking each character to identify digits and operators).
- Using scanf format strings in C, such as scanf("%d%c%d", &num1, &operator, &num2);.

3. Arithmetic Operators

The following arithmetic operators are required:

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulo)

Each operator has a specific functionality:

- +: Adds two numbers.
- -: Subtracts the second number from the first.
- *: Multiplies the numbers.
- /: Divides the first number by the second (must handle division by zero).
- %: Returns the remainder of the division.

In programming languages, these operators are used directly, and we can map user input to these operations using conditional structures.

4. Decision-Making (Conditional Logic)

After parsing the operator, we need to **perform the appropriate operation**. This can be done using:

- if-else statements (e.g., if (operator == '+') result = num1 + num2;)
- switch statements (especially in C or Java, for clean handling of multiple cases).

The conditional logic ensures that based on the operator, the correct mathematical operation is performed.

5. Displaying the Result

Finally, the output should be displayed in a specific format:

- Input expression followed by the = sign and then the result (e.g., 8%4=0).

In both Java and C:

- Use of System.out.println() in Java or printf() in C is used to print the formatted result.

6. Error Handling (Optional but Recommended)

Real-world calculators handle errors like:

- **Division by zero** (/ or % with zero).
- **Invalid operators** (e.g., user inputs 1&2 which is not valid).

Although not strictly required in the sample problem, adding basic error checking improves the program's robustness.

Source code (C):

```
#include <stdio.h>

int main() {
    double num1, num2;
    char charecter, press;

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

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

    printf("Enter +,-,*,/,%c: ", '%'); // % needs to be escaped in printf
    scanf(" %c", &charecter); // note the space before %c to consume any leftover newline

    printf("Enter =: ");
    scanf(" %c", &press);

    if (press == '=') {
        if (charecter == '+') {
            printf("%.2lf + %.2lf = %.2lf\n", num1, num2, num1 + num2);
        } else if (charecter == '-') {
            printf("%.2lf - %.2lf = %.2lf\n", num1, num2, num1 - num2);
        } else if (charecter == '*') {
            printf("%.2lf * %.2lf = %.2lf\n", num1, num2, num1 * num2);
        } else if (charecter == '/') {
            if (num2 != 0)
                printf("%.2lf / %.2lf = %.2lf\n", num1, num2, num1 / num2);
            else
                printf("Division by zero is not allowed.\n");
        } else if (charecter == '%') {
            int int1 = (int)num1;
            int int2 = (int)num2;
            if (int2 != 0)
                printf("%d %% %d = %d\n", int1, int2, int1 % int2);
            else
                printf("Modulo by zero is not allowed.\n");
        } else {
            printf("Invalid operator.\n");
        }
    }

    return 0;
}
```

Output:

```
Enter num1: 5
Enter num2: 10
Enter +,-,*,/,%: +
Enter =: =
5.00 + 10.00 = 15.00
```

Problem No: 02
Problem Name: Write a program in "JAVA" or "C" that will take two 'n' integers as input until a particular operator and produce 'n' output.
Sample input: 4 5 7 8 20 40 +; **Sample output:** 9 15 60

Objectives:
- ❖ To write a program that reads a sequence of integers terminated by an operator.
- ❖ To perform operations (such as addition) on consecutive integer pairs.
- ❖ To improve understanding of input parsing, array handling, and loop control.

Theory:
This problem is a practical application of basic programming principles involving input handling, data processing, and arithmetic operations. It focuses on building a program that can take multiple integers as input, followed by a single arithmetic operator, and then perform the specified operation in pairs of the given integers. The final output consists of the results of these pairwise operations.

Input Format and Interpretation
The input is a single line containing:
1. A list of space-separated integers.
2. An arithmetic operator at the end (such as +;, -;, *;, /;, or %;), which indicates the operation to perform.

The semicolon (;) signifies the end of the input and helps isolate the operator. The program needs to:
- ➢ Read the full line as a single string.
- ➢ Separate the integers from the operator.
- ➢ Convert the numeric strings to actual integer values for computation.

Processing the Input
After reading the input, the program processes it as follows:

1. **Tokenization**:
   The string is split into individual parts using spaces. The last token is identified as the operator (with the semicolon), and the rest are considered integers.

2. **Operator Extraction**:
   The operator symbol is extracted by removing the semicolon from the last token. It must be one of the valid operators:
   - ✓ + for addition
   - ✓ - for subtraction
   - ✓ * for multiplication
   - ✓ / for division
   - ✓ % for modulo

3. **Conversion of Strings to Numbers**:
   All remaining tokens are parsed and stored as integer values in a list or array for computation.

Validating the Input

It is important to ensure:

- The total number of integers is **even** since the operation is done in **pairs**. If an odd number is provided, the last number would not have a partner for the operation.
- The operator is valid and recognized.
- Division and modulo operations do not attempt to divide by zero, which would result in a runtime error.

Pairwise Arithmetic Operations

Once the input is validated and parsed:

- The program iterates through the list of integers in steps of two.
- For each pair, the specified operator is applied.
- The result of each operation is stored or printed immediately.

For example, in the case of operator * and input numbers 2 3 4 5, the operations performed would be:

- 2 * 3 = 6
- 4 * 5 = 20

Final output: 6 20

This step requires conditional logic to match the operator with the corresponding arithmetic computation.

Source Code (C):

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

int main() {
    int arr1[100], arr2[100];
    char op;
    int n = 0;

    printf("Enter number of elements (n): ");
    scanf("%d", &n);

    if (n <= 0 || n > 100) {
        printf("Invalid number of elements.\n");
        return 1;
    }

    printf("Enter %d integers for the first set:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr1[i]);
    }
    printf("Enter %d integers for the second set:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr2[i]);
    }

    printf("Enter operator (+, -, *, /) followed by ; : ");
    scanf(" %c;", &op);  // The space before %c skips whitespace

    printf("Output:\n");
    for (int i = 0; i < n; i++) {
        switch (op) {
            case '+':
                printf("%d ", arr1[i] + arr2[i]);
                break;
            case '-':
                printf("%d ", arr1[i] - arr2[i]);
                break;
            case '*':
                printf("%d ", arr1[i] * arr2[i]);
                break;
            case '/':
                if (arr2[i] != 0)
                    printf("%d ", arr1[i] / arr2[i]);
                else
                    printf("Inf ");  // handle division by zero
                break;
            default:
                printf("\nInvalid operator.\n");
                return 1;
        }
    }

    printf("\n");
    return 0;
}
```

Output:

```
Enter number of elements (n): 3
Enter 3 integers for the first set:
1 2 3
Enter 3 integers for the second set:
2 3 4
Enter operator (+, -, *, /) followed by ; : +
Output:
3 5 7
```

Problem No: 03
Problem Name: Write a program in "JAVA" or "C" to check whether a number or string is palindrome or not.
N.B.: Your program must not take any test case number such as 1 or 2 for the desired cases from the user. Program user will insert a number or string as input directly and the program will display the exact result in the output console.

Objectives:
- ❖ To write a program that checks whether a given input (number or string) is a palindrome.
- ❖ To practice handling both numeric and string inputs.
- ❖ To strengthen understanding of string manipulation and conditional logic.

Theory:
A palindrome is a word, phrase, number, or sequence that reads the same forward and backward. This property of symmetry makes palindromes a common subject in algorithm design and programming exercises. Examples include simple numbers like 121 or words like "madam", both of which remain unchanged when reversed.
In this problem, the program should determine whether the user-input value, which can be either a number or a string, is a palindrome or not. The input is taken directly from the user and should not rely on predefined test cases or case numbers.

Understanding the Input and Output
The user inputs a single string or number. This input can be:
- ✓ A numeric string, such as 12321 or 4554
- ✓ An alphabetic string, such as "racecar" or "level"
- ✓ A mixed-case string, such as "Noon" or "Madam"

The output should be:
- ✓ A confirmation message like “It is a palindrome” if the input matches its reversed version
- ✓ Or “It is not a palindrome” if it doesn’t

The logic must consistently treat both numbers and strings since the characters of a number like 121 can be processed similarly to characters in a word like "wow".

Core Logical Steps for Palindrome Detection
1. **Input Reading:**
   The program reads a single input from the user. Regardless of whether the input is a number or a string, it is typically read as a **string** type. This simplifies the reversal process, as each character (including digits) can be easily compared.

2. **Reversing the Input:**
   The main operation is to reverse the input string and compare it with the original input:
   - ➢ If the reversed version is the same as the original, it is a palindrome.
   - ➢ Otherwise, it is not.

There are several ways to reverse a string:
- ➢ By iterating from the end to the beginning and building a new reversed string
- ➢ Using built-in functions (in Java, for example, StringBuilder.reverse())

3. **Comparison:**
   Once the string is reversed, a comparison is made between the original input and the reversed version:
   - ➢ If they match exactly (including case sensitivity), the input is a palindrome.
   - ➢ If they don't, it is not.

4. **Case Sensitivity Handling:**
   In some cases, strings like "Madam" and "madam" may be considered palindromes. To handle such inputs, the comparison can be made **case-insensitive** by converting both strings to lowercase (or uppercase) before comparison.

Special Considerations

- ❖ **Numbers vs. Strings:**
  Although numbers and strings are different data types, for the purpose of checking palindromes, numbers can be **converted to strings** and processed the same way.
  For instance:
  - ✓ Number 1221 can be turned into string "1221" and reversed to check for symmetry.

- ❖ **Single Characters and Empty Strings:**
  - ✓ A single-character input (like "a" or "1") is always a palindrome.
  - ✓ An empty string is usually considered a palindrome as well.

- ❖ **Whitespace and Special Characters:**
  - ✓ If the program is to check sentences or phrases (e.g., "Was it a car or a cat I saw"), it must ignore spaces and possibly punctuation. However, this problem assumes simpler cases like single words or numbers.

- ❖ **Real-time Input Without Test Case Number:**
  The problem explicitly mentions that the program should **not ask the user for any test case numbers**. This means the program directly takes a single line of input and processes it without asking how many inputs will follow.

Significance of Palindrome Checking in Programming

Checking for palindromes is a classic problem that:

- ✓ Helps develop understanding of **string manipulation** and **character arrays**
- ✓ Introduces concepts such as **string reversal**, **input validation**, and **conditional logic**
- ✓ Is a building block for more complex problems like **pattern recognition**, **DNA sequence analysis**, and **language processing**

Source code (C):

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

int isPalindrome(char str[]) {
    int left = 0;
    int right = strlen(str) - 1;

    while (left < right) {
        if (str[left] != str[right])
            return 0;
        left++;
        right--;
    }
    return 1;
}

int main() {
    char input[100];

    printf("Enter a number or string: ");
    scanf("%s", input);

    // Convert to lowercase for case-insensitive comparison
    for (int i = 0; input[i]; i++) {
        input[i] = tolower(input[i]);
    }

    if (isPalindrome(input))
        printf("'%s' is a palindrome.\n", input);
    else
        printf("'%s' is not a palindrome.\n", input);

    return 0;
}
```

Output: A message indicating whether the input is a palindrome or not.

Examples:

- madam is a palindrome.
- 121 is a palindrome.
- hello is not a palindrome.

Problem No -04:
Problem Name: Write down the ATM system specifications and report the various bugs.

Objectives:

- ❖ Provide users with a secure and convenient way to perform banking transactions.
- ❖ Allow users to withdraw cash, check balance, deposit money, and transfer funds.
- ❖ Authenticate users using a PIN code.
- ❖ Ensure transaction records are maintained for auditing.
- ❖ Handle error conditions such as invalid input, insufficient funds, or hardware failures gracefully.

Theory:
An **Automated Teller Machine (ATM)** is an electronic banking device that allows customers to perform basic financial transactions such as withdrawals, deposits, balance inquiries, and fund transfers without the need for human interaction at a bank branch. Designing an ATM system requires a clear understanding of its specifications (functional and non-functional requirements), components, workflows, and potential bugs that may arise during its operation.

ATM System Specifications
The specifications of an ATM system can be broadly categorized into functional and non-functional requirements.

1. Functional Specifications
These describe what the system should do and include:

a. User Authentication:

- ✓ ATM must authenticate users by asking for a card number and a PIN (Personal Identification Number).
- ✓ After 3 incorrect attempts, the account may be temporarily blocked for security.

b. Account Access:

- ✓ Support for **multiple accounts** (savings, checking, etc.) under the same user.
- ✓ Ability to select account type before performing operations.

c. Transactions:

- ✓ **Cash Withdrawal**: User enters the amount, and system checks for sufficient balance and ATM cash availability.
- ✓ **Balance Inquiry**: Displays current account balance.
- ✓ **Deposit Function**: Accepts cash or checks, confirms amount, and credits to the account.
- ✓ **Fund Transfer**: Transfers money from one account to another (within same bank or to a different bank).

d. Transaction Receipts:

- ✓ Option to print or skip a transaction receipt.
- ✓ Display a summary of each transaction on screen.

e. Language Selection:

- ✓ Users can select from multiple languages to operate the ATM.

f. Session Timeout:

- ✓ If the user does not interact for a certain time, the session is automatically terminated.

g. Card Ejection:

- ✓ After the session ends, the card is ejected automatically.

2. Non-Functional Specifications

These are the quality attributes of the ATM system:

a. Security:

- ✓ Encrypted communication between ATM and the bank server.
- ✓ Secure storage of PINs using hashing.
- ✓ Tamper-proof hardware with alarm systems.

b. Reliability:

- ✓ System should be fault-tolerant and operational 24/7.
- ✓ Backup power support in case of outages.

c. Usability:

- ✓ User-friendly interface with clear instructions.
- ✓ Touchscreen or button-based navigation.

d. Performance:

- ✓ Fast response time (each operation should complete within a few seconds).
- ✓ High uptime and low failure rates.

e. Maintainability:

- ✓ Easy to update software remotely.
- ✓ Routine hardware and software diagnostics.

Components of an ATM System

- ❖ **Card Reader**: Reads data from the magnetic strip or chip of the ATM card.
- ❖ **Keypad**: Used for entering PINs and transaction inputs.
- ❖ **Display Screen**: Provides information and prompts to the user.
- ❖ **Cash Dispenser**: Dispenses the requested amount.
- ❖ **Deposit Slot**: Accepts cash or checks.
- ❖ **Receipt Printer**: Prints transaction summaries.

- **Network Interface**: Connects to the bank server for real-time updates and verification.

Common Bugs in ATM Systems
Bugs can be due to software logic errors, hardware malfunctions, or integration problems. Below are various bugs that could be found during testing or use:

1. Software Bugs

a. Transaction Miscalculation:
- ✓ Wrong amount dispensed or deducted due to floating-point errors or incorrect logic.

b. Session Not Ending Properly:
- ✓ System stays logged in even after card ejection or timeout, allowing unauthorized use.

c. PIN Validation Flaws:
- ✓ Accepting incorrect PINs due to logical flaws in the authentication algorithm.

d. Inconsistent Balance Display:
- ✓ Showing incorrect balance due to server sync issues or data fetching errors.

e. Receipt Misprinting:
- ✓ Wrong details printed, such as incorrect time, transaction ID, or amount.

f. Language Translation Errors:
- ✓ Mismatched or incorrectly translated messages in multi-language interfaces.

2. Hardware Bugs
a. Card Reader Failure:
- ✓ Card not detected or rejected repeatedly due to sensor malfunction.

b. Cash Jam:
- ✓ Notes stuck in the dispenser, causing failed transactions.

c. Keypad Input Errors:
- ✓ Unresponsive or wrongly mapped keys leading to input mistakes.

d. Printer Jam or Low Ink:
- ✓ Receipts not printed or printed incompletely.

e. Display Malfunction:
- ✓ Blank screen or unreadable text affecting usability.

3. Security and Integration Bugs
a. Data Leakage:
- ✓ PIN or transaction details not encrypted during communication with the server.

b. Denial of Service (DoS):
- ✓ System becomes unresponsive if misused or repeatedly accessed without input.

c. Incomplete Fund Transfers:
- ✓ Money debited from one account but not credited to the receiver due to network failure or bugs in transaction logic.

Problem No -05:
Problem Name: Write a program in "JAVA" or "C" to find out the factorial of a number using a while or for loop. Also, verify the results obtained from each case.

Objectives:
- To write a program that calculates the factorial of a given number using a for loop.
- To write a program that calculates the factorial of a given number using a while loop.
- To verify and compare the results obtained from both looping methods.
- To understand the implementation of loops and iterative control in C.

Theory:
A **factorial** is a mathematical function that multiplies a positive integer by all the positive integers less than it. The factorial of a number n is denoted as n! and defined as:

$$n!=n\times(n-1)\times(n-2)\times\cdots\times 1$$

For example:

- 5! = 5×4×3×2×1 = 120
- 3! = 3×2×1 = 63

Additionally, the factorial of 0 is always defined as:

$$0!=1$$

Factorials are widely used in mathematics, particularly in combinatorics, permutations, probability, and algorithms.

Understanding the Problem
In this problem, the task is to write a program in either C or Java that:
1. Takes an integer n as input.
2. Computes the factorial using a loop (either for or while loop).
3. Displays the result.
4. Optionally, verifies the result (for instance, by comparing with a built-in function or re-calculating using another loop for cross-checking).

This task does not allow the use of recursion—only iterative solutions using loops are allowed.

Why Use Loops to Calculate Factorials
There are two main ways to compute factorials:
1. **Recursively** (a function calling itself)
2. **Iteratively** using loops

Using **loops** is efficient in terms of memory usage and simpler to trace and debug, especially for beginners. Loop-based solutions avoid the risk of **stack overflow** that can happen in recursive calls for very large values of n.

There are two loop types suitable for this:

- ✓ for loop: best when the number of iterations is known (e.g., from 1 to n).
- ✓ while loop: useful when we continue a process until a certain condition is false.

Logical Steps for Factorial Using a Loop

1. **Input**:
   - ✓ Read an integer n from the user.
   - ✓ Ensure that n is **non-negative** since factorials are not defined for negative numbers.
2. **Initialization**:
   - ✓ Set a variable fact to 1 (since multiplication identity is 1).
   - ✓ This variable will hold the factorial result as it is calculated.
3. **Loop Execution**:
   - ✓ Use a loop to multiply numbers from 1 to n.
     - In a for loop: iterate from i = 1 to i <= n
     - In a while loop: continue multiplying while i <= n and increment i
4. **Output**:
   - ✓ Display the final result stored in the fact variable.

Source code:

```
#include <stdio.h>

int main() {
   int num;
   long long factorial_for = 1, factorial_while = 1;

   printf("Enter a positive integer: ");
   scanf("%d", &num);

   if (num < 0) {
      printf("Factorial is not defined for negative numbers.\n");
      return 1;
   }

   // Factorial using for loop
   for (int i = 1; i <= num; i++) {
      factorial_for *= i;
   }

   // Factorial using while loop
   int i = 1;
   while (i <= num) {
      factorial_while *= i;
```

```
        i++;
    }

    // Display both results
    printf("Factorial using for loop: %lld\n", factorial_for);
    printf("Factorial using while loop: %lld\n", factorial_while);

    // Verify results
    if (factorial_for == factorial_while) {
        printf("Verification successful: Both methods give the same result.\n");
    } else {
        printf(" Verification failed: Results are different.\n");
    }

    return 0;
}
```

Output:

```
Enter a positive integer: 4
Factorial using for loop: 24
Factorial using while loop: 24
```

Problem No -06:
Problem Name: Write a program in "JAVA" or "C" that will find the sum and average of an array using a do-while loop and 2 user-defined functions.

Objectives:

- ❖ To create a program that calculates the sum of elements in an array using a do-while loop.
- ❖ To calculate the average of the array elements using a user-defined function.
- ❖ To practice the use of user-defined functions and do-while loops in C programming.
- ❖ To display the results clearly to the user.

Theory:
An array stores multiple values of the same data type. Calculating the sum of all elements involves iterating through the array and accumulating each value. The average is computed by dividing the total sum by the number of elements.
The do-while loop executes the block at least once before checking the loop condition, making it useful when the number of iterations is determined during execution.
User-defined functions promote modular programming by separating logic for sum and average calculations, improving code readability and reusability.

<u>Algorithm</u>

Step 1: Start
Step 2: Input Array Size

- ✓ Prompt the user to enter the total number of elements (n).

Step 3: Declare the Array

- ✓ Declare an array of size n.

Step 4: Input Array Elements

- ✓ Prompt the user to enter n integer values.
- ✓ Store them in the array.

Step 5: Call Function calculateSum(array, n)

- ✓ Pass the array and its size to the function calculateSum.

Step 6: Inside calculateSum(array, n)

- ✓ Initialize a variable sum = 0.
- ✓ Initialize index variable i = 0.
- ✓ Use a **do-while loop** to iterate through the array:
  - ▪ Add array[i] to sum.
  - ▪ Increment i.
  - ▪ Repeat until i < n.

- ✓ Return sum to the main function.

Step 7: Call Function calculateAverage(sum, n)

- ✓ Pass the sum and size of the array to calculateAverage.

Step 8: Inside calculateAverage(sum, n)

- ✓ Calculate average = sum / n (make sure result is in float or double format).
- ✓ Return average to the main function.

Step 9: Display Result

- ✓ Print the returned sum.
- ✓ Print the returned average.

Step 10: End

Source Code (C):

```
#include <stdio.h>

// Function to calculate sum using do-while loop
int calculateSum(int arr[], int n) {
   int sum = 0, i = 0;
   do {
      sum += arr[i];
      i++;
   } while (i < n);
   return sum;
}

// Function to calculate average (reuses sum function)
float calculateAverage(int arr[], int n) {
   int sum = calculateSum(arr, n);
   return (float)sum / n;
}

int main() {
   int n, arr[100];

   // Input array size
   printf("Enter number of elements in the array: ");
   scanf("%d", &n);

   // Input array elements
```

```
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Call functions
    int sum = calculateSum(arr, n);
    float avg = calculateAverage(arr, n);

    // Output results
    printf("Sum = %d\n", sum);
    printf("Average = %.2f\n", avg);

    return 0;
}
```

Output:

```
Enter number of elements in the array: 3
Enter 3 integers:
1 2 3
Sum = 6
Average = 2.00
```

Problem No -07:
Problem Name: Write a simple "JAVA" program to explain classNotFound Exception and endOfFile (EOF) exception.

Objectives:

- ❖ To understand and demonstrate how ClassNotFoundException is thrown and handled in Java.
- ❖ To understand and demonstrate how EOFException is thrown and handled during file reading.
- ❖ To learn exception handling using try-catch blocks.

Theory:
In C programming, exceptions like ClassNotFoundException and EOFException do not exist as built-in exception types as they do in Java, because C does not support exception handling in the same way. However, similar error conditions can be detected and handled manually using conditional statements and standard library functions. For example, the equivalent of EOFException occurs when reading from a file and the end-of-file is reached, which is indicated by the EOF constant returned by functions like fgetc() or fscanf(). Programmers typically use feof() or check for EOF to handle such cases. The ClassNotFoundException has no direct C equivalent because C does not support dynamic class loading, but errors related to missing files, libraries, or invalid function pointers can be conceptually similar and are handled using error codes, NULL checks, and manual validation.

<u>Algorithm</u>

Step 1: Start the program

Step 2: Simulate "ClassNotFoundException" Concept

- ✓ Prompt the user to enter the name of a file (representing a "class file").
- ✓ Try to open the file using fopen().
- ✓ If fopen() returns NULL, print an error message like:
    - ▪ "Error: Class file not found."

Step 3: Simulate "EOFException" Concept

- ✓ If the file opens successfully:
    - ▪ Use fgetc() or fscanf() to read characters or data from the file.
    - ▪ Use a loop (e.g., while) to read until end of file.
    - ▪ Check for end-of-file using feof() or EOF constant.
    - ▪ When EOF is reached, display a message like:
        - "End of file reached."

Step 4: Close the file using fclose()

Step 5: End the program

Source Code (Java):

```
import java.io.*;

public class ExceptionDemo {

    public static void main(String[] args) {

        // 1. ClassNotFoundException demo
        try {
            System.out.println("Trying to load a non-existent class...");
            Class.forName("NonExistentClass");
        } catch (ClassNotFoundException e) {
            System.out.println("Caught ClassNotFoundException: " + e.getMessage());
        }

        // 2. EOFException demo
        try {
            System.out.println("\nCreating file and writing one object...");
            FileOutputStream fos = new FileOutputStream("sample.dat");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeInt(42);
            oos.close();

            System.out.println("Reading two integers from file...");
            FileInputStream fis = new FileInputStream("sample.dat");
            ObjectInputStream ois = new ObjectInputStream(fis);

            int first = ois.readInt();
            System.out.println("First int: " + first);

            int second = ois.readInt();  // Will trigger EOFException
            System.out.println("Second int: " + second);

            ois.close();
        } catch (EOFException e) {
            System.out.println("Caught EOFException: Reached end of file unexpectedly.");
        } catch (IOException e) {
            System.out.println("Caught IOException: " + e.getMessage());
        }
    }
}
```

Output:

```
Creating file and writing one object...
Reading two integers from file...
First int: 42
Caught EOFException: Reached end of file unexpectedly.
```

Problem No -08:
Problem Name: Write a program in "JAVA" or "C" that will read a input.txt file containing n positive integers and calculate addition, subtraction, multiplication and division in separate output.txt file.
Sample input: 5 5 9 8;
Sample output: Case-1: 10 0 25 1; **Case-2:** 171 72 1

Objectives:

- ❖ To read a list of positive integers from a text file (input.txt).
- ❖ To perform arithmetic operations (addition, subtraction, multiplication, division).
- ❖ To write the results of each operation into an output file (output.txt).
- ❖ To use file input/output handling and loops in Java.

Theory:
In C programming, file handling allows reading from and writing to files using standard functions like fopen(), fscanf(), fprintf(), and fclose(). For this problem, the program will read a list of positive integers from a file named input.txt, perform addition, subtraction, multiplication, and division, and then write the results to a file named output.txt.
When reading the file, integers are typically separated by spaces or new lines. The program will:

- ✓ **Open** the input.txt file in read mode.
- ✓ **Read** all integers using a loop with fscanf().
- ✓ **Store** or accumulate values while performing:
  - ▪ **Addition**: Sum all integers.
  - ▪ **Subtraction**: Start from the first integer and subtract the rest.
  - ▪ **Multiplication**: Multiply all integers.
  - ▪ **Division**: Start from the first integer and divide by the rest (avoiding division by zero).

- ✓ **Write** each result to output.txt using fprintf().

This approach demonstrates both file input/output and arithmetic operation handling in C, and requires careful attention to file existence and zero division errors.

Source Code:

```
#include <stdio.h>

int main() {
   FILE *inputFile, *outputFile;
   inputFile = fopen("input.txt", "r");
   outputFile = fopen("output.txt", "w");
```

```
if (inputFile == NULL || outputFile == NULL) {
    printf("Error opening file.\n");
    return 1;
}

int num, count = 0;
int sum = 0, first = 1, subtraction = 0;
int product = 1;
float division = 0;
int numbers[100]; // assuming at most 100 integers

// Reading numbers from input.txt
while (fscanf(inputFile, "%d", &num) == 1) {
    numbers[count++] = num;
    sum += num;
    product *= num;
}

if (count == 0) {
    fprintf(outputFile, "No numbers found in input file.\n");
    fclose(inputFile);
    fclose(outputFile);
    return 0;
}

// Initialize subtraction and division with the first number
subtraction = numbers[0];
division = (float) numbers[0];

for (int i = 1; i < count; i++) {
    subtraction -= numbers[i];
    if (numbers[i] != 0)
        division /= numbers[i];
    else {
        fprintf(outputFile, "Division by zero encountered. Cannot continue.\n");
        fclose(inputFile);
        fclose(outputFile);
        return 1;
    }
}

// Writing results to output.txt
fprintf(outputFile, "Sum: %d\n", sum);
```

```
    fprintf(outputFile, "Subtraction (left-to-right): %d\n", subtraction);
    fprintf(outputFile, "Multiplication: %d\n", product);
    fprintf(outputFile, "Division (left-to-right): %.2f\n", division);

    fclose(inputFile);
    fclose(outputFile);

    return 0;
}
```

Output:

```
Input.txt:  5 5 9 8
Output.txt:
Sum: 27
Subtraction (left-to-right): -17
Multiplication: 1800
Division (left-to-right): 0.01
```

Problem No -09
Problem Name: Explain the role of software engineering in Biomedical Engineering and in the field of Artificial Intelligence and Robotics.

Objectives:

- ❖ To understand the applications of software engineering in Biomedical Engineering.
- ❖ To explore the importance of software development in Artificial Intelligence (AI) and Robotics.
- ❖ To analyze how software engineering contributes to innovation, safety, and efficiency in these fields

Theory:
Role of Software Engineering in Biomedical Engineering
Software engineering plays a vital role in biomedical engineering by providing the tools, methods, and frameworks necessary to develop reliable, efficient, and safe software systems that support healthcare and medical applications. Biomedical engineering involves designing and creating devices, diagnostic tools, imaging systems, and therapeutic equipment, all of which rely heavily on software for operation, data processing, and analysis.

- **Medical Device Software**: Software engineering ensures that embedded software in medical devices such as pacemakers, MRI machines, and infusion pumps works correctly and safely under strict regulatory standards.
- **Healthcare Data Management**: Developing software for managing large volumes of patient data, electronic health records (EHR), and medical imaging requires robust data structures, security, and interoperability protocols.
- **Signal and Image Processing**: Biomedical signals like ECG, EEG, and medical images require sophisticated algorithms and software to process, analyze, and visualize data for diagnosis and treatment.
- **Simulation and Modeling**: Software engineering supports the creation of simulations for biological systems, drug interactions, and surgical procedures, aiding research and education.
- **Compliance and Safety**: Biomedical software must meet rigorous quality assurance, validation, and compliance requirements (e.g., FDA regulations), which are ensured through disciplined software engineering practices like testing, documentation, and version control.

Role of Software Engineering in Artificial Intelligence (AI) and Robotics
In AI and robotics, software engineering forms the backbone of designing, implementing, and maintaining intelligent systems that can perceive, reason, learn, and interact with their environment.

- **Algorithm Development**: AI systems rely on complex algorithms for machine learning, natural language processing, computer vision, and decision-making, all of which require software engineering principles for efficient coding and optimization.

- **System Integration**: Robotics involves integrating various sensors, actuators, control systems, and AI components through software to create autonomous or semi-autonomous machines.
- **Real-Time Processing**: Robotics systems often operate in real-time and safety-critical environments, necessitating software engineering techniques to handle concurrency, timing constraints, and fault tolerance.
- **Scalability and Maintainability**: AI applications often involve large datasets and continuous learning, requiring software systems that are scalable, modular, and maintainable.
- **Testing and Validation**: Rigorous software testing, simulation, and validation are essential to ensure AI models and robotic systems behave as expected in diverse scenarios.
- **Human-Robot Interaction**: Software engineering enables designing interfaces and communication protocols for seamless interaction between humans and robots.

Problem No -10
Problem Name: Study the various phases of Water-fall model. Which phase is the most dominated one?

Objectives:

- To study and understand each phase of the Waterfall model.
- To analyze the flow of software development in a sequential model.
- To identify the most dominant and critical phase of the Waterfall model.

Theory:
The Waterfall Model is one of the earliest models of the Software Development Life Cycle (SDLC). It is a linear sequential model where each phase must be completed before the next begins. The model flows in a single direction — like a waterfall — hence the name.

Phases of the Waterfall Model:

1. **Requirement Analysis**
    - All possible requirements of the system are gathered and documented.
    - The goal is to understand *what* the customer needs.
2. **System Design**
    - The system's architecture and design are prepared from the requirements.
    - Defines hardware, software, data, and system interfaces.
3. **Implementation (Coding)**
    - Actual source code is written in this phase using appropriate programming languages and tools.
4. **Testing**
    - After coding, the system is tested thoroughly for bugs and errors.
    - This phase ensures the software meets the desired requirements.
5. **Deployment**
    - The finished product is delivered to the customer and deployed in the target environment.
6. **Maintenance**
    - Once the software is in use, issues are fixed and improvements are made.
    - Includes bug fixes, upgrades, and minor enhancements.

Problem No -11
Problem Name: Using COCOMO model estimate effort for specific problem in industrial domain.

Objectives:

- ❖ To apply the COCOMO (Constructive Cost Model) for software effort estimation.
- ❖ To estimate the development time and effort for a sample industrial problem.
- ❖ To understand how project size and type influence the cost estimation.

Theory:
The Constructive Cost Model (COCOMO) is a widely used algorithmic software cost estimation model developed by Barry Boehm. It helps project managers and engineers estimate the effort, time, and cost required to develop software based on the size of the project and various project attributes.
In the industrial domain, where software projects tend to be complex and critical, accurate effort estimation is crucial for resource allocation, scheduling, budgeting, and risk management.

There are three categories:

1. **Organic:** Small, simple software projects.
2. **Semi-Detached:** Medium complexity.
3. **Embedded:** Complex software with tight hardware, software, or regulatory constraints.

Basic COCOMO Equations:

$$Effort\,(E) = a \times (KLOC)^b$$

Development Time $(TDEV) = c \times (E)^d$

Steps to Estimate Effort Using COCOMO in Industrial Domain:

1. **Determine Project Type**:
   Identify whether the software is organic, semi-detached, or embedded based on project complexity, team experience, and requirements stability.

2. **Estimate Size (KLOC)**:
   Estimate the size of the software in thousands of lines of code. This can be based on previous similar projects, expert judgment, or early design documents.

3. **Apply Basic COCOMO Formula**:
   Use the appropriate constants (a, b) for the project type to calculate the initial effort estimate in person-months.

4. **Adjust Using Cost Drivers (Intermediate/ Detailed COCOMO)**:
   Consider additional project attributes like product reliability, complexity, developer

experience, tooling, and schedule constraints to adjust the effort estimate with multiplicative factors.

5. **Calculate Development Time and Staffing**:
   From the effort, derive the development time (months) and required staff size using further COCOMO formulas.

Importance in Industrial Domain:

- ✓ Industrial software projects often have **strict reliability and safety requirements**, making effort estimation critical.
- ✓ Accurate estimates help in **budgeting and scheduling** complex projects like manufacturing control systems, robotics software, or embedded systems.
- ✓ Helps **manage risks** by anticipating resource needs and timelines.
- ✓ Supports decision-making in **outsourcing, staffing, and tool selection**.

Problem No -12
Problem Name: Identify the reasons behind software crisis and explain the possible solutions for the following scenario:

Case 1: "Air ticket reservation software was delivered to the customer and was installed in an airport at 12.00 AM (midnight) as per the plan. The system worked quite fine till the next day 12.00 PM (noon). The system crashed at 12.00 PM and the airport authorities could not continue using software for ticket reservation till 5.00 PM. It took 5 hours to fix the defect in the software."

Case 2: "Software for financial systems was delivered to the customer. Customer informed the development team about a malfunction in the system. As the software was huge and complex, the development team could not identify the defect in the software."

Objectives:

- ❖ To identify and understand the reasons behind software crisis.
- ❖ To analyze real-world cases of software failure.
- ❖ To suggest appropriate preventive and corrective solutions for each case.

Theory:
Reasons Behind the Software Crisis
The **software crisis** refers to the difficulties and challenges faced in developing, delivering, and maintaining software systems that are reliable, timely, and within budget. Key reasons include:

1. **Complexity**: Software systems can become highly complex, making them difficult to design, implement, test, and maintain.

2. **Poor Requirements**: Ambiguous, incomplete, or changing requirements lead to misunderstandings and errors.
3. **Inadequate Testing**: Insufficient testing or lack of proper quality assurance allows defects to remain undetected until deployment.

4. **Time Pressure**: Strict deadlines may force teams to rush development, resulting in low-quality code.

5. **Lack of Proper Documentation**: Poor documentation makes understanding and fixing defects challenging.

6. **Poor Project Management**: Ineffective planning, resource allocation, and communication cause delays and errors.

7. **Inadequate Maintenance Process**: Defects found post-deployment take longer to fix if maintenance procedures are weak.

Case 1: Ticket Reservation Software Crash at 12:00 PM

Problem:
The software worked fine for 12 hours but crashed exactly at noon, causing a 5-hour downtime.

Likely Causes:

- ✓ **Time-related bug or overflow**: The crash at an exact time suggests a bug linked to date/time handling, such as:
    - Incorrect handling of timers, time zone, or daylight saving changes.
    - Integer overflow or variable reset after a certain count (e.g., 12-hour format mismanagement).

- ✓ **Resource exhaustion**: The system might have run out of memory or system resources after running continuously.

- ✓ **Insufficient testing**: Lack of testing for long-running stability and boundary time conditions.

- ✓ **Poor error handling**: The system failed to recover automatically, resulting in prolonged downtime.

Possible Solutions:

- ✓ Implement rigorous **time and boundary testing**, including 24-hour and daylight saving changes.
- ✓ Use **automated monitoring** and **logging** to detect anomalies early.
- ✓ Design **fault-tolerant and self-recovering systems** that handle exceptions without crashing.
- ✓ Conduct **stress and endurance testing** to evaluate software stability over time.
- ✓ Ensure proper **resource management** to prevent leaks.
- ✓ Schedule **maintenance windows** and backup systems for quick recovery.

Case 2: Financial Software with Undetected Defect in Complex System

Problem:
Customer reports malfunction, but development team struggles to locate the defect due to software complexity.

Likely Causes:

- ✓ **Large codebase and poor modularization**: Difficulty in isolating the problematic component.

- ✓ **Lack of documentation**: Makes understanding system behavior and tracing errors difficult.

- ✓ **Inadequate debugging tools**: Absence of sophisticated tools to trace and analyze software behavior.

- ✓ **Poor requirements traceability**: Difficulty mapping defects to requirements or modules.

- ✓ **Insufficient automated testing**: Hard to reproduce and locate intermittent or complex bugs.

Possible Solutions:

- ✓ Apply **modular design and coding practices** to reduce complexity.
- ✓ Maintain **comprehensive documentation** and **design diagrams**.
- ✓ Use **version control and issue tracking systems** for defect management.
- ✓ Integrate **automated testing frameworks** (unit, integration, system tests).
- ✓ Employ **advanced debugging and profiling tools** to analyze runtime behavior.
- ✓ Use **logging and monitoring** extensively to capture runtime errors.
- ✓ Adopt **agile development** or **continuous integration** practices to detect issues early.
- ✓ Conduct **code reviews and pair programming** to improve code quality.