Notes

Introduction:

The idea of dynamic programming is with two key ideas:

- Identifying optimal substructure:
    - Optimum to the big problem is built on the optimum of smaller sub-problems
- Taking advantage of overlapping sub-problems:
    - Only solve each sub-problem once, then repeat it

Dynamic Programming: Longest Common Subsequence

Looking at the example of two frogs:

DNA:
AGCCCTAAGGGCTACCTAGCTT

DNA:
GACAGCCTACAAGCGTTAGCTTG

They look similar but to make sure we need to look at the DNA:

- AGCCCTAAGGGCTACCTAGCTT
- GACAGCCTACAAGCGTTAGCTTG

From the naked eye it is hard to see any similarities but they do share a long subsequence:

- AGCCTAAGCTTAGCTT

But how did we find that? We need to discover the longest common subsequence (LCS)

Outside of this example the problem of finding the LCS is one that is in a lot of computer science: Cryptography, svn and git all use them as well as unix's diff command.

We could brute force it but that just isn't realistic for large datasets. So we need to use Dynamic programming to solve the problem.

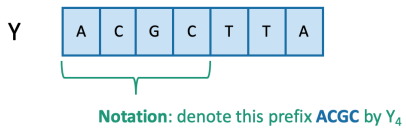
DF Formulation

We know we have two steps when we approach a problem with DP:

1. Identify optimal substructure and finding overlapping subproblems
2. The recursive formulation to solve the problem

Before we can identify the optimal substructure we need to establish notation.

Y4 will represent the subsequence ACGC within a larger sequence "ACGCTTA":



And we will use a matrix C to keep track of the LCS in smaller subsequences. C[i,j[will denote the length of the LCS in subsequences Xi and Yj.

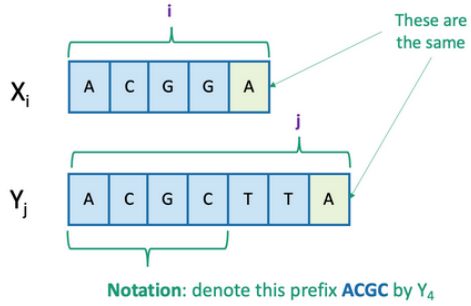
Since the subproblem is finding LCS's prefixes of X and Y we know there is a relationship between LCS's prefixes and the whole sequence.

So we know the optimal substructure to the problem now we need to figure out a recursive formula.

When we have two elements of both Xi and Yj we may find they point to the same location or not. So when we increase the length of X and Y from i-1 and i from j-1 to j, then the LCS has changed as the new letter should contribute to the LCS. So we can add one to our matrix **C[i-1, j-1]** that way we just need to increment the current count. So we get the formula:

$$C[i,j] = 1 + C[i-1, j-1]$$

As a figure to make it much more clear:



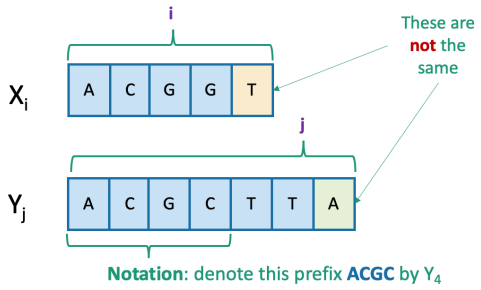
Then C[i,j] = 1 + C[i-1,j-1].

- because $LCS(X_i,Y_j) = LCS(X_{i-1},Y_{j-1})$ followed by A

In the second case that X and Y do not match with a new number then we "either excluding the latest element in sequence **X**, that is **X[i]** - or we can consider the value of LCS by excluding the latest element in sequence **Y**, that is: **Y[j]**." using this formula:

$$\max\{C[i,j-1], C[i-1,j]\}$$

Or again visualised in this handy diagram:



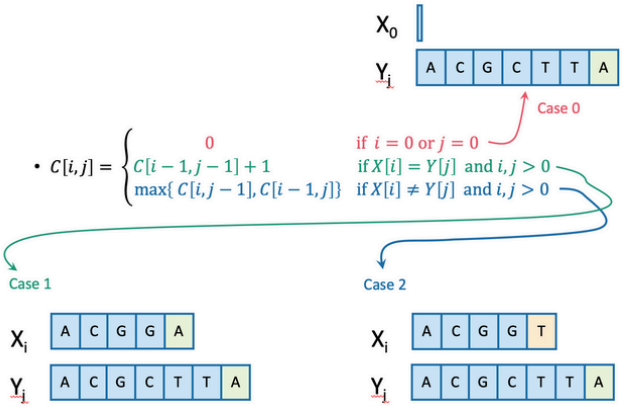
Then C[i,j] = max{ C[i-1,j], C[i,j-1] }.

- either $LCS(X_i,Y_j) = LCS(X_{i-1},Y_j)$ and T is not involved,
- or $LCS(X_i,Y_j) = LCS(X_i,Y_{j-1})$ and A is not involved,

So now we can create the formula:

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + C[i\text{-}1, j\text{-}1] & \text{if } X[i] = Y[i] \\ \max C[i, j\text{-}1], C[i\text{-}1,j] & \text{if } X[i] \mathrel{!}= Y[i] \end{cases}$$

And the figure:



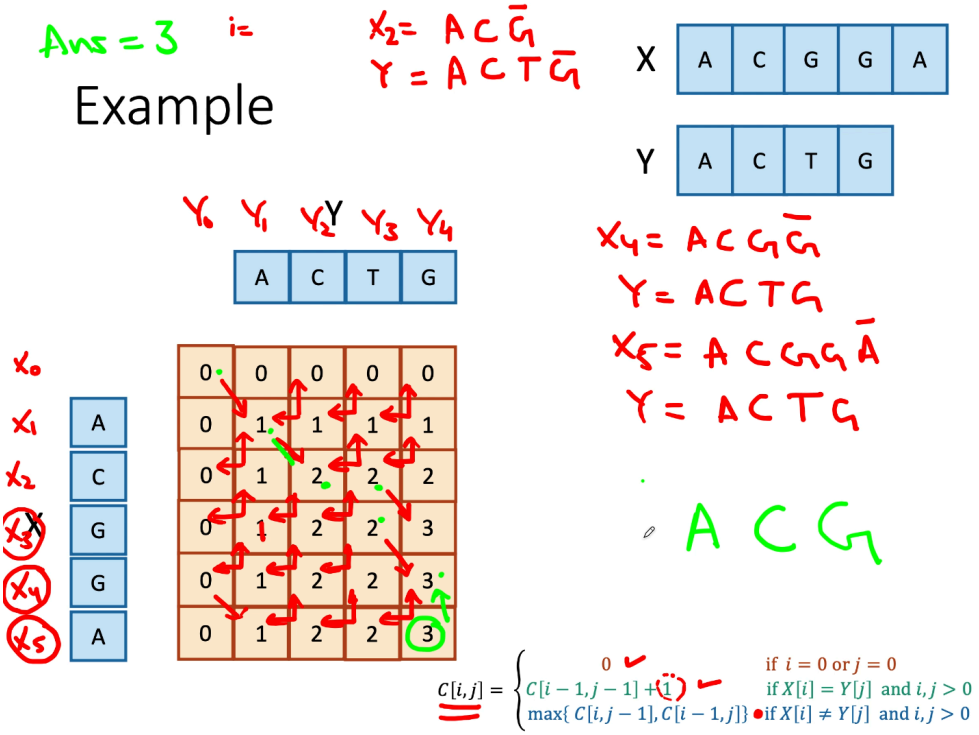
Algorithm

So we have identified the steps to take. Now we formulate it into a formula!

- **LCS**(X, Y):
  - C[i,0] = C[0,j] = 0 for all i = 1,…,m, j=1,…n.
  - **For** i = 1,…,m and j = 1,…,n:
    - **If** X[i] = Y[j]:
      - C[i,j] = C[i-1,j-1] + 1
    - **Else:**
      - C[i,j] = max{ C[i,j-1], C[i-1,j] }

This can be programmed with relative ease as well.

The lecture took each step but this is the final solution which can be followed along:



Complexity analysis

It is O(mn) where n and m are the length of the subsequences.

Dynamic Programming: Knapsack Problem

Dynamic programming is good at finding solutions for optimisation problems. The one we are looking at is the Knapsack problem. We have to pack n items in the knapsack, but it has a capacity, such as weight.

Example is:

We can store 10 items but we need to maximise the value.
This can be expanded out to more items, maybe 1000 each with varied values. So we need to sort out finding items with good value.

Each element is denoted as i with wi and Vi (weight and value for item i). Out knapsack has W for capacity.

There are two variants on the knapsack problem depending on if we are allowed to pack an item more than once.

That gives us Unbound or 0/1.

Dynamic Programming: Unbounded Knapsack Problem

So as our first step we need to identify the optimal substructure and identify the overlapping subproblems then move onto the recursive formula.

Let's start then with a small knapsack since the solution does not change but makes it more workable. Then we move onto larger:
Small -> large -> larger

When we look at an item (i) that is at the capacity (x - wi) with a value of (V - vi) we have the optimal solution. So we work back from out max and how much each item takes away from the optimal.

K[x] is out optimal capacity value for x. So:

$$\mathrm{K}[x] = \max_i \{\mathrm{K}[x - w_i] + v_i\}$$

$$\text{such that } \; w_i \leq x$$

So we look over all the items and choose the one that leads us to K[x].
But we need to only consider the items who weigh less than our current capacity of x. So wi hast to be smaller than x.
We can create an algorithm like this: - that also changes if k[x] is update when a new item is found:

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = ∅
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Dynamic Programming: 0/1 Knapsack Problem

The real difference between this solution and the last is simply the capacity. Since we have similar subproblems.

So the key difference here is that once we have taken an item it can no longer be picked again! So we need to keep track of each item.

We start by using a two dimensional array. This will store the optimal value for items (j) and a capacity (x). So now we have a two dimensional structure that stores different values for j and x.
So we have two possible options when we consider item j for capacity x:

- J will improve the optimal solution meaning:

$$\mathrm{K}(x, j) = \mathrm{K}(x, j-1)$$

- J will not improve the solution:

$$\mathrm{K}(x, j) = \mathrm{K}(x - w_j, j-1) + v_j$$

This is similar to the unbound bestion, so we will look for an item which k(x,j) is maximised. When we combine both we get:

$$K[x,j] = \max\{ K[x, j-1] , K[x - w_j, j-1] + v_j \}$$

Case 1 Case 2

Case 1 is when we just keep j-1 since it is better. But if we have another that is better then we go through case 2.

Algorithm

Zero-One-Knapsack(W, n, w, v):

- K[x,0] = 0 for all x = 0,…,W
- K[0,i] = 0 for all i = 0,…,n
- **for** x = 1,…,W:
  - **for** j = 1,…,n:
    - K[x,j] = K[x, j-1] (Case 1)
    - **if** $w_j \leq x$:
      - K[x,j] = max{ K[x,j], $K[x - w_j, j-1] + v_j$ } (Case 2)
- **return** K[W,n]

Complexity: "**O(nW)**, where **n** is the number of items and **W** is the capacity of knapsack."