

```

    for (j = 0; j <= n; j++) {
        bc[j][j] = 1;
    }

    for (i = 2; i <= n; i++) {
        for (j = 1; j < i; j++) {
            bc[i][j] = bc[i-1][j-1] + bc[i-1][j];
        }
    }

    return(bc[n][k]);
}

```

Study this function carefully to make sure you see how we did it. The rest of this chapter will focus more on formulating and analyzing the appropriate recurrence than the mechanics of table manipulation demonstrated here.

10.2 Approximate String Matching

Searching for patterns in text strings is a problem of unquestionable importance. Back in Section 6.7 (page 188) I presented algorithms for *exact* string matching—finding where the pattern string P occurs as a substring of the text string T . But life is often not that simple. Words in either the text or pattern can be misspelled (sic), robbing us of exact similarity. Evolutionary changes in genomic sequences or language usage mean that we often search with archaic patterns in mind: “Thou shalt not kill” morphs over time into “You should not murder.”

How can we search for the substring closest to a given pattern, to compensate for spelling errors? To deal with inexact string matching, we must first define a cost function telling us how far apart two strings are. A reasonable distance measure reflects the number of *changes* that must be made to convert one string to another. There are three natural types of changes:

- *Substitution* – Replace a single character in pattern P with a different character, such as changing *shot* to *spot*.
- *Insertion* – Insert a single character into pattern P to help it match text T , such as changing *ago* to *agog*.
- *Deletion* – Delete a single character from pattern P to help it match text T , such as changing *hour* to *our*.

Properly posing the question of string similarity requires us to set the cost of each such transform operation. Assigning each operation an equal cost of 1 defines the *edit distance* between two strings. Approximate string matching arises in many applications, as detailed in Section 21.4 (page 688).

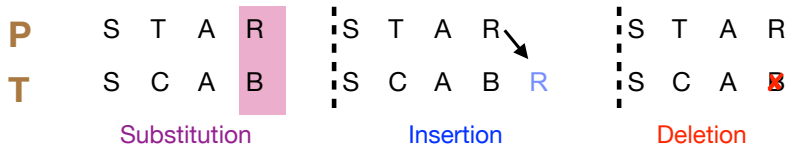


Figure 10.4: In a single string edit operation, the last character must be either matched/substituted, inserted, or deleted.

Approximate string matching seems like a difficult problem, because we must decide exactly where to best perform a complicated sequence of insert/delete operations in pattern and text. To solve it, let's think about the problem in reverse. What information would we need to select the final operation correctly? What can happen to the last character in the matching for each string?

10.2.1 Edit Distance by Recursion

We can define a recursive algorithm using the observation that the last character in the string must either be matched, substituted, inserted, or deleted. There is no other possible choice, as shown in Figure 10.4. Chopping off the characters involved in this last edit operation leaves a pair of smaller strings. Let i and j be the indices of the last character of the relevant prefix of P and T , respectively. There are three pairs of shorter strings after the last operation, corresponding to the strings after a match/substitution, insertion, or deletion. If we knew the cost of editing these three pairs of smaller strings, we could decide which option leads to the best solution and choose that option accordingly. We *can* learn this cost through the magic of recursion.

More precisely, let $D[i, j]$ be the minimum number of differences between the substrings $P_1P_2 \dots P_i$ and $T_1T_2 \dots T_j$. $D[i, j]$ is the *minimum* of the three possible ways to extend smaller strings:

- If $(P_i = T_j)$, then $D[i - 1, j - 1]$, else $D[i - 1, j - 1] + 1$. This means we either match or substitute the i th and j th characters, depending upon whether these tail characters are the same. More generally, the cost of a single character substitution can be returned by a function $match(P_i, T_j)$.
- $D[i, j - 1] + 1$. This means that there is an extra character in the text to account for, so we do not advance the pattern pointer and we pay the cost of an insertion. More generally, the cost of a single character insertion can be returned by a function $indel(T_j)$.
- $D[i - 1, j] + 1$. This means that there is an extra character in the pattern to remove, so we do not advance the text pointer and we pay the cost of a deletion. More generally, the cost of a single character deletion can be returned by a function $indel(P_i)$.

```

#define MATCH      0      /* enumerated type symbol for match */
#define INSERT     1      /* enumerated type symbol for insert */
#define DELETE     2      /* enumerated type symbol for delete */

int string_compare_r(char *s, char *t, int i, int j) {
    int k;                /* counter */
    int opt[3];           /* cost of the three options */
    int lowest_cost;      /* lowest cost */

    if (i == 0) {        /* indel is the cost of an insertion or deletion */
        return(j * indel(' '));
    }

    if (j == 0) {
        return(i * indel(' '));
    }

    /* match is the cost of a match/substitution */

    opt[MATCH] = string_compare_r(s,t,i-1,j-1) + match(s[i],t[j]);
    opt[INSERT] = string_compare_r(s,t,i,j-1) + indel(t[j]);
    opt[DELETE] = string_compare_r(s,t,i-1,j) + indel(s[i]);

    lowest_cost = opt[MATCH];
    for (k = INSERT; k <= DELETE; k++) {
        if (opt[k] < lowest_cost) {
            lowest_cost = opt[k];
        }
    }

    return(lowest_cost);
}

```

This program is absolutely correct—convince yourself. It also turns out to be impossibly slow. Running on my computer, the computation takes several seconds to compare two 11-character strings, and disappears into Never-Never Land on anything longer.

Why is the algorithm so slow? It takes exponential time because it re-computes values again and again and again. At every position in the string, the recursion branches three ways, meaning it grows at a rate of at least 3^n —indeed, even faster since most of the calls reduce only one of the two indices, not both of them.

10.2.2 Edit Distance by Dynamic Programming

So, how can we make this algorithm practical? The important observation is that most of these recursive calls compute things that have been previously computed. How do we know? There can only be $|P| \cdot |T|$ possible unique recursive calls, since there are only that many distinct (i, j) pairs to serve as the argument parameters of the recursive calls. By storing the values for each of these (i, j) pairs in a table, we can look them up as needed and avoid recomputing them.

A table-based, dynamic programming implementation of this algorithm is given below. The table is a two-dimensional matrix m where each of the $|P| \cdot |T|$ cells contains the cost of the optimal solution to a subproblem, as well as a parent field explaining how we got to this location:

```
typedef struct {
    int cost;           /* cost of reaching this cell */
    int parent;         /* parent cell */
} cell;

cell m[MAXLEN+1][MAXLEN+1]; /* dynamic programming table */
```

Our dynamic programming implementation has three differences from the recursive version. First, it gets its intermediate values using table lookup instead of recursive calls. Second, it updates the `parent` field of each cell, which will enable us to reconstruct the edit sequence later. Third, it is implemented using a more general `goal_cell()` function instead of just returning `m[|P|][|T|].cost`. This will enable us to apply this routine to a wider class of problems.

Be aware that we adhere to special string and index conventions in the routine below. In particular, we assume that each string has been padded with an initial blank character, so the first real character of string `s` sits in `s[1]`. Why did we do this? It enables us to keep the matrix indices in sync with those of the strings for clarity. Recall that we must dedicate the zeroth row and column of `m` to store the boundary values matching the empty prefix. Alternatively, we could have left the input strings intact and adjusted the indices accordingly.

```
int string_compare(char *s, char *t, cell m[MAXLEN+1][MAXLEN+1]) {
    int i, j, k;        /* counters */
    int opt[3];          /* cost of the three options */

    for (i = 0; i <= MAXLEN; i++) {
        row_init(i, m);
        column_init(i, m);
    }
}
```

```

    for (i = 1; i < strlen(s); i++) {
        for (j = 1; j < strlen(t); j++) {
            opt[MATCH] = m[i-1][j-1].cost + match(s[i], t[j]);
            opt[INSERT] = m[i][j-1].cost + indel(t[j]);
            opt[DELETE] = m[i-1][j].cost + indel(s[i]);

            m[i][j].cost = opt[MATCH];
            m[i][j].parent = MATCH;
            for (k = INSERT; k <= DELETE; k++) {
                if (opt[k] < m[i][j].cost) {
                    m[i][j].cost = opt[k];
                    m[i][j].parent = k;
                }
            }
        }
    }

    goal_cell(s, t, &i, &j);
    return(m[i][j].cost);
}

```

To determine the value of cell (i, j) , we need to have three values sitting and waiting for us in matrix m —namely, the cells $m(i-1, j-1)$, $m(i, j-1)$, and $m(i-1, j)$. Any evaluation order with this property will do, including the row-major order used in this program.¹ The two nested loops do in fact evaluate m for every pair of string prefixes, one row at a time. Recall that the strings are padded such that $s[1]$ and $t[1]$ hold the first character of each input string, so the lengths (`strlen`) of the padded strings are one character greater than those of the input strings.

As an example, we show the cost matrix for turning $P = \text{“thou shalt”}$ into $T = \text{“you should”}$ in five moves in Figure 10.5. I encourage you to evaluate this example matrix by hand, to nail down exactly how dynamic programming works.

10.2.3 Reconstructing the Path

The string comparison function returns the cost of the optimal alignment, but not the alignment itself. Knowing you can convert “thou shalt” to “you should” in only five moves is dandy, but what is the sequence of editing operations that does it?

The possible solutions to a given dynamic programming problem are described by paths through the dynamic programming matrix, starting from the

¹Suppose we create a graph with a vertex for every matrix cell, and a directed edge (x, y) , when the value of cell x is needed to compute the value of cell y . Any topological sort on the resulting DAG (why must it be a DAG?) defines an acceptable evaluation order.

	<i>T</i>		y	o	u	-	s	h	o	u	l	d
<i>P</i>	pos	0	1	2	3	4	5	6	7	8	9	10
:		<u>0</u>	1	2	3	4	5	6	7	8	9	10
t:	1	<u>1</u>	1	2	3	4	5	6	7	8	9	10
h:	2	2	<u>2</u>	2	3	4	5	5	6	7	8	9
o:	3	3	3	<u>2</u>	3	4	5	6	5	6	7	8
u:	4	4	4	3	<u>2</u>	3	4	5	6	5	6	7
-:	5	5	5	4	3	<u>2</u>	3	4	5	6	6	7
s:	6	6	6	5	4	3	<u>2</u>	3	4	5	6	7
h:	7	7	7	6	5	4	3	<u>2</u>	<u>3</u>	4	5	6
a:	8	8	8	7	6	5	4	3	3	<u>4</u>	5	6
l:	9	9	9	8	7	6	5	4	4	4	<u>4</u>	5
t:	10	10	10	9	8	7	6	5	5	5	5	<u>5</u>

Figure 10.5: Example of a dynamic programming matrix for editing distance computation, with the underlined entries appearing on the optimal alignment path. Blue values denote insertions, green values deletions, and red values match/substitution.

initial configuration (the pair of empty strings $(0,0)$) down to the final goal state (the pair of full strings $(|P|,|T|)$). The key to building the solution is reconstructing the decisions made at every step along the optimal path that leads to the goal state. These decisions have been recorded in the **parent** field of each array cell.

Reconstructing these decisions is done by walking backward from the goal state, following the **parent** pointer back to an earlier cell. We repeat this process until we arrive back at the initial cell, analogous to how we reconstructed the path found by BFS or Dijkstra’s algorithm. The **parent** field for $m[i][j]$ tells us whether the operation at (i,j) was **MATCH**, **INSERT**, or **DELETE**. Tracing back through the parent matrix in Figure 10.6 yields the edit sequence **DSMMMMISMS** from “thou_shalt” to “you_should”—meaning delete the first “t”; replace the “h” with “y”; match the next five characters before inserting an “o”; replace “a” with “u”; and finally replace the “t” with a “d”.

Walking backward reconstructs the solution in reverse order. However, clever use of recursion can do the reversing for us:

```
void reconstruct_path(char *s, char *t, int i, int j,
                     cell m[MAXLEN+1][MAXLEN+1]) {
    if (m[i][j].parent == -1) {
        return;
    }

    if (m[i][j].parent == MATCH) {
        reconstruct_path(s, t, i-1, j-1, m);
    }
}
```

<i>P</i>	<i>T</i>											
	pos	0	y	o	u	-	s	h	o	u	l	d
	0	<u>-1</u>	1	1	1	1	1	1	1	1	1	1
t:	1	<u>2</u>	0	0	0	0	0	0	0	0	0	0
h:	2	2	<u>0</u>	0	0	0	0	0	1	1	1	1
o:	3	2	0	<u>0</u>	0	0	0	0	0	1	1	1
u:	4	2	0	2	<u>0</u>	1	1	1	1	0	1	1
-:	5	2	0	2	2	<u>0</u>	1	1	1	1	0	0
s:	6	2	0	2	2	2	<u>0</u>	1	1	1	1	0
h:	7	2	0	2	2	2	2	<u>0</u>	<u>1</u>	1	1	1
a:	8	2	0	2	2	2	2	2	0	<u>0</u>	0	0
l:	9	2	0	2	2	2	2	2	0	0	<u>0</u>	1
t:	10	2	0	2	2	2	2	2	0	0	0	<u>0</u>

Figure 10.6: Parent matrix for edit distance computation, with the optimal alignment path underlined to highlight. Again, blue values denote insertions, green values deletions, and red values match/substitution.

```

    match_out(s, t, i, j);
    return;
}

if (m[i][j].parent == INSERT) {
    reconstruct_path(s, t, i, j-1, m);
    insert_out(t, j);
    return;
}

if (m[i][j].parent == DELETE) {
    reconstruct_path(s, t, i-1, j, m);
    delete_out(s, i);
    return;
}
}

```

For many problems, including edit distance, the solution can be reconstructed from the cost matrix without explicitly retaining the last-move array. In edit distance, the trick is working backward from the costs of the three possible ancestor cells and corresponding string characters to reconstruct the move that took you to the current cell at the given cost. But it is cleaner and easier to explicitly store the moves.

10.2.4 Varieties of Edit Distance

The `string_compare` and path reconstruction routines reference several functions that we have not yet defined. These fall into four categories:

- *Table initialization* – The functions `row_init` and `column_init` initialize the zeroth row and column of the dynamic programming table, respectively. For the string edit distance problem, cells $(i, 0)$ and $(0, i)$ correspond to matching length- i strings against the empty string. This requires exactly i insertions/deletions, so the definition of these functions is clear:

```
row_init(int i)                column_init(int i)
{                               {
    m[0][i].cost = i;           m[i][0].cost = i;
    if (i>0)                    if (i>0)
        m[0][i].parent = INSERT;    m[i][0].parent = DELETE;
    else
        m[0][i].parent = -1;       m[i][0].parent = -1;
}                               }
```

- *Penalty costs* – The functions `match(c,d)` and `indel(c)` present the costs for transforming character c to d and inserting/deleting character c . For standard edit distance, `match` should cost 0 if the characters are identical, and 1 otherwise; while `indel` returns 1 regardless of what the argument is. But application-specific cost functions can be employed, perhaps with substitution more forgiving for characters located near each other on standard keyboard layouts or those that sound or look similar.

```
int match(char c, char d)      int indel(char c)
{                               {
    if (c == d) return(0);      return(1);
    else return(1);            }
}
```

- *Goal cell identification* – The function `goal_cell` returns the indices of the cell marking the endpoint of the solution. For edit distance, this is always defined by the length of the two input strings. However, other applications we will soon encounter do not have fixed goal locations.

```
void goal_cell(char *s, char *t, int *i, int *j) {
    *i = strlen(s) - 1;
    *j = strlen(t) - 1;
}
```

- *Traceback actions* – The functions `match_out`, `insert_out`, and `delete_out` perform the appropriate actions for each edit operation during traceback.

For edit distance, this might mean printing out the name of the operation or character involved, as determined by the needs of the application.

```

insert_out(char *t, int j)      match_out(char *s, char *t,
{                               int i, int j)
    printf("I");               {
                                if (s[i]==t[j]) printf("M");
                                else printf("S");
                                }
delete_out(char *s, int i)      }
{
    printf("D");
}

```

All of these functions are quite simple for edit distance computation. However, we must confess it is difficult to get the boundary conditions and index manipulations correct. Although dynamic programming algorithms are easy to design once you understand the technique, getting the details right requires clear thinking and thorough testing.

This may seem like a lot of infrastructure to develop for such a simple algorithm. However, several important problems can be solved as special cases of edit distance using only minor changes to some of these stub functions:

- *Substring matching* – Suppose we want to find where a short pattern P best occurs within a long text T —say searching for “Skiena” in all its misspellings (Skienna, Skena, Skina, ...) within a long file. Plugging this search into our original edit distance function will achieve little sensitivity, since the vast majority of any edit cost will consist of deleting all that is not “Skiena” from the body of the text. Indeed, matching any scattered ... $S \dots k \dots i \dots e \dots n \dots a \dots$ and deleting the rest will yield an optimal solution.

We want an edit distance search where the cost of starting the match is independent of the position in the text, so that we are not prejudiced against a match that starts in the middle of the text. Now the goal state is not necessarily at the end of both strings, but the cheapest place to match the entire pattern somewhere in the text. Modifying these two functions gives us the correct solution:

```

void row_init(int i, cell m[MAXLEN+1][MAXLEN+1]) {
    m[0][i].cost = 0;           /* NOTE CHANGE */
    m[0][i].parent = -1;        /* NOTE CHANGE */
}

```

```

void goal_cell(char *s, char *t, int *i, int *j) {
    int k;    /* counter */

    *i = strlen(s) - 1;
    *j = 0;

    for (k = 1; k < strlen(t); k++) {
        if (m[*i][k].cost < m[*i][*j].cost) {
            *j = k;
        }
    }
}

```

- *Longest common subsequence* – Perhaps we are interested in finding the longest scattered string of characters included within both strings, without changing their relative order. Indeed, this problem will be discussed in Section 21.8. Do Democrats and Republicans have anything in common? Certainly! The *longest common subsequence* (LCS) between “democrats” and “republicans” is *ecas*.

A common subsequence is defined by all the identical-character matches in an edit trace. To maximize the number of such matches, we must prevent substitution of non-identical characters. With substitution forbidden, the only way to get rid of the non-common subsequence will be through insertion and deletion. The minimum cost alignment has the fewest such “in-dels,” so it must preserve the longest common substring. We get the alignment we want by changing the match-cost function to make substitutions expensive:

```

int match(char c, char d) {
    if (c == d) {
        return(0);
    }
    return(MAXLEN);
}

```

Actually, it suffices to make the substitution penalty greater than that of an insertion plus a deletion for substitution to lose any allure as a possible edit operation.

- *Maximum monotone subsequence* – A numerical sequence is *monotonically increasing* if the i th element is at least as big as the $(i - 1)$ st element. The *maximum monotone subsequence* problem seeks to delete the fewest number of elements from an input string S to leave a monotonically increasing subsequence. A maximum monotone subsequence of 243517698 is 23568.

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

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