This is the End

Kenneth A. Ribet

UC Berkeley

Math 10B
April 28, 2016

Social Events

- Tomorrow at 12:30PM: Faculty Club lunch (just show up).
- Wednesday, May 4, Field trip to the Big C. (Meet 2PM at the south end of the Foothill parking lot.)
- May 6, 8:30AM: Faculty Club breakfast. This is a new event—send email to sign up (limit 20 people).
- May 6, 12:30PM: Faculty Club lunch (just show up).

Don’t leave early

After our discussion of dynamic programming, I’ll attempt to orchestrate a class photo: everyone come to the front, and we’ll grab someone from outside to snap a photo.

Final Exam, May 9

- It's here (F295 Haas).
- It's a 3-hour exam that begins at 11:30AM (not 11:40AM) on Monday, May 9. (Didn't I mention the date somewhere already?)
- You can bring one two-sided sheet of notes. No devices, calculators, watches. . . .
- Don't you love bullet points?
- The exam will be harder than the midterms (by your request, as communicated by the GSIs).
- It covers everything we have done together.
- It will probably stress the last third of the course ever so slightly.

Course evaluation

Course evaluations for this class are open.

“Although students received an invitation email and reminders along the way, previous research demonstrates that a personal reminder from the instructor and an explanation of how evaluations are used to inform your teaching can make a positive impact on response rate and quality.”

RRR Week

This class will meet as usual on Tuesday and Thursday at 2:10PM in this room. With luck, GSIs will be on hand on Tuesday to conduct a review. Please prepare questions for Thursday.

If you don't see me at 2:10 on Tuesday, we will probably have United Airlines to blame. I will be away on Monday and on Tuesday morning.

I will hold office hours as follows next week:

- Thursday 10:30–noon,
- Friday 10:45–12:15 (lunch to follow).

Dynamic programming

According to Wikipedia, dynamic programming is

> *a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions—ideally, using a memory-based data structure. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space.*

The Wikipedia page gives a large number of examples where dynamic programming is useful. Our "textbook" `Dynamics.pdf` provides a simplified account of the Needleman–Wunsch algorithm. The general problem is one of sequence alignment.

In the problem, we are trying to compare two finite sequences of letters

$$a_1 a_2 \cdots a_m, \qquad b_1 b_2 \cdots b_n,$$

where each a and b is taken from the alphabet $\{ A, C, G, T \}$.

The rule is that we are allowed to "expand" either sequence by adding blank characters. The aim is to insert blank characters (if necessary) so as to maximize the number of indices i such that $a_i = b_i \in \{ A, C, G, T \}$.

The "textbook" gives the example of the two strings

$$GCTA, \qquad GTAG.$$

The second string can be expanded to *G_TAG*, which then agrees with *GCTA* in three places. It is 99% clear by inspection that three places of agreement is the best that we can do in this example.

What about the general case?

Actually, the Needleman–Wunsch algorithm was published by Saul B. Needleman and Christian D. Wunsch in 1970.

The strings a and b are presumably of different length. We are allowed to add spaces at the beginning and/or end of either string (as well as in the middle of either). Adding places at the beginning of a, for example, can be viewed as sliding a along b. Thus we are aligning a alongside b. The number of places of agreement is then the *score* of the resulting *alignment*.

We can mentally imagine truncating a so that it stops at a_j and b so that it stops at b_i. (If j is less than m or i is less than n, this makes the problem shorter and presumably easier.) The optimal (i.e., largest) score for an alignment of $a_1 \cdots a_j$ with $b_1 \cdots b_i$ will be denoted M_{ij}.

If there is no truncation, then $j = m$, $i = n$. We seek to compute M_{nm}.

The alignment that we are after—the "one" with optimal score—need not be unique. For example, we can get the score "1" from GA and AG in two ways: we can align the two Gs vertically or align the two As vertically:

$$\begin{array}{ccc} \_ & G & A \\ A & G & \_ \end{array}$$

or

$$\begin{array}{ccc} G & A & \_ \\ \_ & A & G. \end{array}$$

A key point is that there is a recursive relation among the various M_{ij}. If you know the numbers for small indices, you can compute them for larger indices:

Suppose you want to align $a_1 \cdots a_j$ with $b_1 \cdots b_i$. If the alignment has a_j above b_i, then $M_{ij} = M_{i-1,j-1} + S(i,j)$, where $S(i,j)$ is 0 if $a_j \neq b_i$ and is 1 if $a_j = b_i$. If not, then the rightmost element of a is to the right of the last element of b, or vice versa. If a_j sticks out to the right of b_i, then a_j is irrelevant. Similarly, if b_i protrudes to the right of a_j, then b_i is irrelevant. Hence Mij is the maximum of these three numbers:

$$M_{i-1,j}, \quad M_{i,j-1}, \quad M_{i-1,j-1} + S(i,j).$$

This makes sense even if i or j is 1—we just let $M(i,0) = 0$ for all i and do the same for $M(0,j)$.

So far, so good. However, the mere fact that a function is recursively defined does not mean that it is easy to compute. One can say in fact that dynamic programming is a technique for computing recursive functions by storing intermediate results in an intelligent way.

As Wikipedia explains, the recursive definition of the Fibonacci sequence makes it onerous to compute even for small indices. The fifth Fibonacci number is the sum of the fourth and third; the fourth is the sum of the third and second; the third is the sum of the second and first,.... If you don't pay attention, you'll write the fifth Fibonacci number is an eight-term sum and the nth number will be a sum with 2^{n-2} terms. When n is large, there's hell to pay.

The Needleman–Wunsch algorithm finds an optimal alignment of two sequences using two tables (= matrices): a scoring table, which holds numbers, and a traceback table, which holds arrows that keep track of where the numbers came from.

I will illustrate the algorithm (on the document camera) in two examples.

The sequence $a_1 \cdots a_n$ will be called "A" and the sequence $b_1 \cdots b_m$ will be called "B."

For reference, here is how we deduce the sequences at the end from the traceback table:

- $\rightarrow$ means that we write down the last unused letter of A above a blank (for B);
- $\downarrow$ means that we write down the last unused letter of B below a blank (for A);
- $\searrow$ means that we write down the last unused letter of A above the last unused letter of B.

We take letters of A and of B from right to left.

Example 1

This example comes from the online “textbook”:

$$\text{Sequence } A = G\,C\,T\,A.$$

$$\text{Sequence } B = G\,T\,A\,G.$$

The algorithm produced

$$\begin{array}{cccccc} G & C & T & A & \_ \\ G & \_ & T & A & G. \end{array}$$

There are three matches after alignment, and the optimal score is 3.

Note that we are scoring ‘1’ for each match and ‘0’ for everything else. We could have different scores for each of these possibilities:

- a match,
- a mismatch: two different letters,
- a letter lined up with a blank.

The algorithm works whenever scores are assigned to the three possibilities.

The second example comes from Lior Pachter's second MT (earlier this month):

$$\text{Sequence } A = G\,A\,T\,T\,A\,C\,A\,.$$

$$\text{Sequence } B = A\,T\,A\,T\,A\,A\,C.$$

When I did this at home, I ended up with

$$\begin{array}{cccccccccc} G & A & T & \_ & T & \_ & A & C & A \\ \_ & A & T & A & T & A & A & C & \_, \end{array}$$

which has five matches.

An important comment is that the path to the optimal score is not unique (in general). In this example, it seems that there were quite a few cells in the traceback array where there were two arrow choices. Accordingly, it is plausible to me that there could be more than one path back from the southeast corner of the traceback array to the northwest corner.