## Edit Distance notes, Rice University Algorithms course

…https://www.clear.rice.edu/comp130/12spring/editdist/

### Inductive Representation of the Recurrence Relations

Now that we have a hold on the fundamental recurrence relations in the problem, we can bring some formalism to bear on the problem.   The process of breaking a problem down into successive steps is formalized in mathematics as [induction](http://en.wikipedia.org/wiki/Mathematical_induction).    We can use this formalism to bring some structure and concreteness to our emerging algorithm.

Key to solving a problem using dynamic programming is to clearly state the recurrence relations which state how the sub-pieces relate to the whole.   This amounts to the statement of the base and inductive cases that are fundamental to inductive proofs:

## Base cases:

The edit distance between any string, S,  (i.e. either string A or B) and an empty string is len(S), i.e. just insert all the characters in S or delete all the characters in S, depending on which way you look at it.

## Inductive cases:

We can analyze two substrings, one of length i, one of length j, in terms of smaller substrings, breaking the analysis into 4 distinct situations:

* Both smaller substrings are one element smaller, i.e.   
   lengths i-1 & j-1 respectively: A[:i-1], B[:j-1]  
  + This corresponds to doing nothing if the current character is the same for both A and B, or
  + This corresponds to replacing the last letter in A with the last letter in B if the current characters are different.
* Only one substring is smaller, i.e
  + A[:i-1], B[:j] This corresponds to performing an insertion on the "B"  
     substring to make it the same as the "A" string.
  + A[:i],  B[:j-1] This corresponds to performing a deletion on the "B" substring  
     ("**j**" indices) to make it the same as the "A" string ("i" indices).

It is important to note that the details about "replacements" (substitutions), "insertions" and "deletions" etc are almost immaterial to what is going on here, for several reasons:

* Other edit distance definitions include more types of operations
* For more general problems, such as DNA sequence analysis, the edit operations are different.
* The edit distance algorithm doesn’t actually care about what the operation performed is, just that it was performed, thus raising the edit count by one.

…

If we didn't have to worry about how fast our algorithm ran, the top-down approach would be just fine.   It's easy to understand and is easy to code.   For small problems, we could stop right there and be perfectly happy...but we're never satisfied, are we?    Let's see if another approach will ease the efficiency problems of our top-down approach, though be warned, we may be trading one set of problems for another!

Let's try solving all the sub-problems first, i.e. a bottom-up approach.   But we need to keep track of the what we've solved so that we can use them in subsequent calculations.  Since the sub-problems involve all possible sub-strings in all possible configurations, we need to find some way to easily represent all those possible combinations.  ==> Matrices fit our design bill very well.

A matrix represents all possible combinations of one set of entities with another, creating a 2-dimensional space of all combinations.

So, one way to represent all the possible sub-problems is to create a matix, "D", that holds the results (minimum edit distances) for all the sub-problems where

* D[i,j] is the edit distance between the substring of A of length i to the substring of B of length j.
* String A is across rows and string B is across the columns, starting at position #1 (not 0!).
  + Technically, each row or column position in the matrix represents the *entire* sub-string where the last character in the substring is the one currently being compared.
* The first column and row,  D[i,0] and D[0,j] represents the distance of the substrings to the empty string

*Aside*: One way to think of the first colum/row as of "padding" and empty string in front of the original strings. The technique of "padding" the A and B strings with a "dummy values  is a very common technique in both mathematical proofs and numerical methods and can be found in diverse applications from computer graphics to quantum physics.

NOTE: This discussion uses the standard mathematical notation for matrix indexing, e.g. "D[i,j]".   In Python code, since 2-D matrices can be represented as lists of lists, *e.g.*   
  
a 2x3 matrix could be D = [[0, 1, 2], [3, 4, 5]].  
  
Indexing a value in the matrix would be written in Python as D[i][j].

Let's see how the recurrence relationships manifest themselves in this implementation:

## Base cases:

The base case edit distance values are trivial to calculate:  They are just the lengths of the sub-strings.   We can just fill our matrix with those values using simple loops.

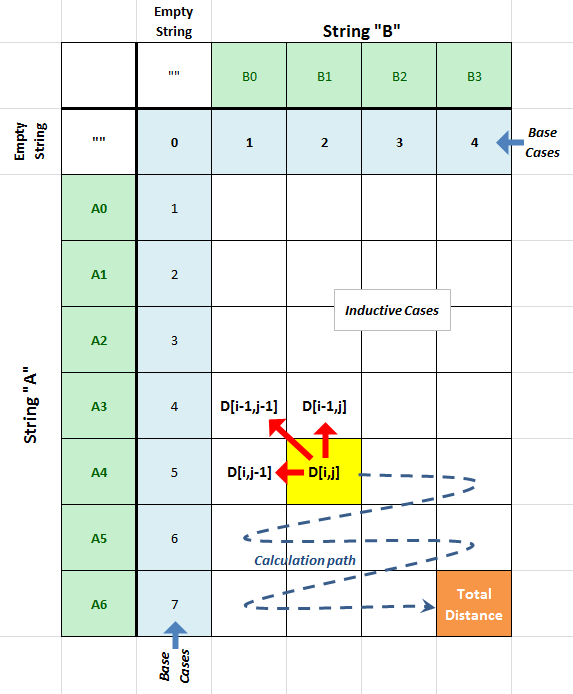
## Inductive Cases:

The smallest edit distance between two strings has two possibilities, depending on the last characters in the two strings:

1. If the *last characters are equal*: The edit distance is the edit distance of the two substrings, both one character shorter than the current substrings, i.e.   
     
   D[i,j] = 0+D[i-1,j-1]
2. If the *last characters are not equal*: The edit distance is one more edit than the edit distance of the smallest edit distance of any of the 3 possible substring situations, i.e.:  
     
   D[i,j] = 1 + min(D[i, j-1], D[i-1, j], D[i-1, j-1])

Notice that for the inductive case, the sub-problem values that any spot in the matrix depends on are only to the left and above that point in the matrix (see the diagram below).   That is, the edit distance value at any given point in the matrix does NOT depend on any values to its right or below it.    This means that if we iterate through the matrix from left to right and from top to bottom, we will always be iterating into positions in the matrix whose values only depend on values that we've already calculated!

*The last element in the matrix, at the lower right corner, holds the edit distance for the entire source string* ***being transformed*** *into the entire target string, and thus, is our final answer to the whole problem.*

  
**IMPORTANT: In the headers above, "Ai" and "Bj" refer to sub-strings of the A and B strings respectively, from their start up to and including the i'th/j'th character.**