6.047/6.878 Lecture 2: Sequence Alignment and Dynamic Programming

Guilherme Issao Fuijwara, Pete Kruskal (2007)
Arkajit Dey, Carlos Pards (2008)
Victor Costan, Marten van Dijk (2009)
Andreea Bodnari, Wes Brown (2010)
Sarah Spencer (2011)
Nathaniel Parrish (2012)

September 10, 2013

Contents

1 Introduction **3**

2 Aligning Sequences **3**
- 2.1 Example Alignment . . . 3
- 2.2 Solving Sequence Alignment . . . 3

3 Problem Formulations **5**
- 3.1 Formulation 1: Longest Common Substring . . . 5
- 3.2 Formulation 2: Longest Common Subsequence (LCS) . . . 6
- 3.3 Formulation 3: Sequence Alignment as Edit Distance . . . 7
- 3.4 Formulation 4: Varying Gap Cost Models . . . 7
- 3.5 Enumeration . . . 7

4 Dynamic Programming **8**
- 4.1 Theory of Dynamic Programming . . . 8
- 4.2 Fibonacci Numbers . . . 9
  - 4.2.1 The Naïve Solution . . . 9
  - 4.2.2 The Memoization Solution . . . 10
  - 4.2.3 The Dynamic Programming Solution . . . 10
- 4.3 Sequence Alignment using Dynamic Programming . . . 11

5 The Needleman-Wunsch Algorithm **11**
- 5.1 Dynamic programming vs. memoization . . . 11
- 5.2 Problem Statement . . . 11
- 5.3 Index space of subproblems . . . 12
- 5.4 Local optimality . . . 12
- 5.5 Optimal Solution . . . 13
- 5.6 Solution Analysis . . . 13
- 5.7 Needleman-Wunsch in practice . . . 13
- 5.8 Optimizations . . . 14
  - 5.8.1 Bounded Dynamic Programming . . . 14
  - 5.8.2 Linear Space Alignment . . . 16

6 Multiple alignment **17**
- 6.1 Aligning three sequences . . . 17
- 6.2 Heuristic multiple alignment . . . 17

7 Current Research Directions **18**

8 Further Reading **18**

9 Tools and Techniques **18**

10 What Have We Learned? **18**

11 Appendix **18**
- 11.1 Homology . . . 18
- 11.2 Natural Selection . . . 18
- 11.3 Dynamic Programming v. Greedy Algorithms . . . 19
- 11.4 Pseudocode for the Needleman-Wunsch Algorithm . . . 20

1 Introduction

Evolution has preserved functional elements in the genome. Such preserved elements between species are often homologs[1] – either orthologous or paralogous sequences (refer to Appendix 11.1). Orthologous gene sequences are of higher interest in the study of evolutionary paths due to the higher influence of purifying selection[2], since such regions are extremely well preserved. A common approach to the analysis of evolutionary similarities is the method of aligning sequences, primarily solved using computational methods (e.g., dynamic programming). These notes discuss the sequence alignment problem, the technique of dynamic programming, and a specific solution to the problem using this technique.

2 Aligning Sequences

Sequence alignment represents the method of comparing two or more genetic strands, such as DNA or RNA. These comparisons help with the discovery of genetic commonalities and with the (implicit) tracing of strand evolution. There are two main types of alignment:

- Global alignment: an attempt to align every element in a genetic strand, most useful when the genetic strands under consideration are of roughly equal size. Global alignment can also end in gaps.
- Local alignment: an attempt to align regions of sequences that contain similar sequence motifs within a larger context.

2.1 Example Alignment

Within orthologous gene sequences, there are islands of conservation, or relatively large stretches of nucleotides that are preserved between generations. These conserved regions typically imply functional elements and vice versa. As an example, we considered the alignment of the Gal10-Gal1 intergenic region for four different yeast species, the first cross-species whole genome alignment (Figure 1). As we look at this alignment, we note that some areas are more conserved than others. In particular, we note some small conserved motifs such as CGG and CGC, which in fact are functional elements in the binding of Gal4[7]. This example illustrates how we can read evolution to find functional elements.

We have to be cautious with our interpretations, however, because conservation does sometimes occur by random chance. In order to extract accurate biological information from sequence alignments we have to separate true signatures from noise. The most common approach to this problem involves modeling the evolutionary process. By using known codon substitution frequencies and RNA secondary structure constraints, for example, we can calculate the probability that evolution acted to preserve a biological function. See Chapter **??** for an in-depth discussion of evolutionary modeling and functional conservation in the context of genome annotation.

2.2 Solving Sequence Alignment

The genome changes over time, and, lacking a time machine, we cannot compare genomes of living species with their ancestors. Thus, we are limited to comparing just the genomes of living descendants. The goal of sequence alignment is to infer the 'edit operations' that change a genome by looking only at these endpoints.

We must make some assumptions when performing sequence alignment, if only because we must transform a biological problem into a computationally feasible one and we require a model with relative simplicity and tractability. In practice, the majority of sequence evolution occurs in the form of nucleotide mutations, deletions, and insertions (Figure 2). Thus, our sequence alignment model will only consider these three operations and will ignore other realistic events that occur with lower probability.

1. A nucleotide **mutation** occurs when some nucleotide in a sequence changes to some other nucleotide during the course of evolution.

[1] Homologous sequences are genomic sequences descended from a common ancestor.

[2] In the human genome, only $\approx 5\%$ of the nucleotides are under selection. Appendix 11.2 has more details on types of selection.

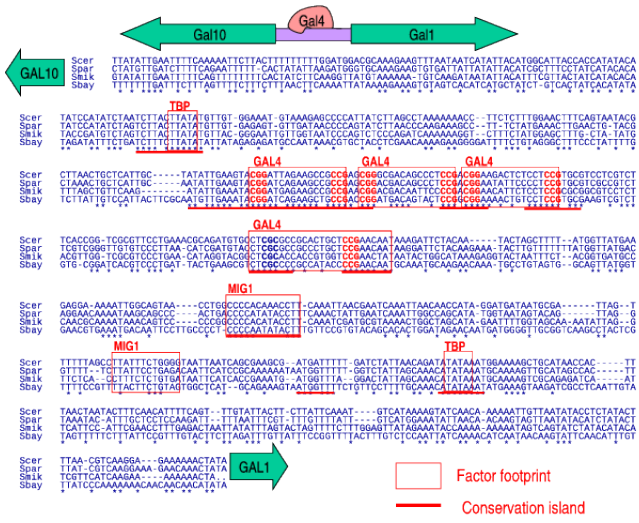

Figure 1: Sequence alignment of Gal10-Gal1 between four yeast strains. Asterisks mark conserved nucleotides.

2. A nucleotide **deletion** occurs when some nucleotide is deleted from a sequence during the course of evolution.

3. A nucleotide **insertion** occurs when some nucleotide is added to a sequence during the course of evolution.

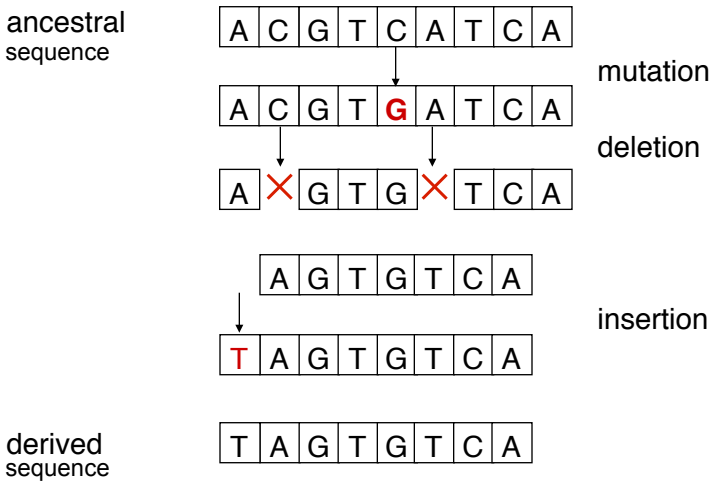

Figure 2: Evolutionary changes of a genetic sequence

Note that these three events are all reversible. For example, if a nucleotide N mutates into some nucleotide M, it is also possible that nucleotide M can mutate into nucleotide N. Similarly, if nucleotide N is deleted, the event may be reversed if nucleotide N is (re)inserted. Clearly, an insertion event is reversed by a corresponding deletion event.

This reversibility is part of a larger design assumption: time-reversibility. Specifically, any event in our model is reversible in time. For example, a nucleotide deletion going forward in time may be viewed as a nucleotide insertion going backward in time. This is useful because we will be aligning sequences which both exist in the present. In order to compare evolutionary relatedness, we will think of ourselves following one sequence backwards in time to a common ancestor and then continuing forward in time to the other sequence. In doing so, we can avoid the problem of not having an ancestral nucleotide sequence.

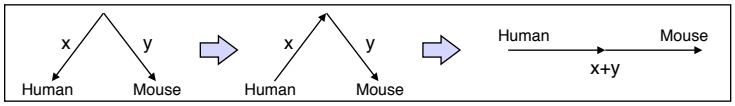

Figure 3: Aligning human to mouse sequences is analogous to tracing backward from the human to a common ancestor, then forward to the mouse

Note that time-reversibility is useful in solving some biological problems but does not actually apply to biological systems. For example, CpG (where p denotes the phosphate backbone in a DNA strand) may incorrectly pair with a TpG or CpA during DNA replication, but the reverse operation cannot occur; it is not time-reversible. To be very clear, time-reversibility is simply a design decision in our model; it is not inherent to the biology[3].

We also need some way to evaluate our alignments. There are many possible sequences of events that could change one genome into another. Perhaps the most obvious ones minimize the number of events (i.e., mutations, insertions, and deletions) between two genomes, but sequences of events in which many insertions are followed by corresponding deletions are also possible. We wish to establish an optimality criterion that allows us to pick the 'best' series of events describing changes between genomes.

We choose to invoke Occam's razor and select a maximum parsimony method as our optimality criterion. That is, in general, we wish to minimize the number of events used to explain the differences between two nucleotide sequences. In practice, we find that point mutations are more likely to occur than insertions and deletions, and certain mutations are more likely than others. Our parsimony method must take these and other inequalities into account when maximizing parsimony. This leads to the idea of a substitution matrix and a gap penalty, which are developed in the following sections. Note that we did not need to choose a maximum parsimony method for our optimality criterion. We could choose a probabilistic method, for example using Hidden Markov Models (HMMs), that would assign a probability measure over the space of possible event paths and use other methods for evaluating alignments (e.g., Bayesian methods). Note the duality between these two approaches; our maximum parsimony method reflects a belief that mutation events have low probability, thus in searching for solutions that minimize the number of events we are implicitly maximizing their likelihood.

3 Problem Formulations

In this section, we introduce a simple problem, analyze it, and iteratively increase its complexity until it closely resembles the sequence alignment problem. This section should be viewed as a warm-up for Section 5 on the Needleman-Wunsch algorithm.

3.1 Formulation 1: Longest Common Substring

As a first attempt, suppose we treat the nucleotide sequences as strings over the alphabet A, C, G, and T. Given two such strings, S1 and S2, we might try to align them by finding the longest common substring between them. In particular, these substrings cannot have gaps in them.

As an example, if S1 = ACGTCATCA and S2 = TAGTGTCA (refer to Figure 4), the longest common substring between them is GTCA. So in this formulation, we could align S1 and S2 along their longest common substring, GTCA, to get the most matches. A simple algorithm would be to try aligning S1 with different offsets of S2 and keeping track of the longest substring match found thus far. Note that this algorithm is quadratic in the lengths of S1 and S2, which is slower than we would prefer for such a simple problem.

[3]This is an example where understanding the biology helps the design greatly, and illustrates the general principle that success in computational biology requires strong knowledge of the foundations of both CS and biology. Warning: computer scientists who ignore biology will work too hard.

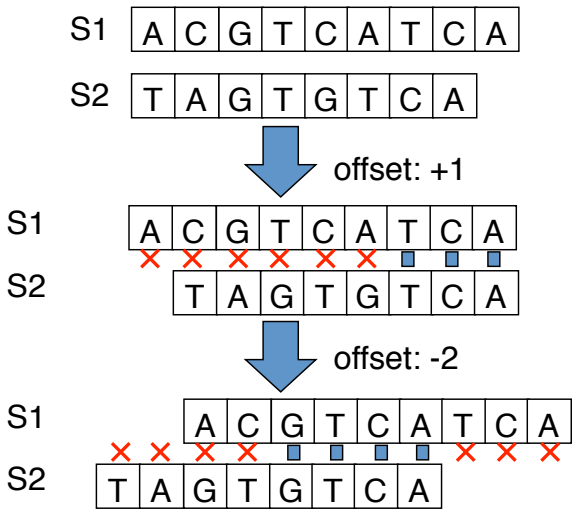

Figure 4: Example of longest common substring formulation

3.2 Formulation 2: Longest Common Subsequence (LCS)

Another formulation is to allow gaps in our subsequences and not just limit ourselves to substrings with no gaps. Given a sequence X = (x_1 ,..., x_m), we formally define $Z = (z_1, \ldots, z_k)$ to be a subsequence of X if there exists a strictly increasing sequence $i_1 < i_2 < \ldots < i_k$ of indices of X such that for all j, $1 \leq j \leq k$, we have $x_{i_j} = z_j$(CLRS 350-1).

In the longest common subsequence (LCS) problem, we're given two sequences X and Y and we want to find the maximum-length common subsequence Z. Consider the example of sequences S1 = ACGTCATCA and S2 = TAGTGTCA (refer to Figure 5). The longest common subsequence is AGTTCA, a longer match than just the longest common substring.

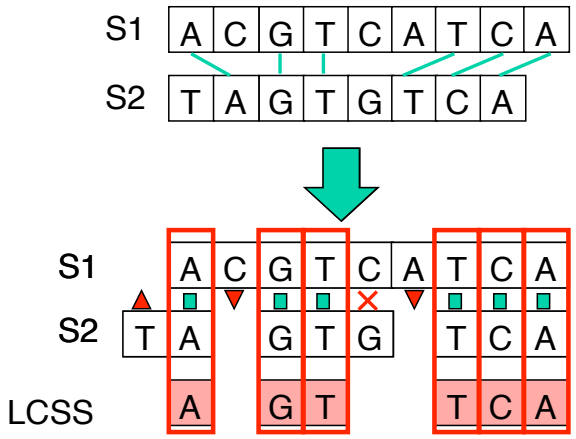

Figure 5: Example of longest common subsequence formulation

3.3 Formulation 3: Sequence Alignment as Edit Distance

The previous LCS formulation is close to the full sequence alignment problem, but so far we have not specified any cost functions that can differentiate between the three types of edit operations (insertion, deletions, and substitutions). Implicitly, our cost function has been uniform, implying that all operations are equally likely. Since substitutions are much more likely, we want to bias our LCS solution with a cost function that prefers substitutions over insertions and deletions.

We recast sequence alignment as a special case of the classic Edit-Distance[4] problem in computer science (CLRS 366). We add varying penalties for different edit operations to reflect biological occurrences. One biological reasoning for this scoring decision is the probabilities of bases being transcribed incorrectly during polymerization. Of the four nucleotide bases, A and G are purines (larger, two fused rings), while C and T are pyrimidines (smaller, one ring). Thus RNA polymerase[5] is much more likely to confuse two purines or two pyrimidines since they are similar in structure. The scoring matrix in Figure 6 models the considerations above. Note that the table is symmetric - this supports our time-reversible design.

	A	G	T	C
A	+1	-½	-1	-1
G	-½	+1	-1	-1
T	-1	-1	+1	-½
C	-1	-1	-½	+1

Figure 6: Cost matrix for matches and mismatches

Calculating the scores implies alternating between the probabilistic interpretation of how often biological events occur and the algorithmic interpretation of assigning a score for every operation. The problem is to the find the least expensive (as per the cost matrix) operation sequence which can transform the initial nucleotide sequence into the final nucleotide sequence.

3.4 Formulation 4: Varying Gap Cost Models

Biologically, the cost of creating a gap is more expensive than the cost of extending an already created gap[6]. Thus, we could create a model that accounts for this cost variation. There are many such models we could use, including the following:

- **Linear gap penalty:** Fixed cost for all gaps (same as formulation 3).
- **Affine gap penalty:** Impose a large initial cost for opening a gap, then a small incremental cost for each gap extension.
- **General gap penalty:** Allow any cost function. Note this may change the asymptotic runtime of our algorithm.
- **Frame-aware gap penalty:** Tailor the cost function to take into account disruptions to the coding frame.

3.5 Enumeration

Recall that in order to solve the Longest Common Substring formulation, we could simply enumerate all possible alignments, evaluate each one, and select the best. This was because there were only $O(n)$ alignments

[4]Edit-distance or Levenshtein distance is a metric for measuring the amount of difference between two sequences (e.g., the Levenshtein distance applied to two strings represents the minimum number of edits necessary for transforming one string into another).

[5]RNA polymerase is an enzyme that helps transcribe a nucleotide sequence into mRNA.

[6]Insertions and deletions (indels) are rare, therefore it is more likely that a single, longer indel will occur than multiple shorter indels.

of the two sequences. Once we allow gaps in our alignment, however, this is no longer the case. It is a known issue that the number of all possible gapped alignments cannot be enumerated (at least when the sequences are lengthy). For example, with two sequences of length 1000, the number of possible alignments exceeds the number of atoms in the universe.

Given a metric to score a given alignment, the simple brute-force algorithm enumerates all possible alignments, computes the score of each one, and picks the alignment with the maximum score. This leads to the question, 'How many possible alignments are there?' If you consider only NBAs[7] and $n > m$, the number of alignments is

$$\binom{n}{m} = \frac{(n+m)!}{n!m!} \approx \frac{(2n)!}{(n!)^2} \approx \frac{\sqrt{4\pi n}\frac{(2n)^{2n}}{e^{2n}}}{(\sqrt{2\pi n}\frac{(n)^n}{e^n})} = \frac{2^{2n}}{\sqrt{\pi n}} \tag{1}$$

For some small values of n such as 100, the number of alignments is already too big ($> 10^{60}$) for this enumeration strategy to be feasible. Thus, using a better algorithm than brute-force is a necessity.

4 Dynamic Programming

Before proceeding to a solution of the sequence alignment problem, we first discuss dynamic programming, a general and powerful method for solving problems with certain types of structure.

4.1 Theory of Dynamic Programming

Dynamic programming may be used to solve problems with:

1. **Optimal Substructure**: The optimal solution to an instance of the problem contains optimal solutions to subproblems.
2. **Overlapping Subproblems**: There are a limited number of subproblems, many/most of which are repeated many times.

Dynamic programming is usually, but not always, used to solve optimization problems, similar to greedy algorithms. Unlike greedy algorithms, which require a greedy choice property to be valid, dynamic programming works on a range of problems in which locally optimal choices do not produce globally optimal results. Appendix 11.3 discusses the distinction between greedy algorithms and dynamic programming in more detail; generally speaking, greedy algorithms solve a smaller class of problems than dynamic programming.

In practice, solving a problem using dynamic programming involves two main parts: Setting up dynamic programming and then performing computation. Setting up dynamic programming usually requires the following 5 steps:

1. Find a 'matrix' parameterization of the problem. Determine the number of dimensions (variables).
2. Ensure the subproblem space is polynomial (not exponential). Note that if a small portion of subproblems are used, then memoization may be better; similarly, if subproblem reuse is not extensive, dynamic programming may not be the best solution for the problem.
3. Determine an effective transversal order. Subproblems must be ready (solved) when they are needed, so computation order matters.
4. Determine a recursive formula: A larger problem is typically solved as a function of its subparts.
5. Remember choices: Typically, the recursive formula involves a minimization or maximization step. Moreover, a representation for storing transversal pointers is often needed, and the representation should be polynomial.

Once dynamic programming is setup, computation is typically straight-forward:

1. Systematically fill in the table of results (and usually traceback pointers) and find an optimal score.
2. Traceback from the optimal score through the pointers to determine an optimal solution.

[7] Non-Boring Alignments, or alignments where gaps are always paired with nucleotides.

4.2 Fibonacci Numbers

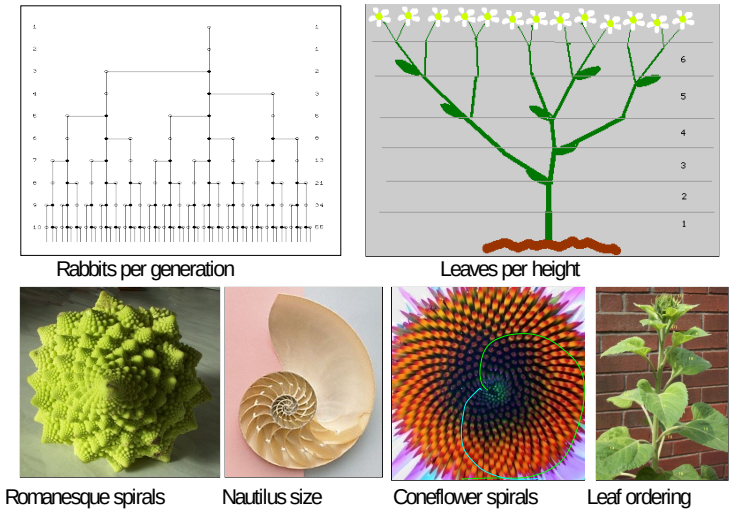

Figure 7: Examples of Finonacci numbers in nature are ubiquitous.

The Fibonacci numbers provide an instructive example of the benefits of dynamic programming. The Fibonacci sequence is recursively defined as $F_0 = F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for $n \leq 2$. We develop an algorithm to compute the n^{th} Fibonacci number, and then refine it first using memoization and later using dynamic programming to illustrate key concepts.

4.2.1 The Naïve Solution

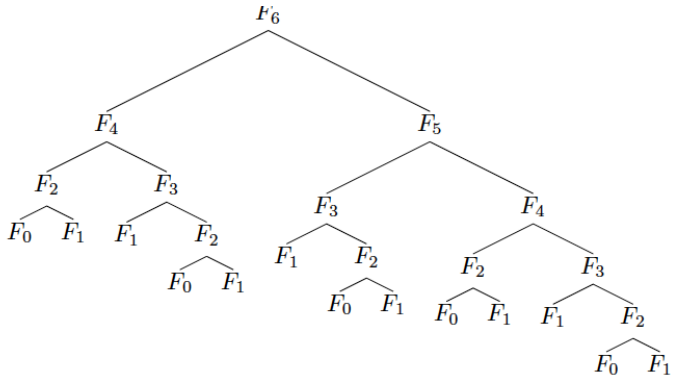

Figure 8: The recursion tree for the fib procedure showing repeated subproblems. The size of the tree is $O(\phi(n))$, where ϕ is the golden ratio.

The simple top-down approach is to just apply the recursive definition. Listing 1 shows a simple Python implementation.

Listing 1: Python implementation for computing Fibonacci numbers recursively.

```
# Assume n is a non-negative integer.
def fib(n):
    if n==0 or n==1:
```

```
    return 1
  else:
    return fib(n-1) + fib(n-2)
```

But this top-down algorithm runs in exponential time. That is, if $T(n)$ is how long it takes to compute the n^{th} Fibonacci number, we have that $T(n) = T(n-1) + T(n-2)$, so $T(n) = O(\phi^n)$ [8]. The problem is that we are repeating work by solving the same subproblem many times.

4.2.2 The Memoization Solution

A better solution that still utilizes the top-down approach is to memoize the answers to the subproblems. Listing 2 gives a Python implementation that uses memoization.

Listing 2: Python implementation for computing Fibonacci numbers using memoization.

```
# Assume n is a non-negative integer.
fibs = {0:1,1:1} # stores subproblem answers
def fib(n):
  if not(n in fibs):
    x = fib(n-2)
    y = fib(n-1)
    fibs[n] = x+y
  return fibs[n]
```

Note that this implementation now runs in $T(n) = O(n)$ time because each subproblem is computed at most once.

4.2.3 The Dynamic Programming Solution

For calculating the n^{th} Fibonacci number, instead of beginning with $F(n)$ and using recursion, we can start computation from the bottom since we know we are going to need all of the subproblems anyway. In this way, we will omit much of the repeated work that would be done by the naïve top-down approach, and we will be able to compute the n^{th} Fibonacci number in $O(n)$ time.

As a formal exercise, we can apply the steps outlined in section 4.1:

1. **Find a 'matrix' parameterization:** In this case, the matrix is one-dimensional; there is only one parameter to any subproblem $F(x)$.
2. **Ensure the subproblem space is polynomial:** Since there are only $n-1$ subproblems, the space is polynomial.
3. **Determine an effective transversal order:** As mentioned above, we will apply a bottom-up transversal order (that is, compute the subproblems in order).
4. **Determine a recursive formula:** This is simply the well-known recurrance $F(n) = F(n-1)+F(n-2)$.
5. **Remember choices:** In this case there is nothing to remember, as no choices were made in the recursive formula.

Listing 3 shows a Python implementation of this approach.

Listing 3: Python implementation for computing Fibonacci numbers iteratively using dynamic programming.

```
# Assume n is a non-negative integer
def fib(n):
  x = y = 1
  for i in range(1,n):
    x, y = y, x + y
  return x
```

[8] ϕ is the **golden ratio**, i.e. $\frac{1+\sqrt{5}}{2}$

This method is optimized to only use constant space instead of an entire table since we only need the answer to each subproblem once. But in general dynamic programming solutions, we want to store the solutions to subproblems in a table since we may need to use them multiple times without recomputing their answers. Such solutions would look somewhat like the memoization solution in Listing 2, but they will generally be bottom-up instead of top-down. In this particular example, the distinction between the memoization solution and the dynamic programming solution is minimal as both approaches compute all subproblem solutions and use them the same number of times. In general, memoization is useful when not all subproblems will be computed, while dynamic programming may not have as much overhead as memoization when all subproblem solutions must be calculated. Additional dynamic programming examples may be found online [6].

4.3 Sequence Alignment using Dynamic Programming

We are now ready to solve the more difficult problem of sequence alignment using dynamic programming, which is presented in depth in the next section. Note that the key insight in solving the sequence alignment problem is that alignment scores are additive. This allows us to create a matrix M indexed by i and j, which are positions in two sequences S and T to be aligned. The best alignment of S and T corresponds with the best path through the matrix M after it is filled in using a recursive formula.

By using dynamic programming to solve the sequence alignment problem, we achieve a provably optimal solution that is far more tractable than brute-force enumeration.

5 The Needleman-Wunsch Algorithm

We will now use dynamic programming to tackle the harder problem of general sequence alignment. Given two strings S $=(S_1, \ldots, S_n)$ and T $=(T_1, \ldots, T_m)$, we want to find the longest common subsequence, which may or may not contain gaps. Rather than maximizing the length of a common subsequence we want to compute the common subsequence that optimizes the score as defined by our scoring function. Let d denote the gap penalty cost and s(x; y) the score of aligning a base x and a base y. These are inferred from insertion/deletion and substitution probabilities which can be determined experimentally or by looking at sequences that we know are closely related. The algorithm we will develop in the following sections to solve sequence alignment is known as the Needleman-Wunsch algorithm.

5.1 Dynamic programming vs. memoization

Before we dive into the algorithm, a final note on memoization[9] is in order. Much like the Fibonacci problem, the sequence alignment problem can be solved in either a top-down or bottom-up approach.

In a *top-down recursive approach* we can use memoization to create a potentially large dictionary indexed by each of the subproblems that we are solving (aligned sequences). This needs $O(n^2m^2)$ space if we index each subproblem by the starting and end points of the subsequences for which an optimal alignment needs to be computed. The advantage is that we solve each subproblem at most once: if it is not in the dictionary, the problem gets computed and then inserted into dictionary for further reference.

In a *bottom-up iterative approach* we can use dynamic programming. We define the order of computing sub-problems in such a way that a solution to a problem is computed once the relevant sub-problems have been solved. In particular, simpler sub-problems will come before more complex ones. This removes the need for keeping track of which sub-problems have been solved (the dictionary in memoization turns into a matrix) and ensures that there is no duplicated work (each sub-alignment is computed only once).

5.2 Problem Statement

Suppose we have an optimal alignment for two sequences S and T in which S_i matches T_j. The key insight is that this optimal alignment is composed of an optimal alignment between $(S_1, \ldots, S_{i-1})$ and $(T_1, \ldots, T_{j-1})$

[9]Memoization is a technique that reduces processing time by storing the answers to calculations or subproblems that are called many times.

and an optimal alignment between $(S_{i+1}, \ldots, S_n)$ and $(T_{j+1}, \ldots, T_m)$. This follows from a cut-and-paste argument: if one of these partial alignments is suboptimal, then we cut-and-paste a better alignment in place of the suboptimal one. This achieves a higher score of the overall alignment and thus contradicts the optimality of the initial global alignment. In other words, every subpath in an optimal path must also be optimal. Notice that the scores are additive, so the score of the overall alignment equals the addition of the scores of the alignments of the subsequences. This implicitly assumes that the sub-problems of computing the optimal scoring alignments of the subsequences are independent. We need to biologically motivate that such an assumption leads to meaningful results.

5.3 Index space of subproblems

We now need to index the space of subproblems. Let $F_{i,j}$ be the score of the optimal alignment of $(S_1, \ldots, S_i)$ and $(T_1, \ldots, T_j)$. The space of subproblems is $\{F_{i,j}\ \forall i \in [1, |S|], \forall j \in [1, |T|]\}$. This allows us to maintain an $m \times n$ matrix F with the solutions (i.e. optimal scores) for all the subproblems.

5.4 Local optimality

We can compute the optimal solution for a subproblem by making a locally optimal choice based on the results from the smaller sub-problems. Thus, we need to establish a recursive function that shows how the solution to a given problem depends on its subproblems. And we use this recursive definition to fill up the table F in a bottom-up fashion.

We can consider the 4 possibilities (insert, delete, substitute, match) and evaluate each of them based on the results we have computed for smaller subproblems. To initialize the table, we set $F_{0,j} = -j \cdot d$ $F_{i,0} = -i \cdot d$ since those are the scores of aligning $(T_1, \ldots, T_j\ )$ with j gaps and $(S_1, \ldots, S_i)$ with i gaps (aka zero overlap between the two sequences). Then we traverse the matrix column by column computing the optimal score for each alignment subproblem by considering the four possibilities:

- Sequence S has a gap at the current alignment position.
- Sequence T has a gap at the current alignment position.
- There is a mutation (nucleotide substitution) at the current position.
- There is a match at the current position.

We then use the possibility that produces the maximum score. We express this mathematically by the recursive formula for $F_{i,j}$:

$$\begin{aligned} \textit{Initialization} \quad &: \quad \begin{array}{l} F(0,0) = 0 \\ F(i,0) = F(i-1,0) - d \\ F(0,j) = F(0,j-1) - d \end{array} \\ \textit{Iteration} \quad &: \quad F(i,j) = \max \begin{cases} F(i-1,j) - d & \text{insert gap in } S \\ F(i,j-1) - d & \text{insert gap in } T \\ F(i-1,j-1) + s(x_i, y_j) & \text{match or mutation} \end{cases} \\ \textit{Termination} \quad &: \quad \textit{Bottom right} \end{aligned}$$

After traversing the matrix, the optimal score for the global alignment is given by $F_{m,n}$. The traversal order needs to be such that we have solutions to given subproblems when we need them. Namely, to compute $F_{i,j}$, we need to know the values to the left, up, and diagonally above $F_{i,j}$ in the table. Thus we can traverse the table in row or column major order or even diagonally from the top left cell to the bottom right cell. Now, to obtain the actual alignment we just have to remember the choices that we made at each step.

5.5 Optimal Solution

Note the duality between paths through the matrix F and sequence alignments. In evaluating each cell $F_{i,j}$ we make a choice by selecting the maximum of the three possibilities. Thus the value of each (uninitialized) cell in the matrix is determined either by the cell to its left, above it, or diagonally to the left above it. A match and a substitution are both represented as traveling in the diagonal direction; however, a different cost can be applied for each, depending on whether the two base pairs we are aligning match or not. To construct the actual optimal alignment, we need to traceback through our choices in the matrix. It is helpful to maintain a pointer for each cell while filling up the table that shows which choice was made to get the score for that cell. Then we can just follow our pointers backwards to reconstruct the optimal alignment.

5.6 Solution Analysis

The runtime analysis of this algorithm is very simple. Each update takes $O(1)$ time, and since there are mn elements in the matrix F, the total running time is $O(mn)$. Similarly, the total storage space is $O(mn)$. For the more general case where the update rule is more complicated, the running time may be more expensive. For instance, if the update rule requires testing all sizes of gaps (e.g. the cost of a gap is not linear), then the running time would be $O(mn(m+n))$.

5.7 Needleman-Wunsch in practice

Assume we want to align two sequences S and T, where

S = AGT

T = AAGC

The first step is placing the two sequences along the margins of a matrix and initializing the matrix cells. To initialize we assign a 0 to the first entry in the matrix and then fill in the first row and column based on the incremental addition of gap penalties, as in Figure 9 below. Although the algorithm could fill in the first row and column through iteration, it is important to clearly define and set boundaries on the problem.

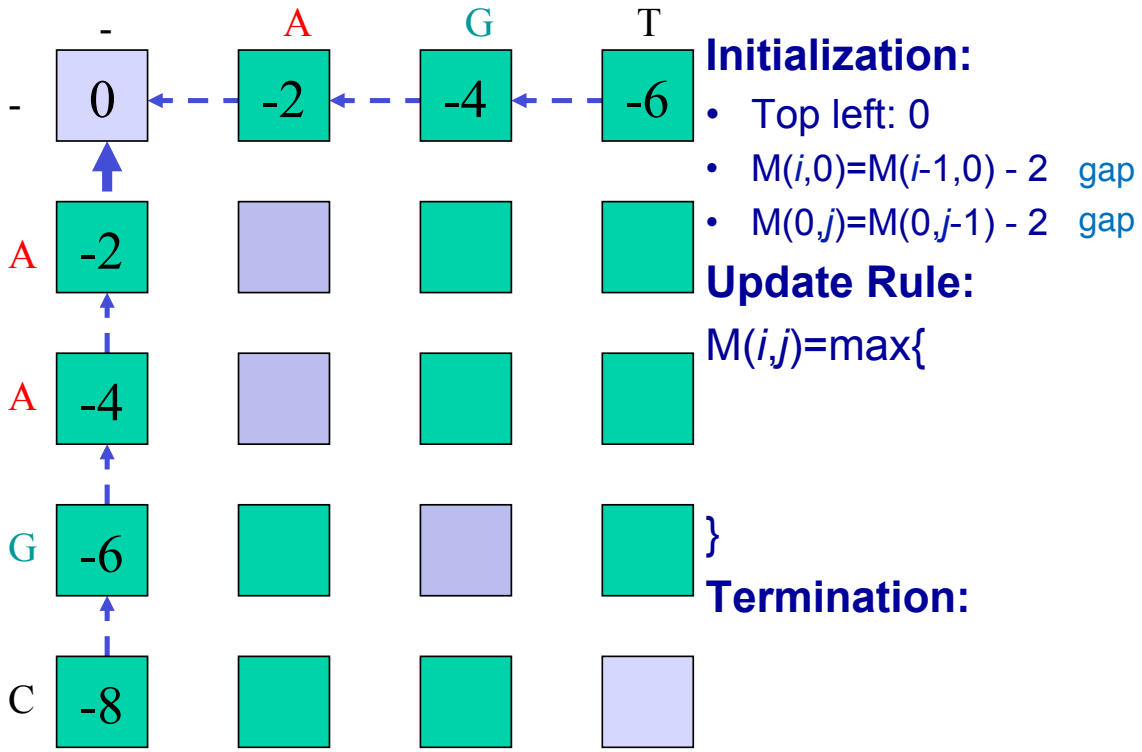

Figure 9: (Example) Initial setup for Needleman-Wunsch

The next step is iteration through the matrix. The algorithm proceeds either along rows or along columns, considering one cell at time. For each cell three scores are calculated, depending on the scores of three adjacent matrix cells (specifically the entry above, the one diagonally up and to the left, and the one to the left). The maximum score of these three possible tracebacks is assigned to the entry and the corresponding pointer is also stored. Termination occurs when the algorithm reaches the bottom right corner. In Figure 10 the alignment matrix for sequences S and T has been filled in with scores and pointers.

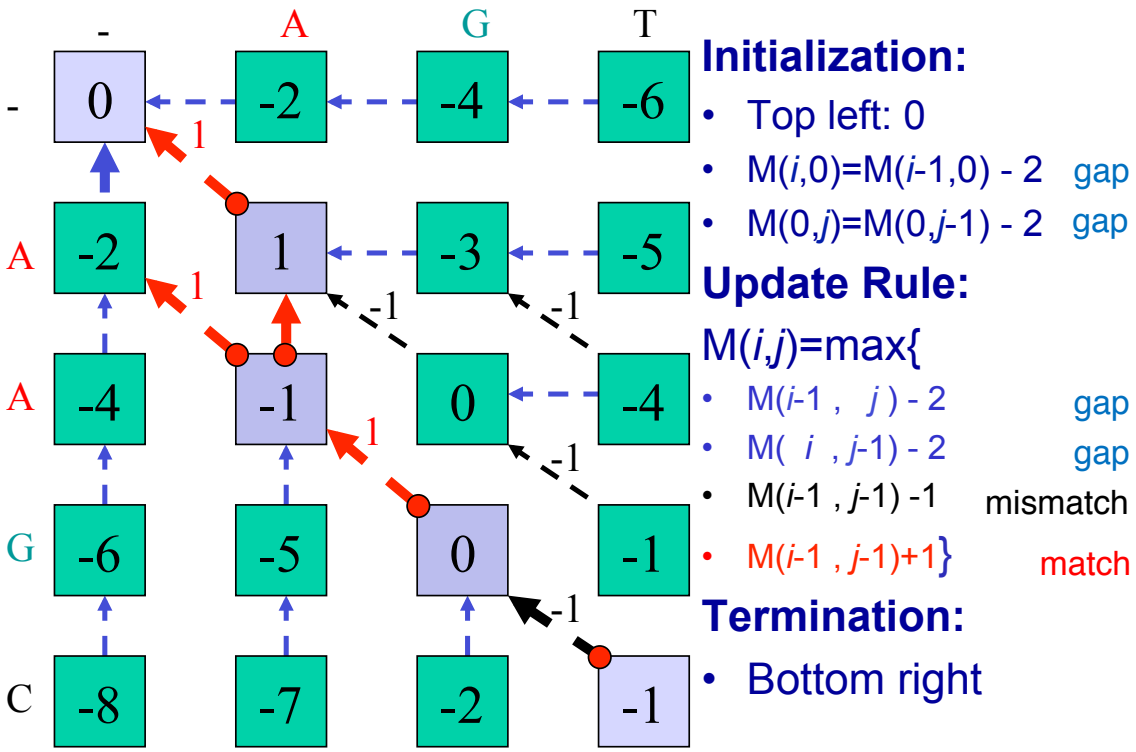

Figure 10: (Example) Half-way through the second step of Needleman-Wunsch

The final step of the algorithm is optimal path traceback. In our example we start at the bottom right corner and follow the available pointers to the top left corner. By recording the alignment decisions made at each cell during traceback, we can reconstruct the optimal sequence alignment from end to beginning and then invert it. Note that in this particular case, multiple optimal pathways are possible (Figure 11). A pseudocode implementation of the Needleman-Wunsch algorithm is included in Appendix 11.4

5.8 Optimizations

The algorithm we presented is much faster than the brute-force strategy of enumerating alignments and it performs well for sequences up to 10 kilo-bases long. Nevertheless, at the scale of whole genome alignments the algorithm given is not feasible. In order to align much larger sequences we can make modifications to the algorithm and further improve its performance.

5.8.1 Bounded Dynamic Programming

One possible optimization is to ignore Mildly Boring Alignments (MBAs), or alignments that have too many gaps. Explicitly, we can limit ourselves to stay within some distance W from the diagonal in the matrix F of subproblems. That is, we assume that the optimizing path in F from $F_{0,0}$ to $F_{m,n}$ is within distance W along the diagonal. This means that recursion (2) only needs to be applied to the entries in F within distance W around the diagonal, and this yields a time/space cost of $O((m+n)W)$ (refer to Figure 12).

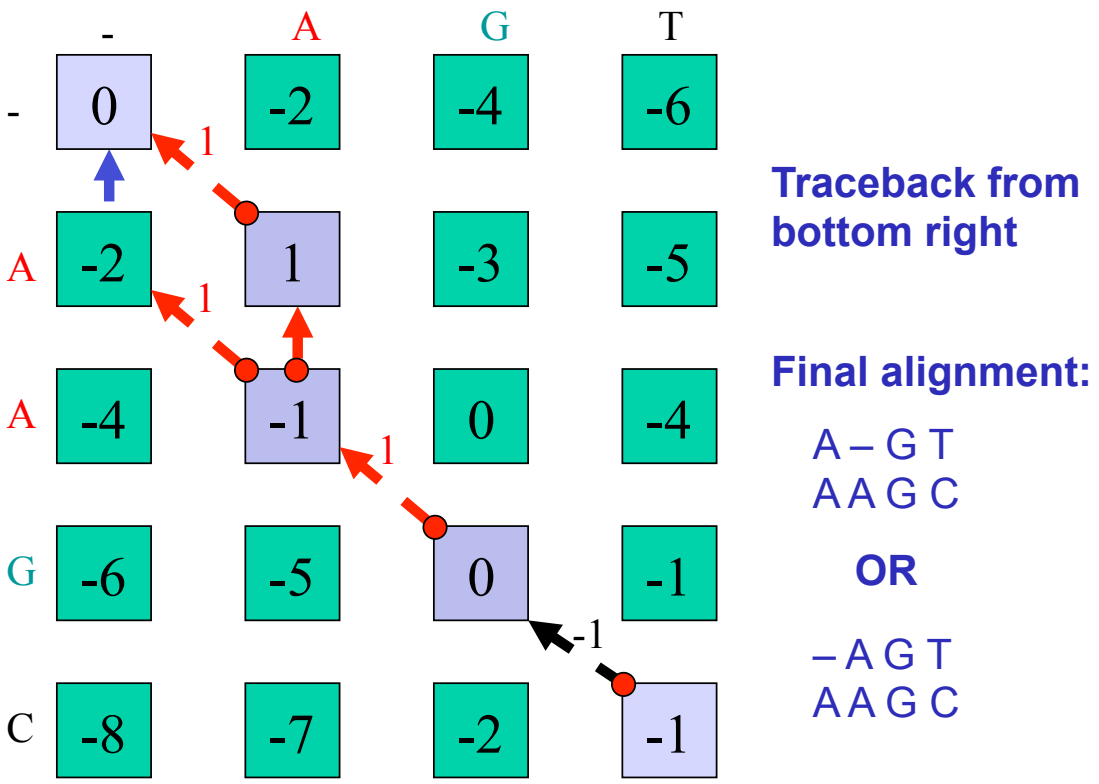

Figure 11: (Example) Tracing the optimal alignment

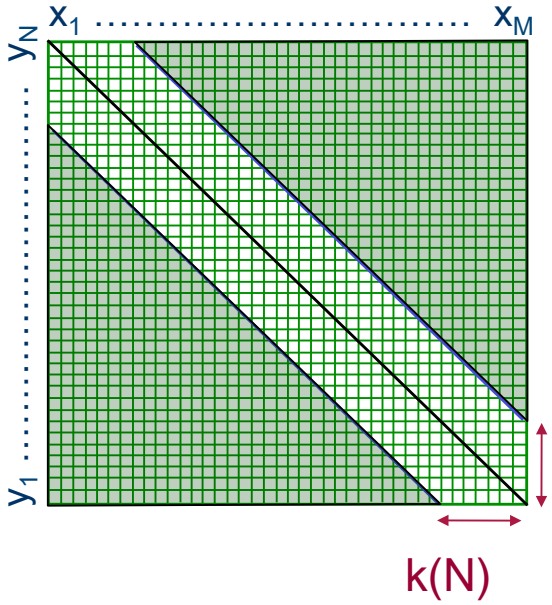

Figure 12: Bounded dynamic programming example

Note, however, that this strategy is heuristic and no longer guarantees an optimal alignment. Instead it attains a lower bound on the optimal score. This can be used in a subsequent step where we discard the recursions in matrix F which, given the lower bound, cannot lead to an optimal alignment.

5.8.2 Linear Space Alignment

Recursion (2) can be solved using only linear space: we update the columns in F from left to right during which we only keep track of the last updated column which costs $O(m)$ space. However, besides the score $F_{m,n}$ of the optimal alignment, we also want to compute a corresponding alignment. If we use trace back, then we need to store pointers for each of the entries in F, and this costs $O(mn)$ space.

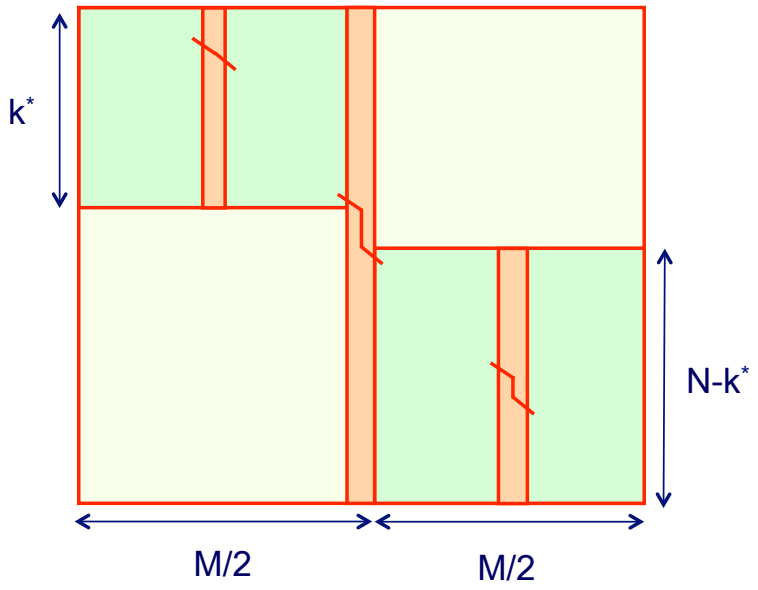

Figure 13: Recovering the sequence alignment with $O(m+n)$ space

It is also possible to find an optimal alignment using only linear space! The goal is to use divide and conquer in order to compute the structure of the optimal alignment for one matrix entry in each step. Figure 13 illustrates the process. The key idea is that a dynamic programming alignment can proceed just as easily in the reverse direction, starting at the bottom right corner and terminating at the top left. So if the matrix is divided in half, then both a forward pass and a reverse pass can run at the same time and converge in the middle column. At the crossing point we can add the two alignment scores together; the cell in the middle column with the maximum score must fall in the overall optimal path.

We can describe this process more formally and quantitatively. First compute the row index $u \in \{1,\ldots,m\}$ that is on the optimal path while crossing the $\frac{n}{2}^{\text{th}}$ column. For $1 \leq i \leq m$ and $\frac{n}{2} \leq j \leq n$ let $C_{i,j}$ denote the row index that is on the optimal path to $F_{i,j}$ while crossing the $\frac{n}{2}^{\text{th}}$ column. Then, while we update the columns of F from left to right, we can also update the columns of C from left to right. So, in $O(mn)$ time and $O(m)$ space we are able to compute the score $F_{m,n}$ and also $C_{m,n}$, which is equal to the row index $u \in \{1,\ldots,m\}$ that is on the optimal path while crossing the $\frac{n}{2}^{\text{th}}$ column.

Now the idea of divide and conquer kicks in. We repeat the above procedure for the upper left $u \times \frac{n}{2}$ submatrix of F and also repeat the above procedure for the lower right $(m-u) \times \frac{n}{2}$ submatrix of F . This can be done using $O(m+n)$ allocated linear space. The running time for the upper left submatrix is $O(\frac{un}{2})$ and the running time for the lower right submatrix is $O(\frac{(m-u)n}{2})$, which added together gives a running time of $O(\frac{mn}{2}) = O(mn)$.

We keep on repeating the above procedure for smaller and smaller submatrices of F while we gather more and more entries of an alignment with optimal score. The total running time is $O(mn)+O(\frac{mn}{2})+O(\frac{mn}{4})+\ldots = O(2mn) = O(mn)$. So, without sacrificing the overall running time (up to a constant factor), divide and conquer leads to a linear space solution.

6 Multiple alignment

6.1 Aligning three sequences

Now that we have seen how to align a pair of sequences, it is natural to extend this idea to *multiple* sequences. Suppose we would like to find the optimal alignment of 3 sequences. How might we proceed?

Recall that when we align two sequences S and T, we choose the maximum of three possibilities for the final position of the alignment (sequence T aligned against a gap, sequence S aligned against a gap, or sequence S aligned against sequence T):

$$F_{i,j} = \max \begin{cases} F_{i,j-1} + d \\ F_{i-1,j} + d \\ F_{i-1,j-1} + s(S_i, T_j) \end{cases}$$

For three sequences S,T, and U, there are seven possibilities for the final position of the alignment. That is, there are three ways to have two gaps in the final position, three ways to have one gap, and one way to have all three sequences aligned ($\binom{3}{1} + \binom{3}{2} + \binom{3}{3} = 7$). The update rule is now:

$$F_{i,j,k} = \max \begin{cases} F_{i-1,j,k} + s(S_i, -, -) \\ F_{i,j-1,k} + s(-, T_j, -) \\ F_{i,j,k-1} + s(-, -, U_k) \\ F_{i-1,j-1,k} + s(S_i, T_j, -) \\ F_{i-1,j,k-1} + s(S_i, -, U_k) \\ F_{i,j-1,k-1} + s(-, T_j, U_k) \\ F_{i-1,j-1,k-1} + s(S_i, T_j, U_k) \end{cases}$$

where s is the function describing gap, match, and mismatch scores.

This approach, however, is exponential in the number of sequences we are aligning. If we have k sequences of length n, computing the optimal alignment using a k-dimensional dynamic programming matrix takes $O((2n)^k)$ time (the factor of 2 results from the fact that a k-cube has 2^k vertices, so we need to take the maximum of $2^k - 1$ neighboring cells for each entry in the score matrix). As you can imagine, this algorithm quickly becomes impractical as the number of sequences increases.

6.2 Heuristic multiple alignment

One commonly used approach for multiple sequence alignment is called *progressive multiple alignment*. Assume that we know the evolutionary tree relating each of our sequences. Then we begin by performing a pairwise alignment of the two most closely-related sequences. This initial alignment is called the *seed alignment*. We then proceed to align the next closest sequence to the seed, and this new alignment replaces the seed. This process continues until the final alignment is produced.

In practice, we generally do not know the evolutionary tree (or *guide tree*), this technique is usually paired with some sort of clustering algorithm that may use a low-resolution similarity measure to generate an estimation of the tree.

While the running time of this heuristic approach is much improved over the previous method (polynomial in the number of sequences rather than exponential), we can no longer guarantee that the final alignment is optimal.

Note that we have not yet explained how to align a sequence against an existing alignment. One possible approach would be to perform pairwise alignments of the new sequence with each sequence already in the seed alignment (we assume that any position in the seed alignment that is already a gap will remain one). Then we can add the new sequence onto the seed alignment based on the best pairwise alignment (this approach was previously described by Feng and Doolittle[4]). Alternatively, we can devise a function for scoring the alignment of a sequence with another alignment (such scoring functions are often based on the pairwise sum of the scores at each position).

Design of better multiple sequence alignment tools is an active area of research. Section 9 details some of the current work in this field.

7 Current Research Directions

8 Further Reading

9 Tools and Techniques

Lalign finds local alignments between two sequences. Dotlet is a browser-based Java applet for visualizing the alignment of two sequences in a dot-matrix.

The following tools are available for multiple sequence alignment:

- Clustal Omega - A multiple sequence alignment program that uses seeded guide trees and HMM profile-profile techniques to generate alignments.[9]
- MUSCLE - MUltiple Sequence Comparison by Log-Expectation[3]
- T-Coffee - Allows you to combine results obtained with several alignment methods[2]
- MAFFT - (Multiple Alignment using Fast Fourier Transform) is a high speed multiple sequence alignment program[5]
- Kalign - A fast and accurate multiple sequence alignment algorithm[8]

10 What Have We Learned?

11 Appendix

11.1 Homology

One of the key goals of sequence alignment is to identify homologous sequences (e.g., genes) in a genome. Two sequences that are homologous are evolutionarily related, specifically by descent from a common ancestor. The two primary types of homologs are orthologous and paralogous (refer to Figure 14[10]). Other forms of homology exist (e.g., xenologs), but they are outside the scope of these notes.

Orthologs arise from speciation events, leading to two organisms with a copy of the same gene. For example, when a single species A speciates into two species B and C, there are genes in species B and C that descend from a common gene in species A, and these genes in B and C are orthologous (the genes continue to evolve independent of each other, but still perform the same relative function).

Paralogs arise from duplication events within a species. For example, when a gene duplication occurs in some species A, the species has an original gene B and a gene copy B′, and the genes B and B′ are paralogus.

Generally, orthologous sequences between two species will be more closely related to each other than paralogous sequences. This occurs because orthologous typically (although not always) preserve function over time, whereas paralogous often change over time, for example by specializing a gene's (sub)function or by evolving a new function. As a result, determining orthologous sequences is generally more important than identifying paralogous sequences when gauging evolutionary relatedness.

11.2 Natural Selection

The topic of natural selection is a too large topic to summarize effectively in just a few short paragraphs; instead, this appendix introduces three broad types of natural selection: positive selection, negative selection, and neutral selection.

- *Positive selection* occurs when a trait is evolutionarily advantageous and increases an individual's fitness, so that an individual with the trait is more likely to have (robust) offspring. It is often associated with the development of new traits.

[10] R.B. - BIOS 60579

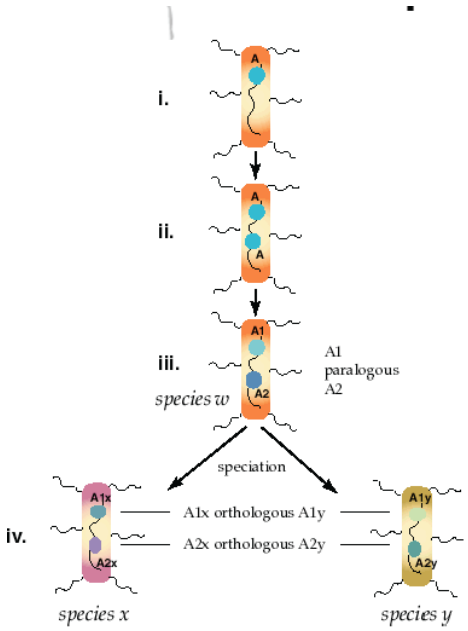

Figure 14: Ortholog and paralog sequences

- *Negative selection* occurs when a trait is evolutionarily disadvantageous and decreases an individual's fitness. Negative selection acts to reduce the prevalence of genetic alleles that reduce a species' fitness. Negative selection is also known as purifying selection due to its tendency to 'purify' genetic alleles until only a single allele exists in the population.
- *Neutral selection* describes evolution that occurs randomly, as a result of alleles not affecting an individual's fitness. In the absence of selective pressures, no positive or negative selection occurs, and the result is neutral selection.

11.3 Dynamic Programming v. Greedy Algorithms

Dynamic programming and greedy algorithms are somewhat similar, and it behooves one to know the distinctions between the two. Problems that may be solved using dynamic programming are typically optimization problems that exhibit two traits:

1. **optimal substructure** and
2. **overlapping subproblems**.

Problems solvable by greedy algorithms require both these traits as well as (3) the **greedy choice property**. When dealing with a problem "in the wild," it is often easy to determine whether it satisfies (1) and (2) but difficult to determine whether it must have the greedy choice property. It is not always clear whether locally optimal choices will yield a globally optimal solution.

For computational biologists, there are two useful points to note concerning whether to employ dynamic programming or greedy programming. First, if a problem may be solved using a greedy algorithm, then it may be solved using dynamic programming, while the converse is not true. Second, the problem structures that allow for greedy algorithms typically do not appear in computational biology.

To elucidate this second point, it could be useful to consider the structures that allow greedy programming to work, but such a discussion would take us too far afield. The interested student (preferably one with a mathematical background) should look at matroids and greedoids, which are structures that have the greedy choice property. For our purposes, we will simply state that biological problems typically involve entities that are highly systemic and that there is little reason to suspect sufficient structure in most problems to employ greedy algorithms.

11.4 Pseudocode for the Needleman-Wunsch Algorithm

The first problem in the first problem set asks you to finish an implementation of the Needleman-Wunsch (NW) algorithm, and working Python code for the algorithm is intentionally omitted. Instead, this appendix summarizes the general steps of the NW algorithm (Section 5) in a single place.

Problem: Given two sequences S and T of length m and n, a substitution matrix vU of matching scores, and a gap penalty G, determine the optimal alignment of S and T and the score of the alignment.

Algorithm:

1. Create two $m+1$ by $n+1$ matrices A and B. A will be the scoring matrix, and B will be the traceback matrix. The entry (i, j) of matrix A will hold the score of the optimal alignment of the sequences $S[1, \ldots, i]$ and $T[1, \ldots, j]$, and the entry (i, j) of matrix B will hold a pointer to the entry from which the optimal alignment was built.

2. Initialize the first row and column of the score matrix A such that the scores account for gap penalties, and initialize the first row and column of the traceback matrix B in the obvious way.

3. Go through the entries (i, j) of matrix A in some reasonable order, determining the optimal alignment of the sequences $S[1, \ldots, i]$ and $T[1, \ldots, j]$ using the entries $(i-1, j-1)$, $(i-1, j)$, and $(i, j-1)$. Set the pointer in the matrix B to the corresponding entry from which the optimal alignment at (i, j) was built.

4. Once all entries of matrices A and B are completed, the score of the optimal alignment may be found in entry (m, n) of matrix A.

5. Construct the optimal alignment by following the path of pointers starting at entry (m, n) of matrix B and ending at entry $(0, 0)$ of matrix B.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, London, third edition, 1964.

[2] Paolo Di Tommaso, Sebastien Moretti, Ioannis Xenarios, Miquel Orobitg, Alberto Montanyola, Jia-Ming Chang, Jean-François Taly, and Cedric Notredame. T-Coffee: a web server for the multiple sequence alignment of protein and RNA sequences using structural information and homology extension. *Nucleic Acids Research*, 39(Web Server issue):W13–W17, 2011.

[3] Robert C Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research*, 32(5):1792–7, January 2004.

[4] D F Feng and R F Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360, 1987.

[5] Kazutaka Katoh, George Asimenos, and Hiroyuki Toh. Multiple alignment of DNA sequences with MAFFT. *Methods In Molecular Biology Clifton Nj*, 537:39–64, 2009.

[6] Manolis Kellis. Dynamic programming practice problems. http://people.csail.mit.edu/bdean/6.046/dp/, September 2010.

[7] Manolis Kellis, Nick Patterson, Matthew Endrizzi, Bruce Birren, and Eric S Lander. Sequencing and comparison of yeast species to identify genes and regulatory elements. *Nature*, 423(6937):241–254, 2003.

[8] Timo Lassmann and Erik L L Sonnhammer. Kalign–an accurate and fast multiple sequence alignment algorithm. *BMC Bioinformatics*, 6(1):298, 2005.

[9] Fabian Sievers, Andreas Wilm, David Dineen, Toby J Gibson, Kevin Karplus, Weizhong Li, Rodrigo Lopez, Hamish McWilliam, Michael Remmert, Johannes Söding, Julie D Thompson, and Desmond G Higgins. Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. *Molecular Systems Biology*, 7(539):539, 2011.