CS262: Lecture #3 – Linear-Space Sequence Alignment

Boyko Kakaradov

January 16, 2007

Sequence Alignment (Recap)

Sequence alignment is the process of introducing gaps in two (or more) sequences which reveals similarity in their structure. We formulate similarity in terms of the edit distance between the sequences. Edit distance is a function of the number of mutations, extensions and deletions needed to transform one sequence to match the other.

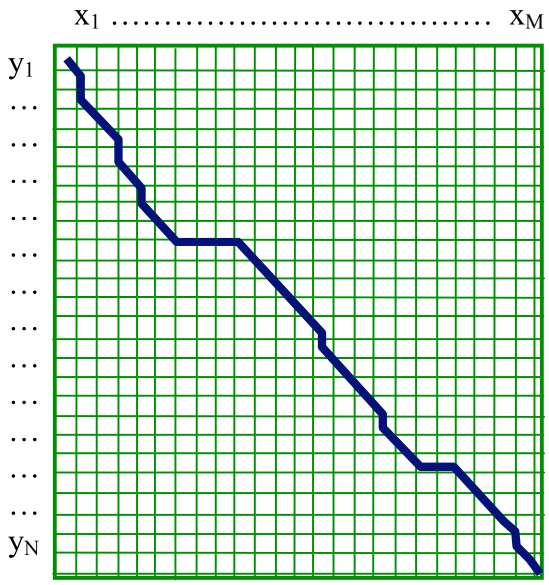
Given two strings $x = x_1x_2...x_M$, $y = y_1y_2...y_N$,
an alignment is an assignment of gaps to positions 0,…, N in x, and 0,…, N in y, so as to line up each letter in one sequence with either a letter, or a gap in the other sequence

For example, given the two sequences on the left below, we introduce gaps in a manner shown on the right which emphasizes the matching letters in both sequences:

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC      -AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAGCTATCACGACCGCGGTCGATTTGCCCGAC     TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC
```

Needleman-Wunsch Alrogithm (Recap)

The Needleman-Wunsch algorithm (NW) is a dynamic programming approach to global sequence alignment which computes the optimal alignment score for telescoping substrings of each sequence in order to find the optimal global alignment.



Every non-decreasing path from (0,0) to (M, N) corresponds to an alignment of the two sequences.
The optimal alignment is the path with

An optimal alignment is composed of optimal sub-alignments

A diagonal segment corresponds to a (mis)match in the sequences. Adds a matching constant to the alignment score, or subtracts a mismatch penalty.

A vertical segment corresponds to a gap in the column sequence.

A horizontal segment corresponds to a gap in the row sequence. Subtracts a gap penalty from scoring function.

Smith-Waterman Algorithm (Recap)

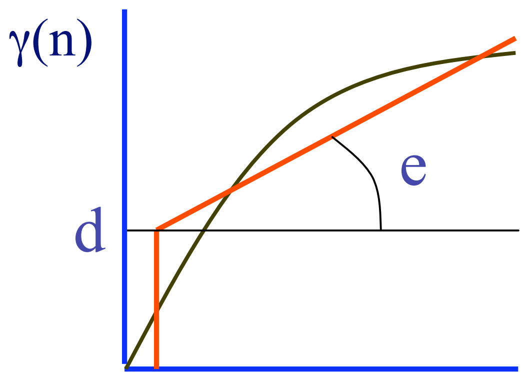
The Smith-Waterman algorithm (SW) is a modification of NW, which finds the optimal local alignment between two sequences. A local alignment is a global alignment between one or more substrings of the original sequences which maximize the global alignment score. The motivation for SW is that the genome consists of highly-preserved coding regions which correspond to genes with common function in an evolutionary tree that are encapsulated by non-conserved regions that are either non-coding or correspond to new genes in species further up the evolutionary tree.

SW is a simple modification of NW. In order to ignore the non-conserved regions, SW adds a zero in the maximization step of the DP recurrences of the optimal alignment sub-score. Thus, if the maximum NW score becomes negative, SW simply restarts the alignment.

Both SW and NW run in quadratic time and require quadratic space: O(nm) for sequences of lengths n and m.

Affine Gaps (Recap)

Affine gaps define a non-constant gap penalty function which more closely models the biological process of extensions and deletions in the genome. The model closest to reality uses a non-linear convex gap function that requires cubic running time due to maximization steps within the main DP maximization step. The affine gap model approximates that with a piecewise linear convex function. It uses two extra matrixes for each linear segment, which is a constant blowup of the quadratic space complexity, but it requires only quadratic running time.



To compute optimal alignment at position i, j, we need to "remember" the best score if gap is open and the best score if gap is not open

F(i, j): optimal score of alignment $x_1 \ldots x_i$ to $y_1 \ldots y_j$ **<u>if</u>** x_i aligns to y_j

G(i, j): optimal score **<u>if</u>** x_i aligns to a gap after y_j

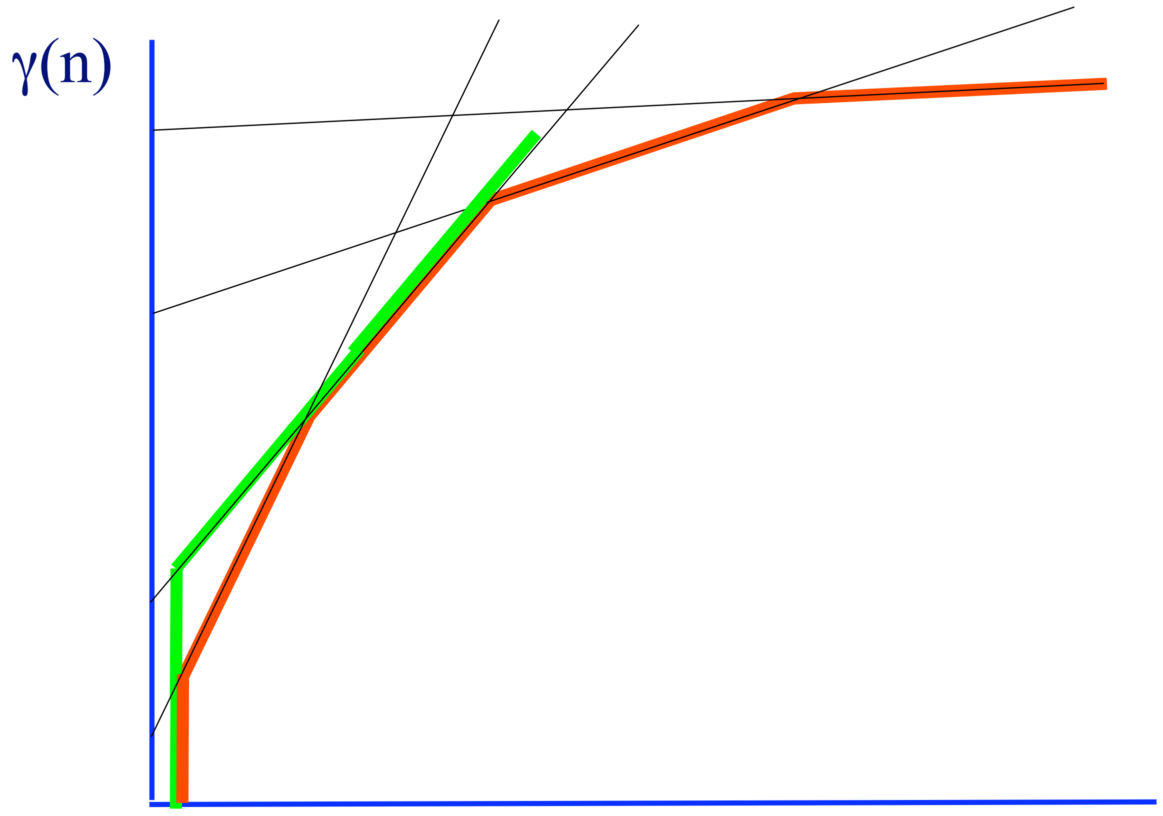
H(i, j): optimal score **<u>if</u>** y_j aligns to a gap after x_i

V(i, j) = optimal score of alignment $x_1 \ldots x_i$ to $y_1 \ldots y_j$

$$\gamma(n) = \underbrace{d}_{\text{gap open}} + (n-1)\times \underbrace{e}_{\text{gap extend}}$$

Generalized Piecewise-Linear Gaps

We extend the affine gap model to more closely approximate the non-linear convex gap function characteristic of extensions and deletions of the genome in real biological systems. Instead of a single linear segment, we can use four as shown below:

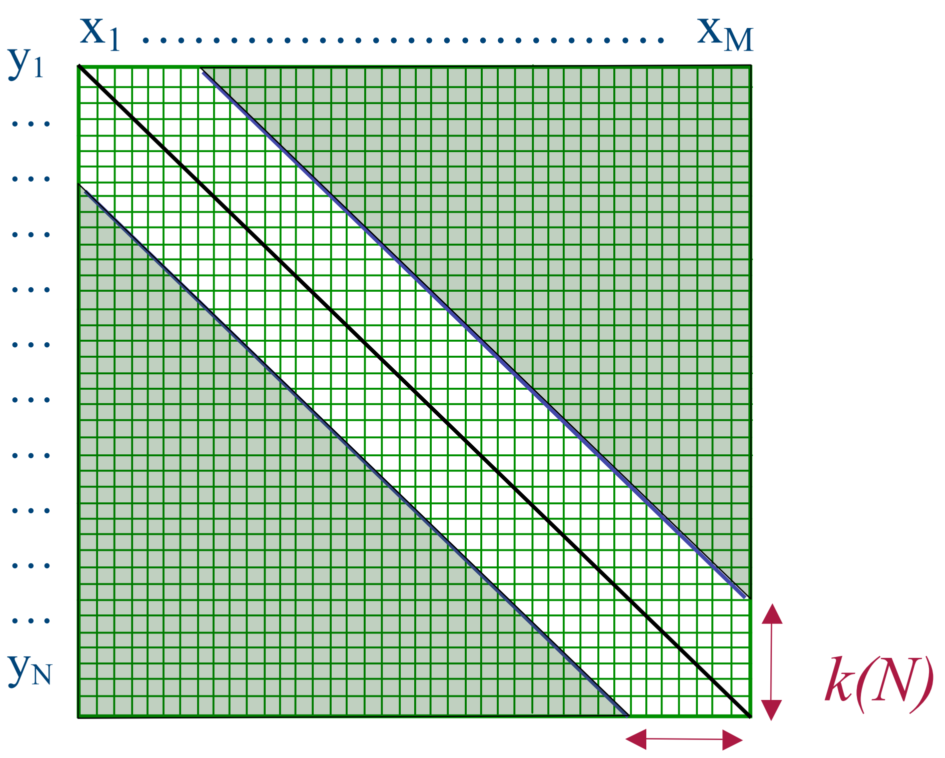


To execute NW using the above gap function while preserving its quadratic time and space complexity, we need to add two matrices for each additional linear segment. For example, in addition to G and H as defined for the affine gap model, we have G' and H' which contain the optimal alignment score assuming an affine gap penalty shown in green above. Note that G and H will contain better scores than G' and H' up to the intersection point of the first and second linear segments above.

Bounded Dynamic Programming

Assuming that the two sequences *x* and *y* are very similar, we can reduce both the time and space complexity of NW. For example, if the number of gaps in the optimal alignment of *x* and *y* is bounded by some constant *k(N)* dependent only on the size of *x* and *y*, we can deduce that if x_i and y_j are aligned, then the difference between their indexes $| i - j |$ which corresponds to the number of gaps in the sub-alignment up to this point is also bounded by *k(N)*. This restriction allows us to limit the problem space in the

DP matrix to a diagonal of width *k(N)* as shown below, which reduces the time and space complexity from $O(N^2)$ to $O(N*k(N))$. BDP is easy to apply to the affine gap model.



To formalize the above restriction, we modify the DP recurrence equations for NW:

Initialization:

F(i,0), F(0,j) undefined for i, j > k

Iteration:

```
For i = 1…M
  For j = max(1, i – k)…min(N, i+k)
```

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i, j-1) - d, \text{ if } j > i - k(N) \\ F(i-1, j) - d, \text{ if } j < i + k(N) \end{cases}$$

Termination: same as NW

Linear-Space Alignment Algorithm

Motivation:
With lengths on the order of billions, biological sequences are so long that even efficient quadratic-space alignment algorithms like NW require too much memory. On the other hand, the quadratic running time is perfectly feasible on modern computers. Therefore, we would like to develop a linear-space alignment (LSA) algorithm which trades off time for space complexity.

Definitions:
A **substring** x' of x is a string of consecutive letters such that $x = ux'v$ for some strings u and v. Alternatively we can write $x' = x_i \dots x_j$ for some $1 \leq i \leq j \leq |x|$.

A **subsequence** x' of x is a substring of x with 0 or more deleted letters. Alternatively we can write $x' = x_{i1} \dots x_{ik}$, for some $1 \leq i_1 \leq \dots \leq i_k \leq |x|$.

Hirschberg's Algorithm:
Given a set of strings $x, y, \dots,$ a ***common subsequence*** is a string u that is a subsequence of all strings $x, y, \dots$ Hirchberg's algorithm finds the score of the longest common subsequence $u = u_1 \dots u_k$ of two strings $x = x_1\, x_2 \dots x_M$ and $y = y_1\, y_2 \dots y_N$. It is a DP algorithm similar to NW with the following modifications:

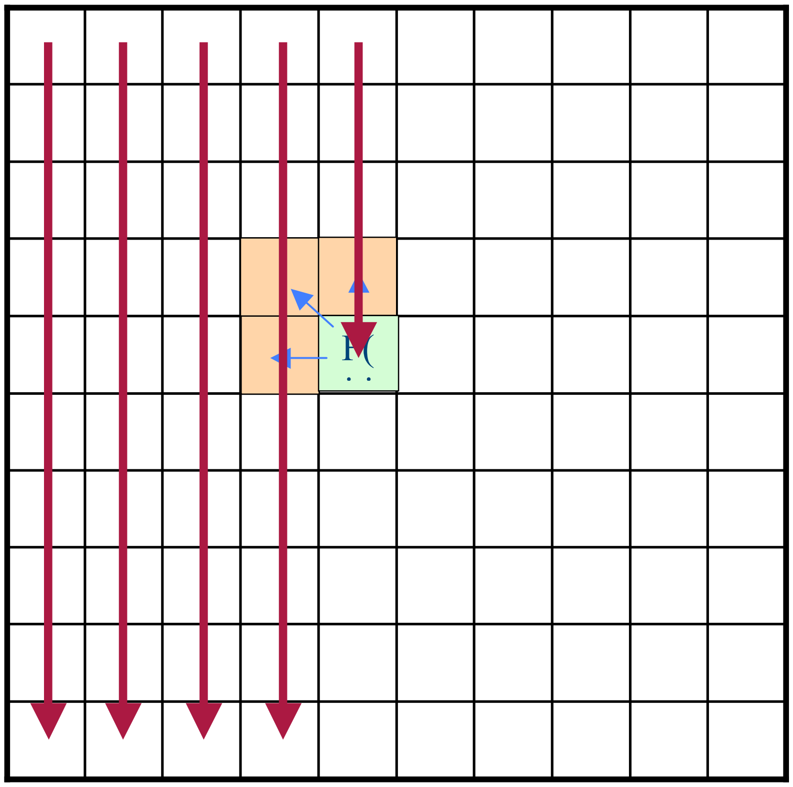
Recurrences:

$$F(i, j) = \max \begin{cases} F(i-1, j) \\ F(i, j-1) \\ F(i-1, j-1) + \mathbf{[1, \text{if } x_i = y_j; 0 \text{ otherwise}]} \end{cases}$$

Ptr(i, j) = *(same as in NW)*

Termination: trace back from Ptr(M, N), and prepend a letter to u whenever Ptr(i, j) = DIAG **and** $F(i-1, j-1) < F(i, j)$

Hirschberg's algorithm runs in linear space by exploiting the fact that F(i, j) only depends on entries in the previous row or column. As shown on the diagram on the next page, we can compute the entries row by row (column by column) from left to right (top to bottom) by keeping only the previous row (column) in memory. Thus, the space complexity of Hirchberg's algorithm is linear. However, the algorithm only gives the optimal score, but not the optimal subsequence as in the process we lose the backward pointers.



```
Allocate ( column[1] )
Allocate ( column[2] )

For   i = 1….M
      If  i > 1, then:
        Free( column[ i – 2 ] )
        Allocate( column[ i ] )
      For   j = 1…N
        F(i, j) = … recurrences
```

Linear-Space Alignment:

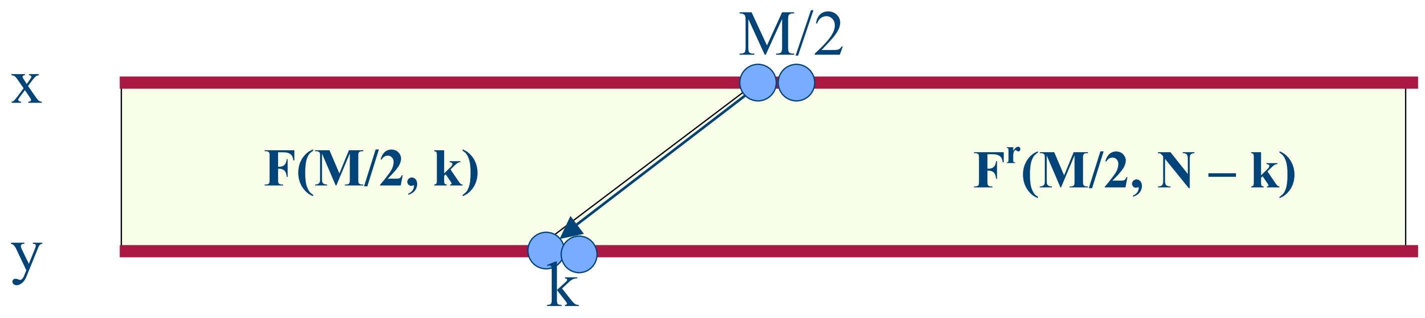
To compute the optimal score and retain the optimal alignment, we use a **divide & conquer** approach to determining which backward pointers to remember. That is, we divide the problem into a logarithmic number of sub-problems and solve each one based on the solutions of the smaller sub-problems. This approach works because of the following result on combining sub-problems:

Lemma: (assume M is even)

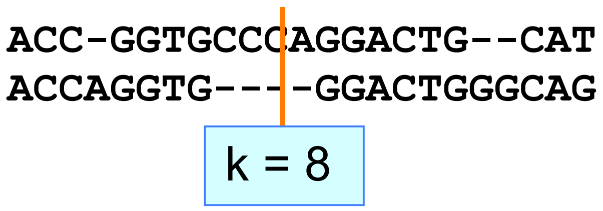
$$F(M, N) = \max_{k=0\ldots N}\left(F(M/2, k) + F^r(M/2, N - k) \right)$$

Where x^r is the reverse of a string x and $F^r(i,j)$ is the optimal score of aligning $x^r_1\ldots x^r_i$ with $y^r_1\ldots y^r_j$ which is the same as aligning $x_{M-i+1}\ldots x_M$ with $y_{N-j+1}\ldots y_N$

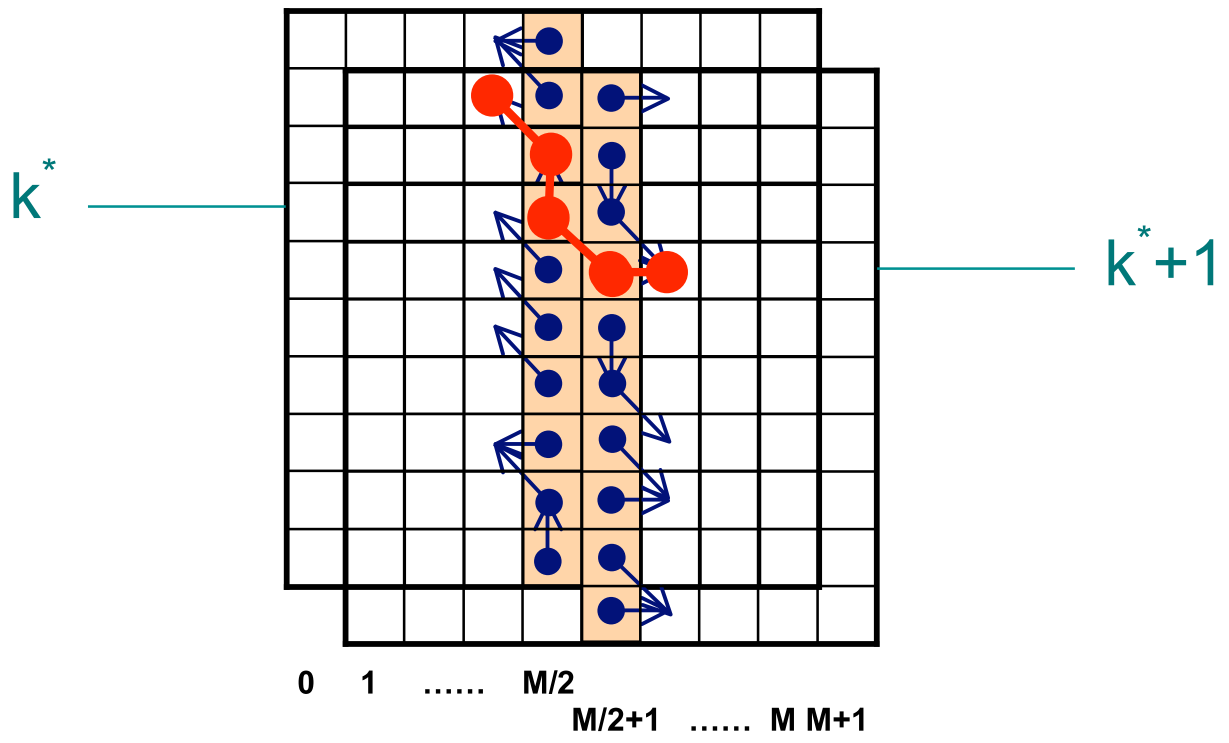
The above lemma states that the global optimal score can be computed recursively by combining the optimal scores of aligning each half of the first string (x) to some division of the second (y) such that their sum is maximal. Graphically, the lemma can be represented as follows (where k is the maximizing length).



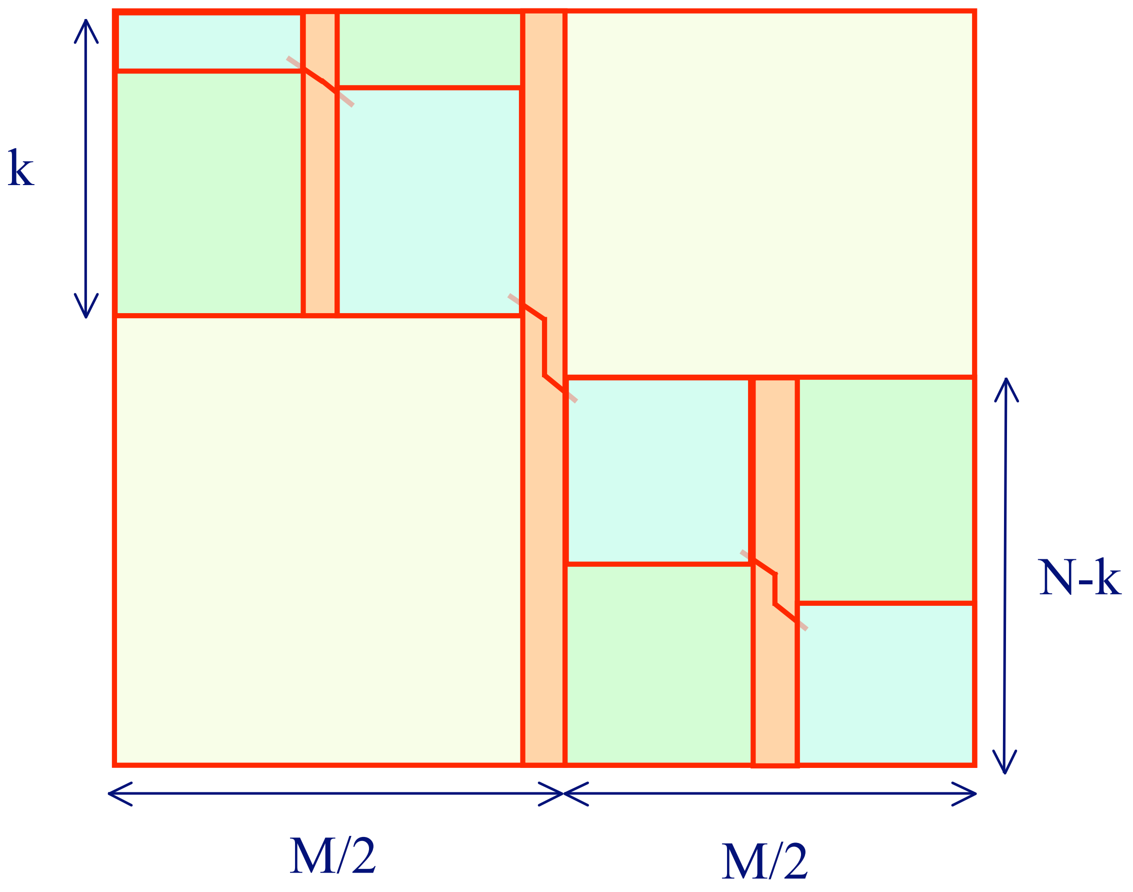
For example, we can apply the lemma to the alignment below to find the maximizing k. In this case, the orange line represents the middle of the first string which is aligned with the first 8 letters of the second string with trailing gaps, so k = 8.



Given the above result, we can compute both the optimal alignment F(M, N) and the back pointers along the division line M/2 to k using 2 columns of the DP matrix as shown below. The left column represents F(M/2, k), the optimal alignment scores between $x_1 \ldots x_{M/2}$ and $y_1 \ldots y_k$ for k = 1…N. The right column represents $F^r(M/2, N – k)$, the optimal alignment scores between $x_{M/2+1} \ldots x_M$ and $y_{k+1} \ldots y_N$. The back pointers for all 2N entries are computed, but only the maximizing connecting path (in red) for k = 4 is memorized.



This constitutes one recursive step of the divide & conquer algorithm. We repeat the procedure on each sub-matrix defined by the optimal choice of k. A second-order recursive call is depicted in the figure below, and the third-order will be performed on the blue sub-matrixes



The maximizing paths through the middle of each sub-matrix are indicated in red. The above algorithm can be formalized as follows:

Hirschberg's Linear-space algorithm:

```
MEMALIGN(l, l', r, r'):                (aligns $x_l \ldots x_{l'}$ with $y_r \ldots y_{r'}$)
1.    Let $h = \lceil (l'-l)/2 \rceil$
2.    Find (in Time $O((l' - l) \times (r' - r))$, Space $O(r' - r)$)
      the optimal path,  $L_h$, entering column $h - 1$, exiting column $h$
      Let $k_1$ = position at column $h - 2$ where $L_h$ enters
          $k_2$ = position at column $h + 1$ where $L_h$ exits

3.    MEMALIGN(l, $h - 2$, r, $k_1$) is alignment to the left

4.    Output $L_h$

5.    MEMALIGN($h + 1$, l', $k_2$, r') is alignment to the right
```

Top level call: MEMALIGN(1, M, 1, N)

Steps 2 above is the maximization step in the Lemma discussed earlier. It performs Hirchberg's algorithm on the two halves of the matrix M containing columns $x_l \ldots x_{l'}$ and rows $y_r \ldots y_{r'}$ – in the left half it's forward and down from (l,r) and in the right half it's backwards and up from (l',r'). Thus, each execution of step two requires $O((l' - l) \times (r' - r))$ time since it must visit each square in M, and $O(r' - r)$ time since it stores two columns of height (r'- r).

Steps 3,4,5 represent an in-order recursive calls of the algorithm on the left and right halves of M, and in the middle outputs the optimal path between the two halves calculated Step 2.

As described above, the first call to Step 2 takes cMN time and 2N space for some constant c. Independent of the particular optimal division, the top-most left and right recursive calls' maximization steps run in a combined time $M/2 \times k + M/2 \times (N - k) = MN/2$ and combined space N. The storage of the optimal path requires precisely M+N space in addition. Thus, the total running time of the MAMALIGN algorithm is still quadratic: O(MN) but the total space is linear: O(M+N).

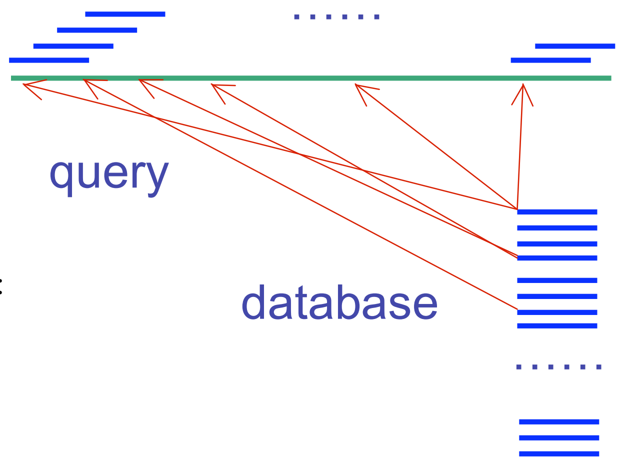
<u>Heuristic Local Aligners</u>

Motivation:
Biological (genomic and protein) sequence databases are growing exponentially around ten times every three years clearly outpacing Moore's Law even if it were still holding. The two primary reasons are exponentially dropping costs of sequencing techniques and growing interest in the scientific community. Currently, around a hundred billion base pairs containing the full genome of twenty-four animal species are already sequenced and available for analysis. Similarly, more than a million known and predicted protein sequences exist.

Problem:
If a scientists discovers a new gene ($\sim 10^4$ base pairs) and wants to determine if it appears in other species and if so, how fast it evolves, (s)he will need to align it to the entire genomic database ($\sim 10^{10}$ to 10^{12} BP). However, performing the Smith-Waterman (S-W) algorithm in quadratic time to find local alignments will take too long. In a different scenario, the scientist might want to compare the newly sequenced genome ($\sim 3 \times 10^9$ BP) from a scientifically significant species to the genomic database.

Solution: Indexing-Based Local Alignment
BLAST, which stands for Basic Local Alignment Search Tool, uses a database of short sequence fragments to perform local alignment using a simple heuristic which makes it orders of magnitude faster than S-W. It is the most useful tool in genomics, and the most cited paper in history of science. The basic idea is to construct a dictionary of all words (short sequence fragments) in the query sequence and to perform local alignment centered at each word matching the database.

BLAST: details
The dictionary or database consists of sequence fragments of some short length (k = 10) from all currently known genomes that are sorted in lexicographic order (in linear or log-linear time). The query sequence of interest is then broken down into consecutively overlapping substrings of the same length (k = 10) which are then compared to the database (each in logarithmic time).



let k = 4 and the query:
ACGCAGCACGT
the 4-letter words are:
ACGC
  CGCA
    GCAG
      ....

When a match to the database is found (similarity score of word to the database is higher than some threshold T1), we initiate a local alignment expanding from the match until the alignment score drops below a certain threshold T2 (see figure below). Once we reach the end of the query, we calculate the total alignment score by adding the individual local alignment scores. If that is above a third threshold T3, the alignment is accepted. These three thresholds are derived statistically from available data.

A note on the running time of BLAST: while fast in practice, the worst-case running time of the above algorithm is cubic. This happens in the case when the query sequence is already in the database resulting in N^2 perfect alignments of all N words.

The following three examples present different ways to handle the extension of the seed match. The first one extends the alignment only linearly and stops after C = 2 mismatches in a row along the diagonal (which drops the alignment score). The second example allows gaps in the extension by searching for the optimal alignment a narrow band along the entire diagonal. The third example combines the first two by allowing gapped extensions but stopping after the score drops C below the current best one.

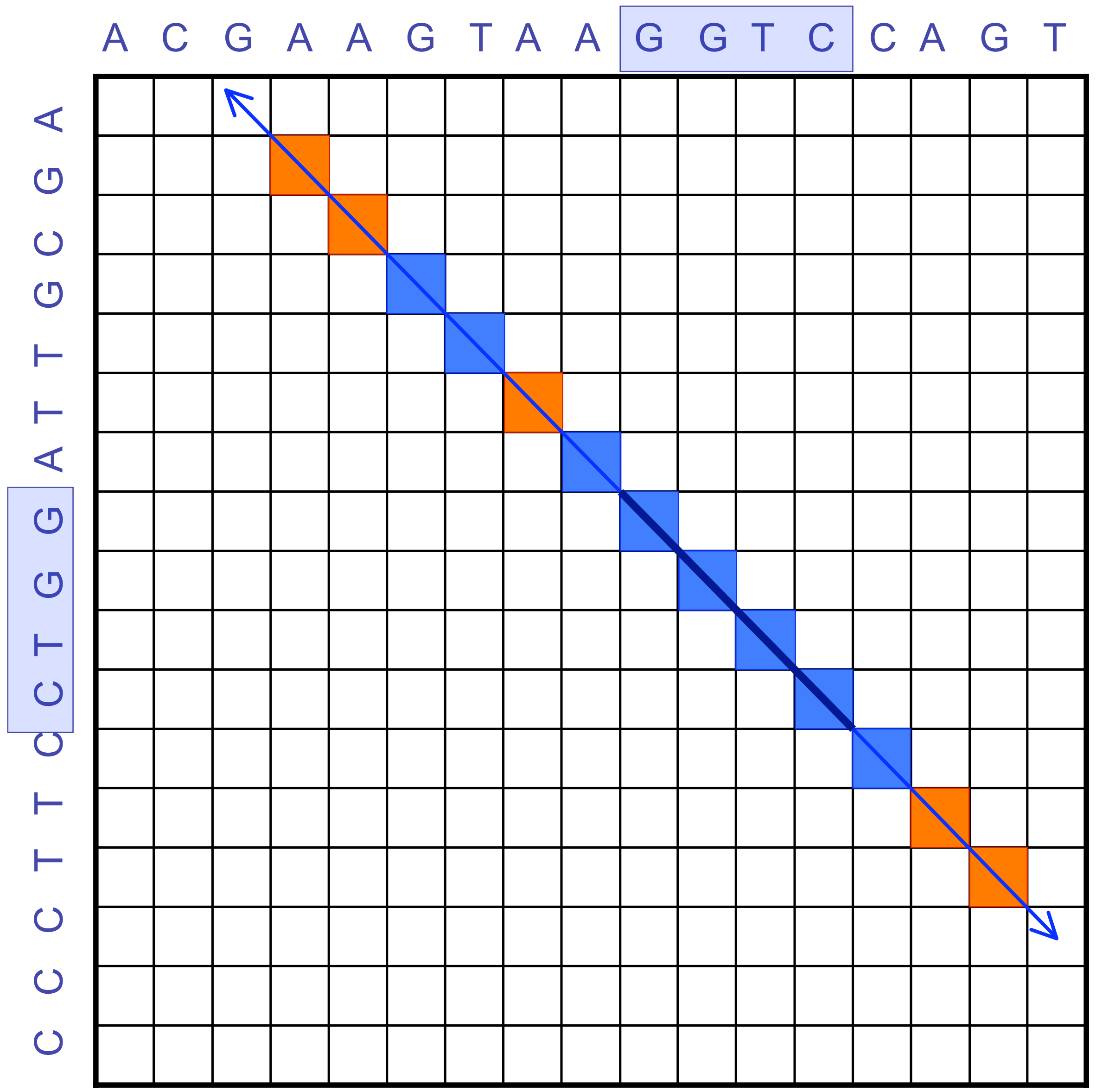
Finally, BLAST is not guaranteed to find the optimal local alignment. For example, consider the case where the word size is K = 10 but the optimal local alignment contains gaps every 9 base pairs. Then, no word from the query will match the database, so no local alignments will be initiated. This leads to a discussion on the **Speed vs Sensitivity tradeoff** – a bigger word size leads to faster searches that might miss the best alignment, while smaller K leads to slower searches which are more likely to return the optimal one.

Example: for k = 4

The matching word GGTC initiates an alignment extending to the left and right with no gaps until alignment falls < C below current best alignment

Output:

```
GTAAGGTCC
GTTAGGTCC
```

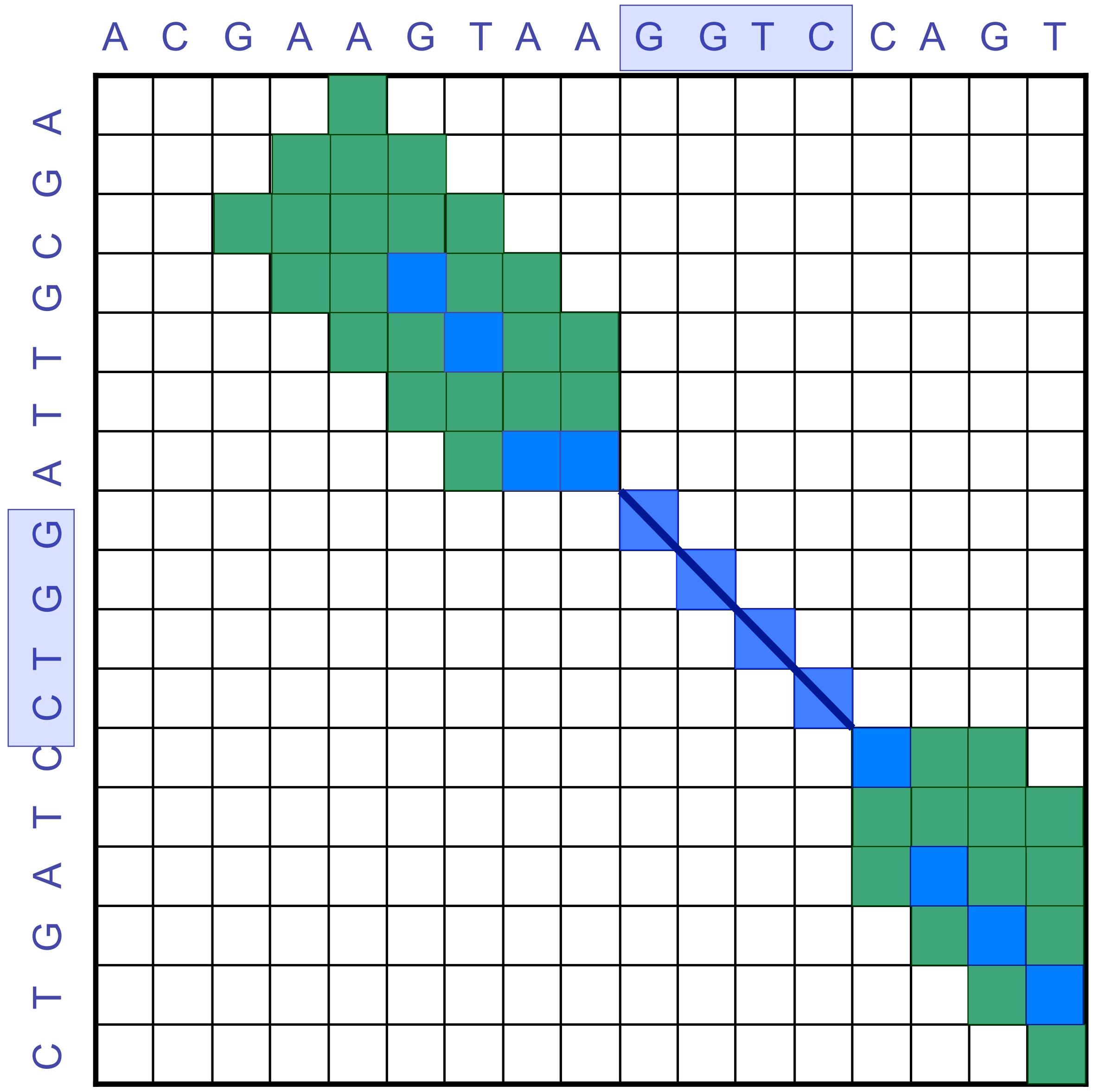


A C G A A G T A A G G T C C A G T
A G C G T A T G G T C C T T C C C

Gapped extensions: Extensions with gaps in a band around anchor

Output:

```
GTAAGGTCCAGT
GTTAGGTC-AGT
```


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

<u>Speed vs Sensitivity tradeoff</u>

Sensitivity is defined as the percent of perfect local alignments (found by S-W) returned by BLAST. For a particular pair of sequences with a certain level of similarity (x%), we would like to estimate the perfect word size which is likely to result in a given sensitivity. The speed of the search depends directly on the word size as well. The figure below shows the theoretical sensitivity of BLAST using a range of word sizes derived from synthetic data with different levels of similarity. For example, a search with K = 7 will have to initiate 13 million local alignments, and depending on the sequence similarity it will return an optimal alignment with probability ranging from 97.4% to 100%. On the other hand, a search with K =14 will have to initiate only 399 local alignments, but those will include an optimal alignment with probability as low as 31% for the 81%-similarity sequences. In practice, scientists consider sequences with 60% similarity. In this case a word size even lower than 7 (corresponding to a very slow search) may be needed to guarantee a certain level of sensitivity.

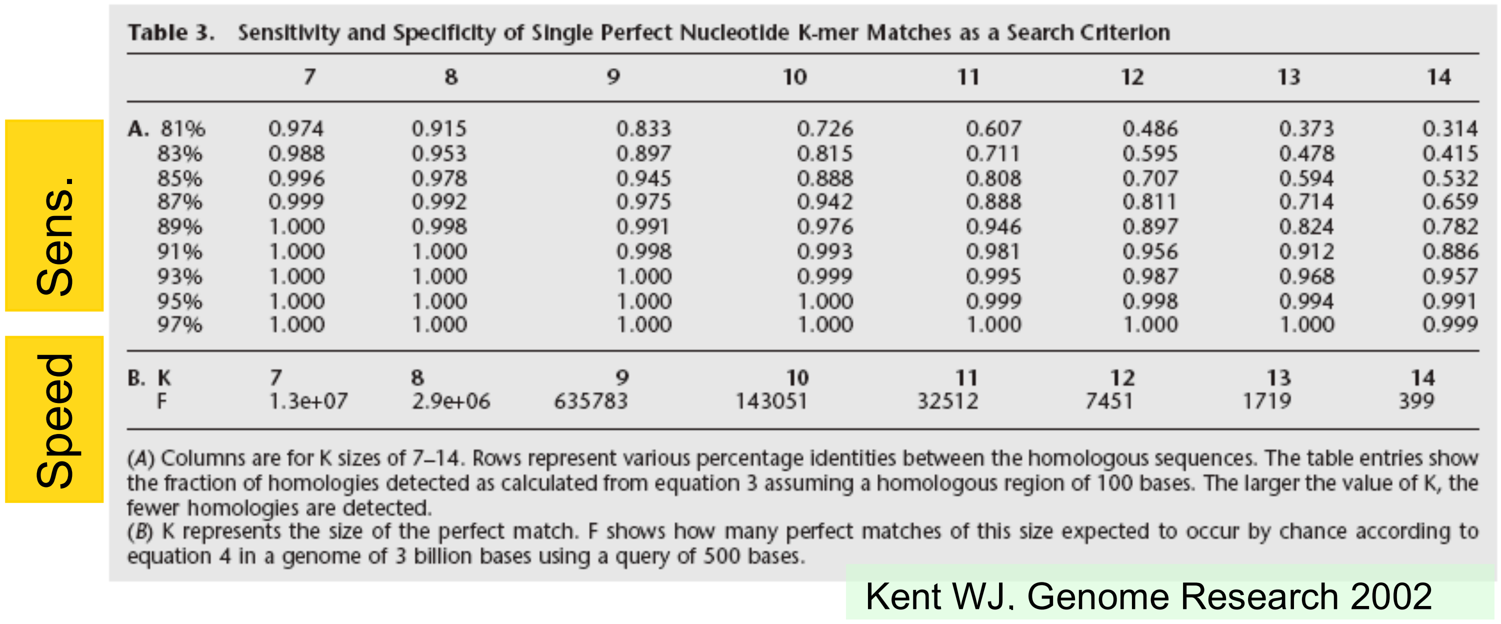


Table 3. Sensitivity and Specificity of Single Perfect Nucleotide K-mer Matches as a Search Criterion

		7	8	9	10	11	12	13	14
Sens.	**A.** 81%	0.974	0.915	0.833	0.726	0.607	0.486	0.373	0.314
	83%	0.988	0.953	0.897	0.815	0.711	0.595	0.478	0.415
	85%	0.996	0.978	0.945	0.888	0.808	0.707	0.594	0.532
	87%	0.999	0.992	0.975	0.942	0.888	0.811	0.714	0.659
	89%	1.000	0.998	0.991	0.976	0.946	0.897	0.824	0.782
	91%	1.000	1.000	0.998	0.993	0.981	0.956	0.912	0.886
	93%	1.000	1.000	1.000	0.999	0.995	0.987	0.968	0.957
	95%	1.000	1.000	1.000	1.000	0.999	0.998	0.994	0.991
	97%	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.999
Speed	**B.** K	7	8	9	10	11	12	13	14
	F	1.3e+07	2.9e+06	635783	143051	32512	7451	1719	399

(*A*) Columns are for K sizes of 7–14. Rows represent various percentage identities between the homologous sequences. The table entries show the fraction of homologies detected as calculated from equation 3 assuming a homologous region of 100 bases. The larger the value of K, the fewer homologies are detected.
(*B*) K represents the size of the perfect match. F shows how many perfect matches of this size expected to occur by chance according to equation 4 in a genome of 3 billion bases using a query of 500 bases.

Kent WJ. Genome Research 2002