

In fact, this is just a longest common subsequence problem, where the second string is the elements of S sorted in increasing order: 123456789. Any common sequence of these two must (a) represent characters in proper order in S , and (b) use only characters with increasing position in the collating sequence—so the longest one does the job. Of course, this approach can be modified to give the longest decreasing sequence simply by reversing the sorted order.

As you can see, our edit distance routine can be made to do many amazing things easily. The trick is observing that your problem is just a special case of approximate string matching.

The alert reader may notice that it is unnecessary to keep all $O(mn)$ cells to compute the cost of an alignment. If we evaluate the recurrence by filling in the columns of the matrix from left to right, we will never need more than two columns of cells to store what is necessary to complete the computation. Thus, $O(m)$ space is sufficient to evaluate the recurrence without changing the time complexity. This is good, but unfortunately we cannot reconstruct the alignment without the full matrix.

Saving space in dynamic programming is very important. Since memory on any computer is limited, using $O(nm)$ space proves more of a bottleneck than $O(nm)$ time. Fortunately, there is a clever divide-and-conquer algorithm that computes the actual alignment in the same $O(nm)$ time but only $O(m)$ space. It is discussed in Section 21.4 (page 688).

10.3 Longest Increasing Subsequence

There are three steps involved in solving a problem by dynamic programming:

1. Formulate the answer you want as a recurrence relation or recursive algorithm.
2. Show that the number of different parameter values taken on by your recurrence is bounded by a (hopefully small) polynomial.
3. Specify an evaluation order for the recurrence so the partial results you need are always available when you need them.

To see how this is done, let's see how we would develop an algorithm to find the longest monotonically increasing subsequence within a sequence of n numbers. Truth be told, this was described as a special case of edit distance in Section 10.2.4 (page 323), where it was called *maximum monotone subsequence*. Still, it is instructive to work it out from scratch. Indeed, dynamic programming algorithms are often easier to reinvent than look up.

We distinguish an increasing sequence from a *run*, where the elements must be physical neighbors of each other. The selected elements of both must be sorted in increasing order from left to right. For example, consider the sequence

$$S = (2, 4, 3, 5, 1, 7, 6, 9, 8)$$

The longest increasing subsequence of S is of length 5: for example, (2,3,5,6,8). In fact, there are eight of this length (can you enumerate them?). There are four increasing runs of length 2: (2, 4), (3, 5), (1, 7), and (6, 9).

Finding the longest increasing *run* in a numerical sequence is straightforward. Indeed, you should be able to easily devise a linear-time algorithm. But finding the longest increasing subsequence is considerably trickier. How can we identify which scattered elements to skip?

To apply dynamic programming, we need to design a recurrence relation for the length of the longest sequence. To find the right recurrence, ask what information about the first $n - 1$ elements of $S = (s_1, \dots, s_n)$ would enable you to find the answer for the entire sequence:

- The length L of the longest increasing sequence in $(s_1, s_2, \dots, s_{n-1})$ seems a useful thing to know. In fact, this will be the length of the longest increasing sequence in S , unless s_n extends some increasing sequence of the same length.

Unfortunately, this length L is not enough information to complete the full solution. Suppose I told you that the longest increasing sequence in $(s_1, s_2, \dots, s_{n-1})$ was of length 5 and that $s_n = 8$. Will the length of the longest increasing subsequence of S be 5 or 6? It depends on whether the length-5 sequence ended with a value < 8 .

- We need to know the length of the longest sequence that s_n will extend. To be certain we know this, we really need the length of the longest sequence ending at *every* possible value s_i .

This provides the idea around which to build a recurrence. Define L_i to be the length of the longest sequence ending with s_i . The longest increasing sequence containing s_n will be formed by appending it to the longest increasing sequence to the left of n that ends on a number smaller than s_n . The following recurrence computes L_i :

$$L_i = 1 + \max_{\substack{0 \leq j < i \\ s_j < s_i}} L_j,$$

$$L_0 = 0$$

These values define the length of the longest increasing sequence ending at each sequence element. The length of the longest increasing subsequence of S is given by $L = \max_{1 \leq i \leq n} L_i$, since the winning sequence must end somewhere. Here is the table associated with our previous example:

Index i	1	2	3	4	5	6	7	8	9
Sequence s_i	2	4	3	5	1	7	6	9	8
Length L_i	1	2	2	3	1	4	4	5	5
Predecessor p_i	–	1	1	2	–	4	4	6	6

What auxiliary information will we need to store to reconstruct the actual sequence instead of its length? For each element s_i , we will store its *predecessor*—the index p_i of the element that appears immediately before s_i in a longest increasing sequence ending at s_i . Since all of these pointers go towards the left, it is a simple matter to start from the last value of the longest sequence and follow the pointers back so as to reconstruct the other items in the sequence.

What is the time complexity of this algorithm? Each one of the n values of L_i is computed by comparing s_i against the $i - 1 \leq n$ values to the left of it, so this analysis gives a total of $O(n^2)$ time. In fact, by using dictionary data structures in a clever way, we can evaluate this recurrence in $O(n \lg n)$ time. However, the simple recurrence would be easy to program and therefore is a good place to start.

Take-Home Lesson: Once you understand dynamic programming, it can be easier to work out such algorithms from scratch than to try to look them up.

10.4 War Story: Text Compression for Bar Codes

Ynjiun waved his laser wand over the torn and crumpled fragments of a bar code label. The system hesitated for a few seconds, then responded with a pleasant *blip* sound. He smiled at me in triumph. “Virtually indestructible.”

I was visiting the research laboratories of Symbol Technologies (now Zebra), the world’s leading manufacturer of bar code scanning equipment. Although we take bar codes for granted, there is a surprising amount of technology behind them. Bar codes exist because conventional optical character recognition (OCR) systems are not sufficiently reliable for inventory operations. The bar code symbology familiar to us on each box of cereal, pack of gum, or can of soup encodes a ten-digit number with enough error correction that it is virtually impossible to scan the wrong number, even if the can is upside-down or dented. Occasionally, the cashier won’t be able to get a label to scan at all, but once you hear that *blip* you know it was read correctly.

The ten-digit capacity of conventional bar code labels provides room enough to only store a single ID number in a label. Thus, any application of supermarket bar codes must have a database mapping (say) 11141-47011 to a particular brand and size of soy sauce. The holy grail of the bar code world had long been the development of higher-capacity bar code symbologies that can store entire documents, yet still be read reliably.

“PDF-417 is our new, two-dimensional bar code symbology,” Ynjiun explained. A sample label is shown in Figure 10.7. Although you may be more familiar with QR codes, PDF-417 is now a well accepted standard. Indeed, the back of every New York State drivers license contains the criminal record of its owner, elegantly rendered in PDF-417.

“How much data can you fit in a typical 1-inch label?” I asked him.

“It depends upon the level of error correction we use, but about 1,000 bytes. That’s enough for a small text file or image,” he said.