Longest Common Subsequences

Here is an important practical problem that has a nice solution using Dynamic Programming. We have two strings A and B and we want to find the longest possible string C that is a *subsequence* of both A and B. When we say that C is a subsequence of A, we mean we can derive C from A by removing some of the elements of A.

For example, "Oi" is a subsequence of "Ohio" and "odor" is a subsequence of "Lord Voldemort.   The longest common subsequence we can make of "Marvin Krislov" and "Oberlin College" has length 5: "rin o".

Clicker Question: What is the length of the longest common subsequence of CGAAGAT and GGTAGCT?

A. 2: AG
B. 3: GGA
C. 4: GGAT
D. 5: GGACT

This is more than just a game. In gene sequencing both genes and DNA strands are treated as strings of base elements represented by the letters GCAT. An exact match of the gene is unlikely, so researchers look for common subsequences of the gene and the DNA strand. If the subsequence is most of the length of the gene sequence, the strand is regarded as containing the gene.

These strings are long -- a DNA strand has millions of base elements and even a simple gene has many thousands of elements. Efficiency is important here.

We can easily find a recursive, inefficient function that gives the length of the longest common subsequence. We will modify this in the usual dynamic programming way to get a more efficient, table-driven solution, and use the table to find the actual common subsequence.

Here is the recursive solution. Let LCS(A, B) be the length of the longest common subsequence of A and B. Our base cases are when either A or B is the empty string, so the length of the common subsequence is 0.

If A and B start with the same letter, we can't do any better than to match this letter and recurse on the rest of the strings. In this case our length is 1+LCS(A.substring(1), B.substring(1)).

Finally, if the first letters of A and B don't match, we throw out the first letter of one of them and find the length of the longest common subsequence of what is left. The result is

```
max( LCS(A.substring(1), B), LCS(A, B.substring(1))
```

Altogether we get the following:

```
int LCS( String A, String B) {
        if (A.length() == 0 || B.length()==0)
                return 0;
        else if (A.charAt(0) == B.charAt(0))
                return 1+LCS(A.substring(1), B.substring(1));
        else
                return Math.max(
                                LCS(A.substring(1), B),
                                LCS(A, B.substring(1)));
}
```

Unfortunately this is massively inefficient. If A has length n and B has length m and they have no elements in common, this checks each of the2^n subsequences of A against each of the 2^m subsequences of B for a total of 2^{m+n} comparisons before deciding they have nothing in common.

For a more efficient solution we want to store partial results in a table. For this it helps to use indexes. Rather than recurse on substrings, we will leave the strings fixed and recurse on indexes: LCS(i, j) will be the length of the longest common subsequence of string A starting with index i and string B starting with index j.

Translating the code to this is easy:

```
int LCS( int i, int j) {
        if (i == A.length() || j==B.length())
                return 0;
        else if (A.charAt(i) == B.charAt(j))
                return 1+LCS(i+1, j+1);
        else
                return Math.max(LCS(i+1, j),  LCS(i, j+1));
}
```

Now we do the usual dynamic programming trick. We will store the result of LCS(i, j) in a 2-dimensional table. Each time we enter the function we look to see if we have a table entry for that combination of i and j; if we do we just return it. If we don't we recurse, and before returning the result we write it into the table

```
int LCS(int i, int j) {
        if (Table[i][j] >= 0)
                return Table[i][j];
        else if (i==A.length() || j==B.length()) {
                Table[i][j] = 0;
                return 0;
        }
        else if (A.charAt(i) == B.charAt(j)) {
                int t = 1+LCS(i+1,j+1);
                Table[i][j] = t;
                return t;
        }
        else {
                int t = Math.max(LCS(i+1,j), LCS(i, j+1));
                Table[i][j] = t;
                return t;
        }
}
```

This is vastly more efficient. Each of the entries that does more than a lookup writes a value into the table. If A has length n and B length m, there are only (n+1)*(m+1) entries in the table. This version runs in time O(n*m), which is a huge improvement over O(2^{n+m}). For example if n and m are both 100 n*m is 10,000 while 2^{n+m} is about 10^{60}.

We can say even more. Consider the table we get from the strings A="ABAC" and B="BAAC":

	0: B	1: A	2: A	3: C	4
0: A	3	3			0
1: B	3		2	1	0
2: A		2	2	1	0
3: C			1	1	0
4	0	0	0	0	0

The [0][0] entry is 3; the value in the row below at [1][0] is also 3, so we threw out the first letter of "ABAC" and matched "BAC" against "BAAC". The B's now match so we pair "AC" against "AAC". The A's match so we pair "C" against "AC", then against "C". This ends when Table[row][col] is 0.

Here is code that looks through the table to find the best subsequence:

```
void PrintSubsequence() {
    int row = 0;
    int col = 0;
    String s = "";
    while (Table[row][col] > 0) {
            if (A.charAt(row)==B.charAt(col)) {
                    s = s + A.charAt(row);
                    row += 1;
                    col += 1;
            }
            else if (Table[row+1][col] == Table[row][col])
                    row += 1;
            else
                    col += 1;
    }
    System.out.println(s);
}
```