NAME: KULSOOM KHURSHID

REG #: SP20-BCS-044

ASSIGNMENT # 4

[Question: 01]

```
#include <bits/stdc++.h>
using namespace std;

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
   if (m == 0 || n == 0)
      return 0;
   if (X[m-1] == Y[n-1])
      return 1 + lcs(X, Y, m-1, n-1);
   else
      return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}
/* Driver code */
int main()
{
   char X[] = "AGGTAB";
   char Y[] = "GXTXAYB";

   int m = strlen(X);
   int n = strlen(Y);

   cout<<"Length of LCS is "<< lcs( X, Y, m, n ) ;

   return 0;
}
```

[Question: 02]

```
#include <bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z) { return min(min(x, y), z); }

int editDist(string str1, string str2, int m, int n)
```

```
{
   // If first string is empty, the only option is to
   // insert all characters of second string into first
   if (m == 0)
      return n;

   // If second string is empty, the only option is to
   // remove all characters of first string
   if (n == 0)
      return m;

   // If last characters of two strings are same, nothing
   // much to do. Ignore last characters and get count for
   // remaining strings.
   if (str1[m - 1] == str2[n - 1])
      return editDist(str1, str2, m - 1, n - 1);

   // If last characters are not same, consider all three
   // operations on last character of first string,
   // recursively compute minimum cost for all three
   // operations and take minimum of three values.
   return 1
        + min(editDist(str1, str2, m, n - 1), // Insert
            editDist(str1, str2, m - 1, n), // Remove
            editDist(str1, str2, m - 1,
                  n - 1) // Replace
          );
}

// Driver code
int main()
{
   // your code goes here
   string str1 = "sunday";
   string str2 = "saturday";

   cout << editDist(str1, str2, str1.length(),
               str2.length());

   return 0;
}
```

[Question: 03]

```
intfindMinInsertionsDP(charstr[], intn)
{
   // Create a table of size n*n. table[i][j]
   // will store minimum number of insertions
   // needed to convert str[i..j] to a palindrome.
   inttable[n][n], l, h, gap;

   // Initialize all table entries as 0
   memset(table, 0, sizeof(table));

   // Fill the table
   for(gap = 1; gap < n; ++gap)
      for(l = 0, h = gap; h < n; ++l, ++h)
         table[l][h] = (str[l] == str[h])?
                  table[l + 1][h - 1] :
                  (min(table[l][h - 1],
                     table[l + 1][h]) + 1);

   // Return minimum number of insertions
   // for str[0..n-1]
   returntable[0][n - 1];
}
```

[Question: 04]

```
#include<bits/stdc++.h>
using namespace std;

// Recursive function to find
// minimum number of insertions
int findMinInsertions(char str[], int l, int h)
{
   // Base Cases
   if (l > h) return INT_MAX;
   if (l == h) return 0;
   if (l == h - 1) return (str[l] == str[h])? 0 : 1;

   // Check if the first and last characters are
   // same. On the basis of the comparison result,
   // decide which subproblem(s) to call
   return (str[l] == str[h])?
               findMinInsertions(str, l + 1, h - 1):
               (min(findMinInsertions(str, l, h - 1),
               findMinInsertions(str, l + 1, h)) + 1);
}
```

```
// Driver code
int main()
{
   char str[] = "geeks";
   cout << findMinInsertions(str, 0, strlen(str) - 1);
   return 0;
}
```

[Question: 05]

Brute force: Try all 2^n possible subsets T.

Divide and Conquer:

Partition the problem into sub problems.
Solve the sub problems.
Combine the solutions to solve the original one.
If the sub problems are not independent, i.e. subproblems share subproblems, then a divide and-conquer algorithm repeatedly solves the common subsubproblems.

Dynamic Programming:

Dynamic programming is a method for solving optimization problems.
The idea: Compute the solutions to the subsub-problems once and store the solutions in a table, so that they can be reused (repeatedly) later.

Remark: We trade space for time.

[Question: 06]

Following are the detailed steps.

Input: Graph and a source vertex src

Output: Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

…..a) Do following for each edge u-v

……If dist[v] >dist[u] + weight of edge uv, then update dist[v]

…….dist[v] = dist[u] + weight of edge uv

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v

……If dist[v] >dist[u] + weight of edge uv, then “Graph contains negative weight cycle”

The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn’t contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

[Question: 07]

The approach to solving this problem is to count frequencies of all characters and consider the most frequent character first and place all occurrences of it as close as possible. After the most frequent character is placed, repeat the same process for the remaining characters.

1. Let the given string be str and size of string be n

2. Traverse str, store all characters and their frequencies in a Max Heap MH(implemented using priority queue). The value of frequency decides the order in MH, i.e., the most frequent character is at the root of MH.

3. Make all characters of str as '\0'.

4. Do the following while MH is not empty.

    - Extract the Most frequent character. Let the extracted character be x and its frequenc be f.
    - Find the first available position in str, i.e., find the first '\0' in str.
    - Let the first position be p. Fill x at p, p+d,.. p+(f-1)d

Time Complexity: Time complexity of above implementation is O(n + mLog(MAX)). Here n is the length of str, m is the count of distinct characters in str[] and MAX is the maximum possible different characters.

[Question: 08]

```
// To add an edge
voidaddEdge(vector <int> adj[], intu,intv)
{
   adj[u].push_back(v);
}
// find the sum of all dependencies
intfindSum(vector<int>adj[], intV)
{
   intsum = 0
   // just find the size at each vector's index
   for(intu = 0; u < V; u++)
      sum += adj[u].size()
   returnsum;
}
```