WIX1002 : FUNDAMENTALS OF PROGRAMMING

GROUP MEMBERS :
JITESH A/L MOGANA RAJA (25006745)
TAN CHEE KEAT (25006123)
TEH XU ZHE (25006355)
LEE MING DAO (25006825)
LIM HONG ZHANG (25006100)

FACULTY: FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

TITLE: VIVA 2 REPORT

SEMESTER SESSION: SEM I 2025/2026

LECTURER NAME : DR. NURUL BINTI JAPAR

GROUP NAME : Ctrl+C Ctrl+V

Question 1

1.1 Problem Statement

The digital root of a positive integer is the **iterative sum of its digits** until a **single digit** remains.
If the sum of digits results in a number with more than one digit, the digits are added again.
This process is repeated until a single-digit number is obtained.

For example:
493193 → 4 + 9 + 3 + 1 + 9 + 3 = 29
29 → 2 + 9 = 11
11 → 1 + 1 = 2

Therefore, the **digital root** of 493193 is **2**.

1.2 Solution Explanation

To solve this problem, the number is processed repeatedly by extracting and summing its digits.

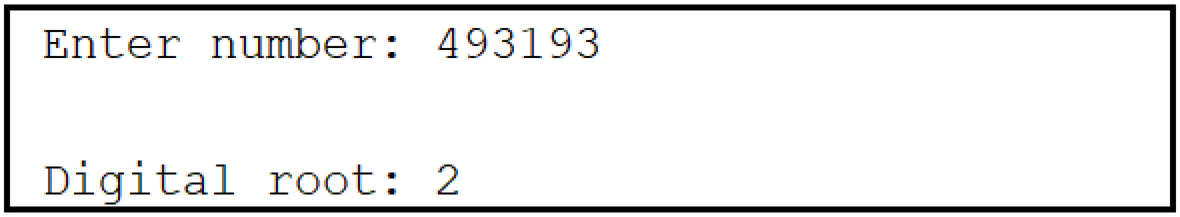
Step-by-step logic:

1. Read a positive integer from the user.
2. While the number has more than one digit:
   - Initialize a variable sum to 0.
   - Extract the last digit using the modulus operator (% 10) and add it to sum.
   - Remove the last digit using integer division (/ 10).
3. Assign the computed sum back to the number.

4. Repeat the process until the number becomes a single digit.

5. The final value is the digital root and is displayed as output.

1.3 Example input and output:



```
Enter number: 493193

Digital root: 2
```

1.4 Source code

```
package viva2;

import java.util.Scanner;

public class Q1 {

  //method to calculate digital root
  public static int digital_root(int n){
    //continue summing digits until single digit is obtained
    while(n>=10){
      int sum=0;
      while(n>0){
        sum+=n%10;
        n=n/10;
      }
      n=sum;
    }
    return n;
  }
  public static void main(String[] args) {
    Scanner sc=new Scanner(System.in);
    //prompt user for input
    System.out.print("Enter number:");
    int number=sc.nextInt();
    //display digital root
    System.out.println("Digital root: "+digital_root(number));
  }

}
```

Question 2

2.1 Problem Statement

The task is to determine whether a given string contains **balanced parentheses**.
An expression is considered *balanced* if every opening parenthesis (has a corresponding closing parenthesis), and the parentheses are properly nested.

The string may include other characters (such as numbers, symbols, and operators). These characters should be ignored; only (and) are used for checking balance.

A method boolean isBalanced(String s) must be implemented to return:

- **true** → if the parentheses are balanced
- **false** → if they are not balanced

The program should then print **“Balanced”** or **“Not balanced”** based on the result.

Example:
Input: (3+5)*(7-(2+1)) → Output: **Balanced**
Input: (5+6*(7-3) → Output: **Not balanced**

2.2 Solution Explanation

To solve this problem, we scan through the string character by character and use a counter to track parentheses:

Step-by-step logic:

1. Initialize a counter (count = 0).
2. Loop through each character in the string.

3. When encountering (, increase the counter (count++).

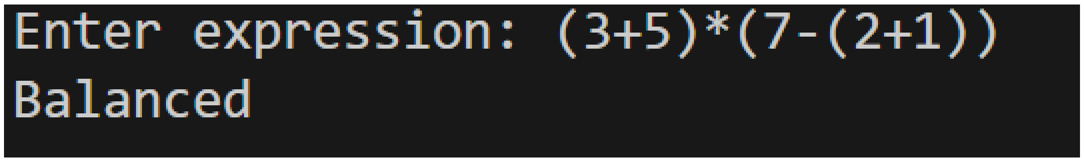
4. When encountering), decrease the counter (count--).

5. If the counter becomes negative at any point, it means a closing parenthesis appears before a matching opening one → expression is **not balanced**.

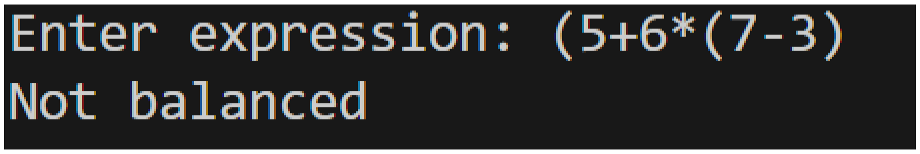
6. After the loop ends, if the counter is exactly zero, the parentheses are **balanced**.

7. If the counter is not zero, some opening parentheses were not closed → **not balanced**.

2.3 Example input and output:



```
Enter expression: (3+5)*(7-(2+1))
Balanced
```



```
Enter expression: (5+6*(7-3)
Not balanced
```

2.4 Source code

```
public class Q2 {

    public static boolean isBalanced(String str) {

        int count = 0;

        for (int i = 0; i < str.length(); i++) {
```

```
        char c = str.charAt(i);
        if (c == '(') {
            count++;
        } else if (c == ')') {
            count--;
        }
        // If closing parenthesis appears before a matching opening one
        if (count < 0) {
            return false;
        }
    }
    // Balanced only if count ends at 0
    return count == 0;
}
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter expression: ");
    String str = sc.nextLine();
    if (isBalanced(str)) {
        System.out.println("Balanced");
        } else {
            System.out.println("Not balanced");
        }
    }
}
```

Question 3

3.1 Problem Statement

A ticket number is considered "Lucky" if the sum of the digits in the first half of the number equals the sum of the digits in the second half. The ticket number must satisfy two conditions: it must consist of digits only, and it must have an even length. If the input string contains non-digit characters or has an odd length, the program must report it as an "Invalid ticket number." Otherwise, it compares the sums and reports whether the ticket is "Lucky" or "Unlucky."

For example:

- 123321 → First half (1+2+3 = 6), Second half (3+2+1 = 6). Since 6 == 6, it is **Lucky**.
- 123456 → First half (1+2+3 = 6), Second half (4+5+6 = 15). Since 6 != 15, it is **Unlucky**.
- 123 → Odd length. **Invalid ticket number.**

3.2 Solution Implementation

IPO Chart

Input

- **User Input:** The program reads a single line of user input from the console.
- **Data Structure:**
    - The input is stored as a String variable (ticketNumber).
    - The string is expected to contain only numeric digits and have an even length.
- **Input Validation:**
    - The program iterates through the string to check two conditions:
        1. **Character Check:** Ensures every character is a digit using Character.isDigit().
        2. **Length Check:** Ensures the string length is even using length % 2 != 0.

    - If either check fails, the ticket is considered invalid immediately.

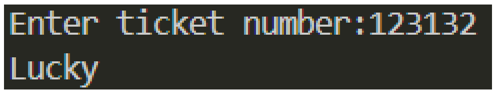
Process

- **Method Call:** The program calls isLuckyTicket(ticketNumber) to validate and process the ticket.
- **Logic within isLuckyTicket():**
    - **Summation Loop:** The program iterates from index 0 to length / 2 (the midpoint).
    - **First Half Calculation:**
        - Extracts the digit at the current index j.
        - Adds it to firstHalfDigit.
    - **Second Half Calculation:**
        - Extracts the digit from the symmetric position at the end of the string (length - 1 - j).
        - Adds it to secondHalfDigit.
    - **Comparison:** Checks if firstHalfDigit is equal to secondHalfDigit.
- **Return Values:**
    - Returns true if the sums match (Lucky).
    - Returns false if the sums do not match (Unlucky) or if the input was invalid.

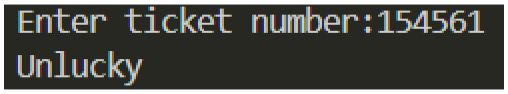
Output

- During the execution of the method, the program directly displays one of the following messages:
    - **Case 1 (Invalid):** Displays Invalid ticket number. if non-digits are found or length is odd.
    - **Case 2 (Lucky):** Displays Lucky if the sum of the first half equals the sum of the second half.
    - **Case 3 (Unlucky):** Displays Unlucky if the sums differ.

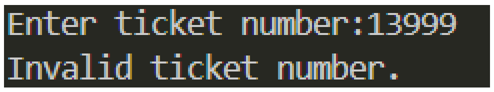
3.3 Sample Input and Output



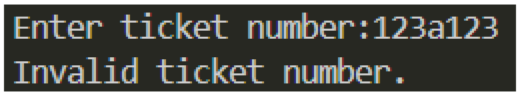
```
Enter ticket number:123132
Lucky
```



```
Enter ticket number:154561
Unlucky
```



```
Enter ticket number:13999
Invalid ticket number.
```



```
Enter ticket number:123a123
Invalid ticket number.
```

3.4 Source Code

```
import java.util.Scanner;
public class Q3 {
  public static boolean isLuckyTicket (String ticketNumber){
      int firstHalfDigit = 0;
      int secondHalfDigit = 0;
      for (int j = 0; j < ticketNumber.length() / 2; j++) {

      firstHalfDigit += Character.getNumericValue(ticketNumber.charAt(j));
      secondHalfDigit +=
Character.getNumericValue(ticketNumber.charAt(ticketNumber.length() - 1 - j));
       }

       if (firstHalfDigit != secondHalfDigit) {
              return false;
       }
       else {
              return true;
       }
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter ticket number:");
```

```
        String ticketNumber = input.nextLine();
        for (int i = 0; i < ticketNumber.length() ; i++) {
          char c = ticketNumber.charAt(i);

          if(!Character.isDigit(c)){
              System.out.println("Invalid ticket number.");
              input.close();
              return;
          }
        }

          if (ticketNumber.length() % 2 != 0) {
              System.out.println("Invalid ticket number.");
              return;
          }

          if (isLuckyTicket(ticketNumber)) {
              System.out.println("Lucky");;
          } else {
              System.out.println("Unlucky");
          }


        input.close();
    }
}
```

Question 4

4.1 Problem Statement

Write a Java program that reads a 3×3 Tic-Tac-Toe board and determines whether **X**, **O**, or **no one** has won. The board contains characters 'X', 'O', and '.'. Implement:

- checkWinner(char[][] board) — returns 'X', 'O', or '.' based on winning rows, columns, or diagonals.

- countMoves(char[][] board, char player) — counts how many times a player appears, useful for validating that X has the same number or one more move than O.

The main method should read the board, call checkWinner(), and print Winner: X, Winner: O, or No winner.

4.2 Solution Implementation

IPO Chart

Input

1. The program reads **three lines** of user input.

2. Each line must contain exactly **three characters**:

    - 'X' → move by player X

    - 'O' → move by player O
    - '.' → empty square

3. These three lines are stored into a **3 × 3 character array** (char[][] board).

4. The program calls countMoves() to count how many moves X and O made.

    - This helps verify that the board is valid:

        - Player X always starts first.

        - Therefore:

            - Number of X moves = number of O moves **OR**

            - Number of X moves = number of O moves + 1

        - If this is not true, the input may not represent a valid game state.

Process

The program calls the method checkWinner(board) to evaluate the board.

Inside checkWinner():

- **Row Check:**
  Each of the 3 rows is checked to see if all three characters are the same and not '.'.
- **Column Check:**
  Each of the 3 columns is checked for the same condition.
- **Diagonal Check:**
  Both diagonals are checked:

    - Left-to-right diagonal
    - Right-to-left diagonal
- If any row, column, or diagonal contains three identical non-. characters:
    - Return 'X' if X wins
    - Return 'O' if O wins

If no winning line is found:

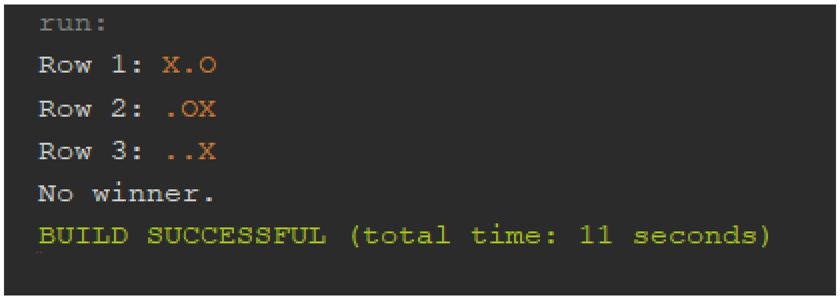
- Return '.' to indicate **no winner**.

Output

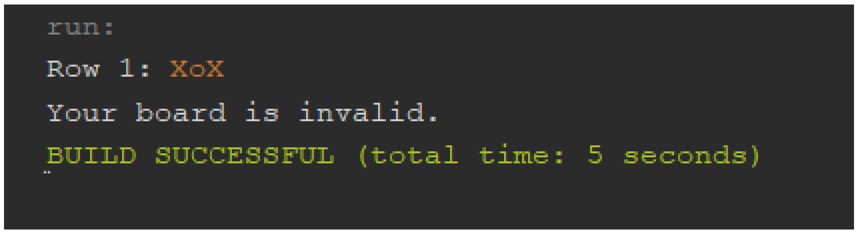
After processing, the program displays one of the following messages:

1. **If X is the winner:**
   Winner: X
2. **If O is the winner:**
   Winner: O
3. **If no winning row, column, or diagonal exists:**
   No winner

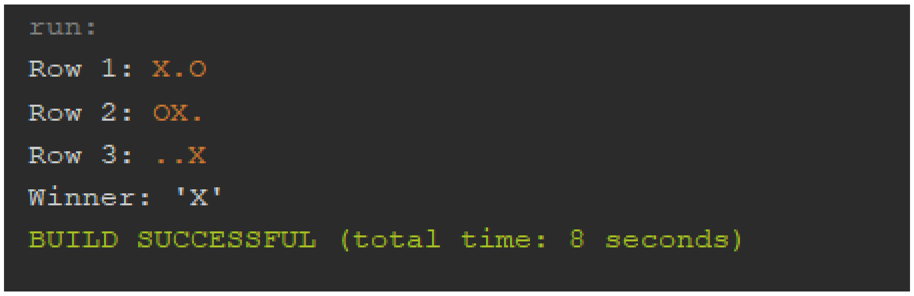
4.3 Sample Input and Output



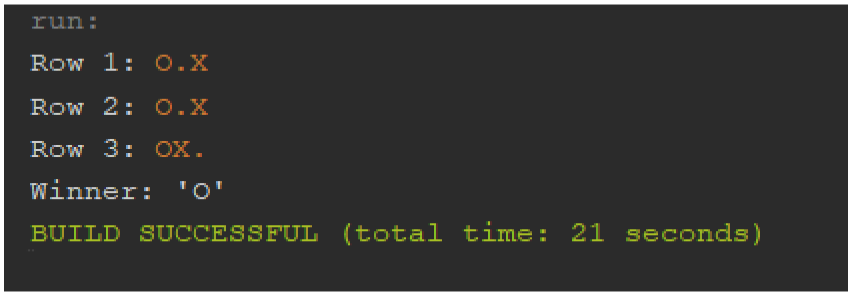
```
run:
Row 1: X.O
Row 2: .OX
Row 3: ..X
No winner.
BUILD SUCCESSFUL (total time: 11 seconds)
```



```
run:
Row 1: XoX
Your board is invalid.
BUILD SUCCESSFUL (total time: 5 seconds)
```



```
run:
Row 1: X.O
Row 2: OX.
Row 3: ..X
Winner: 'X'
BUILD SUCCESSFUL (total time: 8 seconds)
```



```
run:
Row 1: O.X
Row 2: O.X
Row 3: OX.
Winner: 'O'
BUILD SUCCESSFUL (total time: 21 seconds)
```

4.4 Source Code

https://github.com/jiteshmr-2025/viva-

2/blob/main/Q4/src/q4/Q4.java

```
package q4;

import java.util.Scanner;

public class Q4 {

    /**

     * 1. Read the three board rows from the user.

     * 2. Construct a 2D array of

     * characters. 3. Call checkWinner() to determine the result. 4. Display the

     * output as: a. "Winner: X" or "Winner: O" if a player has won, b. "No

     * winner" if the game has no winning line.

     *

     * @param args the command line arguments

     */

    public static void main(String[] args) {

        // Initialize Scanner object

        Scanner input = new Scanner(System.in);



        // Initialize board
```

```
char[][] board = new char[3][3];

for (int i = 0; i < board.length; i++) {

    System.out.printf("Row %d: ", i + 1);

    String temp = input.next();

    // Make sure temp is 3

    if (temp.length() != 3) {

        System.out.println("Invalid number of inputs.");

        return;

    }

    String validChars = "XO.";

    for (char r : temp.toCharArray()) {

        // If "XO." does not contain the character 'r'

        if (validChars.indexOf(r) == -1) {

            System.out.println("Your board is invalid.");

            return;

        }

    }

    for (int j = 0; j < board[0].length; j++) {

        board[i][j] = temp.charAt(j);

    }
```

```
        }


        int countX = countMoves(board, 'X');

        int countO = countMoves(board, 'O');

        // Checking if number of moves is valid

        if (countX == countO || (countX - countO) == 1) {

            // Call checkwinner()

            char winner = checkWinner(board);

            switch (winner) {

                case 'X' ->

                    System.out.println("Winner: 'X' ");

                case 'O' ->

                    System.out.println("Winner: 'O' ");

                default ->

                    System.out.println("No winner.");

            }

        } else {

            System.out.println("Number of moves is not valid.");

        }

    }
```

```
/**
 * Checks how many moves have been played.
 *
 * @param board The main playing board
 * @param player The current character to be checked
 * @return the number of moves made by a player
 */
public static int countMoves(char[][] board, char player) {
    int count = 0;
    for (char[] board1 : board) {
        for (int j = 0; j < board[0].length; j++) {
            if (board1[j] == player) {
                count++;
            }
        }
    }
    return count;
}


/**
```

```
     *
     * @param board The main playing board
     * @return The winner of the game or . if no winner
     */
    public static char checkWinner(char[][] board) {
        // Check if the same symbol appears in any complete row, column, or one of the two
diagonals.
        // Check row and column
        int countRowX = 0;
        int countRowY = 0;
        int countColX = 0;
        int countColY = 0;
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                // Check rows
                if (board[i][j] == 'X') {
                    countRowX++;
                    if (countRowX == 3) {
                        return 'X';
                    }
}

if (board[i][j] == 'O') {

    countRowY++;

    if (countRowY == 3) {

        return 'O';

    }

}

// Check columns

if (board[j][i] == 'X') {

    countColX++;

    if (countColX == 3) {

        return 'X';

    }

}

if (board[j][i] == 'O') {

    countColY++;

    if (countColY == 3) {

        return 'O';

    }

}
```

```
    }

    countRowX = 0;

    countRowY = 0;

    countColX = 0;

    countColY = 0;

}

// Diagonal checking

/*

and also reverse of these

0, 0 -> i = j

1, 1 -> i = j

2, 2 -> i = j

 */

// Everytime method starts we can reset these values

int countDiagForwardX = 0;

int countDiagForwardY = 0;

int countDiagBackX = 0;

int countDiagBackY = 0;

for (int i = 0; i < board.length; i++) {

    for (int j = i; j <= i; j++) {
```

```
if (board[i][j] == 'X') {
    countDiagForwardX++;
    if (countDiagForwardX == 3) {
        return 'X';
    }
}
if (board[i][j] == 'O') {
    countDiagForwardY++;
    if (countDiagForwardY == 3) {
        return 'O';
    }
}
if (board[i][2 - j] == 'X') {
    countDiagBackX++;
    if (countDiagBackX == 3) {
        return 'X';
    }
}
if (board[i][2 - j] == 'O') {
    countDiagBackY++;
```

```
                    if (countDiagBackY == 3) {
                        return 'O';
                    }
                }
            }
        }

        return '.';
    }

}
```

QUESTION 5

5.1 Problems

The program implements Run-Length Encoding (RLE) to either compress or decompress a line of text based on a given mode.

First, the program reads a mode of operation:

- ‘C’ → Compress the input text
- ‘D’ → Decompress the input text

Then, it reads one line of text to process.

Compression (compress)

- Consecutive identical characters are replaced by <count><character>.
- Example:
  AAABCCDDDD → 3A1B2C4D
- Spaces are treated as normal characters and are also compressed.

Decompression (decompress)

- Each encoded group consists of a positive integer followed by a character.
- The method expands the encoded text back to its original form.
- Example:
  3A1B2C4D → AAABCCDDDD
- If the encoded text is invalid (e.g., a number not followed by a character), the program outputs “Invalid encoding.”.

Output Format

- For valid operations:
  Result: <encoded_or_decoded_text>

- For invalid decompression input:
  Invalid encoding.

5.2 Solution Implementation

IPO chart

Input:

- Mode of operation (C for Compress or D for Decompress)
- A line of text (string)

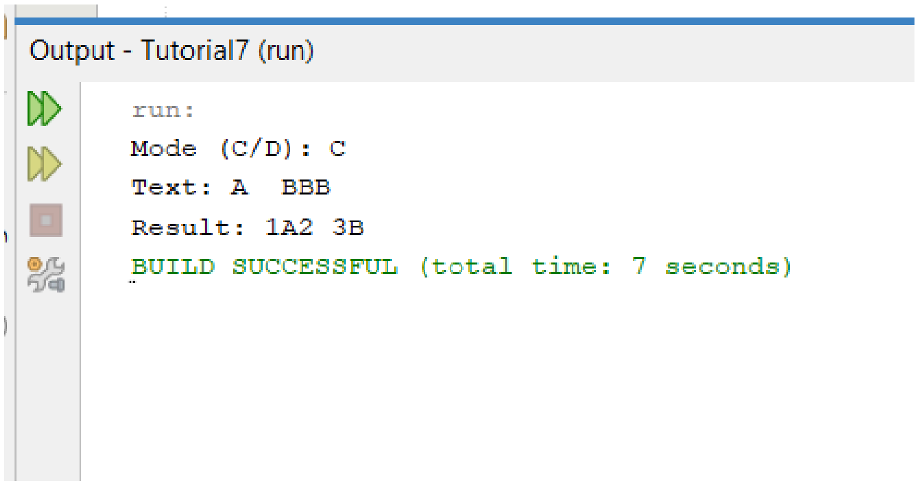
Process:

- If mode is C (Compress):
    - Traverse the string character by character
    - Count consecutive identical characters
    - Append <count><character> to the result
- If mode is D (Decompress):
    - Read digits to form a number
    - Check that each number is followed by a character
    - Repeat the character according to the number
    - If encoding is invalid, stop and display error

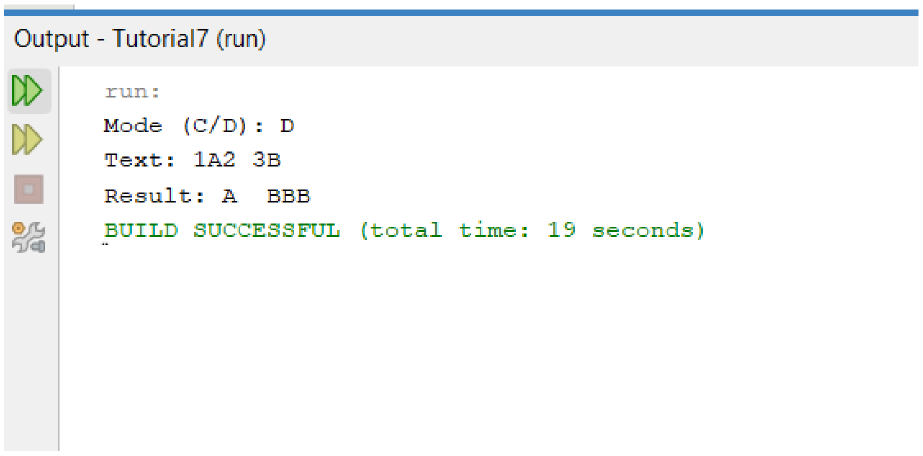
Output:

- For valid operations:
    - Result: <compressed_or_decompressed_text>
- For invalid decompression input:
    - “Invalid encoding.”

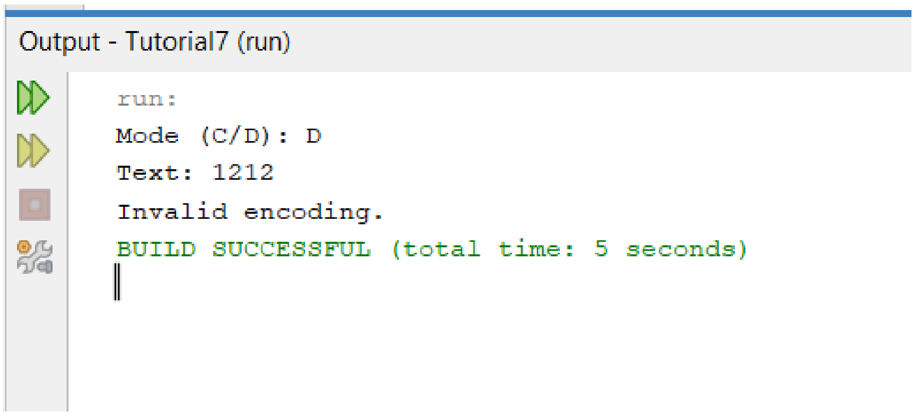
5.3 Sample Input and Output


Output - Tutorial7 (run)

```
run:
Mode (C/D): C
Text: A  BBB
Result: 1A2 3B
BUILD SUCCESSFUL (total time: 7 seconds)
```


Output - Tutorial7 (run)

```
run:
Mode (C/D): D
Text: 1A2 3B
Result: A  BBB
BUILD SUCCESSFUL (total time: 19 seconds)
```


Output - Tutorial7 (run)

```
run:
Mode (C/D): D
Text: 1212
Invalid encoding.
BUILD SUCCESSFUL (total time: 5 seconds)
```

5.4 Source Code

```
import java.util.Scanner;
public class Q5 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Mode (C/D): ");
        char mode = sc.nextLine().charAt(0);

        System.out.print("Text: ");
        String text = sc.nextLine();

        if (mode == 'C') {
            String output = compress(text);
                        if (output.equals("Invalid encoding.")) {
                            System.out.println(output);
                        } else {
                            System.out.println("Result: " + output);
                        }
        }
        else if (mode == 'D') {
            String output = decompress(text);
            if (output.equals("Invalid encoding.")) {
                System.out.println(output);
            } else {
                System.out.println("Result: " + output);
            }
        }
        sc.close();
```

```
}

public static String compress(String s) {
    //check whether is empty string
    if (s.length() == 0) return "";
    for(int i=0;i<s.length();i++){
            if(Character.isDigit(s.charAt(i)))
                    return "Invalid encoding.";
    }

    StringBuilder result = new StringBuilder();
    int count = 1;
    for (int i = 1; i < s.length(); i++) {
        //calculate count for same character by comparing with previous character
        if (s.charAt(i) == s.charAt(i - 1)) {
            count++;
        } else {
            result.append(count).append(s.charAt(i - 1));
            count = 1;
        }
    }
    // append last run
    result.append(count).append(s.charAt(s.length() - 1));
    return result.toString();
}

public static String decompress(String s) {
    StringBuilder result = new StringBuilder();
    int i = 0;
    while (i < s.length()) {
        // must start with a digit
```

```
            if (!Character.isDigit(s.charAt(i))) {
                return "Invalid encoding.";
            }

            int count = 0;
            // read full number (handles multi-digit numbers)
            while (i < s.length() && Character.isDigit(s.charAt(i))) {
                count = count * 10 + (s.charAt(i) - '0');
                i++;
            }
            // number must be followed by a character
            if (i >= s.length()) {
                return "Invalid encoding.";
            }

            char ch = s.charAt(i);
            // expand
            for (int j = 0; j < count; j++) {
                result.append(ch);
            }
            i++; // move to next segment
        }

        return result.toString();
    }
}
```

Github:
https://github.com/jiteshmr-2025/viva-2/blob/20cc966be0f0d71f8dd89a36f238f12f7d468b56/Q5.java

Question 6

6.1 Problem

The problem requires developing a program that checks whether two integer arrays form a mirror pattern. Two arrays are considered mirror patterns if the first element of the first array matches the last element of the second array, the second element matches the second last, and so on.

The program must ensure that both arrays have a valid length between 1 and 50. If the input length is invalid, the user must be prompted to re-enter the data. After validation, the program compares both arrays and determines whether they are mirror patterns. The final result is displayed as either true or false.

6.2 Solution

Read and validate input arrays

a. Prompt the user to enter Array A as comma-separated integers.
b. Split the input into a string array.
c. Check whether the number of elements is between 1 and 50.
d. If invalid, prompt the user to re-enter the input.
e. Repeat the same steps for Array B.

Convert input to integer arrays

a. Create integer arrays for A and B based on their validated lengths.
b. Loop through each string element, remove whitespace, and convert it to an integer using Integer.parseInt.

Check mirror pattern

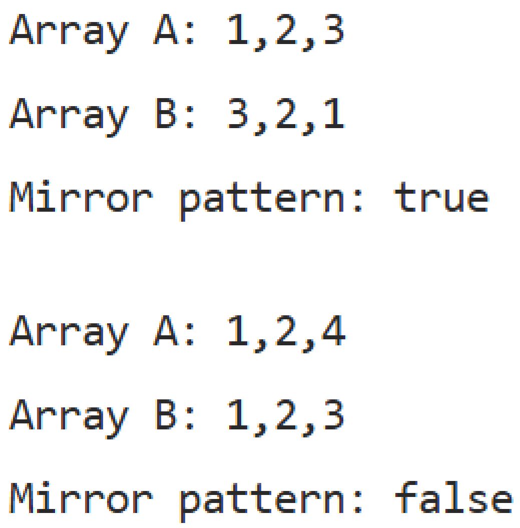
a. Compare the lengths of both arrays.
b. If the lengths are different, return false.
c. Loop through the arrays and compare each element of Array A with the corresponding reversed element of Array B.
d. If any pair does not match, return false immediately.

Display the result

a. If all elements satisfy the mirror condition, return true.

b. Print the mirror pattern result to the user.

6.3 Sample Input and Output



```
Array A: 1,2,3

Array B: 3,2,1

Mirror pattern: true


Array A: 1,2,4

Array B: 1,2,3

Mirror pattern: false
```

6.4 Source Code

```
import java.util.Scanner;
public class Q6 {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String[] aStr;
        String[] bStr;

        do {
            System.out.print("Array A: ");
            aStr = sc.nextLine().split(",");
        } while (!checkValid(aStr));

        do {
            System.out.print("Array B: ");
            bStr = sc.nextLine().split(",");
        } while (!checkValid(bStr));

        int[] a = new int[aStr.length];
        int[] b = new int[bStr.length];

        for (int i = 0; i < a.length; i++)
            a[i] = Integer.parseInt(aStr[i].trim());

        for (int i = 0; i < b.length; i++)
            b[i] = Integer.parseInt(bStr[i].trim());

        System.out.println("Mirror pattern: " + isMirror(a, b));

        sc.close();
```

```java
    }

    // Checks if array length is between 1 and 50
    static boolean checkValid(String[] arr) {
        if (arr.length < 1 || arr.length > 50) {
            System.out.println("Invalid input. Array length must be between 1 and 50.");
            return false;
        }
        return true;
    }

    static boolean isMirror(int[] a, int[] b) {
        if (a.length != b.length) return false;

        for (int i = 0; i < a.length; i++) {
            if (a[i] != b[b.length - 1 - i]) {
                return false;
            }
        }
        return true;
    }
}
```