Dynamic Programming – 1 Week

Graphs – 1 Week

Trees – 1 Week

Stacks/Queues – 1 Week

Strings and Arrays – 1 Week

Linked Lists – 1 Week

Dynamic Programming:

1. **Longest Common Subsequence:**

***Problem Statement:*** Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”. So a string of length n has 2^n-1 different possible subsequences, which implies that the time complexity of the brute force approach will be O(n \* 2n). Note that it takes O(n) time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.

**def lcs(X , Y):**

***# find the length of the strings***

**m = len(X)**

**n = len(Y)**

***# declaring the array for storing the dp values***

**L = [[None]\*(n+1) for i in xrange(m+1)]**

***"""Following steps build L[m+1][n+1] in bottom up fashion***

***Note: L[i][j] contains length of LCS of X[0..i-1]***

***and Y[0..j-1]"""***

**for i in range(m+1):**

**for j in range(n+1):**

**if i == 0 or j == 0:**

**L[i][j] = 0**

**elif X[i-1] == Y[j-1]:**

**L[i][j] = L[i-1][j-1]+1**

**else:**

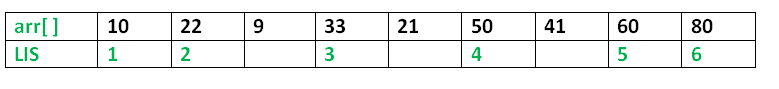
**L[i][j] = max(L[i-1][j], L[i][j-1])**

***# L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]***

**return L[m][n]**

***#end of function lcs***

1. **Longest Increasing subsequence**

***Problem Statement:*** The Longest Increasing Subsequence (LIS) problem 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. For example, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.  


***# lis returns length of the longest increasing subsequence***

***# in arr of size n***

**def lis(arr):**

**n = len(arr)**

***# Declare the list (array) for LIS and initialize LIS***

***# values for all indexes***

**lis = [1]\*n**

***# Compute optimized LIS values in bottom up manner***

**for i in range (1 , n):**

**for j in range(0 , i):**

**if arr[i] > arr[j] and lis[i]< lis[j] + 1 :**

**lis[i] = lis[j]+1**

***# Initialize maximum to 0 to get the maximum of all***

***# LIS***

**maximum = 0**

***# Pick maximum of all LIS values***

**for i in range(n):**

**maximum = max(maximum , lis[i])**

**return maximum**

***# end of list function***

1. **Edit Distance:**

***Problem Statement:*** Given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert ‘str1’ into ‘str2’.

1. Insert
2. Remove
3. Replace

All of the above operations are of equal cost.

***# A Dynamic Programming based Python program for edit***

***# distance problem***

**def editDistDP(str1, str2, m, n):**

***# Create a table to store results of subproblems***

**dp = [[0 for x in range(n+1)] for x in range(m+1)]**

***# Fill d[][] in bottom up manner***

**for i in range(m+1):**

**for j in range(n+1):**

***# If first string is empty, only option is to***

***# insert all characters of second string***

**if i == 0:**

**dp[i][j] = j    # Min. operations = j**

***# If second string is empty, only option is to***

***# remove all characters of second string***

**elif j == 0:**

**dp[i][j] = i    # Min. operations = i**

***# If last characters are same, ignore last char***

***# and recur for remaining string***

**elif str1[i-1] == str2[j-1]:**

**dp[i][j] = dp[i-1][j-1]**

***# If last character are different, consider all***

***# possibilities and find minimum***

**else:**

**dp[i][j] = 1 + min(dp[i][j-1],        # Insert**

**dp[i-1][j],        # Remove**

**dp[i-1][j-1])    # Replace**

**return dp[m][n]**

***# Driver program***

**str1 = "sunday"**

**str2 = "saturday"**

**print(editDistDP(str1, str2, len(str1), len(str2)))**

1. **Count the number of ways to cover a distance**

***Problem statement:*** Given a distance ‘dist, count total number of ways to cover the distance with 1, 2 and 3 steps.

***EX:***

Input: n = 3

Output: 4

Below are the four ways

1 step + 1 step + 1 step

1 step + 2 step

2 step + 1 step

3 step

Input: n = 4

Output: 7

**def printCountDP(dist):**

**count = [0] \* (dist + 1)**

***# Initialize base values. There is***

***# one way to cover 0 and 1 distances***

***# and two ways to cover 2 distance***

**count[0] = 1**

**count[1] = 1**

**count[2] = 2**

***# Fill the count array in bottom***

***# up manner***

**for i in range(3, dist + 1):**

**count[i] = (count[i-1] +**

**count[i-2] + count[i-3])**

**return count[dist];**

***# driver program***

**dist = 4;**

**print( printCountDP(dist))**

1. Find the longest path in a matrix with given constraints

Given a n\*n matric where all numbers are distinct, find the maximum length path (starting from any cell) such that all cells along the path are in increasing order with a difference of 1. We can move in 4 directions from a given cell (i,j), i.e., we can move to (i+1,j) or (i,j+1) or (i-1,j) or (I,j-1) with the condition that adjacent cells have difference of 1.

**n=3**

***# Returns length of the longest path beginning with mat[i][j].***

***# This function mainly uses lookup table dp[n][n]***

**def findLongestFromACell(i, j, mat, dp):**

***# Base case***

**if (i<0 or i>=n or j<0 or j>=n):**

**return 0**

***# If this subproblem is already solved***

**if (dp[i][j] != -1):**

**return dp[i][j]**

***# Since all numbers are unique and in range from 1 to n\*n,***

***# there is atmost one possible direction from any cell***

**if (j<n-1 and ((mat[i][j] +1) == mat[i][j+1])):**

**dp[i][j] = 1 + findLongestFromACell(i,j+1,mat,dp)**

**return dp[i][j]**

**if (j>0 and (mat[i][j] +1 == mat[i][j-1])):**

**dp[i][j] = 1 + findLongestFromACell(i,j-1,mat,dp)**

**return dp[i][j]**

**if (i>0 and (mat[i][j] +1 == mat[i-1][j])):**

**dp[i][j] = 1 + findLongestFromACell(i-1,j,mat,dp)**

**return dp[i][j]**

**if (i<n-1 and (mat[i][j] +1 == mat[i+1][j])):**

**dp[i][j] = 1 + findLongestFromACell(i+1,j,mat,dp)**

**return dp[i][j]**

***# If none of the adjacent fours is one greater***

**dp[i][j] = 1**

**return dp[i][j]**

***# Returns length of the longest path beginning with any cell***

**def finLongestOverAll(mat):**

**result = 1 # Initialize result**

***# Create a lookup table and fill all entries in it as -1***

**dp=[[-1 for i in range(n)]for i in range(n)]**

***# Compute longest path beginning from all cells***

**for i in range(n):**

**for j in range(n):**

**if (dp[i][j] == -1):**

**findLongestFromACell(i, j, mat, dp)**

***# Update result if needed***

**result = max(result, dp[i][j]);**

**return result**

***# Driver program***

**mat = [[1, 2, 9],**

**[5, 3, 8],**

**[4, 6, 7]]**

**print("Length of the longest path is ",finLongestOverAll(mat))**