CompSci 260P Project #2

1. Pranav Udupa Shankaranarayana, UCINet ID: pudupash, Student ID no.: 65085145
2. Abhijeet Suresh Bhute, UCINet ID: abhute, Student ID no.: 81004607

1) Algorithm Description

The algorithms starts by prompting the values for n, x and y.
x and y are then validated to check if the number is lesser than $2^n - 1$.
Here, x and y are converted to their binary representation strings a and b and is used to find the LCS.

1. Find longest subsequence length and populating the DP array

First, we find the length of the longest common sub-sequence using the function find_max_lcs_len.
We achieve this by using the dynamic programming paradigm. We create a 2-dimensional DP array of size n*n. We can observe that at any $i, j < n$ if $a[i] = b[j]$ then the longest sub-sequence that can be formed is 1 more than the longest sub-sequence without using these characters at i, j . This can be formally represented as:

$$dp[i][j] = 1 + dp[i-1][j-1]$$

However, if the two characters are not the same, the longest sub-sequence we can obtain is by either excluding $a[i]$ or by excluding $b[j]$. We take the maximum of the two. This can be formally represented as:

$$dp[i][j] = max(dp[i-1][j], dp[i][j-1])$$

Also, when $i=0 \vee j=0$, we can have no common sub-sequences between the strings. So, we have a base conditions.

$$dp[i][0] = 0 \; for \; all \; i < n$$

and

$$dp[0][j] = 0 \; for \; all \; j < n$$

We can sum the above three up using the code below:

```
int find_max_lcs_len(string p, string q, int n)
{
    for (int i = 0; i <= n; i++)
    {
        for (int j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0)
                dp[i][j] = 0;
            else if (p[i - 1] == q[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
    return dp[n][n];
}
```

2. Find all possible LCS using the DP array

We now need to extract all the possible LCS using the dp matrix we've built above. The function findLCS recursively calculates this using the dp matrix and returns a vector list of all the possible LCS. We can start by comparing the last characters of the two strings a and b. If they are equal, then we know that this character has to be the part of LCS and is the last character. The previous characters can be recursively calculated for both the strings excluding these two characters. This recurrence relationship can be written as follows:

$$a[m-1] + lcs \; for \; all \; lcs \in findLCS(a, q, m-1, n-1)$$

Here we add $a[m-1] \vee b[n-1]$ to all the LCS found using $findLCS(a, q, m-1, n-1)$

But, when the last characters for a and b are not equal, neither can belong to the LCS. We need to recursively find the previous character by excluding either one of the two characters. But, since we need to find the longest sub-sequence, we can just look at the dp array we've built to make a decision. If excluding the last character of a, will return a higher dp[i][j] value than excluding the last character of b, we exclude the last character at a. If not, we do the same for b. We can recursively define this as follows.

$$\begin{gathered} if\,(dp[m-1][n] > dp[m][n-1]) \\ return\; findLCS(a, b, m-1, n); \end{gathered}$$

$$\begin{gathered} if\,(dp[m][n-1] > dp[m-1][n]) \\ return\; findLCS(a, b, m, n-1); \end{gathered}$$

If excluding both these characters gives the same length, then we need find the LCS for both and combine them. Formally,

$$\begin{aligned} & lcs1 = findLCS(a, b, m-1, n); \\ & lcs2 = findLCS(a, b, m, n-1); \\ & return\; merged(lcs1, lcs2) \end{aligned}$$

We can sum the above logic as follows

```
vector<string> findLCS(string a, string b, int m, int n)
{
    if (m == 0 || n == 0)
        return vector<string> (1);

    if (a[m - 1] == b[n - 1])
    {
        vector<string> lcs = findLCS(a, b, m - 1, n - 1);
        vector<string>::iterator it = lcs.begin();
        while(it != lcs.end())
        {
            it->push_back(a[m - 1]);
            it++;
        }
        return lcs;
    }
    if (dp[m - 1][n] > dp[m][n - 1])
        return findLCS(a, b, m - 1, n);

    if (dp[m][n - 1] > dp[m - 1][n])
        return findLCS(a, b, m, n - 1);

    vector<string> v1 = findLCS(a, b, m - 1, n);
    vector<string> v2 = findLCS(a, b, m, n - 1);

    v1.insert(v1.end(), v2.begin(), v2.end());

    return v1;
}
```

3. Remove duplicate LCS

The calculated LCS might have duplicates, we can remove this by using the data structure **set.** We add all the calculated LCS to the set, since they are all hashed into the same key because they have the same value, there will always be one copy of each in the set. This can be done as follows:

```
set<string> lcs(allLCS.begin(), allLCS.end());
```

2) Analysis of the asymptotic worst-case time complexity (θ-notation) as a function of n

We have the following stages of computation:

1. Convert to binary representation

We need to convert both x and y to binary representations, and since we iterate over a digit at a time at most once, this is executed linearly. If we do not have n-digits, we fill the rest with zeros. Hence the total time complexity for this is θ(n) for both x and y.

Time complexity: $\theta(n)$

2. Populate DP array

Here, we need to calculate the DP values for all value of $i, j \leq n$. Clearly, for each $i \rightarrow 0 n$, we have $j \rightarrow 0 n$. Thus, the time complexity will be a quadratic polynomial.

Time complexity: $\theta\left(n^2\right)$

3. Find all LCS

At every recursion level we either skip a character from a, b or both a and b. Hence, the recursion will end when either one of them will reach 0 characters. This always must happen after a maximum of $2 * n$ levels. However, at every level, we need to iterate through all the possible LCS at that level which is not a constant time solution. Let’s assume the number of LCS is actually k . Then we can say that, in our worst case we do k calculations at every recursion level.

Time complexity: $T(n)=T(n-1)+k+c$

4. Remove duplicate LCS

Like above, let's assume the number of LCS is actually k. Then, we remove duplicates by scanning through the vectors, linearly. i.e.

Time complexity: $\theta(k)$

Overall Time complexity

Clearly, either Step 2 or Step 3 is the bottleneck in our algorithm. Hence the overall time complexity can be given as follows:

$$T(n)=\theta\left(max\left(n^2,T(n-1)+k+c\right)\right)$$

Where k is the number of distinct LCS for given x, y

3) Instructions to compile

The code is written in C++, the compilation steps are as follows

1. Save the proj.cpp file in desired directory and move to that directory
2. Execute the following commands on terminal
3. g++ -std=c++11 proj.cpp
4. ./a.out

4) Sample input/output

Case 1

```
$ g++ -std=c++11 proj.cpp && ./a.out
Enter value of n in range [3:20] : 3
Enter value of x in range [0:7] : 4
Enter value of y in range [0:7] : 5

n: 3    x: 4    y: 5
binstring(n,x): 100        binstring(n,y) 101
Number of distinct LCS: 1

List of distinct LCS:
---------------------

10
```

Case 2

```
$ g++ -std=c++11 proj.cpp && ./a.out
Enter value of n in range [3:20] : 6
Enter value of x in range [0:63] : 45
Enter value of y in range [0:63] : 61

n: 6    x: 45  y: 61
binstring(n,x): 101101   binstring(n,y) 111101
Number of distinct LCS: 1
```

```
List of distinct LCS:
---------------------

11101
```

Case 3

```
$ g++ -std=c++11 proj.cpp && ./a.out
Enter value of n in range [3:20] : 11
Enter value of x in range [0:2047] : 1028
Enter value of y in range [0:2047] : 2006

n: 11  x: 1028       y: 2006
binstring(n,x): 10000000100  binstring(n,y) 11111010110
Number of distinct LCS: 2

List of distinct LCS:
---------------------

10010
10100
```

Case 4

```
$ g++ -std=c++11 proj.cpp && ./a.out
Enter value of n in range [3:20] : 16
Enter value of x in range [0:65535] : 65431
Enter value of y in range [0:65535] : 34652

n: 16  x: 65431     y: 34652
binstring(n,x): 1111111110010111 binstring(n,y) 1000011101011100
Number of distinct LCS: 2

List of distinct LCS:
---------------------

1111010111
1111111100
```

Case 5

```
$ g++ -std=c++11 proj.cpp && ./a.out
Enter value of n in range [3:20] : 20
Enter value of x in range [0:1048575] : 921765
Enter value of y in range [0:1048575] : 126532

n: 20  x: 921765   y: 126532
binstring(n,x): 11100001000010100101   binstring(n,y) 00011110111001000100
Number of distinct LCS: 7

List of distinct LCS:
---------------------

0000100000100
0000100100010
0001000100010
1110001000100
```

```
1110100000100
1110100100010
1111000100010
```

5) Bonus Points

The values for integers x and y that yield the largest possible number of distinct LCS's are as follows:

For n=14, x = 2457; y = 10922 ; number of LCS = 50