1. **Sequence Alignment**

Sequence alignment comprises a good initial step for understanding biology, not only in terms of evolutionary steps (what is changed through insertions, deletions, mutations), but also what gives rise to functional elements in the genome (sequence conservation implies function). Therefore, sequence alignments is key to finding important regions in the genome, determining function, and uncovering evolutionary forces. A sequence alignment inserts gaps between two sequences to reveal their similarities.

A sequence can be *edited* through mutations, insertions, or deletions. Therefore, a sequence alignment scoring functions scores according to a match (+m), a mismatch (-s), or a gap (-d). The final score can then be calculated according to the following equation:

\[
Score \ F = (\# \ matches) \times m - (\# \ mismatches) \times s - (\#gaps) \times d
\]

1A. **Dynamic Programming Based Solutions**

*Needleman Wunsch Algorithm*

A dynamic programming based alogrithm which finds the best global alignment for two sequences. (A global alignment occurs when no substrings of the two compared strings are discarded, i.e., when the full strings are compared).

Every possible combination of residues in two strings (nucleotide bases or protein amino acids) are represented in a 2 dimensional array, with an extra row and column added to allow the alignment to begin with a gap of any length in either sequence. Penalty scores are filled in for each square (e.g., the penalty scores in the gap rows will be of increasing value as no value in any string will match a gap). The penalty scores are calculated by searching for the maximum possible value, using the iterative scoring scheme below.

Thus, all possible alignments are represented in this 2 dimensional matrix through pathways. A pathway is denoted by following pointers, as denoted below.

1.  **Initialization.**
   a. \( F(0, 0) = 0 \)
   b. \( F(0, j) = -j \times d \)
   c. \( F(i, 0) = -i \times d \)

2.  **Main Iteration.** Filling-in partial alignments
   a. For each \( i = 1 \ldots \ldots M \)
      For each \( j = 1 \ldots \ldots N \)
      \[
      F(i, j) = \max \begin{cases} 
      F(i-1,j-1) + s(x_i, y_j) & [\text{case 1}] \\
      F(i-1, j) - d & [\text{case 2}] \\
      F(i, j-1) - d & [\text{case 3}] 
      \end{cases}
      \]
PTR(1, j) = DIAG, if [case 1]
left, if [case 2]
UP, if [case 3]

3. **Termination.** F(M, N) is the optimal score, and from PTR(M, N) can trace back optimal alignment

**Smith Waterman Algorithm**

The Smith-Waterman algorithm finds the optimal local alignment between two sequences using the technique of dynamic programming. In many cases, one already knows that two sequences are not globally similar (e.g. in the case of comparing proteins from different families for similar domains). For the optimal local alignment, it searches for two substrings that when aligned, give an optimal global alignment score (best alignment between subsequences).

The Smith-Waterman algorithm is a variant of the Needleman-Wunsch. It has the same maximization step with the added value of 0 (look throughout the matrix for the greatest value). Therefore, the start and end points of the alignment can be anywhere on the matrix.

**1B. Gap Penalty Variations**

**Convex function for gap penalties**

In the simple, linear gap model, a gap of length n incurs penalty $n \times d$ (as in the original Smith-Waterman algorithm). However, this is not realistic as gaps usually occur in groups; once a gap is opened, it is easily extended.

One way to address this concern is by using a convex gap penalty function. In other words, the penalty decreases for additional gap positions (Rather than making the penalty linear in proportion to the length of the gap, the penalty is proportional to the log of the length of the gap). The gap penalties therefore lose their cumulative values and more realistically model the situation where gaps rarely come as singletons.

The convex gap penalty function ($\gamma(n)$), where n is the length of the gap satisfies the following condition:

$$\gamma(n): \text{for all } n, \gamma(n + 1) - \gamma(n) \leq \gamma(n) - \gamma(n - 1)$$
However, the drawbacks to this are in the analysis of the time complexity. Modifying the Needleman-Wunsch algorithm to fit this new model of gap penalties, would result in the following scheme:

1. **Initialization:**
   a. \( F(0, 0) = 0 \)
   b. \( F(0, j) = \gamma(j) \)
   c. \( F(i, 0) = \gamma(i) \)

2. **Main Iteration, Filling-in partial alignments**
   For each \( i = 1 \ldots M \)
   
   For each \( j = 1 \ldots N \)
   
   \[
   F(i, j) = \max \begin{cases} 
   F(i-1, j-1) + s(X[i], Y[j]) & \text{[case 1]} \\
   \max_{k=0 \ldots i-1} F(k,j) - \gamma(i-k) & \text{[case 2]} \\
   \max_{k=0 \ldots j-1} F(i,k) - \gamma(j-k) & \text{[case 3]}
   \end{cases}
   \]

   \[
   \text{Ptr}(i,j) = \begin{cases} 
   \text{DIAG}, & \text{if [case 1]} \\
   \text{LEFT}, & \text{if [case 2]} \\
   \text{UP}, & \text{if [case 3]}
   \end{cases}
   \]

3. **Termination** \( F(M, N) \) is the optimal score, and from \( \text{Ptr}(M, N) \) can trace back optimal alignment

Thus, to calculate the value of each cell, one would need to look at \((i + j + 1)\) potential cells, as opposed to only three cells for comparison as with the Needleman-Wunsch and Smith Waterman algorithms. Although the space complexity remains the same (a single 2 dimensional array): \(O(nm)\), the time complexity increases to \(O(n^2 m)\).

**Compromise: Affine Gaps**

Thus, the compromise between time and accuracy is addressed by affine gaps. An affine gap penalty assigns a higher cost to the introduction of a gap than to the extension of a gap. This is represented by the following function:

\[
\gamma(n) = d + (n - 1) \times e
\]

The variable \( d \) represents the gap open, the first position where the gap is inserted, whereas \( e \) represents the gap extension. Common values for \( d \) tend to be -4 or -5, and common values for \( e \) tend to be -.25 or -.50 (in other words, \( d \) will always be a much greater value than \( e \)).

To compute optimal alignment, at position \((i, j)\), one needs to “remember” the best score if gap is open, and the best score if the gap is unopened. Thus, one needs to know whether the gap is new (use value \( d \)) or whether the gap is being extended (calculate cost with penalty value \( e \)). However, this last cannot be determined until the optimal alignment is calculated. Thus, the algorithm addresses this by keeping track of both sets of values.

In other words, one matrix remembers the best score if the gap is opened, and the other keeps track of the best score if the gap is unopened.

\[
F(i, j): \text{score of alignment } x_1 \ldots x_i \text{ to } y_1 \ldots y_j \text{ if } x_i \text{ aligns to } y_j
\]

\[
G(i, j): \text{score if } x_i \text{ aligns to a gap after } y_j
\]

\[
H(i, j): \text{score if } y_j \text{ aligns to a gap after } x_i
\]
The calculation would then proceed as follows:

If \( x_i \) aligns to \( y_j \) \( \rightarrow \) add -d  
Else if \( x_i \) aligns to a gap \( \rightarrow \) add -c

Since one needs to know if the current gap is a gap extension or a gap opening, two matrices are needed to keep one step back in history.

1. **Initialization:**
   \[
   V(i, 0) = d + (i - 1)\times e  
   V(0, j) = d + (j - 1)\times e
   \]

2. **Iteration:**
   \[
   V(i, j) = \max \{ F(i, j), G(i, j), H(i, j) \} 
   F(i, j) = V(i - 1, j - 1) + s(x_i, y_j)
   \]
   \[
   G(i, j) = \max \{ V(i - 1, j) - d, G(i - 1, j) - e \}
   \]
   \[
   H(i, j) = \max \{ V(i, j - 1) - d, H(i, j - 1) - e \}
   \]

3. **Termination:**
   similar

To generalize a little: think of how you would compute optimal alignment with this gap function… In time \( O(MN) \)?

![Graph](image)

Basically, given these four functions, one will need to fill up four times as many cells as previously.

The affine gap function below tries to model the convex gap function by making a linear approximation with two line segments. To increase accuracy, more line segments can be used, and as long as the number of line segments remains constant, the algorithm can be calculated in time of \( O(nm) \). However, because one would “need” to remember whether or not it belonged to a certain line segment, the number of matrices kept would be equal to the number of line segments.
1C. **Bounded Dynamic Programming**

If one assumes that we know x and y are very similar, one can exclude a large part of dynamic programming matrix. In other words, if the total number of gaps is bounded by a number k, cannot move further than k from the diagonal (one move left or right denotes a gap). In the matrix, each non-diagonal step denotes a gap length increase of one. Thus, the search can be narrowed by excluding all the squares that reside more than k steps away from the diagonal.

Thus, the only squares analysed will be those in the light green area below:

![Diagram of bounded dynamic programming](image)

2. **Linear Space Alignment**

Time should not be the only limiting factor when looking at the sequence alignment algorithms, the calculations of these could also be limited by space or memory. All the algorithms above considered alignments in a 2 dimensional matrix, of space complexity O(mn). However, if one were to align the two genomes of different organisms, each 1 billion pairs long, this would far exceed the RAM in current computers.

2A. **Hirschberg’s algorithm**

This algorithm searches for the longest common subsequence: given sequences s = s_1 s_2… s_m, t = t_1 t_2… t_p. Thus, to find the longest common subsequence u = u_1…u_k, the following function is defined:

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

By definition, this alignment will have the maximum number of matches (LCS).
It is easy to compute $F(M, N)$ in linear space, as to compute the scores of a particular column, one only needs the current column and previous column to yield the optimal score (a number).

However, by computing the score in linear space, one loses the ability to backtrack to obtain the optimal alignment as all pointer information is lost as one moves down the columns. (One can only find the optimal alignment through traceback after all the rows and columns have been filled).

To compute both the optimal score and the optimal alignment, one option would be to use a divide and conquer approach.

Let $x^R$ and $y^R$ denote the reverse of $x$ and $y$ (respectively). In other words, if $x = \text{abc}$, then $x^R = \text{cba}$.

Let $F^R(i, j)$ denote the optimal score of aligning $x^R$ and $y^R$, which is the as aligning the last $i$ characters if $x$ and the last $j$ characters of $y$.

As explained above, the matrix of $F^R(i, j)$ values can be computed in time $O(mn)$ and in linear space.

**Lemma 1:**

$$F(M, N) = \max_{k=0…N} \left( F\left(\frac{M}{2}, \ k\right) + F^R\left(\frac{M}{2}, \ N-k\right) \right)$$

This lemma merely expresses the state where the first half of $x$ is aligned from $k^*$ in $y$, and the second half is aligned to the point after $k^*$. In other words, the middle point of $x$ has to align to some point in $y$. Also, the lemma claims that $k^*$, the optimal breaking point in $y$, is obtained by maximizing over all possible ways of breaking the sequence $y$ into two parts (aligning the first with $x$ until the middle point, and the second part with $x$ after the middle point).

Find the best point in sequence $y$ to align the first half of $x$ to that point, second half of $x$ after that point, and adding those two (Find $k^*$, as the optimal breaking point in $y$). Now, using 2 columns of space, we can compute for $k = 1…M$, $F(m/2, k)$, $F(m/2, n-k)$, as well as the backpointers.

Then, knowing optimal $k^*$, one can keep track of the exact traceback of the columns. Also, one can trace path exiting column $M/2$ from $k^*$.

**Quick Overview**

Thus, first one finds the optimal alignment passing through the middle column (with both values and pointers), as shown in the figure below.
Then, the matrix is divided into four parts based on this point (k*). Alignments only move from upper left to lower right, therefore, the two sub matrices orthogonal to this are discarded. (Only space with possible alignments is calculated). In the figure below, the pink space would be discarded, whereas the algorithm would be called recursively on each of the green boxes.

![Diagram of matrix division](image)

The divide and conquer approach is used to calculate the optimal alignment. Thus, the algorithm can be calculated in linear space and quadratic time.

Hirschberg’s Algorithm

- Find middle point of sequence x → in quadratic time, linear space, one can find optimal F and F' for middle column
- Call the algorithm recursively on the left part
- Output the middle part of the alignment
- Call the algorithm recursively on the right part

*Time/space analysis of Hirschberg’s algorithm*

To compute optimal path at middle column (for box of size m*n):

- Space: 2N
- Time: cMN, for some constant c

Then, left, right calls cost c( M/2 * k* + M/2 * (N-k*) ) = cMN/2

All recursive calls cost:

- Total Time: cMN + cMN/2 + cMN/4 + .. = 2cMN
  - O(MN)
- Total space: O(N+M) to store the optimal alignment

Time: The total time cost will be O(mn)
Space: While the algorithm is running, one uses no more than O(n) space. However, the optimal alignment will be stored in O(n+m) space.
3. **Four Russian Algorithm**
Al'azarov, Dinič, Kronrod, Faradzhev, 1970 (although only one was truly a Russian).

The Four Russian Algorithm is a useful speedup of dynamic programming techniques. It depends upon the main observation that if there is a scoring scheme similar to the alignment one, consider a rectangle of the dynamic programming matrix (seen below). With the values of A, B, and C and substrings x' and y', D can be computed.

Thus, given some pre-computation, the above will allow one to move the matrix using smaller submatrices of the type above. Therefore, by dividing the matrix into t blocks (a t x t square of the dynamic programming matrix), one can precompute the t blocks to allow for faster progression through the large matrix.

Therefore, the size of the table in terms of D is $t^2$.

For each t block, one would compute D given the information above (the values of A, B, and C and substrings x' and y'). The total time costs for this would be equal to $O(N^2/\log^2 N) *$ cost of computing D.

*Lemma 1:*  
Two adjacent cells of $F(.,.)$ differ by at most 1

The lemma above represents the relationship between cells in the Needleman Wunsch matrix. The cell at the right or bottom of the other can be at most one worse than the other one, as $x_i$ can align to a letter or it can align to a gap. Cells can be related in only three ways, as seen below:
Proof of Lemma 1

In the first two cases (same row or same column):
Given that the sequence so far has been aligned, a gap can be added (score − 1), or the current letter was aligned to a gap (score + 1). Therefore, consecutive calls cannot differ by more than one.

In the case of the diagonal:
Given that the sequence so far has been aligned, the diagonal can signify either a match or a mismatch. However, since the movement is in the diagonal direction, no gaps have been inserted. Thus, the score will be either (score + 1) for a match, or (score − 1) for a mismatch.

Precomputation

The offset vector is a t-long vector with values {−1, 0, 1} (see Lemma above), where the first entry is 0. D can be found with the values at A, the top row and right column offset vectors, and the substrings of x and y for the block t.

The offset function is a function that computes the offset vectors of a bottom row and left column for a given t-block.

Precomputation takes a long time, for the computation of the offset function.

Values to precompute:
4^t possible strings given four bases, AGCT.
3^{2(t-1)} possible offset vectors
t^2 t-blocks

Therefore, there are 4^2t x 3^{2(t-1)} x t^2 values to precompute.

These values can then be looked up in a table in O(1) time, if we assume constant lookup RAM for log-sized inputs.

Then, letting t = O( log_3d N / 2 ), precomputation time can be simplified to O( N x log^2 N ), and runtime can be simplified to N^2 / t^c = O(N^2 / log^2 N). This is as best as one can do in running time for this type of dynamic programming based programs.

Alignment

The alignment is generated by dividing the matrix into t-blocks, and initializing all the values in the first row and column of the matrix. Thus, row by row, one can use the offset values at the leftmost column and the top row of each block, and the precomputed offset function to find the offset values at the rightmost column and bottom row. Let Q = total of offsets at row N, then F(N, N) = Q + F(N, 0).
Thus, the above explored two algorithms that go to the boundary of how well one can do in theory, in terms of space and time. However, these are more theoretical than practical, for although Hirschenberg’s algorithm used pretty often in practice, the Four Russian algorithm is not so commonly used.