Iterative Dynamic Programming

Memoization, which is a technique that we saw last time, works in the following way. We start with a recursive function and add a table that maps the function's parameter values to the results computed by the function. Then if this function is ever called twice with the same parameters, we simply look up the answer in the table. Iterative dynamic programming (DP) goes the other way. We start with the table and instead of having a recursive function, we simply fill in the table directly. At the end, we have the answer to our original problem sitting in some cell in the table.

Longest Common Subsequence

This is one of the classic DP problems. We have two strings, S and T. What we want is to *align* them – write T underneath S and insert some gaps into both strings in such a way that the number of matching pairs of characters is maximized. For example, if S="ACGTCGTGT" and T="CTAGTGGAG", then an optimal alignment is

```
S = AC--GTCTGTGT
T = -CTAGTG-GAG-
```

The bold letters are the matching pairs, and they spell out the longest common subsequence of S and T: "CGTGG". LCS has many applications. In web searching, you might be interested in the smallest number of changes that are needed to morph one word into another. A "change" here is either an insertion, deletion or replacement of a single character. In bioinformatics, LCS is used all the time to align DNA, RNA and amino acid sequences to determine evolutionary relationships between organisms.

Once again, let's try to find a recursive solution. Look at the first character of S and T. There are only 3 things we can do with them: match them (write one under the other), insert a gap into S or insert a gap in to T. Inserting gaps into both strings gains us nothing. In all 3 cases, we get a sub-problem of the form, "find the longest common subsequence of X and Y," where X is a suffix of S and Y is a suffix of T.

So let's consider a more general problem of finding the LCS of S[i..m] and T[j..n]. Here, m is the length of S, n is the length of T, S[i..m] is a substring of S starting at character i and going until the end of S and T[j..n] is a substring of T starting at character j and going until the end of T. Look at the first characters: S[i] and T[j]. The 3 options are:

1. Match S[i] to T[j],
2. Insert a gap into S (match T[j] to a gap in S),
3. Insert a gap into T (match S[i] to a gap in T).

In the first case, if S[i] is equal to T[j], we get a matching pair and can count it towards the total length of the LCS. Let LCS(i,j) be the length of the longest common subsequence of S[i..m] and T[j..n]. Then the value of LCS(i,j) is the maximum of 3 numbers:

1. LCS(i+1, j+1) (+1 if S[i] = T[j])
2. LCS(i, j+1)
3. LCS(i+1, j)

We could write a recursive function that computed LCS(i,j) and then use a memoization table to speed up its evaluation. Instead, this time we will work with the table directly. First, we need some base cases. Clearly, when either S or T is empty, the answer is zero – we can't match any characters. This is the only base case we need.

```
int LCS[1024][1024];
int lcs( string S, string T ) {
    int m = S.size(), n = T.size();

    // base cases
    for( int i = 0; i <= m; i++ ) LCS[i][n] = 0;
    for( int j = 0; j <= n; j++ ) LCS[m][j] = 0;

    // iterative DP
    for( int i = m - 1; i >= 0; i-- )
    for( int j = n - 1; j >= 0; j-- ) {
        LCS[i][j] = LCS[i + 1][j + 1];     // matching S[i] to T[j]
        if( S[i] == T[j] ) LCS[i][j]++;    // we get a matching pair

        // the other two cases - inserting a gap
        if( LCS[i][j + 1] > LCS[i][j] ) LCS[i][j] = LCS[i][j + 1];
        if( LCS[i + 1][j] > LCS[i][j] ) LCS[i][j] = LCS[i + 1][j];
    }
    return LCS[0][0];
}
```

First, we take care of the base cases. Note that we need our LCS table to be one row and one column larger than the lengths of the two strings. Then we run the iterative DP loops to fill in each cell in the table. This is like doing recursion backwards, or bottom-up. Note that the value of LCS[i][j] depends on 3 other values (LCS[i+1][j+1], LCS[i][j+1] and LCS[i+1][j]), all of which have larger values of i or j. This is why it is important to get the two loops right. They must go through the table in the order of decreasing i and j values. This will guarantee that when we need to fill in the value of LCS[i][j], we already know the values of all of the cells on which it depends. This also means that we do not need to initialize the table with "don't know" values, like we would have to do in the memoization solution.

The running time of the algorithm is very easy to compute. We only have a single pair of nested loops, which require $O(m)$ time.

This algorithm computes the *length of* the longest commong subsequence, not the subsequence itself. However, we can easily recover the sequence by tracing it through the table. Start at cell (0,0). We know that the value of LCS[0][0] was the maximum of 3 values of the neighbouring cells. So we simply recompute LCS[0][0] and note which cell gave the maximum value. Then we move to that cell (it will be one of (1,1), (0,1) or (1,0)) and repeat this until we hit the boundary of the table. Every time we pass through a cell (i,j) where S[i]=T[j], we have a matching pair and we print S[i]. At the end, we will have printed the longest common subsequence in $O(m+n)$ time.

An alternative approach to path recovery is to keep a separate table that records, for each cell, which direction we came from when computing the value of that cell. At the end, we again start at cell (0,0) and follow these directions until the opposite corner of the table.

Longest Increasing Subsequence

Here is another classic problem. We have a sequence of n integers. We are interested in increasing subsequences. For example, if the sequence, S, is (5,6,2,3,4,-1,9,9,8,9,5), then (5,6), (3,5), (-1,8,9) are all increasing subsequences of S. The longest one of them is (2,3,4,8,9), and we want an algorithm for finding it.

Like we did before, with the LCS, let's concentrate on computing the length of the longest increasing subsequence. Once we have that, we will figure out how to rebuild the subsequence itself. The first step is to come up with a recursive solution. First of all, let's add a negative infinity to the front of the sequence (just pick a number smaller than any other number in S). Then we are guaranteed that any LIS in S must contain that first number.

Also, let's change the problem slightly. We are going to find the longest increasing subsequence in S, given than S[0] (the first element of S) has to go into our subsequence. This is not a problem because that negative infinity at the front guarantees this condition. So we know what the first number in the LIS is. Let's find the second number. We know that it is an element of S, and that it is larger than S[0]. If there is no such element, then we are done – the LIS is the one-element sequence (S[0]). Otherwise, we can find some element S[i] that is larger than S[0] and is to the right of S[0] in S. S[i] could potentially be the second element in our LIS. If it is then we take it and consider the LIS of the sequence (S[i], S[i+1], ..., S[n]). This is a sub-problem, and we can solve it recursively.

More formally, what we need is an array LIS[i] that contains the length of the longest increasing subsequence of (S[i], S[i+1], ..., S[n]), given that we must pick S[i] as the first element of the subsequence. To compute LIS[i], we scan all the nubmers to the right of S[i] and for each S[j] that is larger than S[i], we consider taking it as the second element of the subsequence. Just like the LCS algorithm, instead of using recursion, we will fill in the LIS[] array directly.

```
int LIS[1024];
int lis( vector< int > S ) {
    // insert negative infinity at the front
    S.push_front( INT_MIN );
    int n = S.size();

    // run iterative DP
    for( int i = n – 1; i >= 0; i-- ) {
        LIS[i] = 1;       // pick nothing but the first element

        // try picking a larger second element
        for( int j = i + 1; j < n; j++ )
            if( S[j] > S[i] && LIS[j] + 1 > LIS[i] )
                LIS[i] = LIS[j] + 1;

    // return the length of the LIS, minus the first number we added
    return LIS[0] – 1;
}
```

This algorithm takes $O(n^2)$ time because of the two loops. As in LCS, it is important to run the outer loop from large to small values of i since for each i, we are looking at j > i, and they have to have been computed already. Recovering the subsequence itself can be done in quadratic time, or in linear time if we keep an extra array that remembers for each i, where the next element of the subsequence is.