Dynamic Programming

Sometimes, divide-and-conquer leads to overlapping subproblems and thus to redundant computations. It is not uncommon that the redundancies accumulate and cause an exponential amount of wasted time. We can avoid the waste using a simple idea: **solve each subproblem only once**. To be able to do that, we have to add a certain amount of book-keeping to remember subproblems we have already solved. The technical name for this design paradigm is *dynamic programming*.

Edit distance. We illustrate dynamic programming using the edit distance problem, which is motivated by questions in genetics. We assume a finite set of *characters* or *letters*, Σ, which we refer to as the *alphabet*, and we consider *strings* or *words* formed by concatenating finitely many characters from the alphabet. The *edit distance* between two words is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word to the other. For example, the edit distance between FOOD and MONEY is at most four:

FOOD $\rightarrow$ MOOD $\rightarrow$ MOND $\rightarrow$ MONED $\rightarrow$ MONEY

A better way to display the editing process is the *gap representation* that places the words one above the other, with a gap in the first word for every insertion and a gap in the second word for every deletion:

```
F O O   D
M O N E Y
```

Columns with two different characters correspond to substitutions. The number of editing steps is therefore the number of columns that do not contain the same character twice.

Prefix property. It is not difficult to see that you cannot get from FOOD to MONEY in less than four steps. However, for longer examples it seems considerably more difficult to find the minimum number of steps or to recognize an optimal edit sequence. Consider for example

```
A L G O R   I   T H M
A L   T R U I S T I C
```

Is this optimal or, equivalently, is the edit distance between ALGORITHM and ALTRUISTIC six? Instead of answering this specific question, we develop a dynamic programming algorithm that computes the edit distance between an m-character string $A[1..m]$ and an n-character string $B[1..n]$. Let $E(i, j)$ be the edit distance between the prefixes of length i and j, that is, between $A[1..i]$ and $B[1..j]$. The edit distance between the complete strings is therefore $E(m, n)$. A crucial step towards the development of this algorithm is the following observation about the gap representation of an optimal edit sequence.

PREFIX PROPERTY. If we remove the last column of an optimal edit sequence then the remaining columns represent an optimal edit sequence for the remaining substrings.

We can easily prove this claim by contradiction: if the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence of the original strings.

Recursive formulation. We use the Prefix Property to develop a recurrence relation for E. The dynamic programming algorithm will be a straightforward implementation of that relation. There are a couple of obvious base cases:

- **Erasing:** we need i deletions to erase an i-character string, $E(i, 0) = i$.

- **Creating:** we need j insertions to create a j-character string, $E(0, j) = j$.

In general, there are four possibilities for the last column in an optimal edit sequence.

- **Insertion:** the last entry in the top row is empty, $E(i, j) = E(i, j - 1) + 1$.
- **Deletion:** the last entry in the bottom row is empty, $E(i, j) = E(i - 1, j) + 1$.
- **Substitution:** both rows have characters in the last column that are different, $E(i, j) = E(i - 1, j - 1) + 1$.
- **No action:** both rows end in the same character, $E(i, j) = E(i - 1, j - 1)$.

Let P be the logical proposition $A[i] \neq B[j]$ and denote by $|P|$ its indicator variable: $|P| = 1$ if P is true and $|P| = 0$ if P is false. We can now summarize and for $i, j > 0$ get the edit distance as the smallest of the possibilities:

$$E(i,j) = \min \left\{ \begin{array}{l} E(i, j-1) + 1 \\ E(i-1, j) + 1 \\ E(i-1, j-1) + |P| \end{array} \right\}.$$

The algorithm. If we turned this recurrence relation directly into a divide-and-conquer algorithm, we would have the following horrible recurrence for the running time:

$$\begin{aligned} T(m, n) &= T(m, n-1) + T(m-1, n) \\ &\quad + T(m-1, n-1) + 1. \end{aligned}$$

The solution to this recurrence is exponential in m and n, which is clearly not the way to go. Instead, let us build an $m+1$ times $n+1$ table of possible values of $E(i, j)$. We can start by filling in the base cases, the entries in the 0-th row and column. To fill in any other entry, we need to know the values directly above, directly to the left, and both above and to the left. If we fill the table from top to bottom and from left to right then whenever we reach an entry, the entries it depends on are already available.

```
int EditDistance(int m, n)
  for i = 0 to m do E[i, 0] = i endfor ;
  for j = 1 to n do E[0, j] = j endfor ;
  for i = 1 to m do
    for j = 1 to n do
      E[i, j] = min{E[i, j − 1] + 1, E[i − 1, j] + 1,
                    E[i − 1, j − 1] + |A[i] ≠ B[j]|}
    endfor
  endfor ;
  return E[m, n].
```

Since there are $(m+1)(n+1)$ entries in the table and each takes a constant time to compute, the total running time is in $O(mn)$.

An example. The table constructed for the conversion of ALGORITHM to ALTRUISTIC is shown in Figure 5. Boxed numbers indicate places where the two strings have equal characters. The arrows indicate the predecessors that define the entries. Each direction of arrow corresponds to a different edit operation: horizontal for insertion, vertical for deletion, and diagonal for substitution. Dotted diagonal arrows indicate free substitutions of a letter for itself.

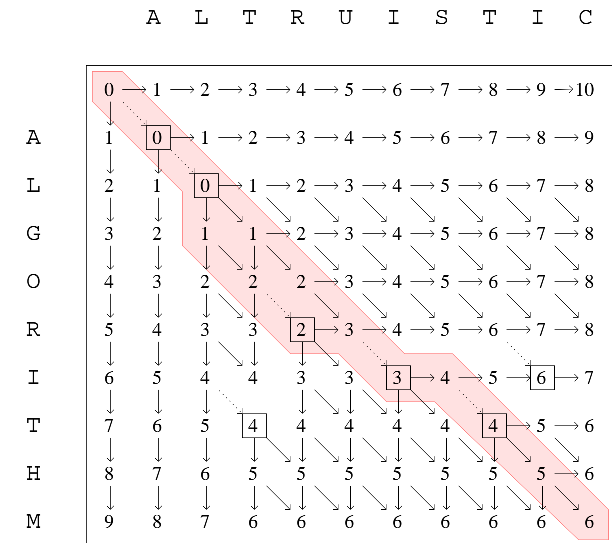

Figure 5: The table of edit distances between all prefixes of ALGORITHM and of ALTRUISTIC. The shaded area highlights the optimal edit sequences, which are paths from the upper left to the lower right corner.

Recovering the edit sequence. By construction, there is at least one path from the upper left to the lower right corner, but often there will be several. Each such path describes an optimal edit sequence. For the example at hand, we have three optimal edit sequences:

```
  A L G O R I   T H M
  A L T R U I S T I C

A L G O R   I   T H M
A L T   R U I S T I C
```

```
A L G O R   I   T H M
A L   T R U I S T I C
```

They are easily recovered by tracing the paths backward from the end to the beginning. The following algorithm recovers an optimal solution that also minimizes the number of insertions and deletions.

```
void R(int i, j)
  if i > 0 or j > 0 then
    switch incoming arrow:
      case ↘: R(i − 1, j − 1); print (A[i], B[j])
      case ↓: R(i − 1, j); print (A[i], _)
      case →: R(i, j − 1); print (_, B[j]).
    endswitch
  endif.
```

Summary. The structure of dynamic programming is again similar to divide-and-conquer, except that the subproblems to be solved overlap. As a consequence we get different recursive paths to the same subproblems. To develop a dynamic programming algorithm that avoids redundant solutions, we generally proceed in two steps:

1. We formulate the problem recursively. In other words, we write down the answer to the whole problem as a combination of the answers to smaller subproblems.
2. We build solutions from bottom up. Starting with the base cases, we work our way up to the final solution and (usually) store intermediate solutions in a table.

For dynamic programming to be effective, we need a structure that leads to at most some polynomial number of different subproblems. Most commonly, we deal with sequences, which have linearly many prefixes and suffixes and quadratically many contiguous substrings.