Experiment Title: To implement programs on problem solving using Dynamic Programming Approach – Scenario2.

Aim/Objective: To understand the concept and implementation of programs on Dynamic Programming approach-based Problems.

Description: The students will understand and able to implement programs on Dynamic Programming Approach based Problems.

Pre-Requisites:

Knowledge: Dynamic Programming approach and its related problems in C/ java/Python.

Tools: Code Blocks/Eclipse IDE.

Pre-Lab:

1. Given two strings s1 and s2, find the length of their longest common subsequence (LCS). A subsequence is a sequence derived from another sequence by deleting some elements without changing the order of the remaining elements.

   Input Format:

   - A string *s1* of length m.
   - A string *s2* of length n.

   Output Format :

   A single integer, the length of the LCS.

   Constraints:

   - $1 \leq m,n \leq 10^3$
   - s1,s2 consist of lowercase English letters.

- **Example Input:**

  abcde

  ace

- **Example Output:**

  3

- **Procedure/Program**:

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

int max(int a, int b) {
    return (a > b) ? a : b;
}

int lcs(char *s1, char *s2, int m, int n) {
    int dp[m + 1][n + 1];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                dp[i][j] = 0;
            else if (s1[i - 1] == s2[j - 1])
                dp[i][j] = 1 + dp[i - 1][j - 1];
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
    return dp[m][n];
}

int main() {
    char s1[1001], s2[1001];
    scanf("%s %s", s1, s2);
    printf("%d", lcs(s1, s2, strlen(s1), strlen(s2)));
    return 0;
}
```

- **Data and Results:**

DATA:

Two input strings are given to find the longest subsequence.

RESULT:

The length of the longest common subsequence is determined.

- **Analysis and Inferences:**

ANALYSIS:

Dynamic programming is used to compute LCS efficiently.

INFERENCES:

LCS helps in text comparison, bioinformatics, and data analysis.

In-Lab:

Christy is interning at Dairymilk. One day she has to distribute some chocolates to her colleagues. She is biased towards her friends and plans to give them more than the others. One of the program managers hears of this and tells her to make sure everyone gets the same number.

To make things difficult, she must equalize the number of chocolates in a series of operations. For each operation, she can give 1,2 or 5 pieces to all but one colleague. Everyone who gets a piece in a round receives the same number of pieces.

Given a starting distribution, calculate the minimum number of operations needed so that every colleague has the same number of pieces.

Example

arr = [1,1,5]

arr represents the starting numbers of pieces for each colleague. She can give 2 pieces to the first two and the distribution is then [3,3,5] . On the next round, she gives the same two 2 pieces each, and everyone has the same number: [5,5,5] . Return the number of rounds, 2.

Function Description:

Complete the *equal* function in the editor below.

equal has the following parameter(s):

- *int arr[n]:* the integers to equalize

Returns int: the minimum number of operations required

Input Format

The first line contains an integer t , the number of test cases.

Each test case has 2 lines.

- The first line contains an integer n , the number of colleagues and the size of arr .

- The second line contains n space-separated integers, , arr[i], the numbers of pieces of chocolate each colleague has at the start.

Constraints

$1 \leq t \leq 100$

$1 \leq n \leq 10000$

The number of chocolates each colleague has initially < 1000.

Sample Input

```
STDIN       Function
----        ------
1           t = 1
4           arr[] size n = 4
2 2 3 7     arr =[2, 2, 3, 7]
```

Sample Output

```
2
```

- **Procedure/Program**:

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

int equal(int arr[], int n) {
   int min_operations = INT_MAX;
   for (int i = 0; i <= 4; i++) {
      int target = arr[0] - i;
      int operations = 0;
      for (int j = 0; j < n; j++) {
         int diff = arr[j] - target;
         if (diff > 0) {
            operations += (diff / 5) + (diff % 5 / 2) + (diff % 5 % 2);
         }
      }
      if (operations < min_operations) {
         min_operations = operations;
      }
   }
   return min_operations;
}

int main() {
   int t = 1;
   int arr[] = {2, 2, 3, 7};
   int n = sizeof(arr) / sizeof(arr[0]);
```

```
  for (int i = 0; i < t; i++) {
    printf("%d\n", equal(arr, n));
  }
  return 0;
}
```

- **Data and Results:**

Data

Given an array, minimize operations to equalize all elements.

Result

Minimum operations required to equalize array elements efficiently.

- **Analysis and Inferences:**

Analysis

Iterate over possible targets, compute operations, find the minimum.

Inferences

Lowering target values reduces overall operations for equalization.

2. You can perform the following operations on the string, *a* :
    1. Capitalize zero or more of *a*'s lowercase letters.
    2. Delete all of the remaining lowercase letters in *a*.

Given two strings, *a and b* , determine if it's possible to make a equal to b as described. If so, print YES on a new line. Otherwise, print NO.

For example, given a = AbcDE and b = ABDE, in a we can convert b and delete c to match b . If a = AbcDE and b = AFDE, matching is not possible because letters may only be capitalized or discarded, not changed.

Function Description

Complete the function *abbreviation* in the editor below. It must return either YES or NO .

abbreviation has the following parameter(s):

a: the string to modify

b: the string to match

Input Format

The first line contains a single integer q , the number of queries.

Each of the next q pairs of lines is as follows:

- The first line of each query contains a single string, a.
- The second line of each query contains a single string, b.

Constraints

$1 \leq q \leq 10$

$1 \leq a,b \leq 1000$

Output Format

For each query, print YES on a new line if it's possible to make string a equal to string b . Otherwise, print NO.

Sample Input

1

daBcd

ABC

Sample Output

YES

- **Procedure/Program**:

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

int abbreviation(char *a, char *b) {
    int m = strlen(a), n = strlen(b);
    int dp[m + 1][n + 1];

    memset(dp, 0, sizeof(dp));
    dp[0][0] = 1;

    for (int i = 1; i <= m; i++) {
        dp[i][0] = dp[i - 1][0] && islower(a[i - 1]);
    }

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (toupper(a[i - 1]) == b[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] || (islower(a[i - 1]) && dp[i - 1][j]);
            } else if (islower(a[i - 1])) {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[m][n];
```

```
}

int main() {
    int q;
    scanf("%d", &q);
    while (q--) {
        char a[1001], b[1001];
        scanf("%s %s", a, b);
        printf("%s\n", abbreviation(a, b) ? "YES" : "NO");
    }
    return 0;
}
```

Post-Lab:

You are given n items, each with a weight w[i] and value v[i], and a knapsack of capacity W. Determine the maximum value you can achieve by selecting items without exceeding the capacity of the knapsack.

Input Format:

- An integer *n*, the number of items.
- An integer *W*, the capacity of the knapsack.
- Two arrays of integers, *w* and *v*, each of size n, where *w[i]* is the weight and *v[i]* is the value of the i^{th} item.

Output Format:

- A single integer, the maximum value that can be achieved.

Constraints:

- $1 \leq n \leq 10^3$
- $1 \leq W \leq 10^4$
- $1 \leq w[i], v[i] \leq 10^3$

Example Input:

4 5

2 3 4 5

3 4 5 6

Example Output:

7

- **Procedure/Program**:

```
#include <stdio.h>

int max(int a, int b) {

   return (a > b) ? a : b;

}

int knapsack(int W, int w[], int v[], int n) {

   int i, j;

   int K[n + 1][W + 1];


   for (i = 0; i <= n; i++) {

      for (j = 0; j <= W; j++) {

         if (i == 0 || j == 0)

            K[i][j] = 0;

         else if (w[i - 1] <= j)

            K[i][j] = max(v[i - 1] + K[i - 1][j - w[i - 1]], K[i - 1][j]);


        else

            K[i][j] = K[i - 1][j];

      }

   }
```

```
    return K[n][W];
}


int main() {
    int n, W;
    scanf("%d %d", &n, &W);
    int w[n], v[n];

    for (int i = 0; i < n; i++)
        scanf("%d", &w[i]);
    for (int i = 0; i < n; i++)
        scanf("%d", &v[i]);

    printf("%d\n", knapsack(W, w, v, n));
    return 0;
}
```

- **Data and Results:**

DATA:

String transformation is analyzed using a dynamic programming approach.

RESULT:

Transformation feasibility is determined and displayed as YES or NO.

- **Analysis and Inferences :**

ANALYSIS:

A 2D table efficiently tracks transformation states step by step.

INFERENCES:

The algorithm effectively processes string modifications within given constraints.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. Why does solving some problems using recursion lead to stack overflow, but not when solved using dynamic programming?

Recursion uses deep calls, leading to stack overflow; DP avoids it with iteration.

2. Can you reduce the space complexity of the Longest Increasing Subsequence problem? How?

Use `O(n log n)` binary search or `O(n)` space DP for LIS.

3. Describe a scenario where a greedy algorithm fails, but dynamic programming succeeds.

Greedy fails in **0/1 Knapsack** as it misses optimal solutions; DP checks all cases.

4. In a DP problem, how do you decide the order of computation in a bottom-up approach?

Compute smaller subproblems first based on dependencies in bottom-up DP.

5. How would you debug a DP solution that gives incorrect results?

Check base cases, print DP table, verify recurrence to debug DP.

Evaluator Remark (if Any):	**Marks Secured ___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.