Lab 7

Due No Due Date **Points** 1

You are suggested to use the teaching servers burrow.soic.indiana.edu or hulk.soic.indiana.edu or tank.soic.indiana.edu for practicing C programs.

Lab 7: Dynamic Programming

Longest common subsequence

A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. For example, the sequence <B,D,E> is a subsequence of <A,B,C,D,E,F>. They should not be confused with substring which is a refinement of subsequence.

The longest common subsequence (LCS) between two string is the longest subsequence common to both the strings.

For example, given two DNA strands:
ACCG**GTC**GA**GT**GCG**CGGAAGCCGGCCGAA** and **GTCG**T**TCGGAA**T**GCCG**TT**GC**T**C**T**G**TA**AA**
the LCS is GTCGTCGGAAGCCGGCCGAA of length 20.

Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. Dynamic programming applies when the subproblems overlap—that is, when subproblems share sub-subproblems.

The idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often when using a more naive method, many of the subproblems are generated and solved many times. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations: once the solution to a given subproblem has been computed, it is stored or "memoized": the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems grows exponentially as a function of the size of the input.

The first step to solve a problem is to divide it into subproblem. For the LCS problem, we can do it the following way:

Let's assume we only care about the **length** of the Longest Common Subsequence for now. For the two input string s1[0....m] and s2[0....n], let p(0≤p≤m) be index for s1, q(0≤q≤n) index for s2. LCS[p][q] denotes the **length** of the LCS between s1[0...**p**] and s2[0...**q**]. What we want is LCS[m,n] and we are going to solve it by combining solutions from subproblems with smaller length. For any p,q, we can devide the problem LCS[p][q] into subproblems in the following way:

Case 1: if s1[p]==s2[q],LCS[p][q] = 1 + LCS[p-1][q-1]

Becasue if the s1[p] and s2[q] matches, first you can have a common subsequence of length 1 + LCS[p-1][q-1]. Is this subsequence the **longest**? Yes, you cannot possibly find another common subsequence for s1[0...p] and s2[0...q] whose length is **strictly** larger than this subsequence. So we have LCS[p][q] = 1 + LCS[p-1][q-1].

Case2: Otherwise,LCS[p][q] = max(LCS[p-1][q], LCS[p][q-1])

The s1[p] and s2[q] does not match but it is still possible for s1[p] to match some other previous character in s2. The same apply to s2[q]. Or maybe neigher of them match with other characters. Note that it is impossible that they both match with some other characters in the opposite string. (Why? please think about it) So with this constraint, we have **either** LCS[p][q] = LCS[p-1][q] **or** LCS[p][q] = LCS[p][q-1]. We try both so we have LCS[p][q] = **max**(LCS[p-1][q], LCS[p][q-1])

And the base case would be LCS[p][q]=0 if either p or q is less than zero.

```
LCS(p, q)
{
        if p==-1 or q ==-1      // base case
                return 0

        if s1[p] == s2[q]       // match case
                return 1 + LCS(p-1, q-1)

        else
                return MAX( LCS(p-1, q), LCS(p, q-1) )
}
```

Memoization

A same subproblem may be called many times. For example, LCS[2][4] can be called from LCS[2][5], LCS[3][4] and LCS[3][5]. Again these 3 subproblems can be called many times from other subproblems. So, rather than calculating LCS[2][4] everytime, we can remember the value so that we can use them next times the same problem is called. This technique is called memoization (not memorization). Without memoization, dynamic program would run in exponential time.

Here we are modifying the previous code to allow memoization, we are using a global CACHE[0...m][0...n].

```
Initialize full CACHE[][] with -1
LCS(p, q)
{
        if p==-1 or q ==-1      // base case
                return 0

        if CACHE[p][q] not equal -1 // this subproblem is already solved, return the cached value from here
                return CACHE[p][q]

        if s1[p] == s2[q]       // match case
                return CACHE[p][q] = 1 + LCS(p-1, q-1)

        else
                return CACHE[p][q] = MAX( LCS(p-1, q), LCS(p, q-1) )
}
```

Tracing the result

The LCS[m][n] would give us the length of the LCS. But how do we find out the exact LCS string? For LCS problem, we are dividing a problem into three subproblem. We will need to mark every time which subproblem we are choosing. The three subproblems are 1)match case, 2)decrease s1 and 3)decrease s2. We will use another matrix array Direction[0...m][0...n] and mark the result the following way:

```
Initialize full CACHE[][] with -1
Initialize full Direction[][] with -1
LCS(p, q)
{
        if p==-1 or q ==-1      // base case
                return 0

        if CACHE[p][q] not equal -1 // this subproblem is already solved, return the cached value from here
                return CACHE[p][q]

        if s1[p] == s2[q]       // match case

                Direction[p][q] = MATCH_CASE
                return CACHE[p][q] = 1 + LCS(p-1, q-1)

        else

                v1 = LCS(p-1, q)
                v2 = LCS(p, q-1)

                if v1 > v2      // decrease s1 case
                        Direction[p][q] = DECREASE_FIRST_STRING
                        CACHE[p][q] = v1
                else            // decrease s2 case
                        Direction[p][q] = DECREASE_SECOND_STRING
                        CACHE[p][q] = v2

                return CACHE[p][q]
}
```

After all the marking of direction done, we can print the LCS with calling PrintPath(m, n):

```
PrintPath(p, q)
{
        if Direction[p][q] == MATCH_CASE

                PrintPath(p-1, q-1)
                print(s1[p]) // or print(s2[q]) they are same

        elseif Direction[p][q] == DECREASE_FIRST_STRING

                PrintPath(p-1, q)

        elseif Direction[p][q] == DECREASE_SECOND_STRING

                PrintPath(p, q-1)

}
```