CPSC 320 Notes: Imlpmenting Memoization and DP!

March 16, 2015

Let's take a problem from recursive solution (recurrence) to memoized and DP implementation.

PROBLEM: Given two strings, finding their Longest Common Subsequence (LCS). The LCS of `A` and `B` is the longest string whose letters appear in order (but not necessarily consecutively) within both `A` and `B`. For example, the LCS of `snow` and `naomi` is the length 2 string `no`.

Our recurrence for the **length** of the LCS of strings `A` with length n and `B` with length m (respectively) is below. (For optimization problems, we usually first find "the value of the best solution" rather than "the best solution".)

$$\text{LCS}(n,m) = \begin{cases} \min(\text{LCS}(n-1,m), \text{LCS}(n,m-1), \text{LCS}(n-1,m-1)+1) & \text{if } m>0,\ n>0, \text{ and } A[n]=B[m] \\ \min(\text{LCS}(n-1,m), \text{LCS}(n,m-1)) & \text{if } m>0,\ n>0, \text{ and } A[n]\neq B[m] \\ 0 & \text{if } m=0 \text{ or } n=0 \end{cases}$$

(In fact, we can change the first case to *just* $\text{LCS}(n-1,m-1)+1$ if we want.)

1 Build Small Examples

Let's write down small examples to test our recurrence beyond the examples we used to understand the problem. We already have the given examples. (`A` = "snow" and `B` = "naomi" have an LCS "no" of length 2 while `A` = "country" and `B` = "tycoon" have an LCS "con" of length 3.) In addition, construct:

1. The smallest, trivial example.

2. Examples to test base cases besides the smallest, trivial example.

3. Example to test when we make progress ($A[n] = B[m]$). (Note that you won't be able to test when, e.g., $\text{LCS}(n-1,m)$ is the right move at the same time as $A[n] = B[m]$ because it never is!)

4. Examples to test the other cases when $A[n] \neq B[m]$. Test both when the right move is to strip a character from A and when the right move is to strip a character from B.

2 Naïve Implementation

You should always start with the most naïve possible implementation.

1. In your favorite language, write a "stub" (a function that just returns 0) for `LCS`.
2. Program your test cases. (In **any** language, you can just write "if LCS(<input>) is equal to <expected result> then output 'yay' else output 'eep'." Hopefully your language also supports exceptions, assertions, unit testing, or the like as well!)
3. Program the recurrence as directly as possible (as the "naïve solution").
4. Test it. How well does it do on the examples we have so far?
5. Write a function that generates a random string of As, Cs, Gs, and Ts of a given length, and test how well the naïve solution scales to two randomly generated strings of various (equal) lengths.

3 Memoization

Converting a recursive solution to a memoized one is surprisingly easy. Use this template to convert yours. (The "global table" mentioned can be an array. In many languages, it works well as a hash table instead. Furthermore, it usually doesn't actually need to be global. E.g., you can build the table in a function that passes it to a helper function to solve the instance and then throw away the table when the helper is done.)

```
Create a global table named Solns large enough
  to store solutions to all subproblems of interest
Initialize all entries of Solns to a flag meaning "empty"

Algorithm(parameters):
  If Solns[parameters] is empty:
    <insert your naive code here, changing anything like

      Return EXPRESSION

    to

      Solns[parameters] = EXPRESSION>

  Return Solns[parameters]
```

1. Write a memoized version of your solution named `LCSM`. Don't forget to rename your recursive calls from `LCS` to `LCSM`, or you'll call the old version!
2. Test out your memoized version on the test cases. How well does it scale?

4 Extracting the Solution

You can generally take the filled-in memoization (or dynamic programming) table and the recurrence and use them to write code to extract the **actual** solution. Call the recurrence with the complete problem and then alter it so instead of calculating the value of the solution, it finds which of the already-stored choices was the right one, recurses into that one to extract the solution, and then outputs whatever portion of the solution that choice represents.

Here's what that might look like as a template:

```
ExtractSolution(parameters):
  <insert your naive code here, but:

     1. In the base case, just return.

     2. In the recursive case:

        a. Determine WHICH choice was the best, not what value it gives.
           (Either use the table or call your memoized version, which
           will itself use the table.)

        b. Call ExtractSolution on the parameters for that choice.

        c. Output whatever that choice MEANS for the solution.>
```

1. Write `LCSM_Solve`, which actually produces the longest common subsequence of two strings.
2. Modify your test cases to work for this function and then test its performance.

5 Dynamic Programming

Dynamic programming requires a bit more effort than memoization because the template requires you to carefully order the solutions:

```
Create a global table named Solns large enough
  to store solutions to all subproblems of interest
Initialize all entries of Solns to a flag meaning "empty"

Algorithm(parameters):
  For all subproblems P in an order that ensures that
    each subproblem P depends on has ALREADY been solved
    when you solve P:

    <insert your naive code here, changing anything like

       RecursiveCall(parameters)

     to

       Solns[parameters]

     and anything like:

       Return EXPRESSION

     to

       Solns[parameters] = EXPRESSION>

  Return Solns[parameters]
```

1. Write a dynamic programming (DP) version of your solution named `LCSDP1`.
2. Test out your DP version on the test cases. How well does it scale?
3. Write a new DP version named `LCSDP2` that solves the problems in a different order and test it.
4. Adapt `LCSM_Solve` to work with `LCSDP1` and call it `LCSDP1_Solve`. (You shouldn't need to change much, since they both build the same table!) Test it.

6 Challenge

It turns out you can **automatically** memoize a function just by using a hash table where the key is the collection of parameters passed to the function. Do that in your favorite language.