Solution to Homework Assignment 2 (Design techniques).

Big Oh Notation

For a function \( g : \mathbb{N} \to \mathbb{N} \), \( O(g(n)) \) denotes the set of all functions \( f : \mathbb{N} \to \mathbb{N} \) for which there exists a constant \( C_1 \) such that \( f(n) \leq C_1 \cdot g(n) \) for all \( n \in \mathbb{N} \). \( \Omega(g(n)) \) denotes the set of all functions \( f \) where for some constant \( C_0 > 0 \) we have \( C_0 \cdot g(n) \leq f(n) \). \( f(n) \in \Theta(g(n)) \) if \( f(n) \in O(g(n)) \) and \( f(n) \in \Omega(g(n)) \). This notion can be extended to multivariate functions, e.g. \( f(m, n) \in O(m \cdot n) \) if \( f(m, n) \leq C_1 \cdot m \cdot n \) for some constant \( C_1 \).

For \( f(n) \in O(g(n)) \), \( f'(n) \in O(g'(n)) \), we have \( f(n)+f'(n) \in O(g(n)+g'(n)) = O(\max\{g(n), g'(n)\}) \), \( f(n) \cdot f'(n) \in O(g(n) \cdot g'(n)) \).

Problem 1

```c
int LCSUBSTR(int i, j)
    if i = 0 then return 0 endif;
    if j = 0 then return 0 endif;
    if A[i] = B[j] then return LCSUBSTR(i - 1, j - 1) + 1;
    if A[i] # B[j] then return max{LCSUBSTR(i - 1, j), LCSUBSTR(i, j - 1)};
```

(a) What can you say about the asymptotic running time of the above procedure?

The worst case running time of LCSUBSTR\((m, n)\) is exponential in \(\min\{m, n\}\). More concretely, let \(T(i, j)\) denote the running time of LCSUBSTR\((i, j)\), we’ll prove that \(T(m, n) \in \Omega(2^{\min\{m, n\}})\).

We have \(T(i, 0) = T(0, j) \geq C_0 \) for some constant \( C_0 > 0 \). Consider the case where \(A\) and \(B\) have no characters in common, i.e. \(A[i] \neq B[j]\) for all \(1 \leq i \leq m, 1 \leq j \leq n\). Here LCSUBSTR\((i, j)\) will recursively call LCSUBSTR\((i - 1, j)\) and LCSUBSTR\((i, j - 1)\), and we can lower bound
\[
T(i, j) \geq T(i - 1, j) + T(i, j - 1) + C
\]
(1)

Where the constant \( C > 0 \) accounts for the work LCSUBSTR\((i, j)\) does besides the recursive calls. Solving this recursion is tricky, so we simplify it using the fact that the running time is monotone, i.e. \(T(i, j) \geq T(i', j')\) whenever \(i' \leq i, j' \leq j\). To see this, note that LCSUBSTR\((i, j)\) will at some point recursively call LCSUBSTR\((i', j')\). Using this, for \(\ell = \min\{m, n\}\)
\[
T(m, n) \geq T(\ell, \ell) \geq T(\ell - 1, \ell) + T(\ell, \ell - 1) + C
\]
\[
\geq 2T(\ell - 1, \ell - 1) + C
\]
\[
= C + 2C + 4C + \ldots + 2^{\ell-1}C + 2^\ell C_0
\]
\[
\geq 2^{\min\{m, n\}}C_0
\]
\[
\in \Omega(2^{\min\{m, n\}})
\]

The running time remains exponential when \(A\) and \(B\) are chosen randomly (and the alphabet has at least two characters.) If \(A\) and \(B\) are identical, the running time is linear \(O(\min\{m, n\})\).
(b) Give a more efficient algorithm for the longest common substring problem. Analyze the efficiency of your solution.

Let \( L[i, j] \) denote the longest common substring of \( A[1..i], B[1..j] \), we can express \( L[i, j] \) by the following recursive relation (we omit explaining this relation as it’s almost identical to the argument for edit distance discussed in the lecture notes.)

\[
L[i, j] = \max\{L[i−1, j], L[i, j−1], L[i−1, j−1] + |A[i] = B[j]|\}
\]

Where \( |A[i] = B[j]| \) is 1 if \( A[i] = B[j] \) and 0 otherwise. From this we get the following dynamic programming algorithm

```c
int LCSUBSEQ(int m, n)
    for i = 0 to m do L[i, 0] = 0 endfor;
    for j = 1 to n do L[0, j] = 0 endfor;
    for i = 1 to m do
        for j = 1 to n do
            L[i, j] = min\{L[i−1, j], L[i, j−1], L[i−1, j−1] + |A[i] = B[j]|\}
        endfor;
    endfor;
    return L[m, n].
```

The running time of our algorithm is roughly the product of the length of the strings, i.e. \( m \cdot n \cdot C \) (for some constant \( C \)).

Optional for two extra points: Give an efficient algorithm that finds the longest common substring of three sequences.

We can extend the above algorithm to compute the longest common substring of three strings \( A[1...i], B[1...j], C[1...k] \), using the relation

\[
L[i, j, k] = \max\{L[i−1, j−1, k−1] + |A[i] = B[j] = C[k]| , L[i−1, j, k] , L[i, j−1, k] , L[i, j, k−1]\}
\]

The algorithm for three strings \( A[1..m], B[1..n], C[1..o] \) is

```c
int LCSUBSTR3(int m, n, o)
    for i = 0 to m do for j = 0 to n do for k = 0 to o do Suff[i, j, k] = 0 endfor endfor endfor;
    for i = 0 to m do for j = 0 to n do for k = 0 to o do Suff[0, j, k] = 0 endfor endfor endfor;
    for i = 1 to m do
        for j = 1 to n do
            for k = 1 to o do
                L[i, j, k] = max\{L[i−1, j−1, k−1] + |A[i] = B[j] = C[k]| , L[i−1, j, k] , L[i, j−1, k] , L[i, j, k−1]\}
            endfor;
        endfor;
    endfor;
    return L[m, n].
```

We can generalize this dynamic programming algorithm to arbitrarily many strings. The running time of this algorithms is the product of the length of the strings, i.e. \( \Theta(m \cdot n \cdot o) \) for the algorithm above. Thus the running time grows rather fast with the number of strings.
(c) Give an efficient algorithm for the longest common subsequence problem. Analyze the efficiency of your solution.

Let $Suff[i, j]$ denote the length of the longest common suffix of $A[1..i]$ and $B[1..j]$. Recall that $S[1..t]$ is a suffix of $A[1...i]$ if $A[i-t..i] = S[1..t]$. E.g. ALGORITHMS and DREAMS have a common suffix MS of length 2.

There is a simple recursive formula for $Suff[i, j]$

$$Suff[i, j] = \begin{cases} 
Suff[i-1, j-1] + 1 & \text{if } A[i] = B[j] \\
0 & \text{if } A[i] \neq B[j]
\end{cases}$$

This recursive formula give a simple dynamic programming algorithm which computes an array containing $Suff[i, j]$ for all $0 \leq i \leq m, 0 \leq j \leq n$. We output the largest value in this array, which is the length of the longest common substring of $A$ and $B$. To see this, assume $S[1...\ell]$ is the longest common substring, and that it appears at positions $A[i-\ell...i] = B[j-\ell...j] = S[1...\ell]$, then $S[1...\ell]$ is a common suffix of $A[1..i], B[1..j]$ and thus $Suff[i,j] = \ell$.

```plaintext
int LCSubstr2(int m, n)
for i = 0 to m do Suff[i, 0] = i endfor;
for j = 0 to n do Suff[0, j] = j endfor;
for i = 1 to m do
    for j = 1 to n do
    endfor;
endfor;
return max{Suff[i, j] | 0 \leq i \leq m, 0 \leq j \leq n}
```

The first two for loops run in time $\Theta(m)$ and $\Theta(n)$, respectively. The nested for loop runs in time $\Theta(m \cdot n)$, and computing the max at the end also has running time $\Theta(m \cdot n)$. Thus the algorithm runs in time $\Theta(m \cdot n)$.

One can also simply compute $Suff[i, j]$ for all $i, j$ separately instead of using dynamic programming. This algorithm runs in time $O(m \cdot n \cdot \min\{m, n\})$. 