Edit Distance for Strings

One important problem in biological computing is to determine how "similar" two primary DNA sequences are. The reason for this is that similarity of the primary sequence may predict similarity of function, for example, of the genes being coded for. Ignoring the biology, we can consider the DNA sequences as finite strings over a four letter alphabet: $\{A, C, G, T\}$. There are many different definitions of similarity; we consider a simple one called "edit distance."

The edit distance between two strings s_1 and s_2 is the *minimum* number of operations to produce s_2 from s_1, where an operation can be the deletion of a letter, the insertion of a letter, or the transformation of a letter into a different letter. For example, here are two strings.

```
AAGTCTTATACAGGC
ATGACTATAGGGCA
```

To indicate how we might have produced one string from the other, we can diagram a possible set of operations as follows.

```
AAGTCTTATACAGGC-
 X X      X
ATGACT-ATAG-GGCA
```

Here we've reproduced the two strings. Each letter of each string corresponds to a letter or a `-` in the other string. If the letters are different, a letter transformation is necessary, and we've marked the pair with an `X`. If there is a letter in the first string and a `-` in the second string, this indicates a deletion of the letter in the first string. If there is a letter in the second string and a `-` in the first string, this indicates an insertion of the letter into the second string. In the editing indicated above, we have obtained the second string from the first by three letter transformations, two deletions, and one insertion, for a total of 6 operations. Thus, the edit distance between these strings is at most 6. But there might be another editing that uses fewer total operations.

Solving the edit distance problem

We illustrate the idea of memoization by using it to obtain a polynomial time algorithm for the edit distance problem. We'll assume that the strings are represented as two vectors of symbols, say `vector1` and `vector2`. We let n denote the length of `vector1` and m denote the length of `vector2`.

A purely recursive algorithm for this problem can be derived as follows. We consider the last character of `vector1` and the last character of `vector2`. If they are equal, then the edit distance between them is equal to the edit distance between the first $n-1$ characters of `vector1` and the first $m-1$ characters of `vector2`. If they are different, then there are three possibilities: the last character of `vector1` is deleted, the last character of `vector2` is inserted, or the last character of `vector1` is transformed into the last character of `vector2`.

When the two last characters are unequal, the edit distance is one more than the minimum of the edit distance between the first $n-1$ characters of `vector1` and the first m characters of `vector2` (the delete case), the edit distance between the first n characters of `vector1` and the first $m-1$ characters of `vector2` (the insert case), and the edit distance between the first $n-1$ characters of `vector1` and the first $m-1$ characters of `vector2`. These recursive calls "reduce" the arguments because one or both of n and m is decreased by 1. The base cases are when either string is empty, and the edit distance is the length of the other string (all inserts or all deletes.)

These considerations lead to the following procedure `changes` (from *Concrete Abstractions*, Chapter 12.)

```
(define changes
  (lambda (vector1 vector2)
    (let ((n (vector-length vector1))
          (m (vector-length vector2)))
      (define changes-in-first-and-elements
        (lambda (i j)
          (cond
            ((= i 0) j)
            ((= j 0) i)
            (else
              (if (equal? (vector-ref vector1 (- i 1)) (vector-ref vector2 (- j 1)))
                  (changes-in-first-and-elements (- i 1) (- j 1))
                  (min (+ 1 (changes-in-first-and-elements (- i 1) j))
                       (+ 1 (changes-in-first-and-elements i (- j 1)))
                       (+ 1 (changes-in-first-and-elements (- i 1) (- j 1)))))))))
(changes-in-first-and-elements n m))))
```

Here is an example run of this procedure that completes relatively quickly.

```
> (changes '#(a c a g g c) '#(t a g g g c a))
4
```

Here is the top of the tree of calls to the auxiliary procedure.

```
                     6 7
             /        |        \
         5 7         6 6        5 6
      /   |    \      |      /   |   \
  4 7    5 6   4 6   5 5   4 6  5 5   4 5
 / | \  / | \  / | \  |   / | \  |     |
.  .  ..  .  ..  .  . .  .  .  . .     .
```

There are repeated calls even in this much of the tree, for example, the pairs $(5,6)$, $(4,6)$, $(5,5)$. As we go deeper in the tree, more and more calls will be repeated. In the worst case, the number of calls is exponential in the lengths of the vectors.

In this case, there are 2912 calls to the auxiliary procedure during the evaluation of `changes` given above. How many *different* pairs of arguments is it called with? No more than 56, because there are 7 possible values for the first argument (from 0 to 6) and 8 possible values for the second argument (from 0 to 7).

When we see many calls with the same arguments, we think: try memoization. We'd like a 2-dimensional table with $m+1$ rows, $n+1$ columns and random access to an element indexed by a pair i and j. In Scheme we can use a vector of vectors to represent the 2-dimensional table.

Memoization to dynamic programming

Dynamic programming is a useful method of designing algorithms. One way to approach dynamic programming is to think in three stages: pure recursive algorithm, memoization, and then analysis of the table built by memoization to see how it can be built efficiently "bottom up," in the style of dynamic programming. If we carry out that analysis for the edit distance problem, we find that the table can be filled in systematically as follows.

First we initialize the table with the base cases, which correspond to $n = 0$ and $m = 0$. The entry for row i and column j is the edit distance of the first i characters of the first string (shown vertically on the left) and the first j characters of the second string (shown horizontally on top.) Thus, the entry 4 in the top row means that to transform the empty string (0 characters) to the string `TAGG`, the edit distance is 4, namely, 4 insert operations. The entry 3 in the leftmost column means that to transform the string `ACA` to the empty string, the edit distance is 3, namely, 3 delete operations.

	-	T	A	G	G	G	C	A
-	0	1	2	3	4	5	6	7
A	1							
C	2							
A	3							
G	4							
G	5							
C	6							

Then we begin filling in the other entries of the table. If the characters in the row and column are equal, the corresponding entry is just copied from the entry diagonally up and to the left. If the characters in the row and column are unequal, the entry is 1 plus the minimum of the three entries above, diagonally up and to the left, and to the left of the desired entry. This rule has been used to partially fill the table below.

	-	T	A	G	G	G	C	A
-	0	1	2	3	4	5	6	7
A	1	1	1	2	3			
C	2	2	2	2				
A	3	3	2	3				
G	4	4						
G	5							
C	6							

For example, the edit distance to transform `ACA` to `TA` is 2, shown by the alignment:

```
A C A
X
T - A
```

Filling in the rest of the table using this rule, we have

	-	T	A	G	G	G	C	A
-	0	1	2	3	4	5	6	7
A	1	1	1	2	3	4	5	6
C	2	2	2	2	3	4	4	5
A	3	3	2	3	3	4	5	4
G	4	4	3	2	3	3	4	5
G	5	5	4	3	2	3	4	5
C	6	6	5	4	3	4	3	4

We can reconstruct a set of operations to achieve the minimum edit distance by tracing back how the number was derived. For example, the 4 in the lower right corner can only be obtained as 1 plus the 3 in the last row of the next to last column. That 3 is obtained (because the characters are the same) from the 3 diagonally up and to its left, and so on. Tracing a path back to the upper left corner, we obtain the following.

```
   -  T  A  G  G  G  C  A
-  0 <1  2  3  4  5  6  7
A  1  1 \1  2  3  4  5  6
C  2  2 ^2  2  3  4  4  5
A  3  3  2 \3  3  4  5  4
G  4  4  3  2 \3  3  4  5
G  5  5  4  3  2 \3  4  5
C  6  6  5  4  3  4 \3 <4
```

The path we chose (there were alternatives) corresponds to the alignment:

```
   -  A  C  A  G  G  C  -
            X
   T  A  -  G  G  G  C  A
```

Thus, this simple (though somewhat mysterious) systematic iterative algorithm to fill in the table using one or three neighboring entries can be viewed as derived from the pure recursive algorithm above.