BioE/MCB/PMB C146/C246 **Spring 2005**

Lecture 1: Course Overview and Sequence Alignment

Lecturer: S. Brenner *Scribe: G. Leung*

1 Course Overview

There are several distinct areas of computational biology. One area is based on *statistical* principles, where associations and distinctive features are discovered by statistical analysis, such as in microarray analysis. Modern genetics is also based on these principles. Another area is based on *physical* principles, which are the basis of protein folding studies. There has been limited success in studying molecular processes, but there has been more success using these principles to study large scale questions, such as how organs move.

The area of computational biology covered in this course will be based on *evolutionary* principles, starting with sequence analysis. The evolutionary conservation of proteins allows us to make functional inferences based on sequence comparison alone. BLAST, the most well-known tool used for sequence comparison, is run 4×10^{14} times a year at NCBI alone, which is indicative of how widely used sequence analysis has become.

Some computational biology courses aim to familiarize students with the application of tools, introducing available bioinformatics software and demonstrating how to use these tools effectively to address biological questions. Other courses are more theory-based, presenting the underlying computer science algorithms used in building effective and efficient bioinformatics tools. This course will present principles and applications, which lies in the middle ground between tools and theory. Existing tools will be presented, but the goal is to be able to critique and evaluate tools that are coming out today.

2 Sequence Alignment and Dynamic Programming

2.1 Naïve Approach

Suppose we would like to align the following two strings:

```
T H A T T I N H A T
C A T I N H A T
```

We can easily produce a reasonable alignment by eye, especially since the last 6 characters of each string exactly match. Suppose we would like to computationally determine the longest common substring between two strings. A naïve approach would be to enumerate all possible substrings for each of the two strings, and then pairwise compare the substrings, keeping track of the longest match found thus far. How long would this take? There are $O(n^2)$ subsequences for each string, considering each starting point and each ending point. Pairwise comparison of these substrings would take $O(n^2) \cdot O(n^2) = O(n^4)$ time.

Clearly, we need to consider a more efficient algorithm, such as dynamic programming. Let's consider a simpler example of DP first.

2.2 Fibonacci Numbers

Fibonacci numbers are recursively defined as follows:

$$\begin{aligned} Fib(n) &= Fib(n-1) + Fib(n-2) \\ Fib(0) &= 1 \\ Fib(1) &= 1 \end{aligned}$$

We can calculate the 6th Fibonacci number, $Fib(6)$, by straightforward application of the above definition:

$$\begin{aligned} Fib(6) &= Fib(5) + Fib(4) \\ &= Fib(4) + Fib(3) + Fib(3) + Fib(2) \\ &= Fib(3) + Fib(2) + Fib(2) + Fib(1) + Fib(2) + Fib(1) + Fib(1) + Fib(0) \\ &= Fib(2) + Fib(1) + Fib(1) + Fib(0) + Fib(1) + Fib(0) + 1 + Fib(1) + Fib(0) + 1 + 1 + 1 \\ &= Fib(1) + Fib(0) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 2 \\ &= 1 + 1 + 1 + 2 + 3 + 3 + 2 \\ &= 13 \end{aligned}$$

This procedure takes exponential time. Note that in the calculations above, $Fib(i)$ for all $i < 5$ are calculated multiple times. This observation can lead us to a more efficient algorithm, where values for $Fib(i)$ are computed only once. Suppose we calculate $Fib(i)$ for $i = 0 \ldots 6$ in increasing order of i:

$Fib(0)$	$Fib(1)$	$Fib(2)$	$Fib(3)$	$Fib(4)$	$Fib(5)$	$Fib(6)$
1	1	$1+1=2$	$1+2=3$	$2+3=5$	$3+5=8$	$5+8=13$

This algorithm runs in linear time. The speed-up clearly comes from storing intermediate results instead of re-calculating them, which is called *memoization*. Note that memoization only works when the intermediate results don't have other intermediate dependencies.

There is also an analytic solution to finding Fibonacci numbers, providing a constant time look-up solution:

$$\begin{aligned} Fib(n-1) &= \frac{\phi^n - \emptyset^n}{\phi - \emptyset} \\ \phi &= \frac{1+\sqrt{5}}{2} \\ \emptyset &= \frac{1-\sqrt{5}}{2} \end{aligned}$$

2.3 Longest Common Substring

How can we solve the longest common substring problem using dynamic programming? Suppose we keep track of potential matches by pairwise comparing the characters of each string to each other. Let 0 represent mismatches and 1 represent matches.

	T	H	A	T	T	I	N	H	A	T
C	0	0	0	0	0	0	0	0	0	0
A	0	0	1	0	0	0	0	0	1	0
T	1	0	0	1	1	0	0	0	0	1
I	0	0	0	0	0	1	0	0	0	0
N	0	0	0	0	0	0	1	0	0	0
H	0	1	0	0	0	0	0	1	0	0
A	0	0	1	0	0	0	0	0	1	0
T	1	0	0	1	1	0	0	0	0	1

The longest common substring, TINHAT, corresponds to the longest diagonal of 1's. Using this method, although it takes $O(n^2)$ time to fill the table, it could take potentially $O(n^3)$ time to trace back through the table - at each cell of the table, the associated diagonal (substring match) needs to be checked. To reduce the trace back time to $O(n^2)$, we can store the count of the longest substring at each cell of the table ($M(i, j)$) by looking at the previous diagonal cell ($M(i-1, j-1)$). The table would then be filled in as follows:

	T	H	A	T	T	I	N	H	A	T
C	0	0	0	0	0	0	0	0	0	0
A	0	0	1	0	0	0	0	0	1	0
T	1	0	0	2	1	0	0	0	0	2
I	0	0	0	0	0	2	0	0	0	0
N	0	0	0	0	0	0	3	0	0	0
H	0	1	0	0	0	0	0	4	0	0
A	0	0	2	0	0	0	0	0	5	0
T	1	0	0	3	1	0	0	0	0	6

Finding the longest common substring is now equivalent to finding the maximum value in the above table and tracing back along that diagonal.

2.4 Dynamic Programming

2.4.1 Initialization

The recursive nature of dynamnic programming necessitates an initialization step where the base case(s) is set. For example, in the DP algorithm for Fibonacci numbers, the initialization sets the first two Fibonacci numbers to 1 ($Fib(0) = 1; Fib(1) = 1$).

For the longest common substring problem, where the previous diagonal cell needs to be considered, a zero row and a zero column need to be added to the table described above.

		T	H	A	T	T	I	N	H	A	T
	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	1	0	0	0	0	0	1	0
T	0	1	0	0	2	1	0	0	0	0	2
I	0	0	0	0	0	0	2	0	0	0	0
N	0	0	0	0	0	0	0	3	0	0	0
H	0	0	1	0	0	0	0	0	4	0	0
A	0	0	0	2	0	0	0	0	0	5	0
T	0	1	0	0	3	1	0	0	0	0	6

2.4.2 Recursion

The recursive step of the Fibonacci dynamic programming algorithm corresponds to the definition of Fibonacci numbers presented above. For the longest common substring problem, each cell $M(i,j)$, which corresponds to the ith character of sequence X aligned to the jth character of sequence Y, can be computed by the following recursion:

$$M(i,j) = \left\{ \begin{array}{cc} M(i-1,j-1)+1 & \text{if } X(i) = Y(j) \\ 0 & \text{otherwise} \end{array} \right\}$$

2.4.3 Trace back

To report the longest matching substring, trace back along the diagonal starting from the cell with the largest number, L. If $M(i^*,j^*) = L$, then the diagonal $M(i^*-L+1, j^*-L+1), M(i^*-L+2, j^*-L+2), \ldots M(i^*,j^*)$ corresponds to the longest common substring. The desired substring of length L is then $X(i^*-L+1)X(i^*-L+2)\ldots X(i^*)$, or equivalently, $Y(i^*-L+1)Y(i^*-L+2)\ldots Y(i^*)$. Note that when there are multiple common substrings of maximum length L, more than one cell in M will contain L, each corresponding to the endpoint of a substring of length L.

2.5 Sequence Alignment

In a *local alignment*, the "TH" in the first string and the "C" in the second string may be left off, producing one of the alignments below:

A	T	T	I	N	H	A	T
A	-	T	I	N	H	A	T

A	T	T	I	N	H	A	T
A	T	-	I	N	H	A	T

T	T	I	N	H	A	T
A	T	I	N	H	A	T

In a *global alignment*, the entirety of both sequences are included in the sequence alignment, while a *semi-global alignment* will only include the entirety of one sequence and a subseqeuence of the other. Another type of global alignment can be implemented which will not penalize for gaps at the ends of the alignment.

In all of these alignments, a scoring scheme needs to be established in order to return the "best-scoring" alignment. For example, each match could score +1, each mismatch -1, and each gap -2. Note that this scoring scheme would result in an optimal local alignment containing only the exact match, TINHAT, while reducing the gap penalty to -1 would allow the two left-hand local alignments above to score equally well as the exact 6-character match. There is no principled way of assigning the gap penalty; in practice, the gap penalty is decided upon emprically.

How many possible alignments are there to consider? If the length of the longer sequence is m and the length of the shorter sequence is n, let $k = m - n$. There are k possible gaps for each sequence, $\binom{m}{k}$ ways to put k gaps in the longer sequence, and $\binom{n}{k}$ ways to put k gaps in the shorter sequence. For any given value of k, there are $\binom{m}{k}\binom{n}{k}$ alignments. To consider gaps of all lengths, there are $\sum_{k=0}^{m}\binom{m}{k}\binom{n}{k}$ alignments, which is $\sim 4^n$ by Stirling's approximation. For sequences of length 100, there are $\sim 10^{75}$ alignments to consider. Given the number of BLAST queries performed, it is clear that efficiency is of paramount importance.