Lecture 23: Dynamic Programming 2

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

String similarity

Let's consider two strings:

- $S_1 =$ ACCGGTCGAGTGCGCGG
- $S_2 =$ GTCGTTCGGAATGCC

Can we come up with some notion of how close these two strings are when the order of the letters is important?

Notions of similarity

- the longest substring common to both strings
- the number of changes to convert one string into another (this is called edit distance)
- the longest string, S_3, such that the characters in S_3 occur in both S_1 and S_2 in the same order, but not necessarily consecutively. This measure is called the longest common subsequence (LCS).

What is the LCS of $S_1 =$ACCGGTCGAGTGCGCGG and $S_2 =$GTCGTTCGGAATGCC?

$S_3 = GGTCGGTGCC$

Longest common subsequence

Let $X = [x_1, x_2, ..., x_m]$ and $Y = [y_1, y_2, ..., y_n]$ be two strings. Further, let $Z = [z_1, z_2, ..., z_k]$ be an LCS of X and Y. Then the following statements hold:

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$ then either:
   2.1 Z is an LCS of X_{m-1} and Y; or
   2.2 Z is an LCS of X_m and Y_{n-1}.

This gives us a very clear optimal substructure problem from which we can easily write a naive recursive solution. If we think about the length of the LCS, then the length can be defined as:

$$l_{i,j} = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ l_{i-1,j-1} + 1, & \text{if } x_i = y_j, \\ \max(l_{i,j-1}, l_{i-1,j}), & \text{otherwise} \end{cases}$$

Naive algorithm

```
function RecursiveLCS(X, Y)
    if X.length=0 or Y.length=0 then
        return ""
    if X[m]=Y[n] then
        return RecursiveLCS(X[1:m-1],Y[1:n-1])+X[m]
    lcs1 $\leftarrow$ RecursiveLCS(X[1:m-1],Y)
    lcs2 $\leftarrow$ RecursiveLCS(X, Y[1:n-1])
    if lcs1.length>lcs2.length then
        return lcs1
    else
        return lcs2
```

Once again, the naive algorithm is exponential, so we have to memoize.

Memoized version

Initialise memo array to be full of -1.

```
function MEMOLCSRECURSE(X, Y)
    if $m = 0$ or $n = 0$ then return ""
    if memo[m,n]≠"-1" then return memo[m,n]
    if X[m]=Y[n] then
        memo[m,n]← MemoLCSRecurse(X[1:m-1],Y[1:n-1])+X[m]
    else
        l1 ← MemoLCSRecurse(X[1:m-1],Y)
        l2 ← MemoLCSRecurse(X,Y[1:n-1])
        if l1.length>l2.length then
            memo[m,n] = l1
        else
            memo[m,n] = l2
    return memo[m,n]
```

Iterative Version

The memoized recursive version is fairly simple and efficient, but the strings do not need to be very long before we will blow the system stack. If we're comparing DNA for example, then we could expect strings millions or billions of characters long. Even with an iterative version, billions of characters will cause a problem. Why?

Nevertheless, an iterative version will work for much larger problems than a recursive one. In previous iterative dynamic programming solutions, we've been able to mirror the recursive version and also work backwards, but in this case, it makes more sense to run the algorithm forwards.

We could have done the other algorithms forwards too, or this one backwards, but it's better to choose the direction that's most natural based on the problem at hand.

The disadvantage of the forward algorithm is that all elements of the array will be filled, whereas the recursive backwards algorithm only fills those elements that are needed.

Iterative Algorithm

```
function IterativeLCS(X,Y)
    memo ← array of size $m + 1 \times n + 1$
    for $i \leftarrow 1$ to $m$ do
        for $j \leftarrow 1$ to $n$ do
            if X[i]=Y[j] then
                memo[i+1,j+1] ← memo[i,j]+X[i]
            else
                if memo[i,j+1].len>memo[i+1,j].len then
                    memo[i+1,j+1] ← memo[i,j+1]
                else
                    memo[i+1,j+1] ← memo[i+1,j]
    return memo[m,n]
```

We can do better than this in terms of space efficiency. There is no need to keep the whole array in memory at one time - we just need to reference the current row and the previous row. That means instead of space with $O(mn)$, we can use only $O(2n)$.

Edit distance

The unix utility *diff* uses edit distance to find differences between two files (or strings). This utility, or others like it, are used in all sorts of places, but most notably in version control systems such as *git* or *subversion*. These version control systems are used to track all changes in a document or a group of documents and are especially useful for tracking changes in source code. Edit distance can also be used to fix spelling errors.

Edit distance is very similar to LCS. In LCS, we effectively only allow deletion from one string or the other in each step. In edit distance, we try to transform one string into the other by allowing operations: insert, delete, or substitute; just as if we were editing the source string. We aren't going to worry about the resulting alignment in the following, but it can be reconstructed from the edit distance array.

Edit distance recurrence

The recurrence for the edit distance is (transform X to Y):

$$d_{ij} = \begin{cases} d_{i-1,j-1}, & \text{if } X[i] = Y[j] \\ \min \begin{cases} d_{i-1,j} + 1, & \text{delete } X[i] \\ d_{i,j-1} + 1, & \text{insert } Y[j] \\ d_{i-1,j-1} + 1, & \text{substitute } Y[j] \text{ for } X[i] \end{cases}, & \text{if } X[i] \neq Y[j] \end{cases}$$

where

$d_{0,j} = j$: insert all characters up to Y[j]

and

$d_{i,0} = i$: delete all characters up to X[i]

Naive algorithm

```
function EditDistance(X,Y)
    if m=0 then return n
    if n=0 then return m
    if X[m]=Y[n] then
        return EditDistance(X[1...m-1],Y[1...n-1])
    delxi = 1+EditDistance(X[1...m-1],Y)
    insyj = 1+EditDistance(X,Y[1...n-1])
    sub = 1+EditDistance(X[1...m-1],Y[1...n-1])
    return min(delxi, insyj, sub)
```

What's the time complexity of EditDistance?

The memoized version and iterative versions are left as exercises.

Relevant parts of the textbook

Longest common subsequence is discussed in Section 15.4.

The edit distance problem is discussed at the end of chapter 5 in the "Problems" section. Confusingly, the "Problems" section doesn't have a section number, and sectioning restarts again at 15.1. Edit distance is discussed in Problems 15.3.

It is also worth reading the main section 15.3 which discusses the essential elements of dynamic programming.