Q.1 Can you explain the logic and working of the Tower of Hanoi algorithm by writing a Java program?

How does the recursion work, and how are the movements of disks between rods accomplished?

The **Tower of Hanoi** is a classic problem in recursion. The problem involves three rods and nnn disks of different sizes, initially stacked in decreasing order on the first rod. The goal is to move all the disks to the third rod following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or on an empty rod.
3. Only the topmost disk of a stack can be moved.

Recursive Logic:

1. Move n−1n-1n−1 disks from **source** to **auxiliary** rod using **destination** as a helper.
2. Move the largest disk (nth disk) directly from **source** to **destination**.
3. Move the n−1n-1n−1 disks from **auxiliary** to **destination** using **source** as a helper.

This forms the recursive breakdown:

T(n)=2T(n−1)+1T(n) = 2T(n-1) + 1T(n)=2T(n−1)+1

Solving this recurrence gives:

T(n)=2n−1T(n) = 2^n - 1T(n)=2n−1

which represents the minimum number of moves needed.

Q.2 Given two strings word1 and word2, return the minimum number of operations required to convert word1 to word2.

Example 1:

Input: word1 = "horse", word2 = "ros"

Output: 3

Explanation:

horse -> rorse (replace 'h' with 'r')

rorse -> rose (remove 'r')

rose -> ros (remove 'e')

We define **edit distance** as the minimum number of operations to convert word1 to word2, using:

1. **Insertion** (Insert a character into word1).
2. **Deletion** (Delete a character from word1).
3. **Replacement** (Replace a character in word1).

Dynamic Programming Approach:

1. **Define dp[i][j]** as the minimum operations required to convert the first i characters of word1 to the first j characters of word2.
2. **Base Case:**
   - If word1 is empty, we need j insertions.
   - If word2 is empty, we need i deletions.
3. **Recurrence Relation:**
   - If word1[i-1] == word2[j-1], no operation is needed: dp[i][j]=dp[i−1][j−1]dp[i][j] = dp[i-1][j-1]dp[i][j]=dp[i−1][j−1]
   - Otherwise, take the minimum of:
     - **Insert** (dp[i][j-1] + 1)
     - **Delete** (dp[i-1][j] + 1)
     - **Replace** (dp[i-1][j-1] + 1)

Step-by-step conversion

1. **Replace** 'h' → 'r' → "rorse"
2. **Delete** 'r' → "rose"
3. **Delete** 'e' → "ros"

Time & Space Complexity

- **Time Complexity:** O(m×n)O(m \times n)O(m×n), since we fill an m×nm \times nm×n table.
- **Space Complexity:** O(m×n)O(m \times n)O(m×n), as we use a 2D DP table.

Space Optimization

We can optimize space to **O(n)O(n)O(n)** by using only two rows (prev and curr), since each row only depends on the previous row.

Example 2:

Input: word1 = "intention", word2 = "execution"

Output: 5

Explanation:

intention -> inention (remove 't')

inention -> enention (replace 'i' with 'e')

enention -> exention (replace 'n' with 'x')

exention -> exection (replace 'n' with 'c')

exection -> execution (insert 'u')

Input:
word1 = "intention"
word2 = "execution"

Step-by-Step Execution:

1. **Remove** 't': "intention" → "inention"
2. **Replace** 'i' with 'e': "inention" → "enention"
3. **Replace** 'n' with 'x': "enention" → "exention"
4. **Replace** 'n' with 'c': "exention" → "exection"
5. **Insert** 'u': "exection" → "execution"

Total operations: 5

Q. 3 Print the max value of the array [13, 1, -3, 22, 5].

Method 1: Using max()

arr = [13, 1, -3, 22, 5]

print("Max value:", max(arr))

Method 2: Using a Loop

arr = [13, 1, -3, 22, 5]

max_val = arr[0] # Initialize with the first element

for num in arr:

**   if num > max_val:**

**   max_val = num**

print("Max value:", max_val)

Q.4 Find the sum of the values of the array [92, 23, 15, -20, 10].

Method 1: Using sum()

python

CopyEdit

```
arr = [92, 23, 15, -20, 10]
print("Sum of array values:", sum(arr))
```

Method 2: Using a Loop

python

CopyEdit

```
arr = [92, 23, 15, -20, 10]
total_sum = 0  # Initialize sum

for num in arr:
   total_sum += num  # Add each element to total_sum

print("Sum of array values:", total_sum)
```

Q.5 Given a number n. Print if it is an armstrong number or not.An armstrong number is a number if the sum of every digit in that number raised to the power of total digits in that number is equal to the number.

Example : 153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153 hence 153 is an armstrong number. (Easy)

Input1 : 153

Output1 : Yes

Input 2 : 134

Output2 : No

```
def is_armstrong_number(n: int) -> str:
    # Convert number to string to easily iterate through digits
    digits = str(n)
    num_digits = len(digits)

    # Sum of digits raised to the power of the number of digits
    sum_of_powers = sum(int(digit) ** num_digits for digit in digits)

    # Check if the sum equals the original number
    if sum_of_powers == n:
        return "Yes"
    else:
        return "No"

# Example Usage
input1 = 153
input2 = 134
print(is_armstrong_number(input1))  # Output: Yes
print(is_armstrong_number(input2))  # Output: No
```