The following content is provided by MIT OpenCourseWare under a Creative Commons license. Additional information about our license and MIT OpenCourseWare, in general, is available at ocw.mit.edu.

GEORGE CHURCH: OK, so welcome to the fourth lecture. This will be the second one on the subject of DNA. The major difference between this lecture and the last one is that last one, so-called DNA 1, we focused in on types of mutants, their really closely related DNA sequences. And this one, we'll talk about the most distant related biopolymer sequences.

So we went through the way you can generate closely related sequences and the way that the populations which are made up of closely related sequences obtain those allele frequencies through mutation drift and selection. And I argued that, deep down, at the most precise level, you've got a binomial distribution behind each of the processes of mutation drift and selection, at least under a certain set of assumptions.

And then we went on to another very valuable statistic, which is the chi-square, which you can use in a number of different scenarios but, in particular, with association studies, where we did a simple case of a two-allele system with two outcomes-- HIV resistance and sensitivity. And then this association led to a broader discussion of alleles and haplotypes and genotypes, in general, and how one can obtain those and, in the process of doing so, expose oneself to random systematic errors, a theme that we'll return to from time to time.

So today, we'll talk not about very closely related sequences but very distant related sequences and what are the algorithms that are available for finding those. These distant related sequences get a very different set of avenues. Here, we're trying to look for hints, for hypotheses in, say, new genome sequences as to what new genes might do.

So we'll begin by comparing different types of algorithms that will allow us to do alignments. In particular, we'll stress the hero of today's show is dynamic programming. It comes up again and again throughout the course, not merely in pairwise sequence alignment but multisequence alignment. And we'll then go on to how the space in your computer, or the memory, dedicated to processing the time determine trade-offs. And in some cases, you will have to even sacrifice accuracy or completeness.

And this also leads to the issue of finding genes, a particular type of distant sequence comparison. The ones interested in is finding motifs that are involved in finding genes. And finally, we'll end on hidden Markov model, the simplest one that I could think of that would really illustrate the idea of Markov models, probabilistic models, and the hiddenness. And so this is illustrated with a single dinucleotide with two states.

So this puts in context-- in the first couple of lectures, we talked about the tree of life and how, right at the core, we had the common and most simple forms, which share the simple genetic code. And that was-- and then last time, we talked about the very tip of one branch of one of these trees-- basically, the human branch of the animal branch. And what has happened at the tip of that the last 5,000 generations, since the fairly significant bottlenecks that resulted that predated the population explosion.

And so this time, we're going to talk about the whole tree and how these very deep branches can be obtained in comparing biopolymer sequences. Some of the earliest trees of life were based on the ribosomal RNA sequence because that's something that's fairly easy to trace back to common ancestors. But you also want to be able to do this with a variety of proteins, some of which go only a short distance traceably in sequence, and some of them go all the way back.

What can we do with dynamic programming? I alluded to its multiple uses throughout the course. Here's some examples. We already mentioned shotgun sequence assembly. This is relatively easy because the sequences are fairly closely related to one another.

Multiple alignments include sequence assembly, where you have multiple similar sequences. But as you get to more and more distant sequences, you can glean more and more about structural and functional conclusions about proteins and nucleic acids when you have a very large number, in the hundreds. And we'll see the challenges that come there. Repeats are a particular-- you can have repeats within a genome, or we can have alignments between different species.

Now, birdsong seems a little out of place here. But historically, it's one of the first applications of dynamic programming where you, in a sense, have a continuous time axis, and you sample that at discrete points, often, in order to sample the intensity of the audio recording. And this will have a more direct analog than sequence alignments, which we'll do today. Later, when we're doing the RNA analyses, gene expression will show a direct time warping algorithm, which is very similar in outline to the birdsong.

And then, finally, hidden Markov models, which would be the last thing today, use dynamic programming as part of the process by which the decisions are made about the model that's represented, the hidden Markov model. And these hidden Markov models, in turn, allow us to do RNA gene searches, structure prediction, distant protein homologies, speech recognition, and so on.

So there's three types of alignments and scores that we'll discuss on the slide number 5. The main dichotomy is between global and local. Originally, in sequence alignment in the '70s, Needleman-Wunsch had a global algorithm, which has been modified-- Smith and Waterman's local algorithm-- about a decade later.

And the major difference here is that in a global algorithm, you have reason to believe that the sequences align end to end. And you really just want to ask how many mismatches and insertions and deletions are there in the middle of the sequence?

So for example, you might have two proteins that really have the same start and stop site. Or you might have an entire chromosome, and you're asking questions about haplotype, as we were in the last class, which would be, what mutations are in cis on one chromosome relative to another?

Another example is you might be scanning a chromosome for a motif that occurs again and again, like an Alu repeat or a transcription factor binding motif. And this might be in the middle of one sequence, at the end of another, or in the middle of both. Here, it's at the end of one and the middle of another. And it's short and local. And so you don't want to constrain the sequence ends to align. And you don't want to penalize if there's some deviation from alignment of the ends.

Taking this one step further so that it's not internal to either one, in an ideal assembly-- shotgun assembly of sequence, as we talked about last time-- you would expect, as one fragment-- one clone fragment-- ends, hopefully, you have a little bit of overlap at the beginning of the next one, and you'll be able to jump along these stepping stones to get to the end of the sequence. Because you can't sequence the whole thing.

But this ideal suffix, where a suffix of one overlaps the suffix, or the verse complement, of another, this sort of alignment can be imperfect because of errors at the ends of these sequences. So generally speaking, we'll be talking about global versus local. And these specific sequences we'll come back to several slides from now.

Now, you want to have a scoring algorithm. We'll use this very simple scoring metric off and on during this talk. And here, we're just giving plus 1 for every perfect match, indicated by a colon where an a matches an a, and a minus 1 for every mismatch, indicated by an x, where c does not match an a, for example.

So we have five perfect matches and five mismatches in this case. And so it's a total of 5 minus 3 is 2. And in the local case, we're not going to require the ends to align. And so we can slip this a little bit so that now, there's four perfect matches, and we're not penalizing the terminal mismatches. And so we have a total of four, which, if these were directly comparable scores, would be a superior score. And you can see the suffixes, here, you're enforcing that it be at the ends. And it's a score of 3.

So now we're going to compare different ways of searching through sequences. Now, that one was an exact match with mismatches, where the mismatches are penalized. And so an exact search, a truly exact search, is fairly rare-- maybe a restriction site, something like that.

You can expand this to a regular expression where the insertions are restricted. They, in this case, an insertion can be indicated by any base. A, C, G, or T can occur. And then the number that you will tolerate is indicated by a numeric range-- zero to nine, in this case. And so the particular example given-- this equals sign just means that this is an example. So C, G are strict at the ends.

And then it happened to have two As there, which an A is an example of a nucleotide that satisfies the abbreviation N. N is all possible. But you could similarly have the zero to nine bracket could refer to a short sequence, like a G sequence and so on. So you could get a known number of repeats. And that could represent the empirical observation that you have zero to nine AG repeats.

Now, in the previous example, we just penalized whenever it was a mismatch by a fixed amount. But you could have a substitution matrix, which would actually codify your observations. When you look through natural sequences and you line them up, how often do you get an A substituting for a G? And so the penalty wouldn't be a strict minus 1. It would be something you'd look up in a table. And in that table, the diagonal would be the matches, and the off-diagonals would be specific mismatches. And we'll show how we get such a matrix and use it.

Now, then you can have a profile matrix. Or this PSI means position-sensitive. And the position-sensitive means you have a different lookup table for every position in your biopolymer. And that makes sense because not all positions have the same set of equally-- or have the same sort of substitutions that are allowed.

So in parentheses along here, these are actual names of programs that are available, either in commercial packages or for free. BLAST is Basic Local Alignment Sequence. And the N refers to nucleotide. And here, PSI-BLAST, again, position-sensitive. These are some of the ones you'll run into most frequently.

The original versions of BLAST were basically aimed at having the largest block of contiguous sequence without gaps. And there were so-called Carlin statistics that would tell you that that's the largest sequence that will give you the best probability.

But then it was widely recognized that when people actually evaluate matches between two sequences, they're not just evaluating the longest ungapped sequences. They're actually interested in the significance of a sequence that can include insertions and deletions. And so BLAST has been extended to include gaps.

And prior to that long history back to the '70s is the Needleman-Wunsch and Smith-Waterman that I mentioned, a global and local, respectively, which would allow a large number of insertions and deletions of single bases and multiple bases, as determined by the parameters that were either manually set or determined empirically. And we'll try to stress ways that this can be determined empirically.

And finally, the hidden Markov models truly take a probabilistic approach to each of these, allowing position-sensitive and models that are extendable not just where each position has a set of probabilities but there can be dependencies upon adjacent positions. And we'll get to this at the very end.

And when we talked in the very first class about different definitions of complexity, one of them we talked about was the computational complexity, or hardness, of the amount of time that it takes, or the amount of space, or space and time, that it takes to solve a problem. And this certainly is a good illustration in today's talk in dynamic programming.

Because when we want to do either a pairwise sequence alignment or a multisequence alignment-- let's start with pairwise, we're aligning k equals 2 sequences of length n, and we're allowing gaps. Now, if we're just comparing them without gaps, as we did in the earlier slides, it's trivial. It's linear with the total number of bases. It's linear within. And that, of course, scales very gracefully.

But if you have a very naive-- and I'm going to set up a straw man here-- you have a very naive algorithm, then you'll go along, and you'll put in a gap at every possible position at the top in combination with every possible position in the bottom. And so that means that there's n on the top and n on the bottom for a total of 2n possible positions that you could put in a gap. And the gap can be any length, as you see on the right here of slide 7 is that you can have a gap up to length n.

So roughly speaking, such a naive algorithm would scale very exponentially n to the 2n power. And this is just enormous for n the dozens. And when n is in the billions, then you can just basically forget about it. We're not even talking about computers or particles in the universe anymore.

And that's for k equals 2, the simplest possible case where you're aligning two sequences. If you're aligning multiple sequences-- we'll get to that in a moment-- it definitely becomes exponential in k, even in a-non naive algorithm. So we're going to show that it's going to be non-exponential in n, the length of the sequences.

But in order to do that, in order to assess algorithms, we can do them-- some of them we can do theoretically. And in fact, for the dynamic programming, we will show that you can convince yourself that an n squared algorithm is precise. But others, we will have to do empirically. And I just want to take an aside here to talk about how a particular comparison of sequence alignment programs was done.

So the critical thing, not just for this test but for many that you'll be seeing in this course and you may want to do in your own projects, you want to set aside a training set in which you run through a number of algorithms or a number of parameters within an algorithm to assess which ones are the best. And once you've determined the algorithms that are best or the parameters that are best, then you want to have a testing set that's independent. That means non-redundant, as well.

So the training set, if you use the training set as your testing set, you may lull yourself into a sense of complacency because it's been overtrained. And you're basically only capable of solving the problems that you set before it. So you want a separate testing set. And that was done in this case.

We need some sort of evaluation criteria. So we talked a little bit of scores that you'd set up in order to score whether one alignment is better than another alignment. But in addition, when you want to compare two algorithms that may give you the same score, you want to have an external evaluation criterion.

Typically, the evaluation criterion that one might have is sensitivity and specificity. Occasionally, you'll find in literature where people will just focus on one or the other. For some reason or other, they don't want to miss anything. So they want to keep their false-- they want to reduce their false positives as much as possible. Or they don't want to plow through mountains of output, and so they keep the false positives down. But you really want to have them both very low.

And sometimes this is restated as sensitivity and specificity, where sensitivity is the number of true hits that you've predicted over the total number of true hits. This is, say, in your training set, where you know what the true hits are from some outside source. And then specificity is the number of true hits that you predicted over all of the hits that you predicted, true and false.

Now, this truth here that comes in your training set, where does that come from? If we had access to truth, in general, then what do we need these algorithms for? And the answer is that we do have access to, maybe, a higher truth or something that's outside of sequence alignment. And this is crystallography, genetics, and biochemistry.

Crystallography, for reasons we'll go into in the protein part of this course, is capable of detecting much more ancient relationships between biopolymers than is sequence alone. Similar genetics in biochemistry can test structural and functional hypotheses through great expense. And so these are expensive. And so that's the reason they're great for making training sets, but they won't necessarily replace sequence alignment and scores.

So that was the setup for this slide, which is that Bill Pearson, who actually developed the FASTA algorithm, among others, did a thorough assessment of various algorithms. FASTA was one that was based on words, meaning exact matches of some fixed lengths the user could set; FASTP, which was based on these maximum-lengths blocks without gaps in it.

Blitz is a variation on the Smith-Waterman algorithm, meaning a full dynamic programming, which we'll be talking about in a short while. This is a highly parallelized version of it, early parallel version of it. So the different algorithms for doing the alignment were compared, different substitution matrices and different databases. Just in case there was some database bias, he included that.

And so we're going to talk about substitution matrices in just a moment. But basically, these are what amino acid or nucleotide-- what amino acid, in this case, can substitute for another amino acid in actual protein segments that have diverged by about the amount that you want to do your test on. And these different numbers just refer to roughly how distantly related the proteins are. The higher the number, the more distantly related they are in the case of the PAM matrices.

So now, why did he do that with a protein level rather than doing it to the nucleic acid level? Well, historically, it's because there weren't any nucleic acid sequences. There were mostly protein sequences. But even today, there is obviously a lot more nucleic acid sequence.

But there's a real reason to do it at the protein level, which is that when you look at the code that we've been talking about in these lectures, something like leucine can be represented by six different codons, which can have wildly different nucleotide sequences. So for example, CUG is valid, and UUA. And those only share one nucleotide out of three.

And over long periods of time, if there's heavy selection on the protein and relatively weak selection on the nucleic acid, or there could even be pressure on the nucleic acid to change for reasons that we'll go into in the second half of this lecture, that pressure on the nucleotide sequence can cause the nucleotide sequence to change a lot and the protein sequence not to change much at all. So an example of pressure is if the tRNAs change in their abundance, then there'd be a pressure to codon usage to change.

There are some reasons to do it at the nucleotide level. For example, if you're comparing sequences which don't encode proteins, that's an obvious reason. If you have a lot of insertion and deletions or a tricky biological phenomenon like RNA splicing that causes the protein to be out of phase-- the inferred protein to be out of phase or hard to infer-- then you would do it at the nucleotide sequence level.

Now, I'm going to show this slide twice. The first time, we're going to take it as a given that we've been given this multisequence alignment, and we're not going to question right now how we got it. But we're going to use that multisequence alignment to derive, or to talk about how we would derive, a substitution matrix.

And here, a substitution matrix, you can think-- this is a multisequence alignment. So essentially, we have a weight matrix, which, if this were position-sensitive, we would say, at this position, C never changes. If we do enough of these proteins and we don't care about position, we can build up a big set of matrices.

And in general, we will find that C tends to substitute for C, and very few other things substitute for it. Eventually, you will find other substitutions. Cysteine and tryptophan, C and W, are relatively rare amino acids, and they're highly conserved. Other ones can be substituted, as you can see here-- threonine, serine, and valine can substitute for one another.

So now let's take a look at how this plays out when we look at all the possible substitutions that can occur. And that's what's in slide 12. If you look along the very top row are the percentages, the abundances, of amino acids in a particular organism, say E. coli. And then there's a single-letter code, A through Y.

And along the diagonal is the substitution matrix, the score which has been determined-- this is the BLOSUM matrix-- for a block represented in distantly related blocks. Here, you can see that the diagonal represents the tendency for the amino acid to substitute for itself. And those amino acids, which generally are not easily substituted-- as I say, are highly conserved, which we pointed out in the previous slide-- were cysteine, C, and tryptophan, W.

And for example, W is the most strongly conserved. It's 22 along the diagonal. And the consequence of that is there'll be relatively few positive values off the diagonal. And in fact, for tryptophan, there are no positive values. The numbers here have been generated in such a way that the negatives will tend to cancel out the positives in alignments, known alignments, of sequences that are about the right evolutionary distance from one another.

You want to pick-- you want to make your matrix, if you're trying to look for very distant related proteins, you want to take the substitutions that you're sampling in your training set to be as far at that same distance-- that is, very distantly related. And this is one of the mistakes that was made in the early PAM matrix.

There were two mistakes, actually-- the mathematics-- well, first, the proteins that were compared were very closely related. Because closely related proteins were more trustworthy. You could align closed sequences more easily. The algorithms didn't have to be sophisticated. And the trees could be more precise.

But then, because they had done-- so that already was a bias because the substitutions you get in closely related sequences aren't really the same. And then they apply to a mathematical extrapolation method, which was not adequate in terms of the actual evolution and also wasn't even correct, mathematically, although this persisted for decades as the most common-- and still-- the most commonly used matrix.

Anyway, you can see that, although tryptophan doesn't have any positive off-diagonal, something like arginine, here, in this blue, has a positive 4 off-diagonal and a positive 10 on-diagonal. So as you might guess, what's the most likely substituted for positively charged arginine? It's positively charged lysine under physiological conditions. And that's why that's off-diagonal. And there are other ones. We've color coded these the same as the genetic code, where the negatively charged amino acids can also substitute for one another.

So the significance of that top row, of the percent abundance, is that if you find two matching As, that's not so significant because that's the most frequently occurring amino acid in this organism. On the other hand, if you find two matching Cs, that's very significant. Because those are rare. And finding two of them at the same place in a particular alignment means it's significant. So both the abundance and the substitution matrix can be useful.

So now, we're going to walk through a actual scoring of some alignments. And we want to do this in this more challenging situation where you allow insertions and deletions. So first, we will, even though we've told how it is that we get the match versus mismatch numbers as a full substitution matrix, here, you can imagine substitution matrix has plus 1's along its diagonal and minus 1's on off-diagonal just so you can do all the calculations I'm going to show you in the next few slides in your head.

But also, simultaneously imagine that it could have the richness of the substitution matrix we just had. We're going to do this, the next few, with nucleic acid. But imagine you could also do it with amino acids. The nucleic acid substitution matrix will be a 4 by 4. The one we just saw was a 20 by 20 for amino acids.

The indels, we'll penalize them by minus 2. But this is an arbitrary number. And you'll see how critical it is in just a moment. But you can imagine that this could be determined empirically, just like the substitution matrix was determined empirically in the previous slide.

The alignment score, then, will be defined as a sum of columns. We're going to be assuming that adjacent positions are independent of one another. And we'll be scoring them independently and then just taking the sum. That gives us the alignment score for a particular alignment, a particular set of indels and a particular set of offsets from one sequence relative to another.

But what we really want to do is go through all those possible alignments to get the optimal alignment, which is the maximal score defined here. To get the optimal alignment, we'd like to do that in less than exponential time with at least less than exponential per n length of the sequences.

So we're going to use this pair of sequences, ATGA, ACTA, twice on this slide. And we're going to use it in subsequent slides where we do it a slightly different way. And we're going to use that very simple scoring metric-- plus 1 from that perfect match, minus 1 for mismatch, minus 2 for indel.

And what we get here is these are just two of many different alignments we could have with different insertions here. On the left, number one, the most extreme case, no insertions or deletions on either sequence. We're only counting mismatches. There are two matches, two mismatches. And so that's two plus 1's, two minus 1's. And they cancel out. And the score is 0 for the one on the left.

Then the one on the right, we've gone to the-- allowed an insertion on each strand, indicated by a dash on the opposite sequence. And now, you see you have three perfect matches, which is an increase in the number of perfect matches, but penalized with two indels, which are both negative 2. So it's plus 2 minus 2 minus 2 for minus 1. So this is not an improvement for the alignment on the left if we accept the scoring metrics that we had in the previous slide-- plus 1, minus 1, and minus 2 for the indels.

If, instead, we say, well, indels really shouldn't be penalized that much-- we can accept insertions and deletions in these kind of sequences, we'll penalize by minus 1, penalize it the same as a mismatch-- now the score is the 3-- score for the one on the right, with three perfect matches and two insertion-deletions, is now plus 1. And it beats the perfectly aligned ones. So whether 1 is better than 2 depends on the indel score that you chose.

STUDENT: Can I ask you a question?

GEORGE CHURCH: Yes, question.

STUDENT: Now that you are aligning two different sequences and in the case of the indel, you are allowing insertions all over the place-- I mean, you could theoretically have millions of those. But in reality, this [INAUDIBLE] the sequences in most cases would be no. You would know what it is. You can't--

GEORGE CHURCH: We know both of these sequences. These are sequences that we're comparing two different organisms.

STUDENT: Right. So if you know both of them, then what's the point in allowing all these indel [INAUDIBLE].

GEORGE CHURCH: So the question is, why are we allowing insertions and deletions? And the reason is that during evolution, say, either lab evolution or ancient, insertions and deletions are valid mutations. And so we're trying to determine where the most likely places that insertions and deletions might have occurred over the course of the divergence of two sequences. And believe me, insertions and deletions are very, very common. So that's why we permit them.

Now, why it is that insertion-deletions might be highly penalized or low penalized might depend on a position in the sequence. So for example, if you have a transcription factor where its precise geometry is important or an alpha helix and a protein or the translation of a genetic code where an insertion will throw the entire frame out of whack, as we had in the chemokine receptor in last class, then you want to penalize an indel very heavily.

On the other hand, if you have a bunch of motifs that are kind of separated by variable linkers, then the insertion-deletion could almost be zero, no penalty at all. So you can see it matters, and it might be position-sensitive. It might not be one size fits all. But these are empirically determined-- can be empirically determined.

So here's the hero-- dynamic programming. We've hopefully motivated that we can do scoring. We can determine empirically useful substitution matrices and indels. Now, how do we apply them?

And dynamic programming extends beyond biology, as I've alluded. And such an algorithm solves every subproblem just once and saves its answer in a table, thereby avoiding the work of recomputing the answer every time. So the strawman that I threw up before of having this exponential problem is very readily solved. And the way it's solved is this is the subproblem way of dealing with it.

And the idea of recursion, which we lightly touched upon when we defined the factorial, n factorial as equal to n times n minus 1 factorial, so defining it in terms of itself. But the key thing behind that definition and the ones we'll have here is that when you define something it terms of itself, you'd better have the call be a simpler problem and eventually terminate.

And so that's what we have here. I'm going to give two examples in slide 17 and in the next slide, one of them global and one of them local. This one will be done in terms of a tetranucleotide comparison, the same one we've been dealing with all along. And the other one will be in a more abstract sequence.

Here, the way do the sub-subproblem by recursion is we say we define the score of aligning these two tetranucleotides as the maximum of-- and then there are three options. It can be either the score of having an insertion on the top strand, and that's the top option. The middle one is having no insertions or deletions on either strand and just evaluating the last base comparison, which, in this case, is an A versus an A.

Now, that is the way that the algorithm terminates. When you have a single-base comparison or a single base compared to an indel, then you look up the scoring algorithm, or the scoring metrics we've been using all along.

So here, let's look at that final right-hand column. The score for an indel versus an A would be that minus 2 that we've been assuming along. And the score of an A versus A would be the substitution matrix diagonal, which would be a plus 1 and then, here, a minus 2.

And so you can see that you're calling up these three possibilities-- indel, no insertion on the top, no insertion on the bottom. And you take the maximum of these, whichever one of these gives the best score.

Now, that requires going back and calling it again. But you're calling it with a simpler-- you're asking for a simpler one. So now, you'll take the max of ATG versus ACT. And that'll ask you to look up the max of AT AC. And finally, it'll get the max of A versus A. And then you end.

STUDENT: Excuse me.

GEORGE CHURCH: Yeah.

STUDENT: Are you assuming that insertion side is always a 1? The insertion side is always 1, right?

GEORGE CHURCH: No. This algorithm allows any number of insertions up to the length of a sequence. And you'll see it when we do this in tabular form, how every possibility. But you do one at a time. There are only three cases here.

By dealing with just three cases at a time, you actually end up having the full generality of any number of insertions and deletions. And that's the beauty of this algorithm. You don't have to explicitly do every possible insertion with every possible deletion. You just have to run them through once. OK.

Now, I said that I was going to do two treatments of certain similarities. These are both dynamic programming of pairwise sequences. The previous one was global. This one is local. The only difference now is that we restrain the score to be greater than 0. We don't permit negative. So that means we're not penalizing the mismatches, for example, at the ends. Remember when I showed you that specific example early on. So now we have four choices-- the same three as before plus 0.

And the other thing that I made a little different here is, rather than having a specific sequence-- that tetra nucleotide-- here, we have a general sequence where you show that the ellipses, the sequence is up to i long and up to j long here. And at this stage in the scoring, you're going to either lop off the i and j sequence element-- this would be a single base, and you do that score in the central scoring here-- or you have an insertion at the top or insertion at the bottom.

So this is just a restatement of the previous one, generalized and made into a local alignment, which, in general, is what people do. People do local alignments rather than global ones. Because it's unsafe to say that the ends of your sequence will align. But we'll work through both of these as examples.

Now, we're going to compute this as a row-by-row algorithm. Now, casually, you could just leave off this frame along the edge. But in order to make the algorithm be the same for the beginning and all the intermediate steps, what you do is you pre-fill this with numbers such that the edges are some very, very small number which is smaller than the sum of all the scores that you could get out of this table so that you can't really come in from those edges. You have to come in from the zero point, which is the global alignment requires the ends to align.

So this is requiring the left-hand end to align. And so then the first comparison-- the only comparison you can really do-- is the A, A. That's the terminal comparison. And that happens to be a perfect match, so it gets a score of plus 1.

Now, the next square that you can do is minus 1. And remember, each of these has three possibilities in the global alignment. Remember, it can be an insertion, a deletion, or just a direct comparison of match versus mismatch. So this first one, the insertion and deletion were ruled out. They weren't going to win the maximum score. So you basically got 1.

It gets a little more interesting when you go to adding this next C. In order to add this C on the horizontal axis without adding anything on the vertical axis, that means that you've got an indel. And that means that you've got your A-A match. But now, to add this C, you've got a negative 2-- a penalty of minus 2. And so for the net result is a minus 1.

And then, for each subsequent one, it's assumed that the extension is the same as the initial indel, which is all negative 2. And so this is an A-A match followed by one insertion, two insertions, or three insertions. And three insertions, of course, that gives you minus 5.

And you just keep walking through this. Each one of these squares, essentially, is the maximum of three possibilities. The diagonal, if you follow the little yellow diagonal line from the 1 to the 0, that means you've taken an A-A match and a C-T mismatch, and the negative 1 is canceled out to positive 1, and you get a 0.

Alternatively, that 0 is actually the maximum of that individual score compared to an indel from a minus 1 plus this mismatch, which is not going to be better than 0, and a minus 1 plus the mismatch coming in horizontally, which is also not going to be better than 0. So you end up with 0, which is the perfect match plus a mismatch, no indels.

And similarly, you can fill up the entire table this way. Finally, now, you can trace what the best scores are going from end to end here, going all the way from your A-A terminal match at the left end to the A-A terminal match at the right end. And you can see the best traceback route is going through the diagonal here.

This 0 is the maximum of three possibilities-- left-right, up-down, and diagonal. Similarly, this minus 1 was the best of three possibilities. Remember, this is a global alignment that allows negative values. And its maximum was along the diagonal and so forth.

That's an example of the two basic steps. You set up the scoring metrics. You set up this n by n or n by m matrix. And then you just fill it up. And that's an n squared operation. It just goes up with the number of-- the length of the two sequences.

STUDENT: [INAUDIBLE].

GEORGE CHURCH: Question.

STUDENT: A diagonal is not the only optimal--

GEORGE CHURCH: It's not. That's true.

STUDENT: [INAUDIBLE]

GEORGE CHURCH: That's right. If you have an off-diagonal that's equivalent, then that's another valid sequence alignment. And actually, it comes up quite commonly, both in global and local alignments. And then the lower right-hand corner of slide 20 shows the specific interpretation of this [? Brown ?] set of errors, the particular traceback that we chose to highlight here, which is not the only one. And that's interpreted in this the same way that we interpreted the-- symbolically in an earlier slide.

Now, this is also from a much earlier slide. This is the one where we had the motif to illustrate the local alignment. And on the left-hand side matrix is for local alignment using a Smith and Waterman algorithm. And on the right-hand side is the global alignment using the Needleman-Wunsch type algorithm, which we just used on a shorter sequence.

And here, we've emphasized the diagonal, which gives a score of 2 and has a traceback along the magenta diagonal and would have the interpretation of the top sequence directly over the bottom sequence. On the other hand, if we look for local alignments and we do not penalize the offsets or the indels, then you can get an example.

And here's another magenta traceback where we've gone-- the A-A match is not on the diagonal for the global sequence alignment, but it hasn't been penalized. So it picks up the 0's from the frame boundary cells and just picks up the positive 1 perfect match. And then, when you add a C, it picks up another one, and another C, another a. And all four add up to 4.

Adding an additional base, however, does not help. Because it has to be a mismatch or an insertion or deletion. So you going from the 4 to an indel causes it to drop by minus 2, giving the two 2's. And going along the diagonal picks up a mismatch, which is a minus 1 penalty. So you just can't do better. And so that determines the edges of your local alignment. So you not only have a score and a traceback but you also have endpoints.

STUDENT: [INAUDIBLE].

GEORGE CHURCH: Yes, question? You, yeah.

STUDENT: Now, when this gets going, though, on the not exactly diagonal but in diagonal [INAUDIBLE]--

GEORGE CHURCH: Yes, you'll get a longer--

STUDENT: B, which is obviously not as good--

GEORGE CHURCH: And then 4.

STUDENT: And then you keep going, you get back to 4.

GEORGE CHURCH: And that's another example, just like the previous one. That's another valid alignment. It's still a local alignment. It doesn't have the total global endpoints. And it has an equal score to the shorter motif.

STUDENT: So how do these two compare? I mean, would you--

GEORGE CHURCH: They're equal.

STUDENT: --say because the other one is a longer [INAUDIBLE]

GEORGE CHURCH: Well, in this case, the scoring algorithm was set up in such a way that length didn't factor into it, other than the fact that longer sequences have more chances to have more perfect matches. So in this case, they would be equivalent. As you get to more detailed substitution matrices, the chances of getting two identical scores are weaker. But with nucleic acids with this kind of simple scoring, it comes up all the time.

So that's fine. We've now bargained our way down from a horrible exponential potential way of doing alignments to something which scales by n times m where those are the lengths of the two sequences. You could think of this as a rectangular matrix such as the ones we've been doing.

And both the time and the space or memory requirements for the algorithm will scale by this quadratic relationship. And the amount of time in memory is modest. So in absolute terms, it would be on the order of one comparison-- that's that maximum comparison-- three multiplication steps and in computing the entry. And the memory could be on the order of a byte. Data structure could be integer, or it could be floating point. And again, you have to have some way of finding the entries in the table.

So that's fine. It scales gracefully. But how big is it? Let's say we had two megabase genomes. In order to find entries of that size, you might want to set aside 4 bytes. And so you have the 4 bytes times 10 to the sixth squared. Or this is just ballpark. There are various ways you could squeeze this a bit. But this is 4 gigabytes of memory.

And for a gigahertz CPU, you might be able to do a million entries per second so that with 10 to the 6 squared entries, that's about 10 days. Now, that's a fairly small genome. Most genomes are bigger than 1 megabase.

And so when we had the discussions at the beginning of the Genome Project, one of the things the computer people brought up was, how are we going to compare a billion base pairs with a billion base pairs if the goal of this project is to do the three billion base pair human genome? And of course, back then, most computers were 4 gigabytes, and a gigahertz was a quite remarkable computer.

And of course, the answer was that we weren't going to do a full dynamic programming of the human genome against itself. We were going to cheat in various ways. And it took the recognition that it really was practical and, biologically, not much of a shortcut to look for little anchor points that would tell you that maybe the sequences don't align end to end, but there's some anchor point where you have enough bases or enough amino acids in a row that you can say, OK, here's one point where they definitely align. Let's now make reasonable assumptions by how many indels there can be, for example, by knowing how different the two sequences are.

And so if you know the differences of sequences, then you can say, I'm not going to allow more than a reasonable number of indels based on how different the sequences are. And you make a band which is a narrow width-- here's a fairly extreme example where we have a width of 3. And so rather than doing a full n squared matrix where you filled up the entire thing, we just do this band, which is on the order of the width of the band times the length of the longest sequence.

Now, this doesn't look very impressive for this case because n is small, and w is relatively large. But if n were billions and w were, say, 3 to 5, then it would be a very significant savings. So there's two key things here. One is the banding, and the other is getting the anchor points.

So summary for this half of the talk is that dynamic programming is really the rigorous way to compare two sequences. And after the break, we'll see how you can compare multiple sequences. We need to work towards a statistical interpretation of these alignments. That's going to require some test sets-- sorry, some training sets-- where you can see how it actually behaves on real biological populations of sequence alignments.

We need to compute either a global or a local alignment. And you've seen algorithms for doing each of those and how there's important but subtle differences between them. And we've talked about ways that we can improve the algorithm tremendously using the simple scoring functions or complicated ones that are determined empirically. So let's take a little break. And we'll come back and talk about multisequence alignment.