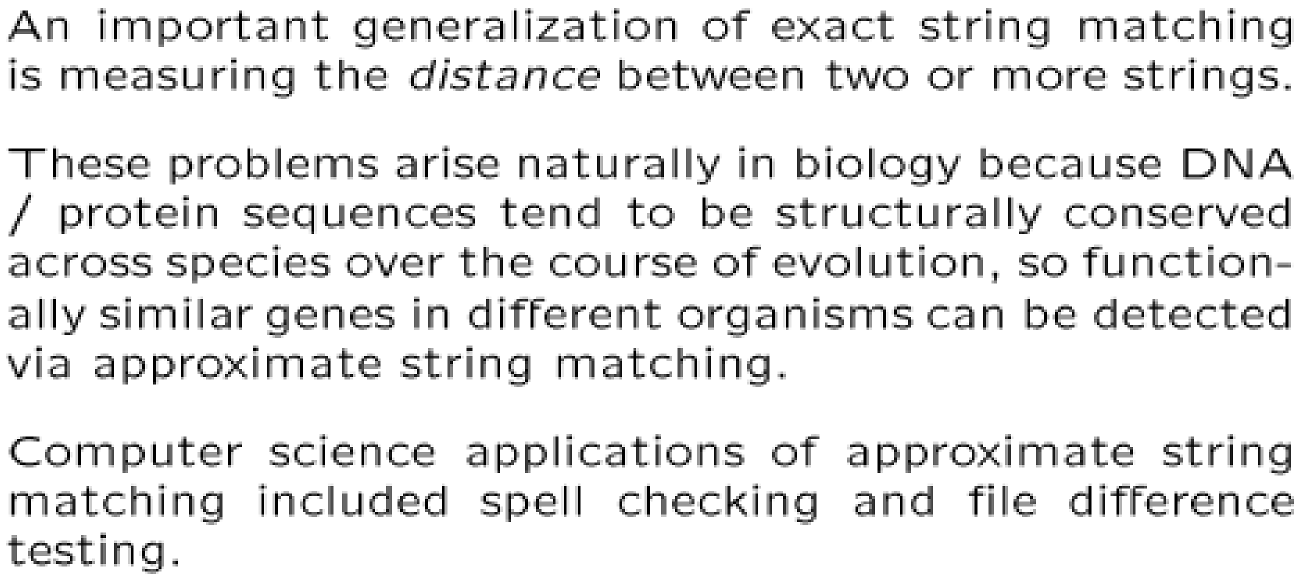
Approximate String Matching

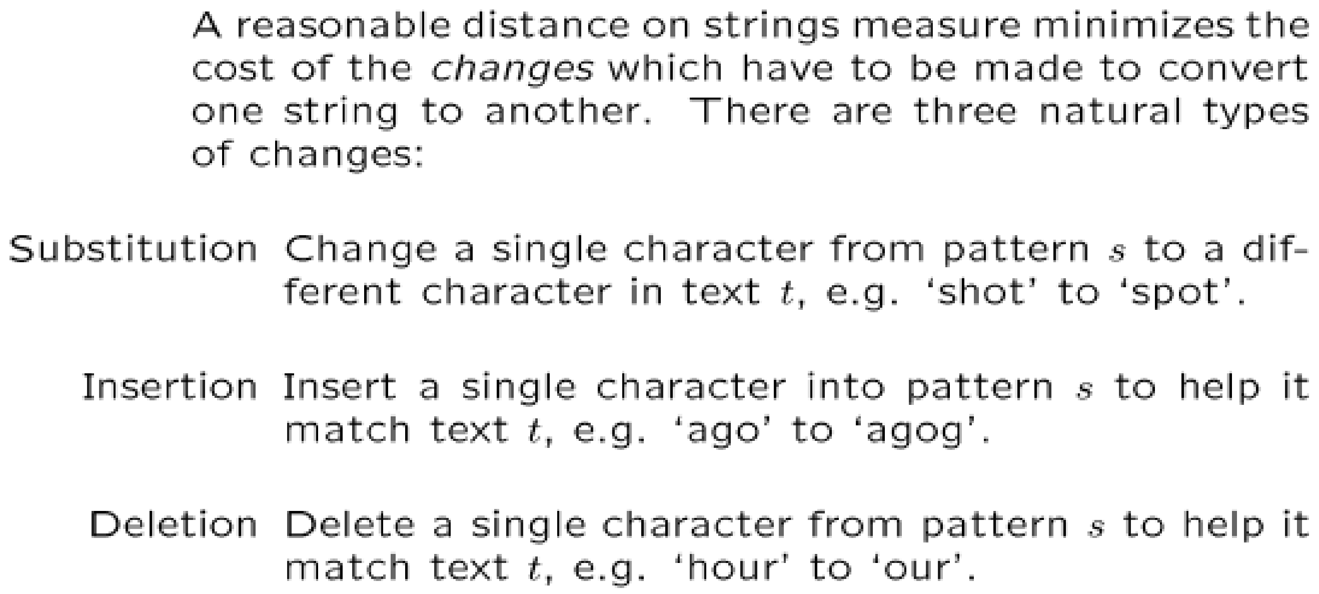


An important generalization of exact string matching is measuring the *distance* between two or more strings.

These problems arise naturally in biology because DNA / protein sequences tend to be structurally conserved across species over the course of evolution, so function-ally similar genes in different organisms can be detected via approximate string matching.

Computer science applications of approximate string matching included spell checking and file difference testing.

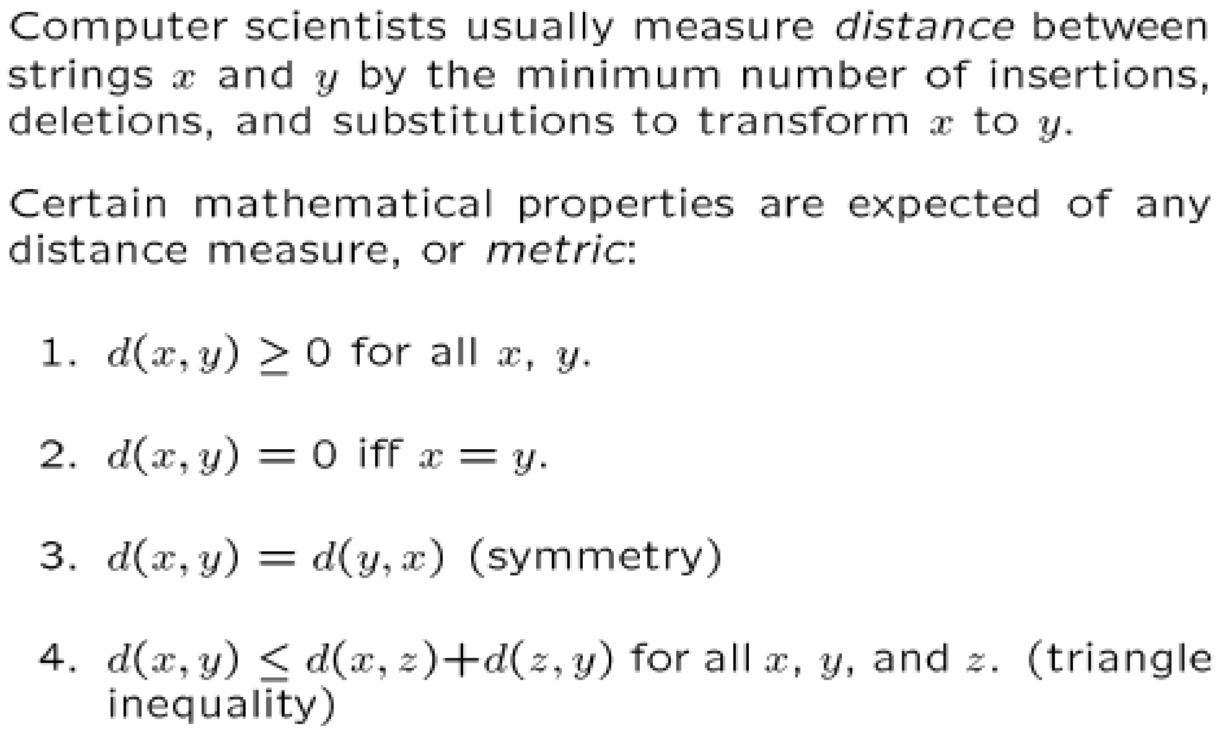
Approximate String Matching



A reasonable distance on strings measure minimizes the cost of the *changes* which have to be made to convert one string to another. There are three natural types of changes:

Substitution	Change a single character from pattern s to a different character in text t, e.g. ‘shot’ to ‘spot’.
Insertion	Insert a single character into pattern s to help it match text t, e.g. ‘ago’ to ‘agog’.
Deletion	Delete a single character from pattern s to help it match text t, e.g. ‘hour’ to ‘our’.

Edit Distance and Similarity Scores



Computer scientists usually measure *distance* between strings x and y by the minimum number of insertions, deletions, and substitutions to transform x to y.

Certain mathematical properties are expected of any distance measure, or *metric*:

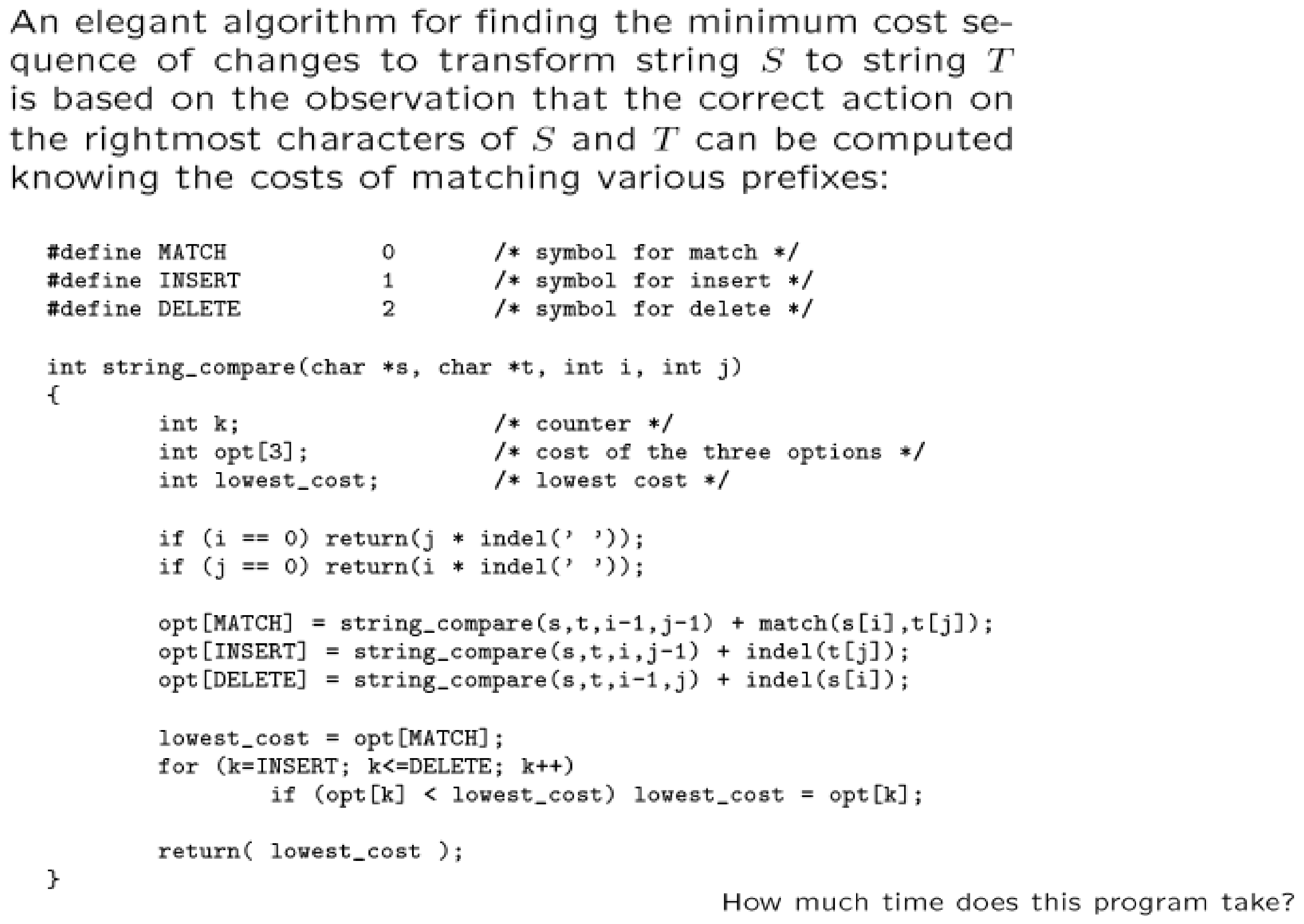
1. $d(x,y) \geq 0$ for all x, y.
2. $d(x,y) = 0$ iff $x = y$.
3. $d(x,y) = d(y,x)$ (symmetry)
4. $d(x,y) \leq d(x,z) + d(z,y)$ for all x, y, and z. (triangle inequality)

Edit Distance and Similarity Scores

Biologists typically instead measure a sequence *similarity score* which gets larger the more similar the sequences are.

Similar algorithms can be used to optimize both measures.

Pairwise string alignment



An elegant algorithm for finding the minimum cost sequence of changes to transform string S to string T is based on the observation that the correct action on the rightmost characters of S and T can be computed knowing the costs of matching various prefixes:

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

int string_compare(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) return(j * indel(' '));
        if (j == 0) return(i * indel(' '));

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

How much time does this program take?

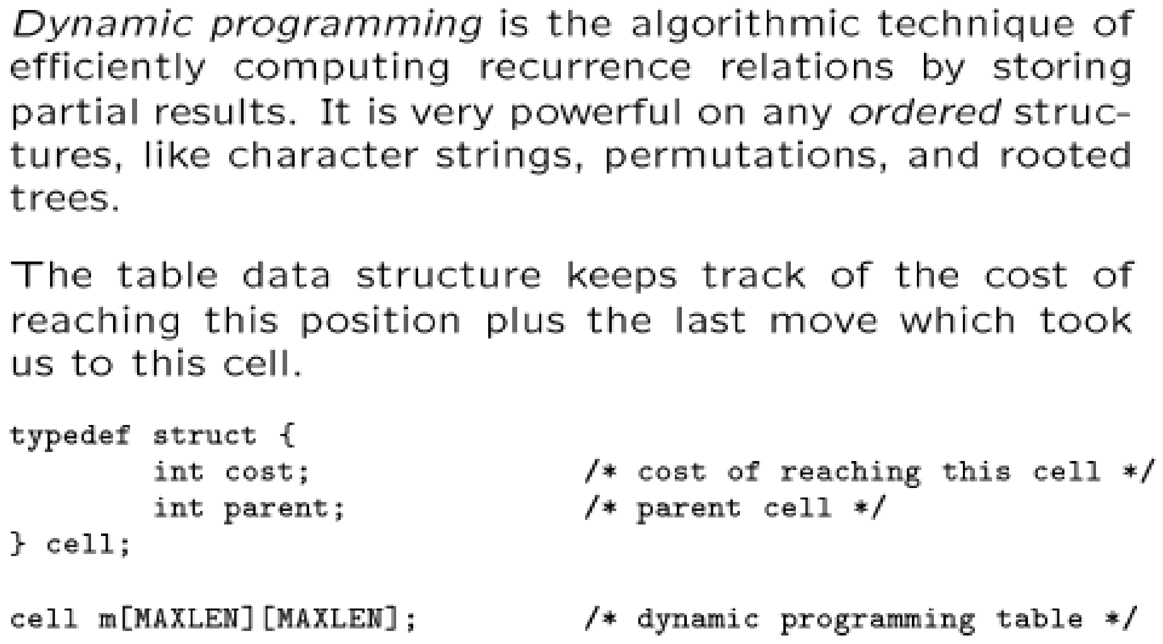
Dynamic Programming

Making this tractible requires realizing that we are repeatedly performing the same computations on each pair of prefixes.

The entire state of the recursive call is governed by the index positions into the strings. Thus there are only $|S| \times |T|$ different calls.

By storing the answers in a table and looking them up instead of recomputing, the algorithm takes quadratic time.

Dynamic Programming

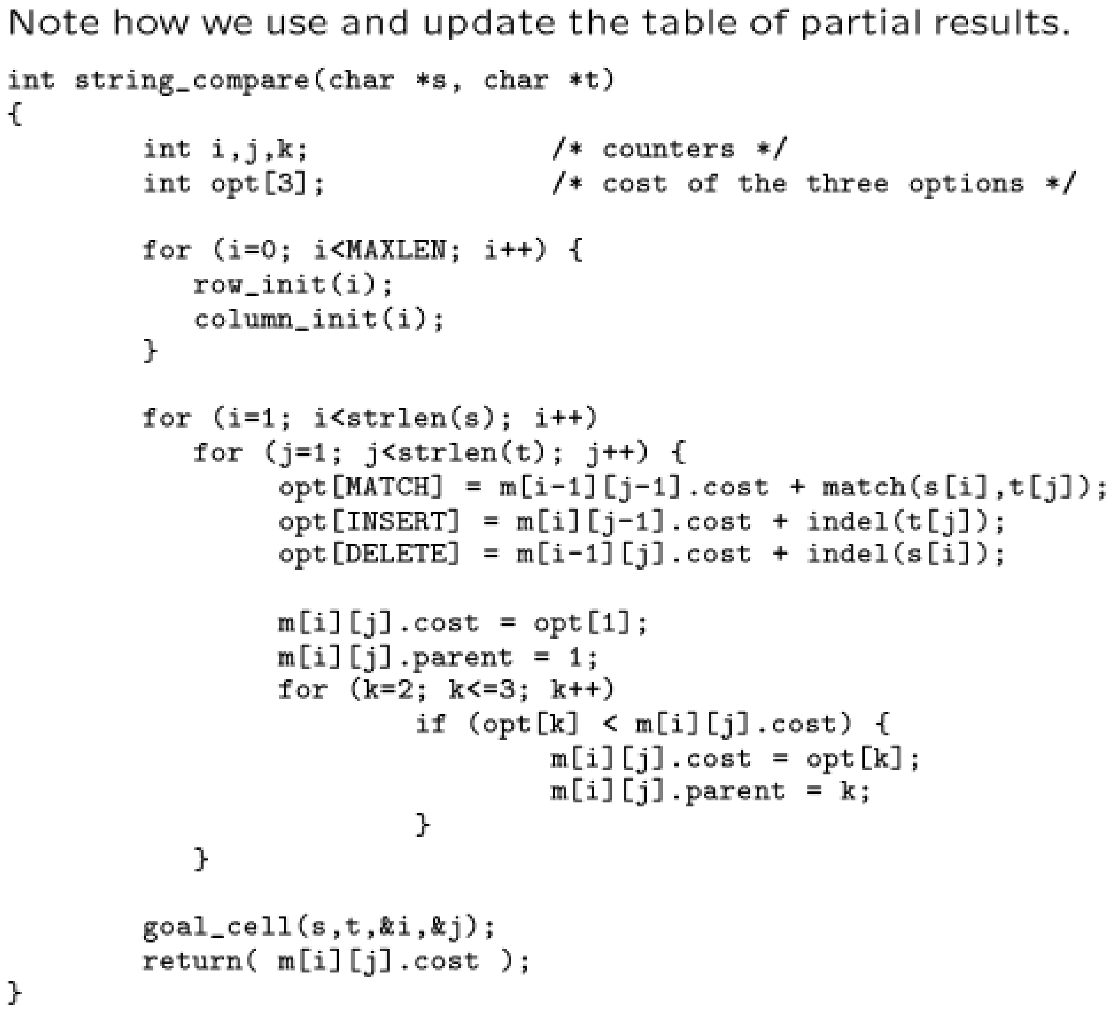


Dynamic programming is the algorithmic technique of efficiently computing recurrence relations by storing partial results. It is very powerful on any *ordered* structures, like character strings, permutations, and rooted trees.

The table data structure keeps track of the cost of reaching this position plus the last move which took us to this cell.

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

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

General edit distance using dynamic programming



Note how we use and update the table of partial results.

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

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

        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[1];
                m[i][j].parent = 1;
                for (k=2; k<=3; 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 );
}
```

General edit distance using dynamic programming

To determine the value of cell (i, j), we need the the cells $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$. Any evaluation order with this property will do, including the row-major order we used.

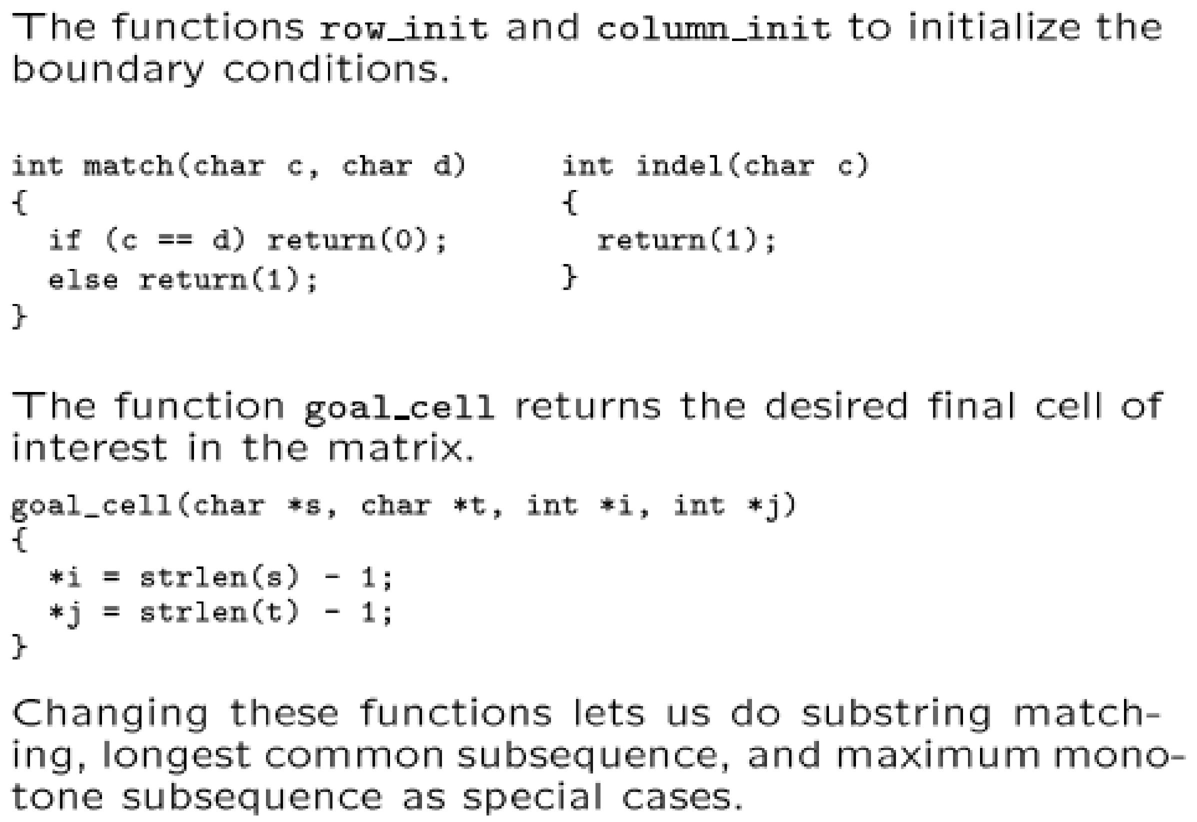
The function `string_compare` is very general, and must be customized to a particular application.

It uses problem-specific subroutines `match` and `indel` to return the costs of character pair transitions:

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

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

General edit distance using dynamic programming



The functions `row_init` and `column_init` to initialize the boundary conditions.

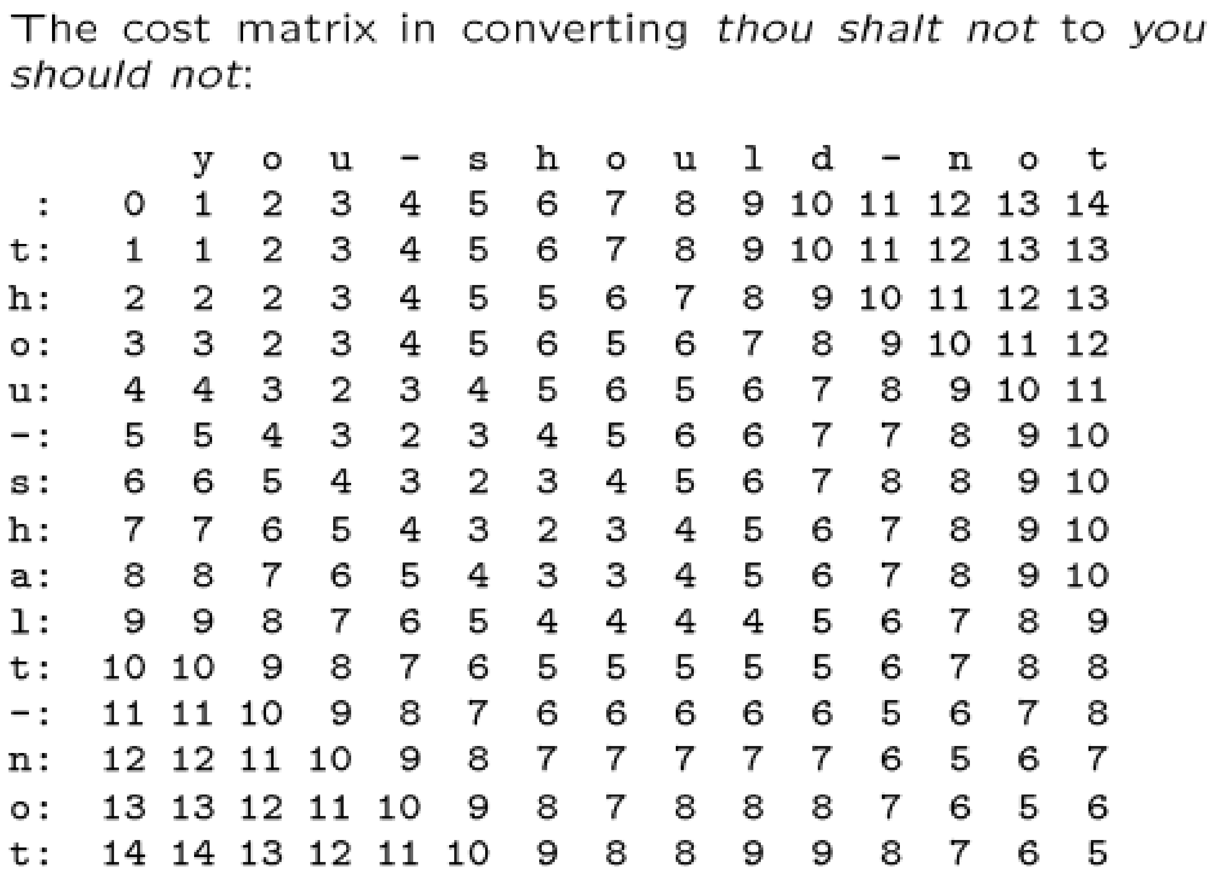
```
int match(char c, char d)       int indel(char c)
{                               {
  if (c == d) return(0);          return(1);
  else return(1);               }
}
```

The function `goal_cell` returns the desired final cell of interest in the matrix.

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

Changing these functions lets us do substring matching, longest common subsequence, and maximum monotone subsequence as special cases.

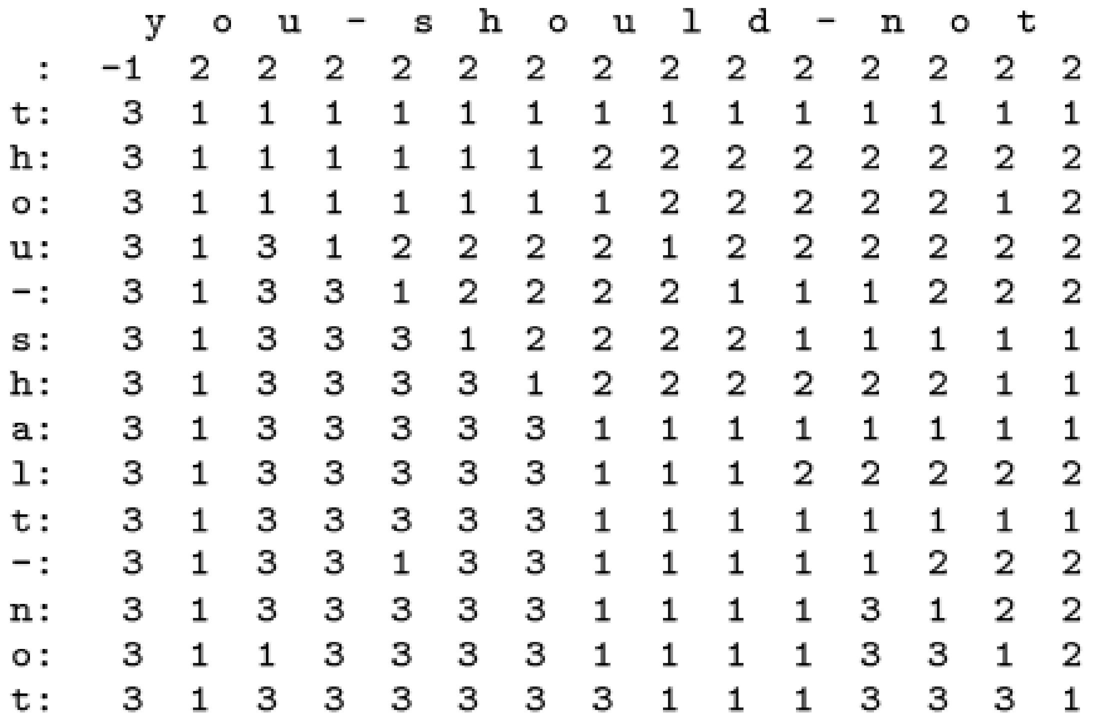
String Matching example: Cost matrix



The cost matrix in converting *thou shalt not* to *you should not*:

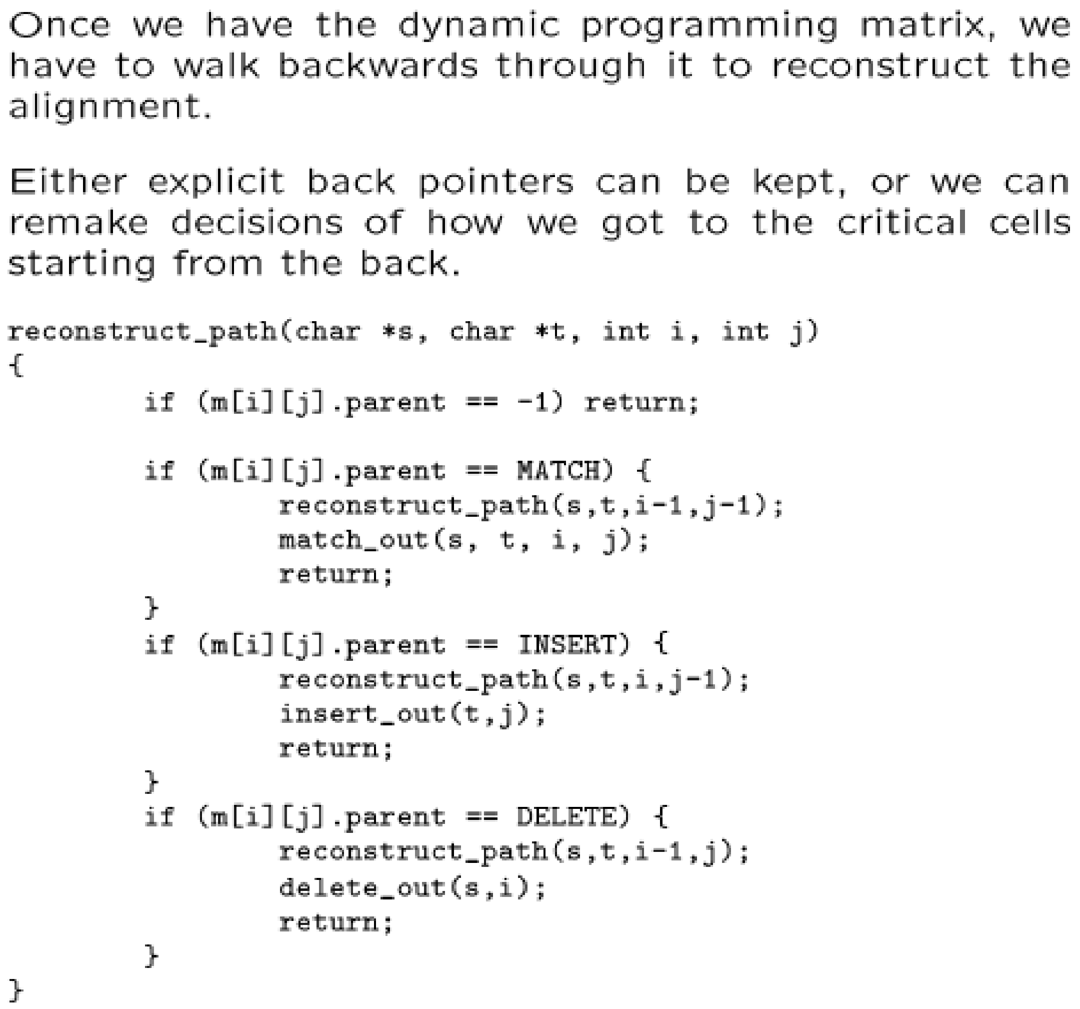
		y	o	u	-	s	h	o	u	l	d	-	n	o	t
:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	1	2	3	4	5	6	7	8	9	10	11	12	13	13
h:	2	2	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	2	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	3	2	3	4	5	6	5	6	7	8	9	10	11
-:	5	5	4	3	2	3	4	5	6	6	7	7	8	9	10
s:	6	6	5	4	3	2	3	4	5	6	7	8	8	9	10
h:	7	7	6	5	4	3	2	3	4	5	6	7	8	9	10
a:	8	8	7	6	5	4	3	3	4	5	6	7	8	9	10
l:	9	9	8	7	6	5	4	4	4	4	5	6	7	8	9
t:	10	10	9	8	7	6	5	5	5	5	5	6	7	8	8
-:	11	11	10	9	8	7	6	6	6	6	6	5	6	7	8
n:	12	12	11	10	9	8	7	7	7	7	7	6	5	6	7
o:	13	13	12	11	10	9	8	7	8	8	8	7	6	5	6
t:	14	14	13	12	11	10	9	8	8	9	9	8	7	6	5

String Matching example: Parent matrix



		y	o	u	-	s	h	o	u	l	d	-	n	o	t
:	-1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
t:	3	1	1	1	1	1	1	1	1	1	1	1	1	1	1
h:	3	1	1	1	1	1	1	2	2	2	2	2	2	2	2
o:	3	1	1	1	1	1	1	1	2	2	2	2	2	1	2
u:	3	1	3	1	2	2	2	2	1	2	2	2	2	2	2
-:	3	1	3	3	1	2	2	2	2	1	1	1	2	2	2
s:	3	1	3	3	3	1	2	2	2	2	1	1	1	1	1
h:	3	1	3	3	3	3	1	2	2	2	2	2	2	1	1
a:	3	1	3	3	3	3	3	1	1	1	1	1	1	1	1
l:	3	1	3	3	3	3	3	1	1	1	2	2	2	2	2
t:	3	1	3	3	3	3	3	1	1	1	1	1	1	1	1
-:	3	1	3	3	1	3	3	1	1	1	1	1	2	2	2
n:	3	1	3	3	3	3	3	1	1	1	1	3	1	2	2
o:	3	1	1	3	3	3	3	1	1	1	1	3	3	1	2
t:	3	1	3	3	3	3	3	3	1	1	1	3	3	3	1

Reconstructing the alignment

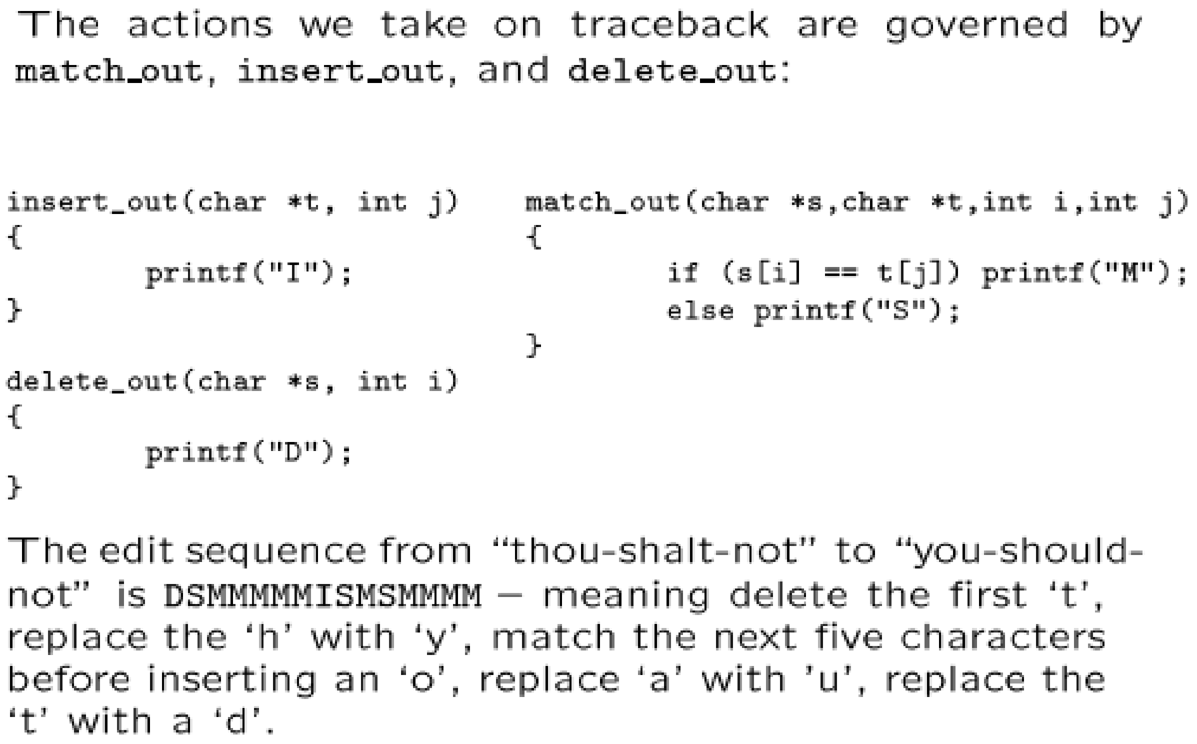


Once we have the dynamic programming matrix, we have to walk backwards through it to reconstruct the alignment.

Either explicit back pointers can be kept, or we can remake decisions of how we got to the critical cells starting from the back.

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

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

Reconstructing the alignment



The actions we take on traceback are governed by **match_out**, **insert_out**, and **delete_out**:

```
insert_out(char *t, int j)
{
        printf("I");
}

delete_out(char *s, int i)
{
        printf("D");
}
```

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

The edit sequence from "thou-shalt-not" to "you-should-not" is DSMMMMMISMSMMMM – meaning delete the first 't', replace the 'h' with 'y', match the next five characters before inserting an 'o', replace 'a' with 'u', replace the 't' with a 'd'.

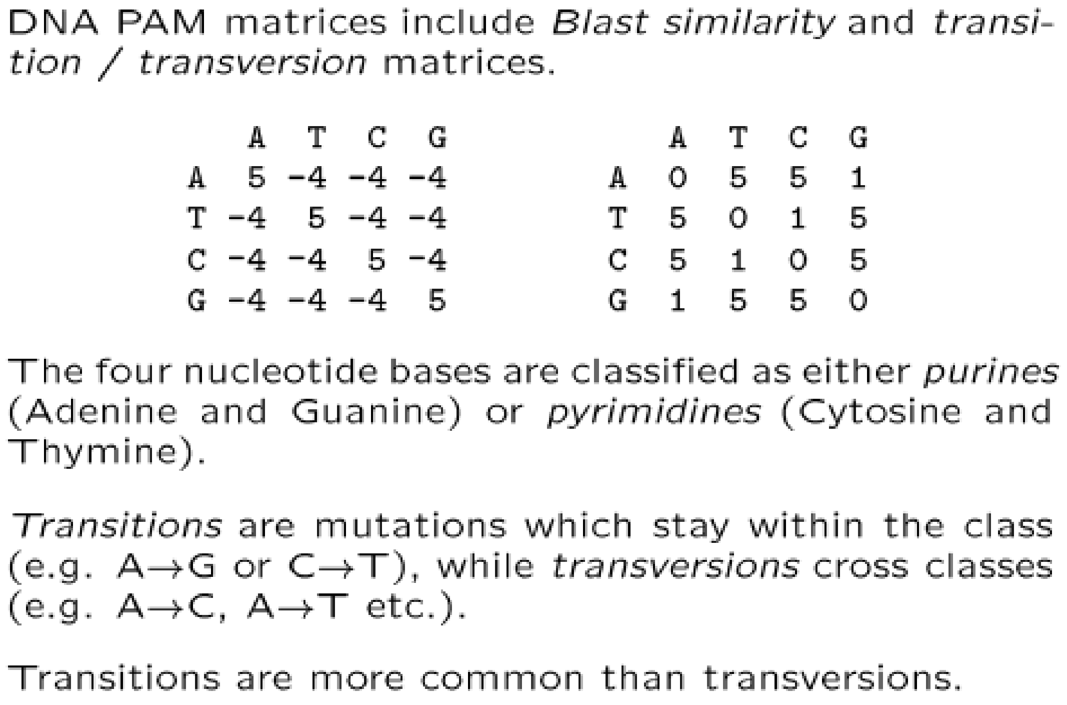
Biological sequence comparison

Constructing a meaningful alignment of two sequences requires using a appropriate function to measure the cost of changing between each possible pair symbols.

In correcting text entered by a fast typist, we might penalize pairs of symbols near each other on the keyboard less than those on different sides, for example.

For genomic sequences, the weights/scores governing the cost of changing between bases are given by *PAM matrices* for “point accepted mutations”

Biological sequence comparison



DNA PAM matrices include *Blast similarity* and *transition / transversion* matrices.

	A	T	C	G
A	5	-4	-4	-4
T	-4	5	-4	-4
C	-4	-4	5	-4
G	-4	-4	-4	5

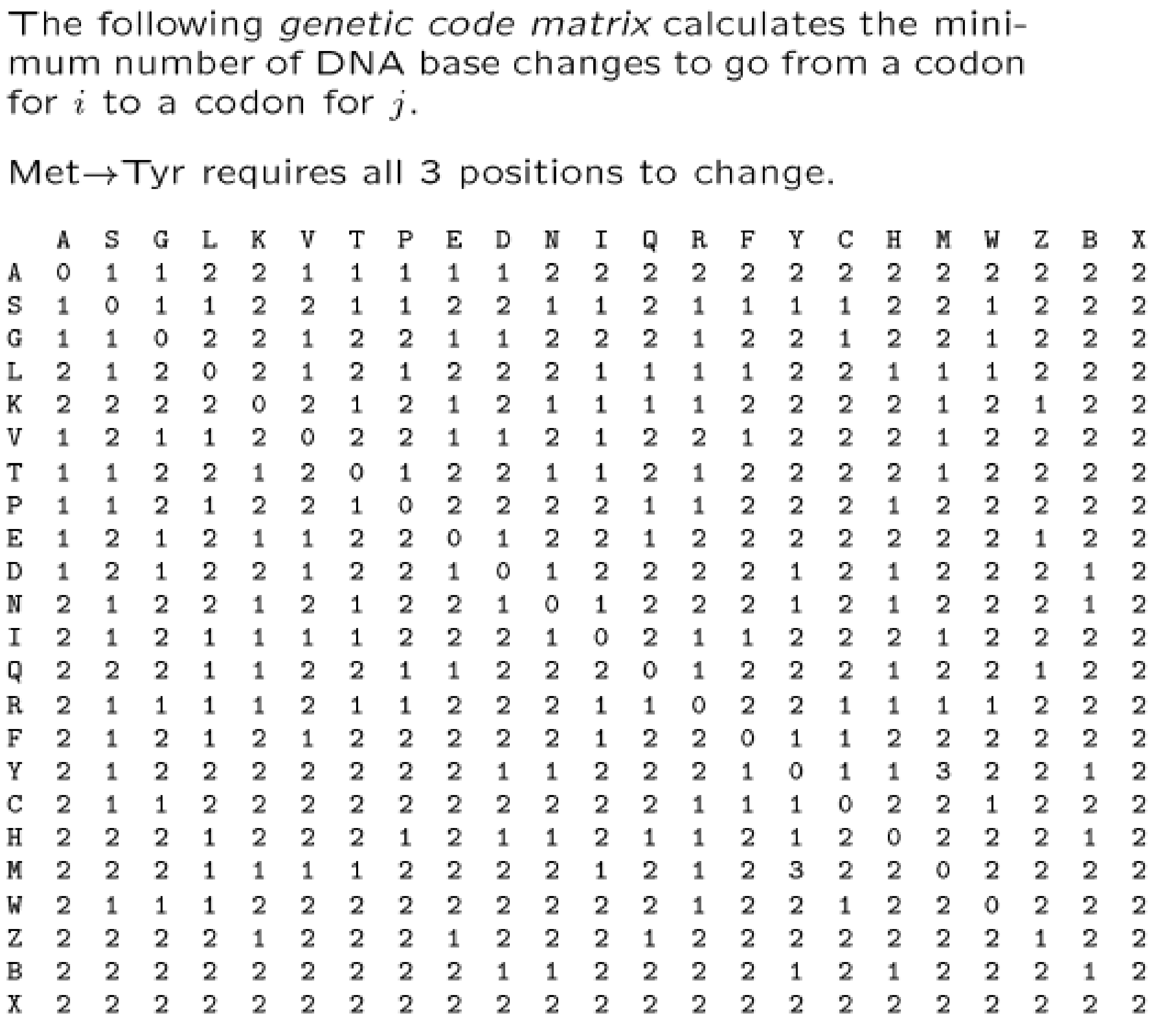
	A	T	C	G
A	0	5	5	1
T	5	0	1	5
C	5	1	0	5
G	1	5	5	0

The four nucleotide bases are classified as either *purines* (Adenine and Guanine) or *pyrimidines* (Cytosine and Thymine).

Transitions are mutations which stay within the class (e.g. A→G or C→T), while *transversions* cross classes (e.g. A→C, A→T etc.).

Transitions are more common than transversions.

Genetic code matrix

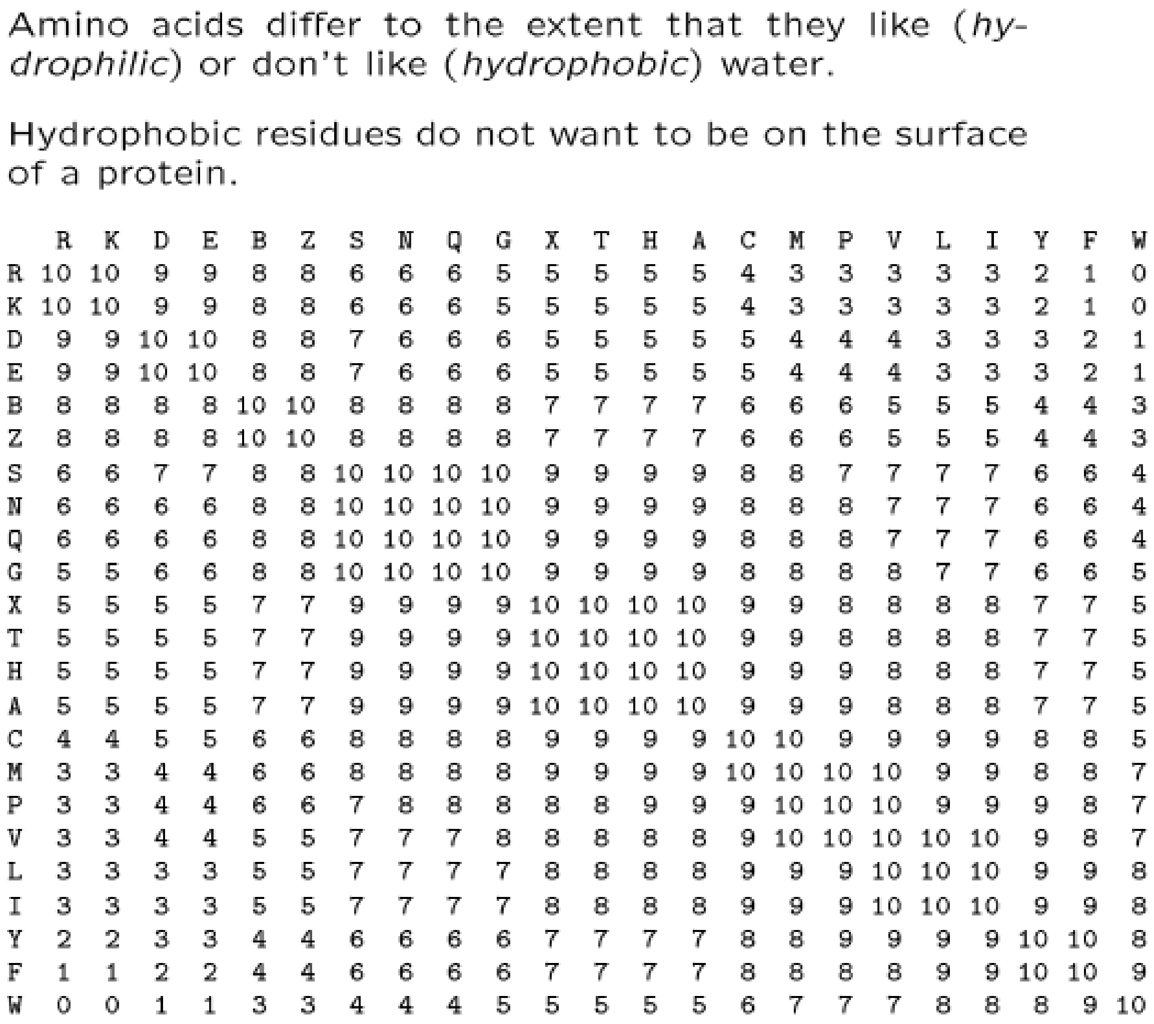


The following *genetic code matrix* calculates the minimum number of DNA base changes to go from a codon for i to a codon for j.

Met→Tyr requires all 3 positions to change.

	A	S	G	L	K	V	T	P	E	D	N	I	Q	R	F	Y	C	H	M	W	Z	B	X
A	0	1	1	2	2	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2
S	1	0	1	1	2	2	1	1	2	2	1	1	2	1	1	1	1	2	2	1	2	2	2
G	1	1	0	2	2	1	2	2	1	1	2	2	2	1	2	2	1	2	2	1	2	2	2
L	2	1	2	0	2	1	2	1	2	2	2	1	1	1	1	2	2	1	1	1	2	2	2
K	2	2	2	2	0	2	1	2	1	2	1	1	1	1	2	2	2	2	1	2	1	2	2
V	1	2	1	1	2	0	2	2	1	1	2	1	2	2	1	2	2	2	1	2	2	2	2
T	1	1	2	2	1	2	0	1	2	2	1	1	2	1	2	2	2	2	1	2	2	2	2
P	1	1	2	1	2	2	1	0	2	2	2	2	1	1	2	2	2	1	2	2	2	2	2
E	1	2	1	2	1	1	2	2	0	1	2	2	1	2	2	2	2	2	2	2	1	2	2
D	1	2	1	2	2	1	2	2	1	0	1	2	2	2	2	1	2	1	2	2	2	1	2
N	2	1	2	2	1	2	1	2	2	1	0	1	2	2	2	1	2	1	2	2	2	1	2
I	2	1	2	1	1	1	1	2	2	2	1	0	2	1	1	2	2	2	1	2	2	2	2
Q	2	2	2	1	1	2	2	1	1	2	2	2	0	1	2	2	2	1	2	2	1	2	2
R	2	1	1	1	1	2	1	1	2	2	2	1	1	0	2	2	1	1	1	1	2	2	2
F	2	1	2	1	2	1	2	2	2	2	2	1	2	2	0	1	1	2	2	2	2	2	2
Y	2	1	2	2	2	2	2	2	2	1	1	2	2	2	1	0	1	1	3	2	2	1	2
C	2	1	1	2	2	2	2	2	2	2	2	2	2	1	1	1	0	2	2	1	2	2	2
H	2	2	2	1	2	2	2	1	2	1	1	2	1	1	2	1	2	0	2	2	2	1	2
M	2	2	2	1	1	1	1	2	2	2	2	1	2	1	2	3	2	2	0	2	2	2	2
W	2	1	1	1	2	2	2	2	2	2	2	2	2	1	2	2	1	2	2	0	2	2	2
Z	2	2	2	2	1	2	2	2	1	2	2	2	1	2	2	2	2	2	2	2	1	2	2
B	2	2	2	2	2	2	2	2	2	1	1	2	2	2	2	1	2	1	2	2	2	1	2
X	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

Hydrophobicity matrix



Amino acids differ to the extent that they like (*hydrophilic*) or don't like (*hydrophobic*) water.

Hydrophobic residues do not want to be on the surface of a protein.

	R	K	D	E	B	Z	S	N	Q	G	X	T	H	A	C	M	P	V	L	I	Y	F	W
R	10	10	9	9	8	8	6	6	6	5	5	5	5	5	4	3	3	3	3	3	2	1	0
K	10	10	9	9	8	8	6	6	6	5	5	5	5	5	4	3	3	3	3	3	2	1	0
D	9	9	10	10	8	8	7	6	6	6	5	5	5	5	5	4	4	4	3	3	3	2	1
E	9	9	10	10	8	8	7	6	6	6	5	5	5	5	5	4	4	4	3	3	3	2	1
B	8	8	8	8	10	10	8	8	8	8	7	7	7	7	6	6	6	5	5	5	4	4	3
Z	8	8	8	8	10	10	8	8	8	8	7	7	7	7	6	6	6	5	5	5	4	4	3
S	6	6	7	7	8	8	10	10	10	10	9	9	9	9	8	8	7	7	7	7	6	6	4
N	6	6	6	6	8	8	10	10	10	10	9	9	9	9	8	8	8	7	7	7	6	6	4
Q	6	6	6	6	8	8	10	10	10	10	9	9	9	9	8	8	8	7	7	7	6	6	4
G	5	5	6	6	8	8	10	10	10	10	9	9	9	9	8	8	8	8	7	7	6	6	5
X	5	5	5	5	7	7	9	9	9	9	10	10	10	10	9	9	8	8	8	8	7	7	5
T	5	5	5	5	7	7	9	9	9	9	10	10	10	10	9	9	8	8	8	8	7	7	5
H	5	5	5	5	7	7	9	9	9	9	10	10	10	10	9	9	9	8	8	8	7	7	5
A	5	5	5	5	7	7	9	9	9	9	10	10	10	10	9	9	9	8	8	8	7	7	5
C	4	4	5	5	6	6	8	8	8	8	9	9	9	9	10	10	9	9	9	9	8	8	5
M	3	3	4	4	6	6	8	8	8	8	9	9	9	9	10	10	10	10	9	9	8	8	7
P	3	3	4	4	6	6	7	8	8	8	8	8	9	9	9	10	10	10	9	9	9	8	7
V	3	3	4	4	5	5	7	7	7	8	8	8	8	8	9	10	10	10	10	10	9	8	7
L	3	3	3	3	5	5	7	7	7	7	8	8	8	8	9	9	9	10	10	10	9	9	8
I	3	3	3	3	5	5	7	7	7	7	8	8	8	8	9	9	9	10	10	10	9	9	8
Y	2	2	3	3	4	4	6	6	6	6	7	7	7	7	8	8	9	9	9	9	10	10	8
F	1	1	2	2	4	4	6	6	6	6	7	7	7	7	8	8	8	8	9	9	10	10	9
W	0	0	1	1	3	3	4	4	4	5	5	5	5	5	6	7	7	7	8	8	8	9	10

PAM matrices

This variety of possible matrix criteria make judging the significance of an alignment tricky.

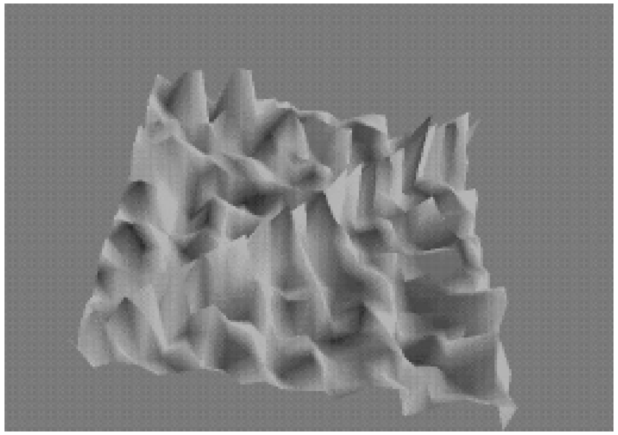
Most widely used are *PAM matrices*, for “point accepted mutations”.

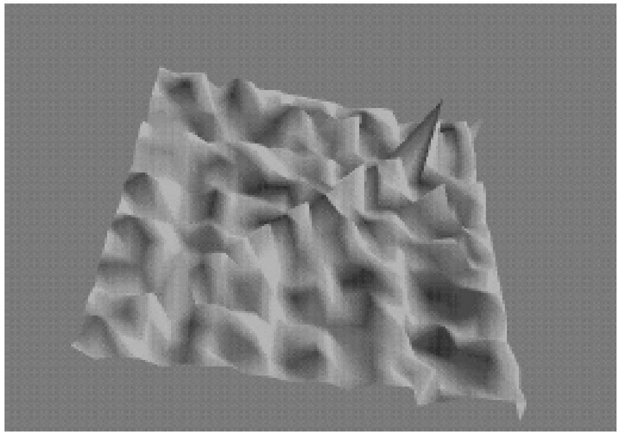
These were constructed by aligning very similar proteins, and tabulating how often each substitution occurred.

The PAM1 matrix scores the transitions when the proteins differ in 1% of the residues.

Raising this to higher powers gives us PAM matrices suited for comparing more distantly related proteins.

PAM matrices





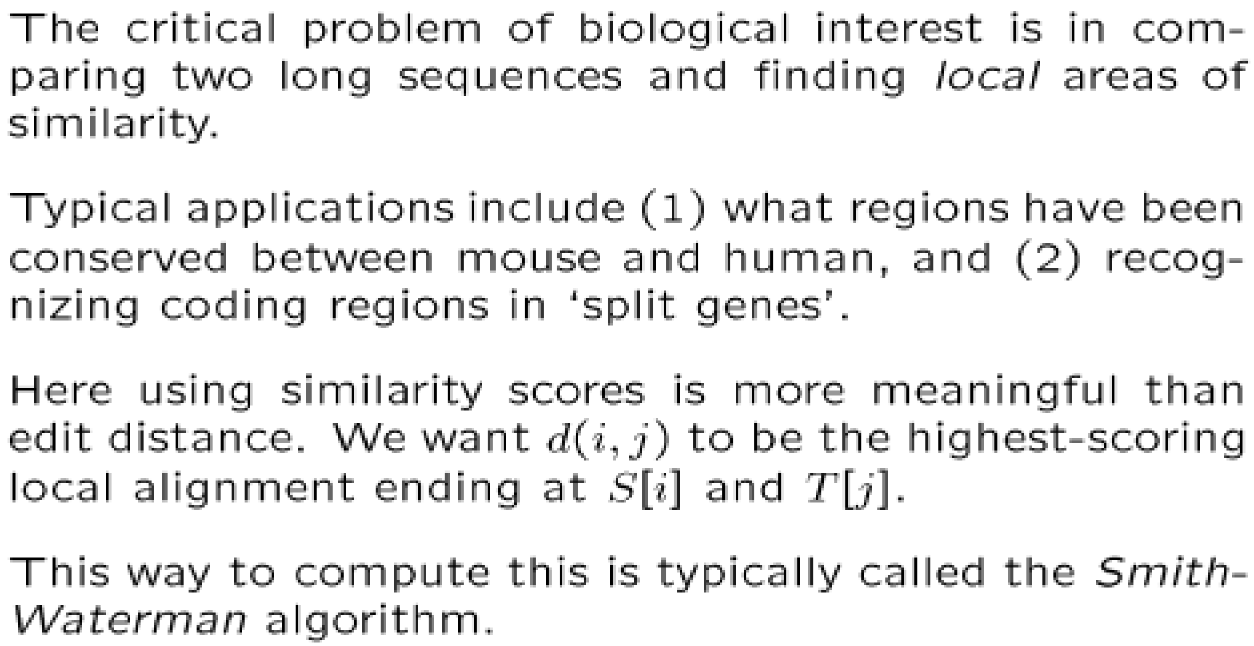
Note the main diagonal on these plots of the PAM50 and PAM250 matrices.

More modern than the PAM matrices are the *Blosum* matrices, reflecting additional sequence alignment data.

Local alignment

The Global Alignment Problem tries to find the longest path between vertices *(0,0)* and (*n,m*) in the edit graph.

The Local Alignment Problem tries to find the longest path among paths between **arbitrary vertices** (*i,j*) and (*i', j'*) in the edit graph.



The critical problem of biological interest is in comparing two long sequences and finding *local* areas of similarity.

Typical applications include (1) what regions have been conserved between mouse and human, and (2) recognizing coding regions in 'split genes'.

Here using similarity scores is more meaningful than edit distance. We want $d(i,j)$ to be the highest-scoring local alignment ending at $S[i]$ and $T[j]$.

This way to compute this is typically called the *Smith-Waterman* algorithm.

Local alignment

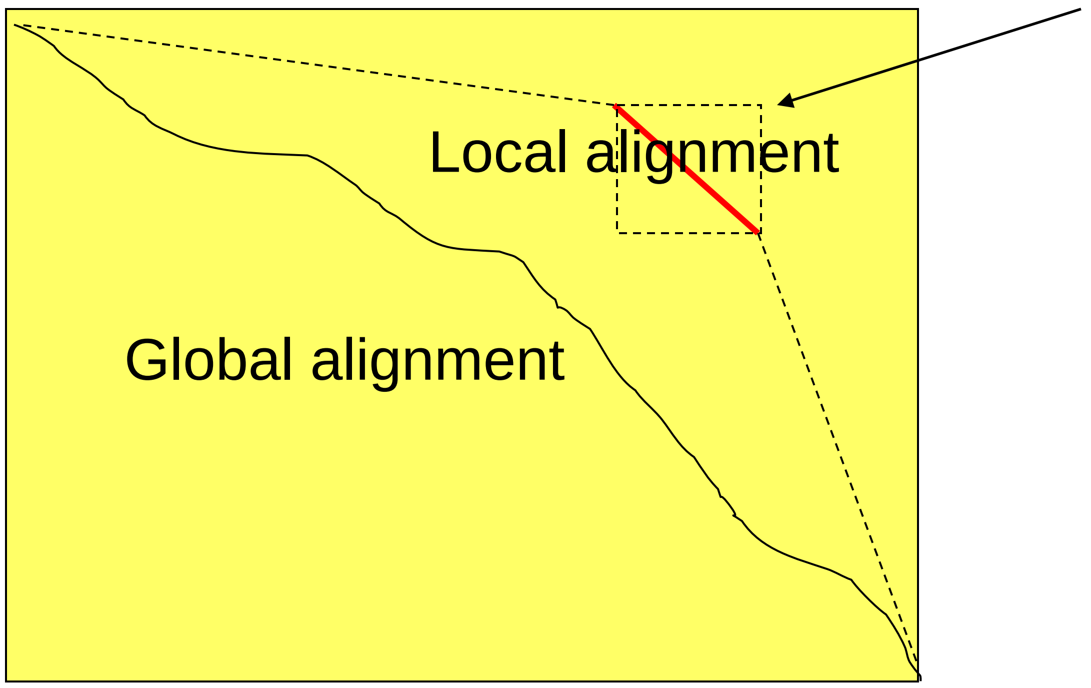
- Global Alignment

```
--T—-CC-C-AGT—-TATGT-CAGGGGACACG—A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || | | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG—T-CAGAT--C
```

- Local Alignment—better alignment to find conserved segment

```
                  tccCAGTTATGTCAGgggacacgagcatgcagagac
                     ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

Local alignment

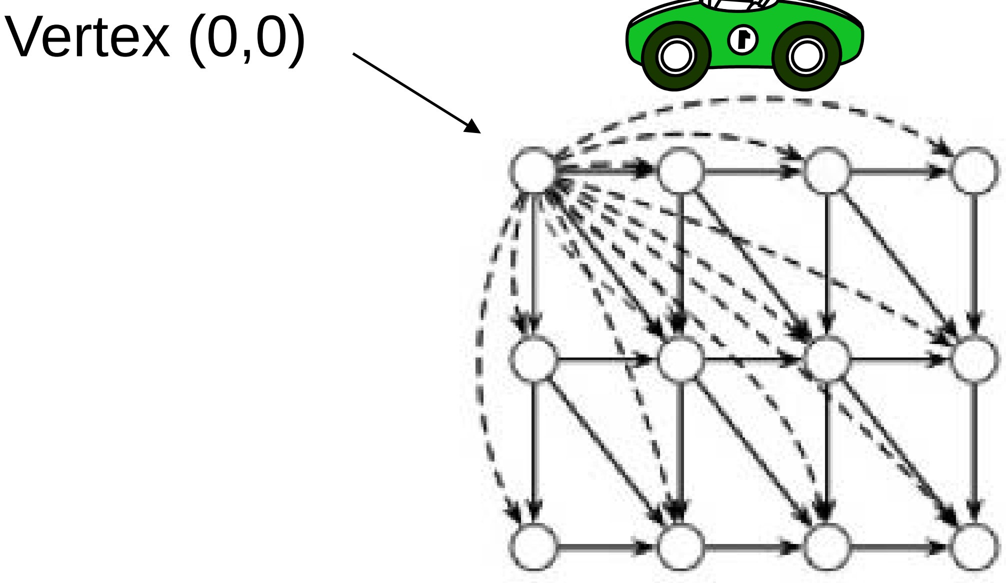


Compute a “mini” Global Alignment to get Local

Local alignment

How much time would it take to compute the best local alignment between two sequences?

Local alignment: Free rides



The dashed edges represent the free rides from (0,0) to every other node.

Local alignment

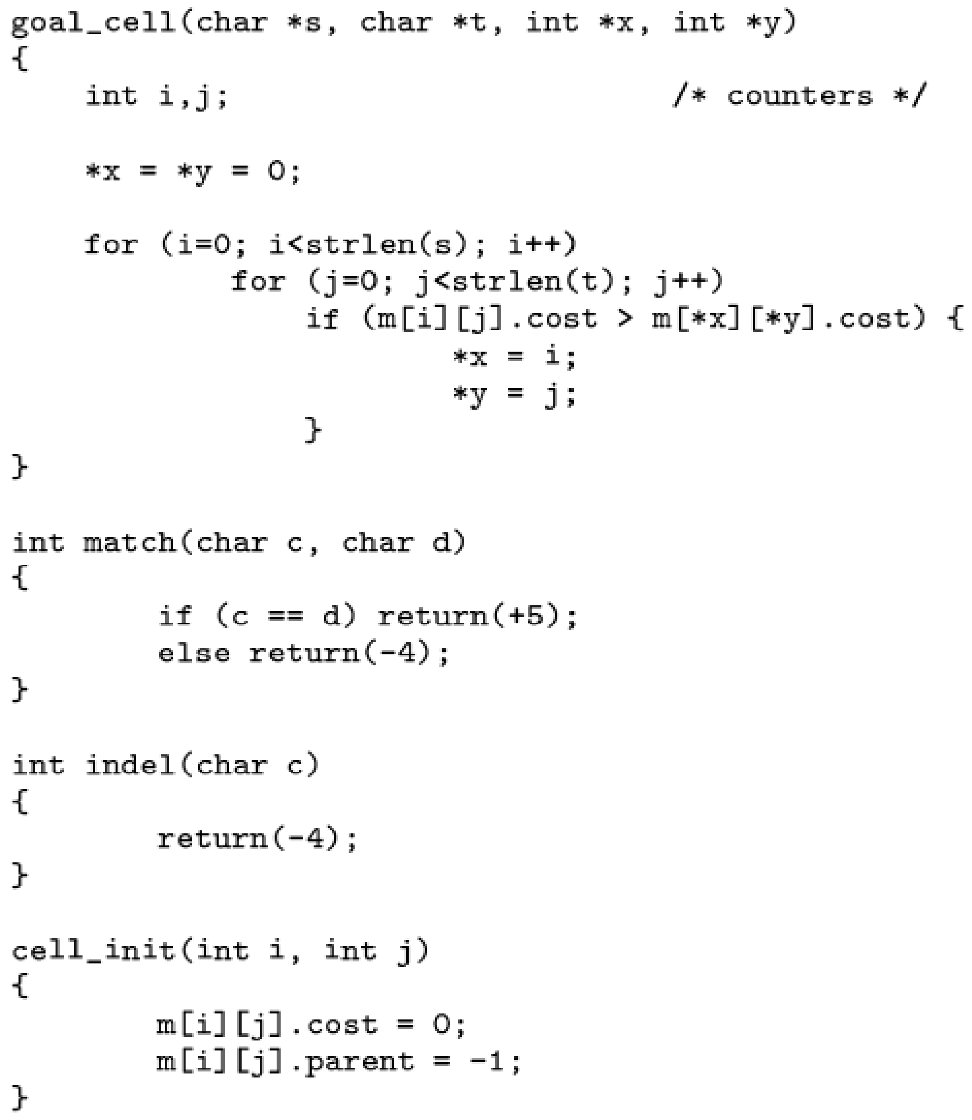
It is the same basic algorithm as for edit distance, except:

- We maximize instead of minimize.
- We have the option of starting our local alignment fresh at each cell in the matrix, i.e. 0 is always an allowable cost.
- We scan all mn cells at the end to see which gives us the best score, not just those along the last row or column.

Smith-Waterman algorithm

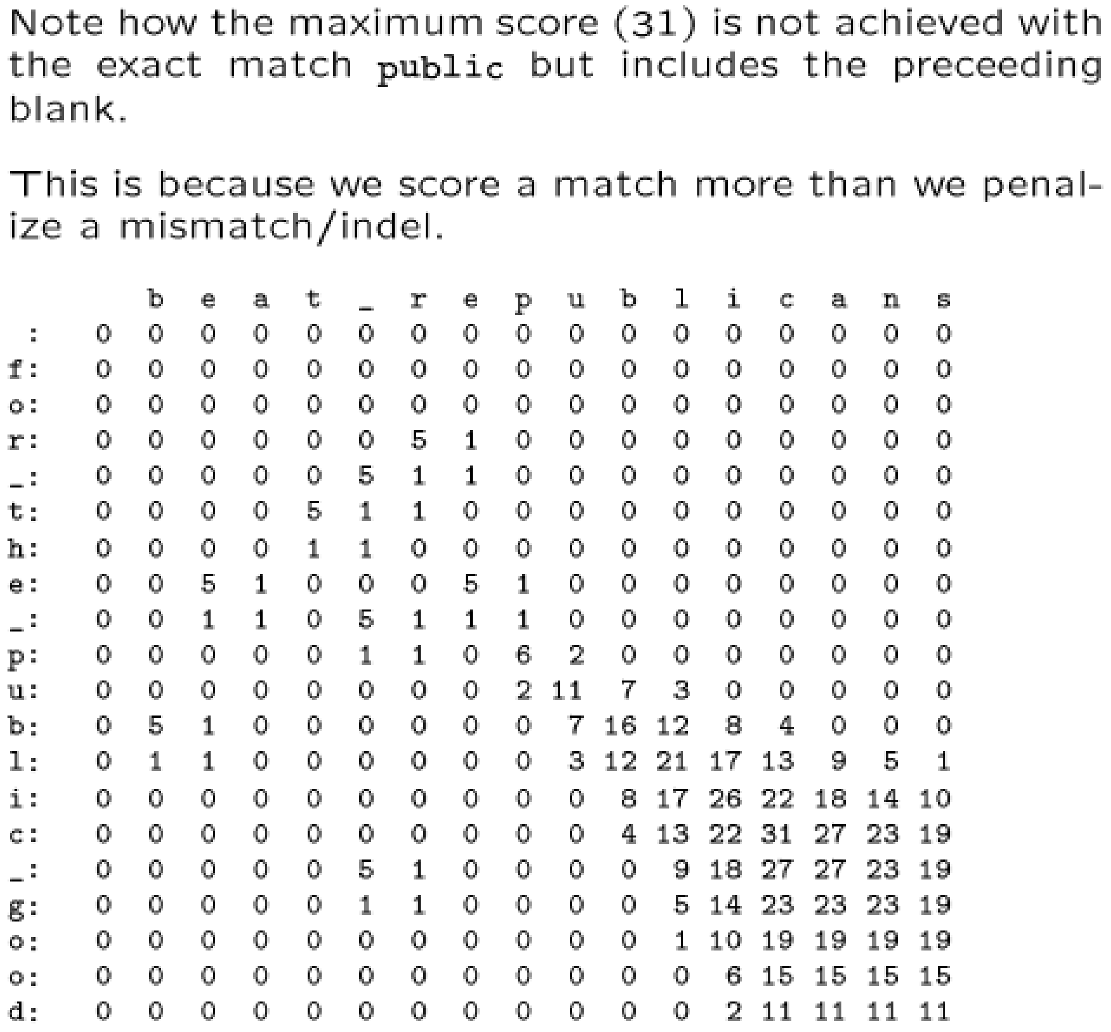
```
int smith_waterman(char *s, char *t)
{
    int i,j,k;                /* counters */
    int opt[3];               /* cost of the three options */
    int match(); int indel();

    for (i=0; i<=strlen(s); i++)
        for (j=0; j<=strlen(t); j++)
            cell_init(i,j);

    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 = 0;
         m[i][j].parent = -1;
         for (k=MATCH; 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 );
}
```

Supporting routines



```
goal_cell(char *s, char *t, int *x, int *y)
{
    int i,j;                                /* counters */

    *x = *y = 0;

    for (i=0; i<strlen(s); i++)
            for (j=0; j<strlen(t); j++)
                if (m[i][j].cost > m[*x][*y].cost) {
                        *x = i;
                        *y = j;
                }
}

int match(char c, char d)
{
        if (c == d) return(+5);
        else return(-4);
}

int indel(char c)
{
        return(-4);
}

cell_init(int i, int j)
{
        m[i][j].cost = 0;
        m[i][j].parent = -1;
}
```

Smith-Waterman example



Note how the maximum score (31) is not achieved with the exact match `public` but includes the preceeding blank.

This is because we score a match more than we penalize a mismatch/indel.

		b	e	a	t	_	r	e	p	u	b	l	i	c	a	n	s
:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
o:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
r:	0	0	0	0	0	0	5	1	0	0	0	0	0	0	0	0	0
_:	0	0	0	0	0	5	1	1	0	0	0	0	0	0	0	0	0
t:	0	0	0	0	5	1	1	0	0	0	0	0	0	0	0	0	0
h:	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
e:	0	0	5	1	0	0	0	5	1	0	0	0	0	0	0	0	0
_:	0	0	1	1	0	5	1	1	1	0	0	0	0	0	0	0	0
p:	0	0	0	0	0	1	1	0	6	2	0	0	0	0	0	0	0
u:	0	0	0	0	0	0	0	0	2	11	7	3	0	0	0	0	0
b:	0	5	1	0	0	0	0	0	0	7	16	12	8	4	0	0	0
l:	0	1	1	0	0	0	0	0	0	3	12	21	17	13	9	5	1
i:	0	0	0	0	0	0	0	0	0	0	8	17	26	22	18	14	10
c:	0	0	0	0	0	0	0	0	0	0	4	13	22	31	27	23	19
_:	0	0	0	0	0	5	1	0	0	0	0	9	18	27	27	23	19
g:	0	0	0	0	0	1	1	0	0	0	0	5	14	23	23	23	19
o:	0	0	0	0	0	0	0	0	0	0	0	1	10	19	19	19	19
o:	0	0	0	0	0	0	0	0	0	0	0	0	6	15	15	15	15
d:	0	0	0	0	0	0	0	0	0	0	0	0	2	11	11	11	11