Algorithms in Bioinformatics

CS 482

Bin Ma

LaTeXed by Sibelius Peng

Preface

Disclaimer Much of the information on this set of notes is transcribed directly/indirectly from the lectures of CS 482 during Winter 2022 as well as other related resources. I do not make any warranties about the completeness, reliability and accuracy of this set of notes. Use at your own risk.

For any questions, send me an email via https://notes.sibeliusp.com/contact.

You can find my notes for other courses on https://notes.sibeliusp.com/.

Sibelius Peng

Contents

1

Introduction

This course is officially titled as "Computational Techniques in Biological Sequence Analysis". However, this is an old title and this course has been offered for over approximately twenty years. It should actually be called "Algorithms in Bioinformatics". Around 1980, this area has a different name: Computational Biology. You may also hear another name: DNA computing. It is another area, and it does not solve biological problem. What is bioinformatics? It is biology + informatics. Biology is the reason, goal, purpose and informatics is the method.

Biology can be studied at different scales. In old days, biology tries to study organisms, namely living things, such as bacteria, animals. This is because people didn't have tools to study at a lower level at that time. Now people study organs and tissues. Then people can look into cell level, molecular level. At molecular level, DNA is chain of nucleotide bases. Protein is chain of amino acids.

There are a lot of public and free molecular data. There are tremendous amount of public biomolecule data and free software. For example:

- NCBI's sequence data bank: https://www.ncbi.nlm.nih.gov/nuccore/NC_045512
- PDB protein structure database: https://www.rcsb.org/structure/6vxx

Why do people do bioinformatics? The goal is to understand life at molecular level. Especially, for human health. For example, sequencing SARS-Cov2 genomes allowed people study the evolution of this virus. Study the structure of the spike protein and its interaction with the host cells. A lot of human diseases are related to genetics.

There are two areas of bioinformatics.

- Determine the molecule information, by analyzing the data produced by measuring instruments. Typically the data is in large scale and high throughput, which is hard for people to look at.
- Use the molecular data to make inference. For example, make prediction given the existing data.

Consider an example of genome sequencing. From wikipedia,

> The Human Genome Project (HGP) was an international scientific research project with the goal of determining the base pairs that make up human DNA, and of identifying, mapping and sequencing all of the genes of the human genome from both a physical and a functional standpoint. It remains the world's largest collaborative biological project. Planning started after the idea was picked up in 1984 by the US government, the project formally launched in 1990, and was declared complete on April 14, 2003. Level "complete genome" was

achieved in May 2021.

Bioinformatics played an essential role in analyzing the data and assemble the genome. Today one can sequence a human's genome with $<$ \$1000 in a couple of weeks. Bioinformatics is the key to utilize the NGS (next generation sequencing) data for genome sequencing. As such, today's cancer treatment starts to become personalized. And many new drugs now require gene sequencing as companion diagnostic.

1.1 Objectives of this course

- Know bioinformatics
  - Purpose and method
  - General topics
- Learn classic problems and algorithms in bioinformatics
- Learn wide-applicable computational techniques
  - String algorithms
  - Hidden Markov Model
  - Log likelihood ratio score
  - Statistical validation
  - A bit of machine learning

Some typical problems:

- Gene prediction problem
- Find the longest shared substring between human and mouse genomes. If we want to find similarities instead of exact matches, this will lead us to the homology search problem.
- Peptide Identification

2

Sequence Alignment

2.1 Biology

Consider two protein sequences:

- AVP78042.1 spike protein: `MLFFL...`
- YP_009724390.1 surface glycoprotein: `MFVFL...`

They look similar, how do we know these two proteins are similar? It's better to visualize them properly so that we can see the similarity. This is done by sequence alignment, and there are many existing tools: such as Clustal Omega. It's a common belief that these two proteins are developed from a common ancestor, then they evolve from an evolution tree. This is called homology.

There are two classes of nucleotide bases:

- Purine: A and G
- Pyrimidine: T and C

Base pairs are bonded by hydrogen bonds. Also, G-C bind stronger because of 3 H-bonds. DNA molecule is oriented.

So we know that DNA is double-helical, with two complementary strands. And the complementary bases: A-T, G-C. For example, the *reverse* complement of `AAGGTAGC` is `GCTACCTT`, because DNA is oriented.

Now consider DNA mutation. DNA mutates with a small probability when inherited by the offspring. For example, one base can be substituted by another, because there might be copy errors. This creates different alleles of the same gene. From wiki, allele is one of two, or more, forms of a given gene variant. When we inherit the DNA from parents, we only inherit half of each parent's genome. These together cause the differences between individuals of the same species.

Single Nucleotide Polymorphisms is a germline substitution of a single nucleotide at a specific position in the genome. Single base variation between members of a species. For Human, 90% of all human genetic variation is caused by SNPs. SNPs occur every 100 to 300 bases along the 3-billion-base human genome. It's a major risk for genetic disease, because when one base pair mutates, it will cause the express protein's functions.

2.2 Compare DNA sequences

The most often used distance on strings in computer science is Hamming distance. This makes some sense on comparing DNA sequences in some cases: substitution. But there are other mutations: insertion/deletion (indel), which cannot be modelled correctly by Hamming distance. Other DNA rearrangements can also happen. But substitutions and indel are the two mutations we concern the most for this course.

Edit distance

Instead, we can use **edit distance**. How "far" away are two sequences from each other? Edit distance is defined to be the minimum number of edit operations needed to convert one to another. Here edit operations include substitutions and indels.

Note that Edit distance is a distance metric:

- Identity: $d(x, y) = 0$ if and only if $x = y$.
- Symmetry: $d(x, y) = d(y, x)$.
- Triangular inequality: $d(x, z) \leq d(x, y) + d(y, z)$.

Now we prepare for the algorithm. For convenience of the proof, we treat each occurrence of the same letter different. For example, `ATAA -> ATA, A` can be done by either deleting the 2nd or 3rd letter `A` from the first string. These are different editing paths. This does not affect our definition of edit distance, but makes our later proof more precise.

Now it's ready to develope the dynamic programming algorithm for edit distance. Let $D[i, j]$ = edit distance between $S[1..i]$ to $T[1..j]$. Consider the edit operations associated with $S[i]$ and $T[j]$ the optimal edit operations. One of the following cases will happen:

1. $S[i]$ is deleted: $D[i, j] = D[i - 1, j] + 1$
2. $T[j]$ is inserted: $D[i, j] = D[i, j - 1] + 1$
3. $S[i]$ becomes $T[j]$: $D[i, j] = D[i - 1, j - 1] + \delta(S[i], T[j])$

where $\delta(S[i], T[j]) = 0$ if $S[i] = T[j]$ and 1 if not.

Algorithm 1: Dynamic Programming Algorithm for Edit distance

```
1 D[0,0] = 0
2 D[0,i] = i for i = 1..|S|
3 D[i,0] = i for i = 1..|T|
4 for i ← 1..|S| do
5 |   for j ← 1..|T| do
6 |   |   D[i,j] = min{D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+δ(S[i],T[j])}
7 return D[|S|,|T|]
```

Longest Common Subsequence

Another way to evaluate the similarity of two sequences is through LCS. A subsequence is obtained by deleting some of the letters from the supersequence and concatenating the remaining letters together. For example, LCS of `ATGCATTTA` and `ATGTACTTTC` is `ATGATTT`. LCS can be computed with dynamic programming as well.

2.3 Alignment

The third way to compare to sequences is through sequence alignment. We want to insert spaces (-) to two sequences so that we can align them together and they are most similar column-wisely. For example,

```
ATGCA-TTTA
|||  |  || |
ATGTACTT-A
```

By "similar", we usually need to use a scoring function. We define the alignment score to be **the total of column scores**. And each column is assigned by a constant score depending on matching conditions. For example, simple score scheme would be

- Match = 1
- Mismatch = -1
- indel = -1

Consider two alignments with the score scheme above:

```
AATGCGA-TTTT
||  |  |||
G-TG--ACTTTC
```

has score 0.

```
AATG-CGATTTT
||  |  ||
G-TGAC-TTTC-
```

has score -1.

As a side note, alignment can "simulate" LCS and edit distance. We have this following table for the scoring system:

	alignment	LCS	edit dist
match	+1	1	0
mismatch	-1	0	-1
indel	-1	0	-1

Now we can develop the develop the dynamic algorithm for the alignment. Let $f(a, b)$ be the scoring scheme for a column with a and b. Here one of a and b can be the dash character -. Thus $f(\text{-}, x)$ and $f(x, \text{-})$ represent scores of indels. The input is S, T. Let $D[i, j]$ be the optimal alignment score for $S[1..i], T[1..j]$.

Now we want to derive a recurrence relation. Suppose we are to align $S[1..i], T[1..j]$. Consider the *last column* of the optimal alignment. Three cases can happen.

$S[1..i-1]$ $S[i]$	$S[1..i-1]$ $S[i]$	$S[1..i-1]$ -
$T[1..j-1]$ $T[j]$	$T[1..j-1]$ -	$T[1..j-1]$ $T[j]$
(1)	(2)	(3)

Note that in each case, the sub-alignment without the last column is an optimal one. Prove by contradiction, assume there's a better sub-alignment without the last column. Then the optimal alignment (for the whole) should have used this better sub-alignment, instead of the current sub-alignment.

Then we have the recurrence for these three cases:

(1) $D[i,j] = D[i-1,j-1] + f(S[i], T[j])$

(2) $D[i,j] = D[i-1,j] + f(S[i], \text{-})$

(3) $D[i,j] = D[i,j-1] + f(\text{-}, T[j])$

Algorithm 2: Sequence alignment

```
1 D[0,0] = 0
2 for i ← 1..m do
3 |   D[i,0] = i× indel
4 for j ← 1..n do
5 |   D[0,j] = j× indel
6 for i ← 1..m do
7 |   for j ← 1..n do
8 |   |   D[i,j] = max { D[i-1,j-1] + f(S[i], T[j])
  |   |                { D[i-1,j] + f(S[i], -)
  |   |                { D[i,j-1] + f(-, T[j])
9 return D[m,n]
```

Then the time complexity is $O(mn)$ where $|S| = m, |T| = n$. In practice, when we run the DP algorithm and build the DP table, we can add arrows from cell to cell: which of the three cases (sub-alignments) is chosen to get the current cell. Then we can backtrack the arrows and get the actual alignment. For backtracking, it is $O(n+m)$. Space complexity is $O(nm)$.

In practice, we don't need to record/store these arrows. Instead, when we backtrack, we can calculate the arrows based on the max of three options dynamically, which takes $O(m+n)$. This saves the cost of storing arrows, which costs $O(mn)$.

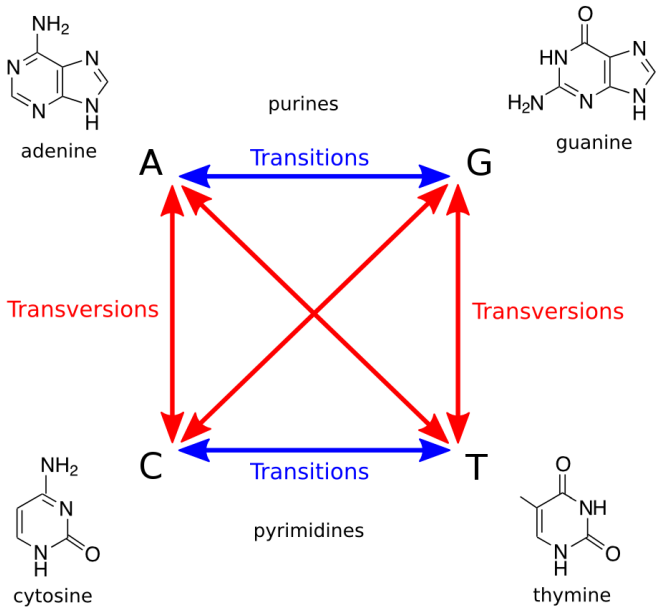
Moreover, if only score is needed, then space complexity can be reduced to linear space.

Note that the algorithm is designed for any score scheme $f(x,y)$. We indeed separate the algorithm and scoring. So we can optimize score scheme and the algorithm independently. Dijkstra once said:

> The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer.

2.4 Score Function

> Transition vs. Transversion
>
> Recall that within DNA, there are two classes. One is purines (A, G), and the other is pyrimidines (C, T). Within the same class, they have similar chemical structure. Moreover, the mutation within the same class is called transitions; across the two classes, is called transversions. Transition is cheaper than transversions.



Picture from https://dodona.ugent.be/en/activities/1351011626/

Transition happens more frequently 2/3 of SNPs are transitions. In other words, transition is easier and therefore should be less penalized. For example,

```
AAAGCAAA        AAAGCAAA
AAAT-AAA   vs   AAA-TAAA
```

Observe that GT is transversion and CT is transition. The right alignment is probably better. This can be easily achieved by changing score scheme $f(a, b)$.

So how to build a score function? First, we need to know what you want. There are two purposes:

1. the optimal alignment reveals the true evolutionary history.
2. high score indicates homology (derived from same ancestor).

We want purpose 1 if possible, but purpose 2 is also useful.

For purpose 1, note that we might be wrong: score function might not have the power to reveal the history. For example, if the history goes as `A -> T -> A`, then in the sequence alignment, the final comparison is between `A` and `A`, which will not give us the true history. So you should never trust that an alignment must be right. It just optimizes the score. Should we give up purpose 1 at all?

For purpose 1, the optimal alignment may be *approximately* correct *under certain conditions* in practice. As long as we know the limitation, we can still use it. For example, for the following alignment, it is "very likely" the alignment is approximately equal to the evolutionary history.

```
ACGTATTACCGG-TTACCG
|||  ||||||||  ||||||
ACGGATTACCGGATTACCG
```

So we should keep in mind that when the score is low, alignment itself is not too useful.

Now let's try to improve our score function. Consider two alignment (with gaps):

```
AGATTTTTTTC     AGATTTTTTTTTC
AGA---TTTTC     AGA-T-T-T-T-C
```

The left seems "simpler" than the right, intuitively. Indeed, during evolution, indels are relatively rare. However, insertion or deletion a segment of k consecutive bases is much easier than k scattered indels. But our current scoring method (adding up column scores) cannot distinguish the two. Currently, a gap of length k costs $k\times$indel. Thus, this is called the **linear gap penalty**. Denote the penalty function by $g(x)$, and it should have negative value. Left's penalty is $g(3)$ and right is $3 \times g(1)$.

Consecutive insertions or deletions are called a gap. Suppose the gap penalty of a length k gap is $g(k)$ instead of the simple $c \times k$. Assume $g(x) + g(y) \leq g(x + y)$, thus a convex function. Otherwise does not serve the purpose of grouping indels. In this case, can the old DP algorithm still work? Recall the three cases:

$S[1..i-1]$ $S[i]$	$S[1..i-1]$ $S[i]$	$S[1..i-1]$ -
$T[1..j-1]$ $T[j]$	$T[1..j-1]$ -	$T[1..j-1]$ $T[j]$
(1)	(2)	(3)

First case is the same. The second case might be wrong. The last column might depend on previous columns. If the second to the last column of T is already a dash, then adding a new dash will increase the length of gap. The we cannot use DP algorithm anymore, in the sense that we cannot prove its correctness. We do not know the contribution of the last column to the gap penalty in the last two cases.

Assume we know the length of gap, then we can apply the recurrence relation. We still use $D[i, j]$ to denote the optimal alignment score of $S[1..i]$ and $T[1..j]$. We change cases 2 and 3 to include the last gap (not the last column). Then three cases change accordingly:

$S[1..i-1]$ $S[i]$	$S[1..i-k]$ $S[i-k+1..i]$	$S[1..i]$ - - - - -
$T[1..j-1]$ $T[j]$	$T[1..j]$ - - - - -	$T[1..j-k]$ $T[j-k+1..j]$
(1)	(2)	(3)

Then $D[i, j] =$ max of the following three cases:

(1) $D[i-1, j-1] + f(S[i], T[j])$

(2) $\max_{1\leq k\leq i} D[i-k, j] + g(k)$

(3) $\max_{1\leq k\leq j} D[i, j-k] + g(k)$

Now the time complexity will change: $O(mn(m+n))$, which is cubic.

In bioinformatics, very often we face the choice between:

- Reality: How close it approximates the real biology.
- Simplicity: How easy it can be computed.

We can simplify $g(k)$ a little bit. We basically want a function that grows slower than linear. We introduce **affine gap penalty**, in contrast to linear gap penalty:

$$g(k) = a + b \cdot k$$

where a is the gap open penalty: whenever we have a gap, we pay for it; b is the gap extension penalty: for example, once we open the gap, we pay less for the second indel within a group.

Example: Affine gap penalty

match $= 1$; mismatch $= -1$; gap open $= -5$; gap extension $= -1$.

```
ATAGG--AAG
||  ||  | |
ATTGGCAATG
```

6 match, 2 mismatch, 1 gap open, 2 gap extension, then the score is

$$6 - 2 + (-5 - 1 \times 2) = -3$$

```
ATAGG-AA-G
||  ||  || |
ATTGGCAATG
```

Similarly, the score is

$$7 - 1 - 5 - 1 - 5 - 1 = -6$$

Now the old algorithm doesn't work anymore. Consider the last column of an alignment again:

```
AT-GG-    ATGG--
||  ||    ||  |
ATTGGC    ATTGGC
```

When considering the last column, we need to know the second to the last column. When the last column is an indel, the added cost depends on the previous column. Because by induction we know the optimal solution for $D[i, j-1]$, we can encode the previous column's configuration. We compute the optimal solution by limiting the last column to one of the following three configurations:

```
ATAGG      ATAGG-     ATAGGC
||  ||     ||  ||     ||  ||
ATTGG      ATTGGC     ATTGG-
```

$D_0[i,j]$ $D_1[i,j]$ $D_2[i,j]$

$D_0[i,j]$ requires that the last column must be $S[i]$ v.s. $T[j]$, not indel. $D_1[i,j]$ is - v.s. $T[j]$, and $D_2[i,j]$ is $S[i]$ v.s. -. We then can develop recursion relationship easier by defining more subproblems. We only distinguish them by the last column, there is no constraint for columns before the last column.

Now let's examine how to calculate $D_0[i,j]$. There are three cases:

$$\begin{matrix} S[1..i-2]S[i-1] & S[i] \\ T[1..j-2]T[j-1] & T[j] \end{matrix} \longrightarrow D_0[i,j] = D_0[i-1,j-1] + f(S[i], T[j])$$

$$\begin{matrix} S[1..i-1] & \text{-} & S[i] \\ T[1..j-2] & T[j-1] & T[j] \end{matrix} \longrightarrow D_0[i,j] = D_1[i-1,j-1] + f(S[i], T[j])$$

$$\begin{matrix} S[1..i-2] & S[i-1] & S[i] \\ T[1..j-1] & \text{-} & T[j] \end{matrix} \longrightarrow D_0[i,j] = D_2[i-1,j-1] + f(S[i], T[j])$$

We then can do the similar thing for $D_1[i,j]$:

$$\begin{matrix} S[1..i-1]S[i] & \text{-} \\ T[1..j-2]T[j-1] & T[j] \end{matrix} \longrightarrow D_1[i,j] = D_0[i,j-1] + \text{gapopen} + \text{gapext}$$

$$\begin{matrix} S[1..i] & \text{-} & \text{-} \\ T[1..j-2] & T[j-1] & T[j] \end{matrix} \longrightarrow D_1[i,j] = D_1[i,j-1] + \text{gapext}$$

$$\begin{matrix} S[1..i-1] & S[i] & \text{-} \\ T[1..j-1] & \text{-} & T[j] \end{matrix} \longrightarrow D_1[i,j] = D_2[i,j-1] + \text{gapopen} + \text{gapext}$$

The relation for $D_2[i,j]$ is symmetric. In summary, we have the recurrence relation

$$D_0[i,j] = f(S[i],T[j]) + \max \begin{cases} D_0[i-1,j-1]; \\ D_1[i-1,j-1]; \\ D_2[i-1,j-1]; \end{cases}$$

$$D_1[i,j] = \text{gapext} + \max \begin{cases} D_0[i,j-1] + \text{gapopen}; \\ D_1[i,j-1]; \\ D_2[i,j-1] + \text{gapopen}; \end{cases}$$

$$D_2[i,j] = \text{gapext} + \max \begin{cases} D_0[i-1,j] + \text{gapopen}; \\ D_1[i-1,j] + \text{gapopen}; \\ D_2[i-1,j]; \end{cases}$$

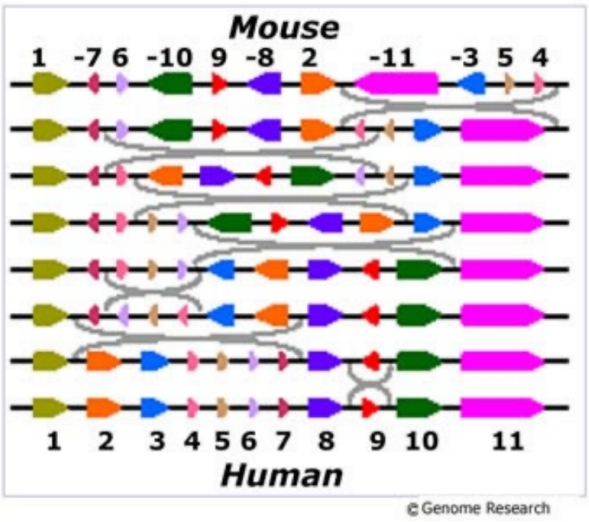
Note the grayed cases can't be optimal so can be safely removed.

This is still DP algorithm. We need to be careful when backtracking. The running time is $O(nm)$, might be approximately 3 times slower, but better than the general gap penalty's cubic time. This is okay because the model is more expressive. This model is first published in 1982, by Gotoh, O.: *An improved algorithm for matching biological sequences*.

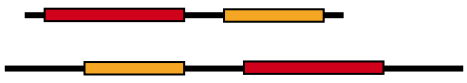
3

Local Alignment and Linear Space Alignment

In biology, we are not comparing S, T directly. Instead, we are comparing small areas of S, T.



Interestingly, if we compare chromosome X of mouse and human, they are quite similar. Each colored block is relatively conserved, but different in orders and orientations. Seven inversions are required to put them in the correct order and orientation. This is called "sorting by reversals". This is an interesting study from Pavel Pevzner.



In this case, if we do the global/sequence alignment, we can align only one part: we will lose yellow part when we align red. Conserved regions are "local" to the genome/chromosome. But previous alignment is "global". We need a proper model to define "local" similarity.

For the problem of local alignment, we are given two sequences S and T. We want to find substrings of S and T that maximizes the alignment score. I.e., The indels at the beginning and end of the two strings are free.

Local alignment score is at least 0, because for the worst case, we have two empty strings. The model only makes sense for alignment but not edit distance nor LCS. Is the optimal local alignment a local part of an optimal "global" alignment? No. If we align `AT` and `TA`, the global alignment would be $\begin{matrix}\texttt{AT}\\ \texttt{TA}\end{matrix}$

if the indel gets the most penalty. While the local alignment would simply be $\begin{matrix}\texttt{A}\\\texttt{A}\end{matrix}$

3.1 Prefix and suffix alignment

Consider a related different problem: **prefix alignment**: find the highest-scoring alignment between two prefixes of the two sequences. We want to find i, j to maximize alignment-score$(S[i..i], T[1..j])$.

Similarly, consider **suffix alignment**. That is, we choose two suffixes, and align them together optimally. We want to compute $\max_{i',j'}$ score$(S[i'..m], T[j'..n])$

Let $D[i, j]$ denote the optimal "suffix alignment" alignment score of $S[1..i]$, $T[1..j]$. That is, $D[i, j]$ is the maximum alignment score for $S[i'..i]$ and $T[j'..j]$ for all i' and j'. Consider the last column of this optimal "suffix" alignment. Four cases arise:

1. $S[i]$ v.s. $T[j]$
2. $S[i]$ v.s. -
3. $T[j]$ v.s. -
4. an empty alignment

Case 4 is the only new case comparing to the basic alignment. Then the DP algorithm's recurrence relation would be

$$D[i,j] = \max \begin{cases} D[i-1, j-1] + f(S[i], T[j]); \\ D[i-1, j] + f(S[i], \text{-}); \\ D[i, j-1] + f(\text{-}, T[j]); \\ 0 \end{cases}$$

Then the $D[m, n]$ will be the last cell in the DP table: optimal suffix alignment between S and T. Previously, initial cases might have negative scores. Due to the new rule here, we just put zeros.

Consider a suffix alignment example where match = 1, mismatch = indel = -1.

		C	A	T	T	C
	0	0	0	0	0	0
A	0	0	1	0	0	0
T	0	0	0	2	1	0
T	0	0	0	1	3	0
G	0	0	0	0	2	2
A	0	0	0	0	1	1

This gives alignment $\begin{matrix} & \texttt{ATTGA}\\ \texttt{C} & \texttt{ATTC-}\end{matrix}$

3.2 Local Alignment

Recall that for suffix alignment, $D[i, j]$ denote the optimal "suffix alignment" alignment score of $S[1..i], T[1..j]$. I.e., $D[i, j]$ is the maximum alignment score for $S[i'..i]$ and $T[j'..j]$ for all i' and j'. Therefore, optimal local alignment score is just $\max_{i,j} Di, j$. The algorithm will be straightforward:

Algorithm 3: Local alignment

1 Fill the dynamic programming table is the same as suffix alignment.
2 Find (i, j) to maximize $D[i, j]$, and backtrack from there.

For example,

		C	A	T	T	C
	0	0	0	0	0	0
A	0	0	1	0	0	0
T	0	0	0	2	1	0
T	0	0	0	1	(3)	0
G	0	0	0	0	2	2
A	0	0	0	0	1	1

Then the local optimal alignment is the optimal suffix alignment of $T[1..4]$ and $S[1..3]$.

The algorithm was first proposed by Temple Smith and Michael Waterman in 1981. It works for both linear and affined gap penalty. It is known popularly as the Smith-Waterman algorithm. The global alignment algorithm was called the Needleman-Wunsch algorithm, which was published in 1970.

Time complexity is quadratic. Space complexity is $O(mn)$. If we only want the score, we can just use a max variable, which takes $O(1)$ space. If we want the end positions, namely i, j, we can still introduce extra two variables $\max I, \max J$.

If we want the start positions, namely i', j', it would be a bit harder. Recall in suffix alignment, there are four cases. For case 4, we have $i' = i + 1, j' = j + 1$. For other three cases, we simply copy i', j' from the smaller alignment (sub-alignment) because they are same. So we just introduce two variables $\max I', \max J'$.

Similarly, we can do affine gap local alignment:

$$D_0[i,j] = f(S[i],T[j]) + \max \begin{cases} D_0[i-1,j-1]; \\ D_1[i-1,j-1]; \\ D_2[i-1,j-1]; \\ 0 \end{cases}$$

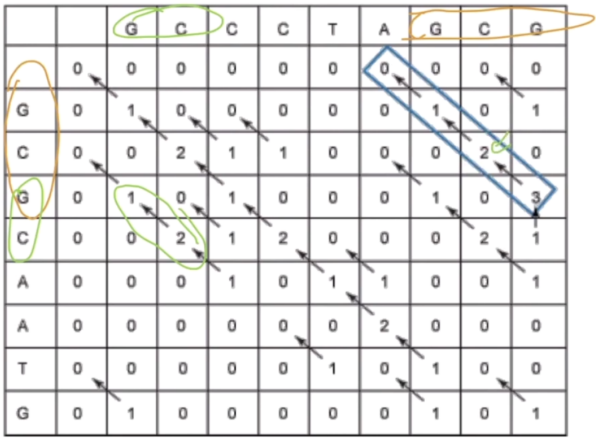
$$D_1[i,j] = \text{gapext} + \max \begin{cases} D_0[i,j-1] + \text{gapopen}; \\ D_1[i,j-1]; \\ D_2[i,j-1] + \text{gapopen}; \\ 0 \end{cases}$$

$$D_2[i,j] = \text{gapext} + \max \begin{cases} D_0[i-1,j] + \text{gapopen}; \\ D_1[i-1,j] + \text{gapopen}; \\ D_2[i-1,j]; \\ 0 \end{cases}$$

Algorithm is as before, except that score is now lower bounded by 0. Afterward, find maximum element in all 3 tables, and backtrack until reaching a 0.

3.3 Many local alignments

It's sometimes useful to find many local alignments of S and T. For example, when there are multiple similar regions between the two input strings. We can let the algorithm output multiple alignments.



		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	0	0	0	0	0	1	0	1
C	0	0	2	1	1	0	0	0	2	0
G	0	1	0	1	0	0	0	1	0	3
C	0	0	2	1	2	0	0	0	2	1
A	0	0	0	1	0	1	1	0	0	1
A	0	0	0	0	0	0	2	0	0	0
T	0	0	0	0	0	1	0	1	0	0
G	0	1	0	0	0	0	0	1	0	1

3.4 Fit Alignment

There are some scenarios where local alignment is not best model. Given sequences S and T. Find a global alignment between S and a substring of T, maximizing the alignment score. We are trying to fit S into T. Deleting the prefix of T is free, deleting the suffix of T is free.

We can use similar idea as before. The DP table aligns T horizontally, S vertically. We can start from anywhere from T, so all zeros. Then for S, we need to initialize properly, $-1, -2, \ldots$. For local alignment, we can stop at any i, j. For fit alignment, we can stop at any j but not any i: we need to find the max value in the last row.

3.5 Linear Space Alignment

Why linear space? Computer RAM used to be very expensive in 80s. There was a prediction " The cost for 128 kilobytes of memory will fall below 100 bucks in the near future" Creative Computing magazine. December 1981, page 6. Even today, keeping everything in the L2 cache may speed up the computation. We have learned the linear space if only alignment score, instead of the alignment, is required. Let's now develop a linear space alignment. We focus on **global** alignment model first.

The idea is to use **divide and conquer**. We want to find j such that the optimal alignment between S and T consists of two parts:

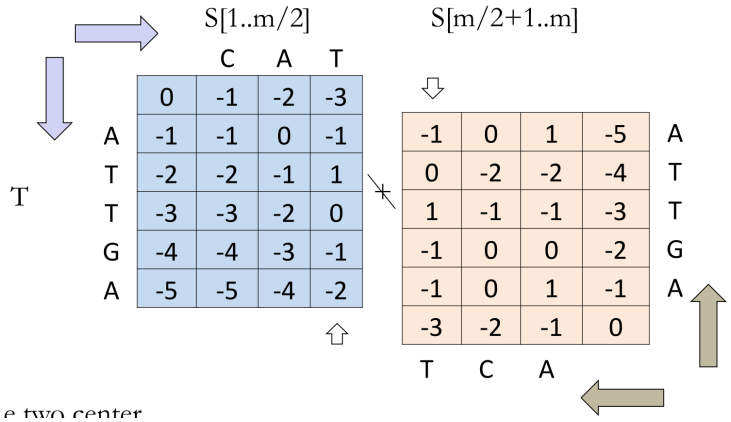
- $S[1..m/2]$ aligns with $T[1..j]$
- $S[m/2+1..m]$ aligns with $T[j+1..n]$

Then we can use divide and conquer. However, we need to compute j in linear space. Note that there may be more than one j satisfying the condition. Any one of them will do the job.

Claim j satisfies the desired condition iff it maximizes

$$\text{alignScore}(S[1..m/2], T[1..j]) + \text{alignScore}(S[m/2+1..m], T[j+1, n])$$

Then we can loop for all j. Let D be the dynamic programming table for aligning S and T. Note that $\text{alignScore}(S[1..m/2], T[1..j])$ is stored in the middle column $(m/2)$ of the table, which takes linear space only. For $\text{alignScore}(S[m/2+1..m], T[j+1, n])$, we can reverse the alignment, which doesn't affect the score. So we can fill DP table backward.



Computing the two center columns requires linear space. Then the algorithm is as follows:

Algorithm 4: Linear Space Alignment

```
1 Align(S, T):
2     if |S| = 1 then
3         return a trivial alignment
4     Use the previous idea to find j that maximizes
        alignScore(S[1..m/2], T[1..j]) + alignScore(S[m/2..m], T[j+1..n])
5     return Concatenation of Align(S[1..m/2], T[1..j]) and Align(S[m/2..m], T[j+1..n])
```

To see its time complexity, we can either use induction or do in in a sloppy way. We know that

$$T(m, n) \leq mn + T(m/2, j) + T(m/2, n-j)$$

Then if we expand the subproblems, the time complexity becomes

$$\begin{aligned} T(m,n) &\leq mn + T(m/2, j) + T(m/2, n-j) \\ &\leq mn + \frac{mn}{2} + \frac{mn}{4} + \cdots \\ &\leq 2mn \end{aligned}$$

But we have linear space complexity.

Related papers:

- *A linear space algorithm for computing longest common subsequences* by D.S. Hirschberg.
- *Optimal alignments in linear space*.

  > "The goal of this paper is to give Hirschberg's idea the visibility it deserves by developing a linear-space version of Gotoh's algorithm."

How to do local alignment in linear space? Recall the trick to find the boundaries of optimal local alignment in linear space. We can use this trick, which takes linear space, to find i, i', j, j'. Then take these four numbers, and then call global linear alignment to get the actual alignment. This is called reduction. Similarly, we can do affined gap penalty in linear space.