1 LCS Revisited

We have been looking at what is called "bottom-up Dynamic Programming". Here is another way of thinking about Dynamic Programming, that also leads to basically the same algorithm, but viewed from the other direction. Sometimes this is called "top-down Dynamic Programming".

Basic Idea (version 2): Suppose you have a recursive algorithm for some problem that gives you a really bad recurrence like $T(n) = 2T(n-1)+n$. However, suppose that many of the subproblems you reach as you go down the recursion tree are the *same*. Then you can hope to get a big savings if you store your computations so that you only compute each *different* subproblem once. You can store these solutions in an array or hash table. This view of Dynamic Programming is often called *memoizing*.

For example, for the LCS problem, using our analysis we had at the beginning we might have produced the following exponential-time recursive program (arrays start at 1):

```
LCS(S,n,T,m):
    if (n==0 OR m==0) return 0
    if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1)     // no harm in matching up
    else result = max[LCS(S,n-1,T,m), LCS(S,n,T,m-1)]   // take the better one
    return result
```

This algorithm runs in exponential time. In fact, if `S` and `T` use completely disjoint sets of characters (so that we never have `S[n]==T[m]`) then the number of times that `LCS(S,1,T,1)` is recursively called equals $\binom{n+m-2}{m-1}$.[1] In the memoized version, we store results in a matrix so that any given set of arguments to `LCS` only produces new work (new recursive calls) once. The memoized version begins by initializing `arr[i][j]` to `unknown` for all `i,j`, and then proceeds as follows:

```
LCS(S,n,T,m):
    if (n==0 || m==0) return 0
    if (arr[n][m] != unknown) return arr[n][m]   // <- added this line (*)
    if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1)
    else result = max[LCS(S,n-1,T,m), LCS(S,n,T,m-1)]
    arr[n][m] = result                           // <- and this line (**)
    return result
```

All we have done is saved our work in line (**) and made sure that we only embark on new recursive calls if we haven't already computed the answer in line (*).

In this memoized version, our running time is now just $O(mn)$. One easy way to see this is as follows. First, notice that we reach line (**) at most mn times (at most once for any given value of the parameters). This means we make at most $2mn$ recursive calls total (at most two calls for each time we reach that line). Any given call of `LCS` involves only $O(1)$ work (performing some equality checks and taking a max or adding 1), so overall the total running time is $O(mn)$.

Comparing bottom-up and top-down dynamic programming, both do almost the same work. The top-down (memoized) version pays a penalty in recursion overhead, but can potentially be faster

[1] This is the number of "monotone walks" between the upper-left and lower-right corners of an n by m grid.

than the bottom-up version in situations where some of the subproblems never get examined at all. These differences, however, are minor: you should use whichever version is easiest and most intuitive for you for the given problem at hand.

More about LCS: Discussion and Extensions. An equivalent problem to LCS is the "minimum edit distance" problem, where the legal operations are insert and delete. (E.g., the unix "diff" command, where S and T are files, and the elements of S and T are lines of text). The minimum edit distance to transform S into T is achieved by doing $|S| - \texttt{LCS}(S,T)$ deletes and $|T| - \texttt{LCS}(S,T)$ inserts.

In computational biology applications, often one has a more general notion of sequence alignment. Many of these different problems all allow for basically the same kind of Dynamic Programming solution.

2 The Knapsack Problem

Imagine you have a homework assignment with different parts labeled A through G. Each part has a "value" (in points) and a "size" (time in hours to complete). For example, say the values and times for our assignment are:

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
time	3	4	2	6	7	3	5

Say you have a total of 15 hours: which parts should you do? If there was partial credit that was proportional to the amount of work done (e.g., one hour spent on problem C earns you 2.5 points) then the best approach is to work on problems in order of points/hour (a greedy strategy). But, what if there is no partial credit? In that case, which parts should you do, and what is the best total value possible?[2]

The above is an instance of the *knapsack problem*, formally defined as follows:

Definition 1 *In the* **knapsack problem** *we are given a set of n items, where each item i is specified by a size s_i and a value v_i. We are also given a size bound S (the size of our knapsack). The goal is to find the subset of items of maximum total value such that sum of their sizes is at most S (they all fit into the knapsack).*

We can solve the knapsack problem in exponential time by trying all possible subsets. With Dynamic Programming, we can reduce this to time $O(nS)$.

Let's do this top down by starting with a simple recursive solution and then trying to memoize it. Let's start by just computing the best possible *total value*, and we afterwards can see how to actually extract the items needed.

```
// Recursive algorithm: either we use the last element or we don't.
Value(n,S)    // S = space left, n = # items still to choose from
    if (n == 0) return 0
    if (s_n > S) result = Value(n-1,S)     // can't use nth item
    else result = max[v_n + Value(n-1, S-s_n), Value(n-1, S)]
    return result
```

[2] Answer: In this case, the optimal strategy is to do parts A, B, F, and G for a total of 34 points. Notice that this doesn't include doing part C which has the most points/hour!

Right now, this takes exponential time. But, notice that there are only $O(nS)$ *different* pairs of values the arguments can possibly take on, so this is perfect for memoizing. As with the LCS problem, let us initialize a 2-d array `arr[i][j]` to "unknown" for all `i,j`.

```
Value(n,S)
    if (n == 0) return 0
    if (arr[n][S] != unknown) return arr[n][S]   // <- added this
    if (s_n > S) result = Value(n-1,S)
    else result = max[v_n + Value(n-1, S-s_n), Value(n-1, S)]
    arr[n][S] = result                                    // <- and this
    return result
```

Since any given pair of arguments to Value can pass through the array check only once, and in doing so produces at most two recursive calls, we have at most $2n(S+1)$ recursive calls total, and the total time is $O(nS)$.

So far we have only discussed computing the *value* of the optimal solution. How can we get the items? As usual for Dynamic Programming, we can do this by just working backwards: if `arr[n][S]` = `arr[n-1][S]` then we *didn't* use the nth item so we just recursively work backwards from `arr[n-1][S]`. Otherwise, we *did* use that item, so we just output the nth item and recursively work backwards from `arr[n-1][S-s_n]`. One can also do bottom-up Dynamic Programming.

3 Matrix product parenthesization

Our final example for Dynamic Programming is the *matrix product parenthesization* problem.

Say we want to multiply three matrices X, Y, and Z. We could do it like $(XY)Z$ or like $X(YZ)$. Which way we do the multiplication doesn't affect the final outcome but it *can* affect the running time to compute it. For example, say X is 100x20, Y is 20x100, and Z is 100x20. So, the end result will be a 100x20 matrix. If we multiply using the usual algorithm, then to multiply an ℓxm matrix by an mxn matrix takes time $O(\ell mn)$. So in this case, which is better, doing $(XY)Z$ or $X(YZ)$?

Answer: $X(YZ)$ is better because computing YZ takes 20x100x20 steps, producing a 20x20 matrix, and then multiplying this by X takes another 20x100x20 steps, for a total of 2x20x100x20. But, doing it the other way takes 100x20x100 steps to compute XY, and then multplying this with Z takes another 100x20x100 steps, so overall this way takes 5 times longer. More generally, what if we want to multiply a series of n matrices?

Definition 2 *The* **Matrix Product Parenthesization** *problem is as follows. Suppose we need to multiply a series of matrices: $A_1 \times A_2 \times A_3 \times \ldots \times A_n$. Given the dimensions of these matrices, what is the best way to parenthesize them, assuming for simplicity that standard matrix multiplication is to be used (e.g., not Strassen)?*

There are an exponential number of different possible parenthesizations, in fact $\binom{2(n-1)}{n-1}/n$, so we don't want to search through all of them. Dynamic Programming gives us a better way.

As before, let's first think: how might we do this recursively? One way is that for each possible split for the final multiplication, recursively solve for the optimal parenthesization of the left and right sides, and calculate the total cost (the sum of the costs returned by the two recursive calls plus the ℓmn cost of the final multiplication, where "m" depends on the location of that split). Then take the overall best top-level split.

For Dynamic Programming, the key question is now: in the above procedure, as you go through the recursion, what do the subproblems look like and how many are there? Answer: each subproblem looks like "what is the best way to multiply some sub-interval of the matrices $A_i \times \ldots \times A_j$?" So, there are only $O(n^2)$ *different* subproblems.

The second question is now: how long does it take to solve a given subproblem assuming you've already solved all the smaller subproblems (i.e., how much time is spent inside any *given* recursive call)? Answer: to figure out how to best multiply $A_i \times \ldots \times A_j$, we just consider all possible middle points k and select the one that minimizes:

	optimal cost to multiply $A_i \ldots A_k$	$\leftarrow$ already computed
+	optimal cost to multiply $A_{k+1} \ldots A_j$	$\leftarrow$ already computed
+	cost to multiply the results.	$\leftarrow$ get this from the dimensions

This just takes $O(1)$ work for any given k, and there are at most n different values k to consider, so overall we just spend $O(n)$ time per subproblem. So, if we use Dynamic Programming to save our results in a lookup table, then since there are only $O(n^2)$ subproblems we will spend only $O(n^3)$ time overall.

If you want to do this using bottom-up Dynamic Programming, you would first solve for all subproblems with $j - i = 1$, then solve for all with $j - i = 2$, and so on, storing your results in an n by n matrix. The main difference between this problem and the two previous ones we have seen is that any *given* subproblem takes time $O(n)$ to solve rather than $O(1)$, which is why we get $O(n^3)$ total running time. It turns out that by being very clever you can actually reduce this to $O(1)$ amortized time per subproblem, producing an $O(n^2)$-time algorithm (Knuth).

4 High-level discussion of Dynamic Programming

What kinds of problems can be solved using Dynamic Programming? One property these problems have is that if the optimal solution involves solving a subproblem, then it uses the *optimal* solution to that subproblem. For instance, say we want to find the shortest path from A to B in a graph, and say this shortest path goes through C. Then it must be using the shortest path from C to B. Or, in the knapsack example, if the optimal solution does not use item n, then it is the optimal solution for the problem in which item n does not exist. The other key property is that there should be only a polynomial number of different subproblems. These two properties together allow us to build the optimal solution to the final problem from optimal solutions to subproblems.

In the top-down view of dynamic programming, the first property above corresponds to being able to write down a recursive procedure for the problem we want to solve. The second property corresponds to making sure that this recursive procedure makes only a polynomial number of *different* recursive calls. In particular, one can often notice this second property by examining the arguments to the recursive procedure: e.g., if there are only two integer arguments that range between 1 and n, then there can be at most n^2 different recursive calls.

Sometimes you need to do a little work on the problem to get the optimal-subproblem-solution property. For instance, suppose we are trying to find paths between locations in a city, and some intersections have no-left-turn rules (this is particulatly bad in San Francisco). Then, just because the fastest way from A to B goes through intersection C, it doesn't necessarily use the fastest way to C because you might need to be coming into C in the correct direction. In fact, the right way to model that problem as a graph is not to have one node per intersection, but rather to have one node per $\langle$*intersection, direction*$\rangle$ pair. That way you recover the property you need.