LAB SHEET -6

Name: AMANDEEP

Roll No.: 2301010200

AIM 🡪 Longest increasing subsequence

The Longest Increasing Subsequence (LIS) problem is a classic problem in computer science. The goal is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

**Problem Definition**

* **Input**: An array of integers.
* **Output**: The length of the longest increasing subsequence.

**Dynamic Programming Approach**

The dynamic programming approach to solve the LIS problem involves maintaining an array **dp** where **dp[i]** represents the length of the longest increasing subsequence that ends with the element at index **i**.

**Steps:**

1. Initialize a **dp** array of the same length as the input array, with all values set to 1 (since the minimum length of an increasing subsequence is 1).
2. For each element in the array, check all previous elements to see if they can form an increasing subsequence.
3. If **arr[j] < arr[i]**, then update **dp[i]** as **dp[i] = max(dp[i], dp[j] + 1)**.
4. The result will be the maximum value in the **dp** array.

**C++ Code Implementation**

Here is the C++ code for finding the length of the Longest Increasing Subsequence:

cpp

RunCopy code

1#include <iostream>

2#include <vector>

3#include <algorithm>

4

5using namespace std;

6

7// Function to find the length of the Longest Increasing Subsequence

8int longestIncreasingSubsequence(const vector<int>& arr) {

9 int n = arr.size();

10 if (n == 0) return 0;

11

12 // Create a DP array initialized to 1

13 vector<int> dp(n, 1);

14

15 // Fill the DP array

16 for (int i = 1; i < n; i++) {

17 for (int j = 0; j < i; j++) {

18 if (arr[j] < arr[i]) {

19 dp[i] = max(dp[i], dp[j] + 1);

20 }

21 }

22 }

23

24 // The length of the longest increasing subsequence is the maximum value in dp

25 return \*max\_element(dp.begin(), dp.end());

26}

27

28int main() {

29 vector<int> arr = {10, 22, 9, 33, 21, 50, 41, 60, 80};

30

31 int length = longestIncreasingSubsequence(arr);

32 cout << "Length of Longest Increasing Subsequence: " << length << endl;

33

34 return 0;

35}

**Explanation of the Code**

1. **Input Initialization**: We define a vector **arr** containing the sequence of integers for which we want to find the LIS.
2. **DP Array Creation**: We create a **dp** vector initialized to 1, as the minimum length of an increasing subsequence is 1 (each element itself).
3. **Filling the DP Array**:
   * We iterate through each element of the array starting from the second element.
   * For each element, we check all previous elements to see if they can form an increasing subsequence.
   * If **arr[j] < arr[i]**, we update **dp[i]** to be the maximum of its current value and **dp[j] + 1**.
4. **Result**: The length of the longest increasing subsequence is found by taking the maximum value from the **dp** array.

**Complexity**

* **Time Complexity**: O(n^2), where **n** is the number of elements in the input array.
* **Space Complexity**: O(n) for the **dp** array.

A close-up of a sign

AI-generated content may be incorrect.

Aim 🡪 Longest common subsequence

The Longest Common Subsequence (LCS) problem is a classic problem in computer science and is often encountered in applications such as version control systems, diff tools, and bioinformatics. The goal is to find the longest subsequence that is common to two sequences.

**Problem Definition**

* **Input**: Two sequences (strings) **X** and **Y**.
* **Output**: The length of the longest common subsequence.

**Dynamic Programming Approach**

The dynamic programming approach to solve the LCS problem involves maintaining a 2D array **dp** where **dp[i][j]** represents the length of the longest common subsequence of the first **i** characters of **X** and the first **j** characters of **Y**.

**Steps:**

1. Initialize a 2D array **dp** of size **(m+1) x (n+1)** with all values set to 0, where **m** is the length of **X** and **n** is the length of **Y**.
2. Iterate through each character of both strings:
   * If **X[i-1] == Y[j-1]**, then **dp[i][j] = dp[i-1][j-1] + 1**.
   * Otherwise, **dp[i][j] = max(dp[i-1][j], dp[i][j-1])**.
3. The result will be found in **dp[m][n]**, which gives the length of the longest common subsequence.

**C++ Code Implementation**

Here is the C++ code for finding the length of the Longest Common Subsequence:

cpp

RunCopy code

1#include <iostream>

2#include <vector>

3#include <string>

4#include <algorithm>

5

6using namespace std;

7

8// Function to find the length of the Longest Common Subsequence

9int longestCommonSubsequence(const string& X, const string& Y) {

10 int m = X.length();

11 int n = Y.length();

12

13 // Create a DP table with dimensions (m+1) x (n+1)

14 vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

15

16 // Fill the DP table

17 for (int i = 1; i <= m; i++) {

18 for (int j = 1; j <= n; j++) {

19 if (X[i - 1] == Y[j - 1]) {

20 dp[i][j] = dp[i - 1][j - 1] + 1; // Characters match

21 } else {

22 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]); // Take the maximum

23 }

24 }

25 }

26

27 // The length of the longest common subsequence is in dp[m][n]

28 return dp[m][n];

29}

30

31int main() {

32 string X = "AGGTAB";

33 string Y = "GXTXAYB";

34

35 int length = longestCommonSubsequence(X, Y);

36 cout << "Length of Longest Common Subsequence: " << length << endl;

37

38 return 0;

39}

**Explanation of the Code**

1. **Input Initialization**: We define two strings **X** and **Y** for which we want to find the LCS.
2. **DP Table Creation**: We create a 2D vector **dp** initialized to 0. The size is **(m+1) x (n+1)** to accommodate the base case (0 characters).
3. **Filling the DP Table**:
   * We iterate through each character of both strings starting from index 1.
   * If the characters at the current indices match (**X[i-1] == Y[j-1]**), we set **dp[i][j]** to **dp[i-1][j-1] + 1**.
   * If they do not match, we take the maximum value from either excluding the current character of **X** or **Y**.
4. **Result**: The length of the longest common subsequence is found in **dp[m][n]**.

**Complexity**

* **Time Complexity**: O(m \* n), where **m** is the length of string **X** and **n** is the length of string **Y**.
* **Space Complexity**: O(m \* n) for the DP table.

