CS142, Programming I **Due: 23:59, Wed 02/08/2023**

Name: Majd Kawak / NetID: mkawa025 / CodeForces ID: mkawa025 / Student ID: 862273310

A. Ski

Submission#: 192560785

In this program, I start off with allocating 2 *int* to get row and column size, two 2D *vector* for input values(heights) "Resort Matrix" and for path to solve at each point "Matrix to Solve". I start by taking input heights for "Resort Matrix", then I go and compute longest path at each point save path value to "Matrix to Solve". I implemented a function that computes all possible paths at each point, checking 4 possible paths using $if\ [i,j] > [(i-1,j)(i-1,j)], [(i+1,j)(i+1,j)], [(i,j-1)(i,j-1)], [(i,j+1)]\ then\ go\ and\ compute\ path$. This is done recursively till all points in "Resort Matrix" are computed. Display longest path at the end. Time complexity is $O(n*n) = O(n^2)$.

B. Maximum Tree Path

Submission#: 192804162

In this program, firstly I declared an *int* that would hold node count n, a *vector* to hold node *weight*, a *vector* to hold node *parent* and a *vector* of *vectors* to hold my *AdjacencyList* tree representation. After taking all inputs I pass all my all vectors and int n to a helper function to calculate max path sum $findMaxSum$, inside that function I declare an int $maxSum$ that would hold root *weight* in case tree size is 1. $findMaxSum$ passes all previous input with new declared int to compute max path sum $findMaxSumPath$, this function uses recursion below:

$$Weight[i] = max \begin{cases} w_i \\ w_i + f_i \\ w_i + f_j + f_k \end{cases}$$

Where:
w_i = weight of the node
$w_i + f_i$ = weight of the node + weight of max child path
$w_i + f_j + f_k$ = weight of the node + weight of 1st max child path+weight of 2nd max child path

$maxSum$ would be updated when calculating every weight node and checks if theres a higher weight node and updates it. For returning a single path to parent node, this state equation is used:

$$Single\ Path = max \begin{cases} w_i \\ w_i + f_j \end{cases}$$

Where:
w_i = weight of the node
$w_i + f_i$ = weight of the node + weight of max child path

Finally I return $maxSum$ which points to the max weighted path in the tree. Visiting all node once would result in a total time complexity = $O(n)$.

C. Symmetry Makes Perfect

Submission#: 192561690

In this program, firstly I declared an *int* that would hold candy string size $string_Size$, an *int* to hold k flavors array size, a list *map* that would take a pair if *char*s and *int*s to order letters and their cost and an *array* of *chars* to store our input string.

After inputing all the above, I declare a function $compute_Cost$ that would take our candies string *array*, string *size* and the *list* of letters and their cost. In this function, a 2D Matrix is allocated to compute our DP solutions. Iterating through 2 nested for loops, starting with $1 - size$ insertion(called it gap) up to $n - size$, each possibility is computed and stored in DP matrix using recursion below:

Recurrence Relation:

$$dp[i][j] = \begin{cases} 0 & \text{(when } i == j\text{)} \\ dp[i+1][j-1] & \text{(when } s[i] == s[j]\text{)} \\ min \begin{cases} dp[i][j-1] + c[s[j]] \\ dp[i+1][j] + c[s[i]] \end{cases} & \text{(when } s[i] \neq s[j]\text{)} \end{cases}$$

Where:

$s[i] = letter\ at\ index\ i$

$s[j] = letter\ at\ index\ j$

$c[s[i]] = letter\ cost\ at\ index\ i$

$c[s[j]] = letter\ cost\ at\ index\ j$

Lastly, element in 2D Matrix at $dp[0][n-1]$ is displayed. Total time complexity $= O(n^2)$.

D. Longest Unimodal Subsequence

Submission#: 192568801

In this program, firstly I declared an *int* that would hold array size, then I declare an *array* with *int size*. I move on to input all values and store them inside my *array*. Then I declare a function called *compute_LUS* that takes my *array* and it's *size* as parameters. Inside *compute_LUS* I compute LIS on all my array elements from LEFT → RIGHT iteratively, saving each index LIS value inside a *vector* called lis_Index_Val using Recurrence Relation below:

$$l_i = max \begin{cases} 1 \\ \underset{0 < j < i, a_j < a_i}{max} & \Big\{ (l_j + 1) \end{cases}$$

Then I compute LDS on all my array elements from RIGHT → LEFT iteratively, saving each index LDS value inside a *vector* called lds_Index_Val using Recurrence Relation below:

$$l_i = max \begin{cases} 1 \\ \underset{0 < j < i, a_j > a_i}{max} & \Big\{ (l_j + 1) \end{cases}$$

Lastly I compute LUS max value using Longest Bitonic Subsequence logic, declare an *array* called lus_Index_Val of *size* identical to my original *array*. Iterate through all indices using Recurrence Relation below:

$$lus\_Index\_Val[i] = (lis\_Index\_Val[i] + lds\_Index\_Val[i])) - 1$$

(where we subtract 1 to remove index double counted in LIS and LDS)

Lastly, I display max value in my $array\ lus\_Index\_Val$ which refers to max LUS value. Total time complexity $= O(n^2) + O(n^2) + O(n) = O(n^2)$.