

Computational methods in bioinformatics

Lecture 2

Measures of sequence similarity

Hamming distance:

Number of positions with mismatching characters.
Defined for two strings of equal length.

```
agtc  
cgta
```

Levenshtein distance:

Minimum number of edit operations (delete, insert, change a single character) needed to change one sequence into another.

```
agtcc  
cgctca
```

Dotplots

A pictorial representation of the similarity between two sequences.

Compare a sequence with itself:

- Repeats

- Palindromic sequences

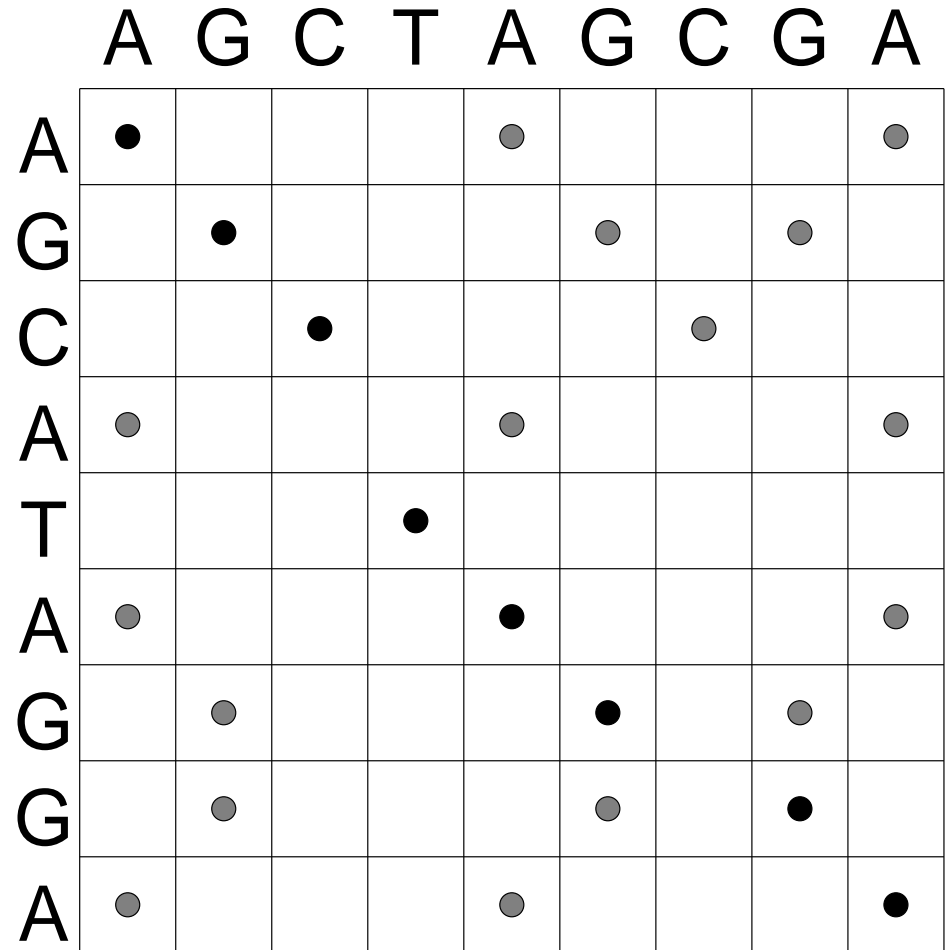
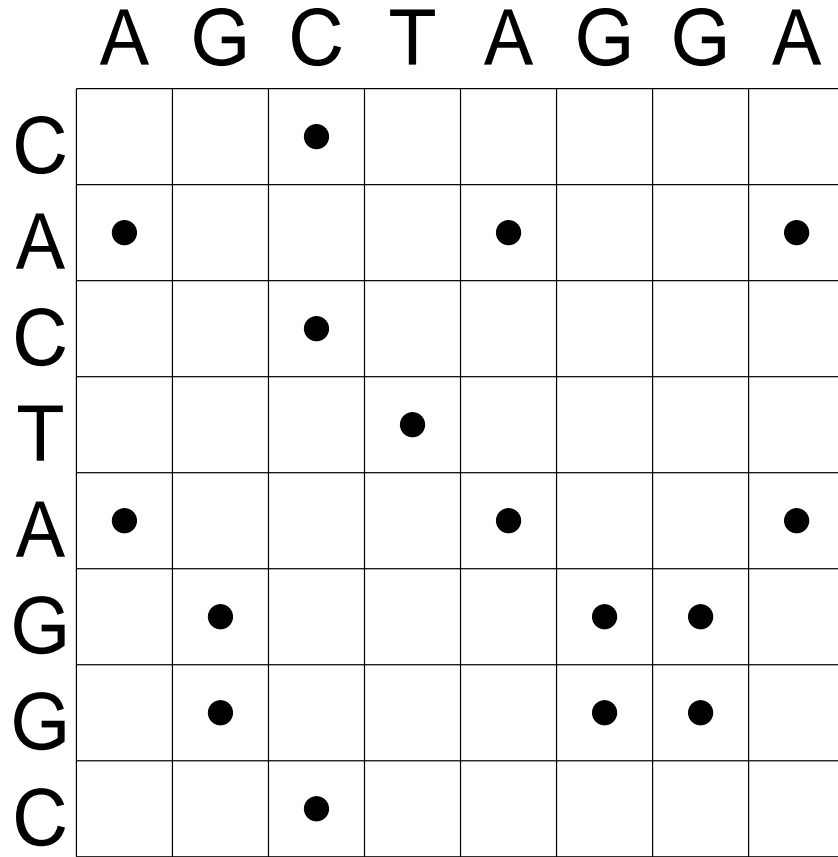
Compare two sequences:

- Any path from upper left to lower right represents an alignment.

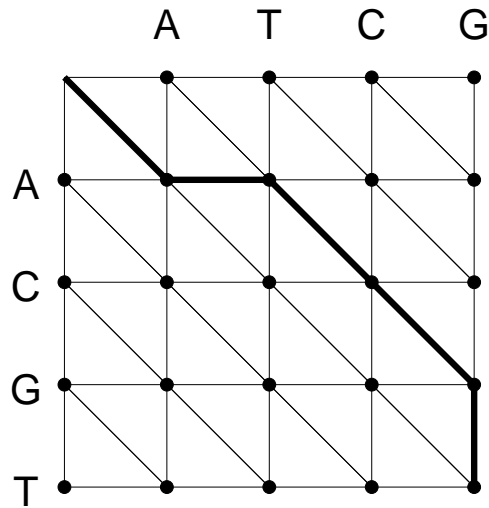
- Horizontal or vertical moves correspond to gaps in one of the sequences.

- Path with highest score corresponds to an optimal alignment.

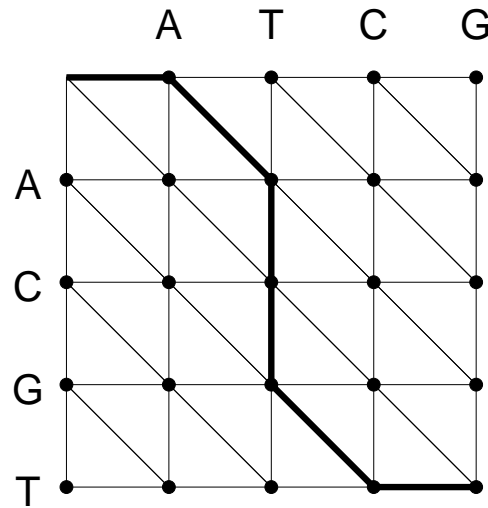
Dotplots



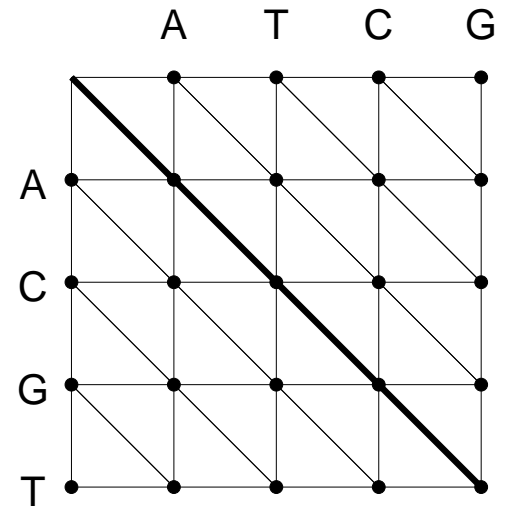
Each path represents an alignment



A-CGT
 | |
 ATCG-



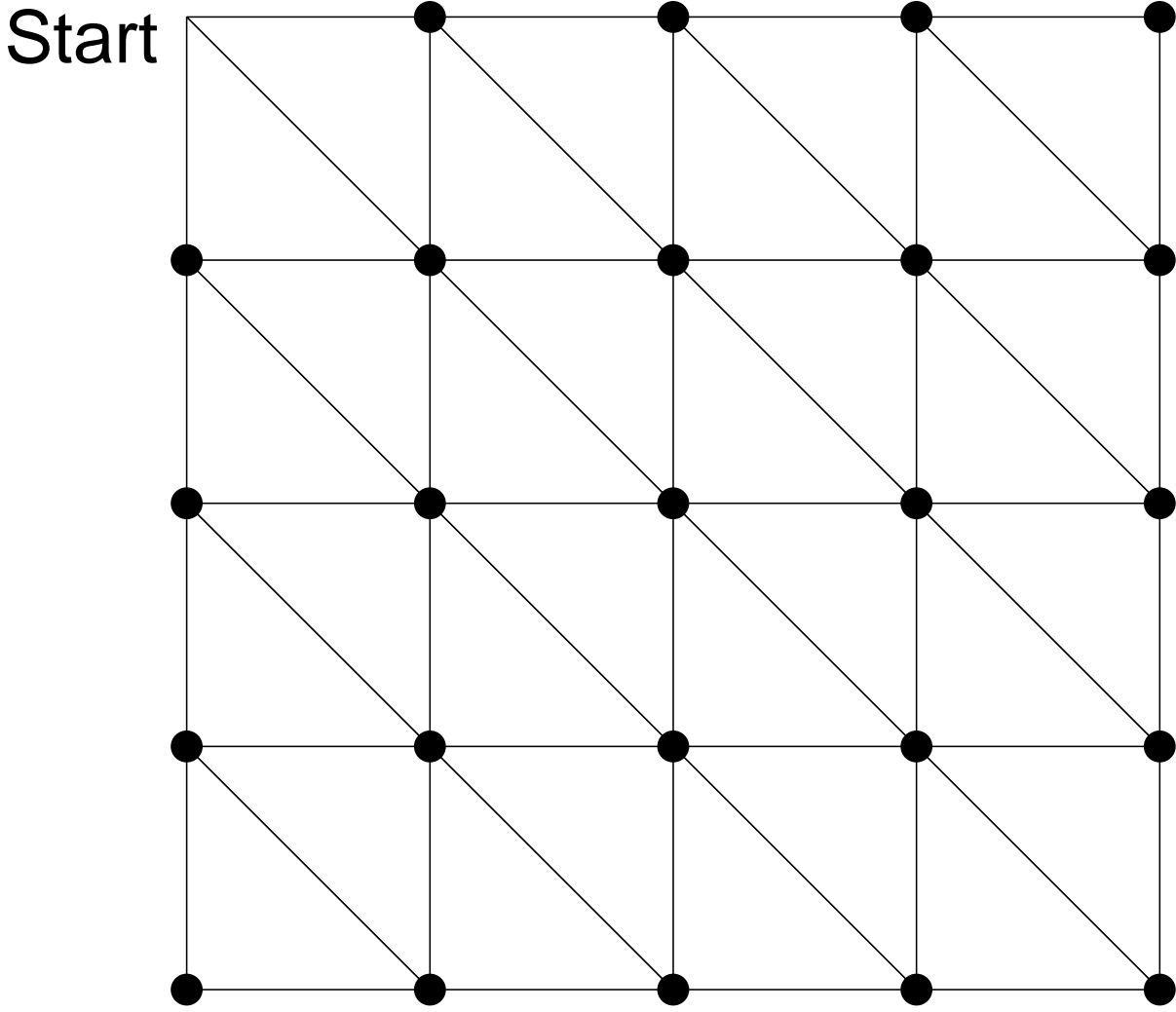
-ACGT-
 | | | |
 AT--CG



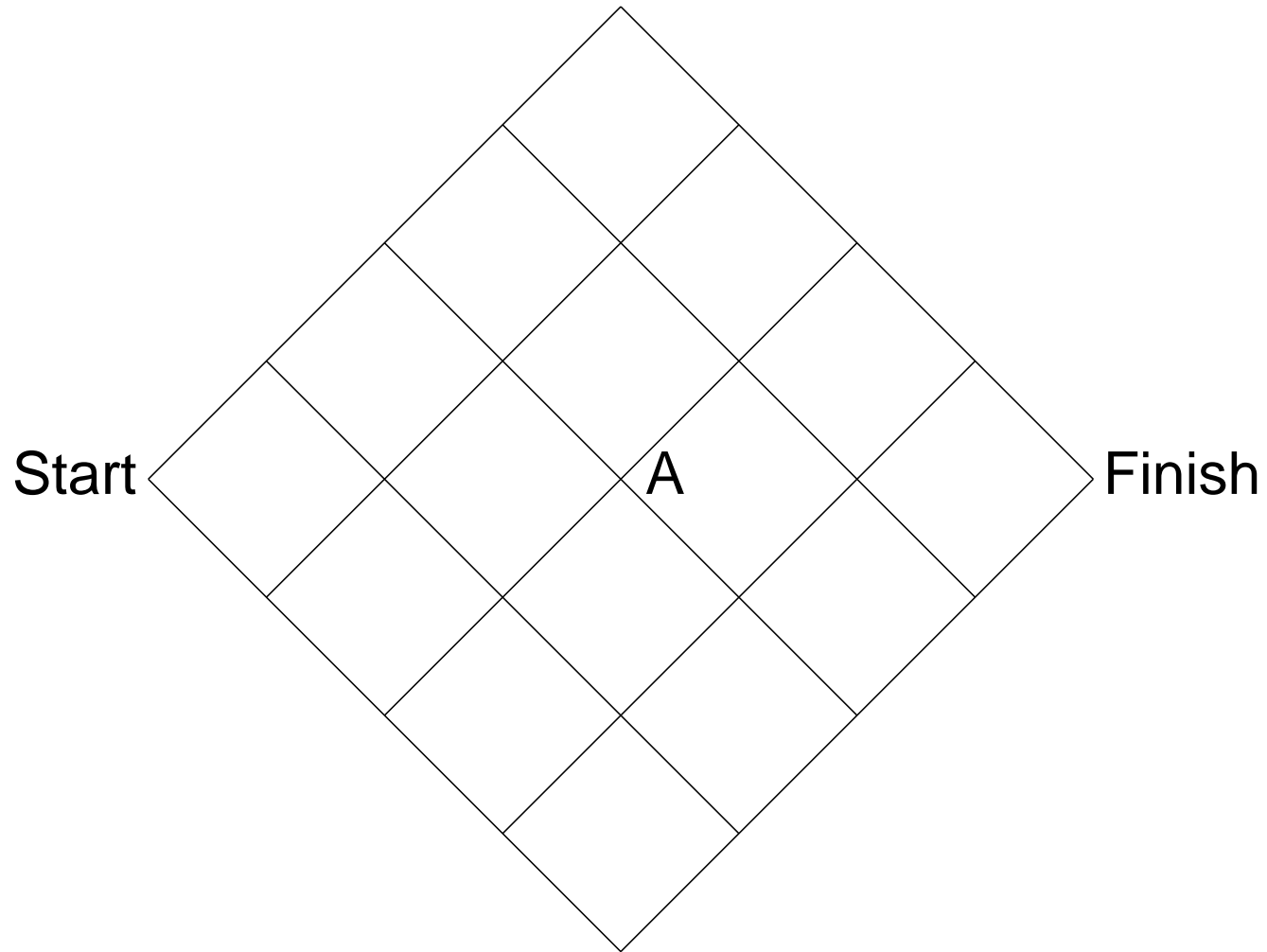
ACGT
 |
 ATCG

- Vertical steps add a gap to the horizontal sequence
- Horizontal steps add a gap to the vertical sequence

How many paths?



Do we have to enumerate all paths?



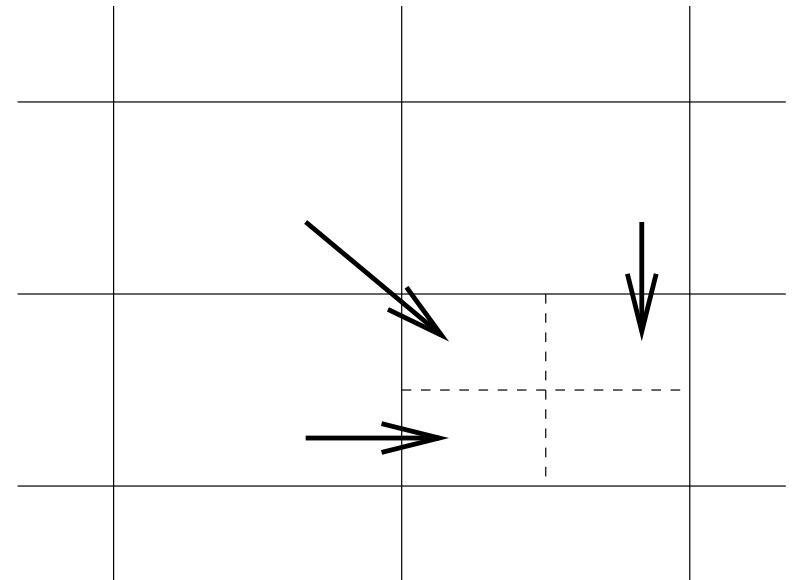
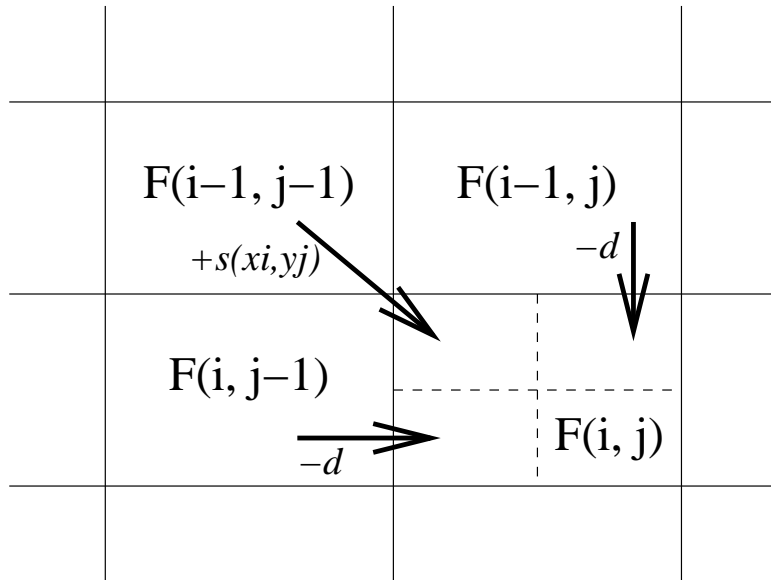
Pairwise global alignment (Needleman-Wunsch algorithm)

Rigorous algorithms use dynamic programming to find an optimal alignment.

- match score
- mismatch score
- gap penalty

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases}$$

Dynamic programming



Score matrix

		A	C	G	T	A	
		■	■	■	■	■	■
A	■		■	■	■	■	■
T	■		■	■	■	■	■
C	■		■	■	■	■	■
G	■		■	■	■	■	■
A	■		■	■	■	■	■

Percent identity

Having obtained an alignment, it is common to quantify the similarity between a pair of sequences by stating the percent identity.

```
-ACGATAG-CGAAACCAAAA  
| | | | | | | |  
CAGC-TAGCCGATGTC----
```

Count the number of alignment positions with matching characters and divide by ... *what?*

- the length of the shortest sequence?
- the length of the alignment?
- the average length of the sequences?
- the number of non-gap positions?
- the number of equivalenced positions excluding overhangs?

Pairwise local alignment (Smith-Waterman algorithm)

Local similarities may be masked by long unrelated regions.

A minor modification to the global alignment algorithm.

- If the score for a subalignment becomes negative, set the score to zero.

$$F(i, j) = \max \begin{cases} 0 \\ F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases}$$

- Trace back from the position in the score matrix with the highest value.
- Stop at cell where score is zero.

Is the similarity significant, or could it be due to chance?

Even if two proteins are unrelated, we would expect some similarity simply by chance.

Is the alignment score significantly higher than random?

Align random permutations of the sequences, and find the mean and standard deviation of the resulting distribution.

The z-score reflects the significance of a global similarity score.

$$z\text{-score} = \frac{\textit{score} - \textit{mean}}{\textit{standard deviation}}$$

Larger values imply greater significance.

BLAST

Basic Local Alignment Search Tool

Less accurate than Smith-Waterman, but over 50 times faster.

1. Find ungapped matches of a small fixed length, w , that score at least T .
2. Extend matches in both directions in an attempt to find an alignment with a score exceeding S .

Segment pairs whose scores cannot be improved by extending or trimming are called high scoring pairs (HSPs).

Typical values for w are 3 when aligning proteins and 11 when aligning nucleic acids.

e-values and p-values

The expected number of HSPs with a score of at least S is given by the formula:

$$E = Kmne^{-\lambda S}$$

Doubling the length of the query sequence (m) or the size of the database (n) should double the number of HSPs.

To obtain score $2x$, score x must be obtained twice in a row. So one expects E to decrease exponentially with score.

The probability of observing a score $\geq S$ is:

$$1 - \exp(-Kmne^{-\lambda S})$$

This is the p-value.

FASTA

k-tuples, strings of length k.

k = 1 - 2 for proteins and 4-6 for nucleic acids.

Construct a look-up table with all k-tuples in the database.

Look up all k-tuples from the query string and mark matching database k-tuples. Sort matches by the difference in their indices (i-j).

Nearby matches on the same diagonal are joined to form an ungapped local alignment region.

Join nearby high scoring regions on different diagonals.

For the best regions, perform dynamic programming in a window around the region.

global_alignment.c

```
#include <stdio.h>

#define MAX_LENGTH 100

#define MATCH_SCORE 2
#define MISMATCH_SCORE -1
#define GAP_PENALTY 2

#define STOP 0
#define UP 1
#define LEFT 2
#define DIAG 3

main()
{
    int i, j;
    int m, n;
    int alignmentLength, score, tmp;
    char X[MAX_LENGTH+1] = "ATCGAT";
    char Y[MAX_LENGTH+1] = "ATACGT";

    int F[MAX_LENGTH+1][MAX_LENGTH+1]; /* score matrix */
    int trace[MAX_LENGTH+1][MAX_LENGTH+1];
    char alignX[MAX_LENGTH*2]; /* aligned X sequence */
    char alignY[MAX_LENGTH*2]; /* aligned Y sequence */

    /*
     * Find lengths of (null-terminated) strings X and Y
     */
    m = 0;
    n = 0;
    while ( X[m] != 0 ) {
        m++;
    }
    while ( Y[n] != 0 ) {
        n++;
    }

    /*
     * Initialise matrices
     */
    F[0][0] = 0;
    trace[0][0] = STOP;
    for ( i=1 ; i<=m ; i++ ) {
        F[i][0] = F[i-1][0] - GAP_PENALTY;
        trace[i][0] = STOP;
    }
    for ( j=1 ; j<=n ; j++ ) {
        F[0][j] = F[0][j-1] - GAP_PENALTY;
        trace[0][j] = STOP;
    }

    /*
     * Fill matrices
     */
    for ( i=1 ; i<=m ; i++ ) {
        for ( j=1 ; j<=n ; j++ ) {
            if ( X[i-1]==Y[j-1] ) {
                score = F[i-1][j-1] + MATCH_SCORE;
            } else {
                score = F[i-1][j-1] + MISMATCH_SCORE;
            }
            trace[i][j] = DIAG;

            tmp = F[i-1][j] - GAP_PENALTY;
            if ( tmp>score ) {
                score = tmp;
                trace[i][j] = UP;
            }

            tmp = F[i][j-1] - GAP_PENALTY;
            if( tmp>score ) {
                score = tmp;
                trace[i][j] = LEFT;
            }

            F[i][j] = score;
        }
    }

    /*
     * Print score matrix
     */
    printf("Score matrix:\n      ");
    for ( j=0 ; j<n ; ++j ) {
        printf("%5c", Y[j]);
    }
    printf("\n");
    for ( i=0 ; i<=m ; i++ ) {
        if ( i==0 ) {
            printf(" ");
        } else {
            printf("%c", X[i-1]);
        }
        for ( j=0 ; j<=n ; j++ ) {
            printf("%5d", F[i][j]);
        }
        printf("\n");
    }
    printf("\n");

    /*
     * Trace back from the lower-right corner of the matrix
     */

    i = m;
    j = n;
    alignmentLength = 0;

    while ( trace[i][j] != STOP ) {
        switch ( trace[i][j] ) {
            case DIAG:
                alignX[alignmentLength] = X[i-1];
                alignY[alignmentLength] = Y[j-1];
                i--;
                j--;
                alignmentLength++;
                break;
            case LEFT:
                alignX[alignmentLength] = '-';
                alignY[alignmentLength] = Y[j-1];
                j--;
                alignmentLength++;
                break;
            case UP:
                alignX[alignmentLength] = X[i-1];
                alignY[alignmentLength] = '-';
                i--;
                alignmentLength++;
        }
    }

    /*
     * Unaligned beginning
     */
    while ( i>0 ) {
        alignX[alignmentLength] = X[i-1];
        alignY[alignmentLength] = '-';
        i--;
        alignmentLength++;
    }
    while ( j>0 ) {
        alignX[alignmentLength] = '-';
        alignY[alignmentLength] = Y[j-1];
        j--;
        alignmentLength++;
    }

    /*
     * Print alignment
     */
    for ( i=alignmentLength-1 ; i>=0 ; i-- ) {
        printf("%c",alignX[i]);
    }
    printf("\n");
    for ( i=alignmentLength-1 ; i>=0 ; i-- ) {
        printf("%c",alignY[i]);
    }
    printf("\n");

    return(1);
}
}
```