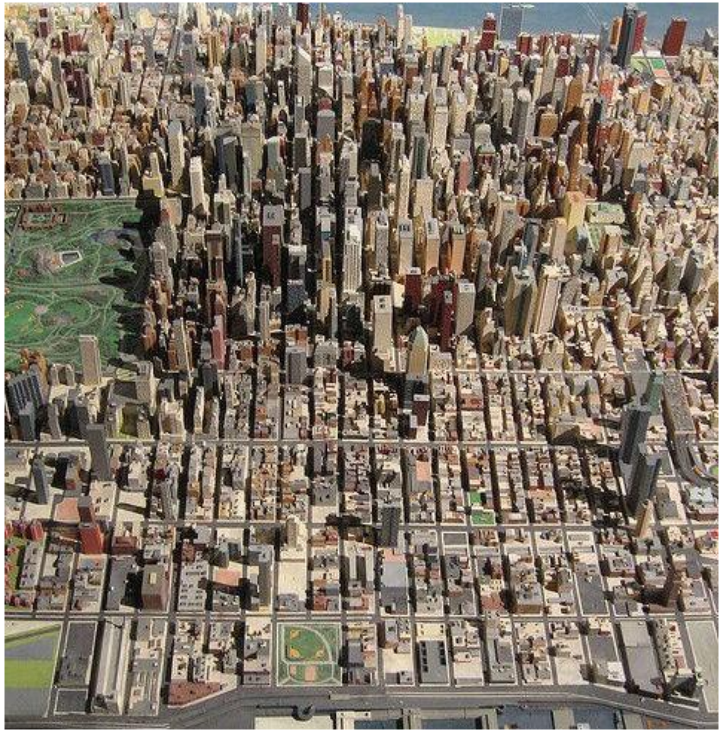
Comp 555 - BioAlgorithms - Spring 2022



- How our Manhattan Tour relates to sequences
- Problem set #3 due tonight
- Look for Problem set #4 later Tonight
- Have a good break

Sequence Alignment

Comparing Sequences

- What makes two sequences similar?
- What is the best measure of similarity?
- Consider the two DNA sequences *v* and *w* :

```
v: TAGACAAT
w: AGAGACAT
   11111100 = 6
```

- The Hamming distance, $d_H(v, w) = 6$, is large but the sequences seem to have more similarity
- What if we allowed for insertions and deletions?

Allowing Insertions and Deletions

- By shifting each sequence over one position:

```
v: _TAGACAAT
w: AGAGACAT_
   110000011 = 4
```

Shifts and gaps:

```
v: _TAGACAAT
w: AGAGAC_AT
   110000100 = 3
```

Another one:

```
v: T_AGACAAT
w: AGAGACA_T
   110000010 = 3
```

- The edit distance: $d_H(v, w) = 3$.
- Hamming distance neglects insertions and deletions

Edit Distance

- Vladimir Levenshtein introduced the notion of an "edit distance" between two strings as the minimum number of elementary operations (insertions, deletions, and substitutions) to transform one string into the other in 1965.

- $d_L(v,w)$ = Minimum number of elementary operations to transform $v \rightarrow w$

- Computing Hamming distance is a trivial task

- Computing edit distance is less trivial

Vladimir Levenshtein

1935 - 2017

Edit Distance: Example

```
TGCATAT → ATCCGAT in 5 steps


TGCATAT →(DELETE last T)
TGCATA  →(DELETE last A)
TGCAT   →(INSERT A at front)
ATGCAT  →(SUBSTITUTE C for G)
ATCCAT  →(INSERT G before last A)
ATCCGAT      (Done)
```

What is the edit distance? 5? (Recall it has to be the *minimum*)

Edit Distance: Example (2nd Try)

```
TGCATAT → ATCCGAT in 4 steps


TGCATAT  →   (INSERT A at front)
ATGCATAT →   (DELETE 2nd T)
ATGCAAT  →   (SUBSTITUTE G for 2nd A)
ATGCGAT  →   (SUBSTITUTE C for 1st G)
ATCCGAT      (Done)
```

But is 4 the minimum edit distance? Is 3 possible?

- Edit sequences are invertible, i.e given *v* → *w*, one can generate *w* → *v*, without recomputing
- A little jargon: Since the effect of insertion in one string can be accomplished via a deletion in the other string these two operations are correlated. Often algorithms will consider them together as a single operation called INDEL

An Aside: Longest Common Subsequence

- A special case of alignment where only *matches, insertions, and deletions* are allowed
- A variant of Edit distance, sometimes called LCS distance, where only indels are allowed
- A subsequence need not be contiguous, but the symbol order must be preserved
  Ex. If v = ATTGCTA then AGCA and TTTA are subsequences of v, but TGTT and ACGA are not
- All substrings of *v* are subsequences, but not vice versa
- Edit distance, d_{LCS}, is related to the length of the LCS, *s*, by the following relationship:

$$d_{LCS}(u,w) = len(v) + len(w) - 2s(u,w)$$

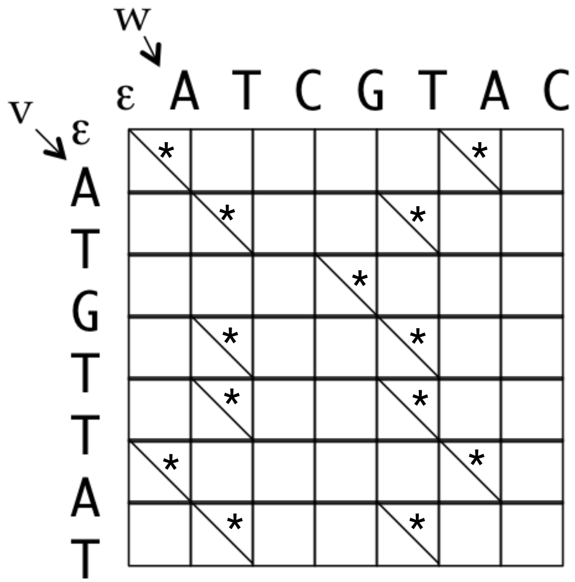
Example:

```
ANUNCLEIKE
UNCBEATDUKE

                   len  LCS
anUNC_lE____iKE     10 - 6 = 4
__UNCb_Eatdu_KE     11 - 6 = 5
```

LCS as a Manhattan Tour (Dynamic Program)

There are similarities between the LCS and MTP

- All possible possible alignments can be represented as a path from the string's beginning (source) to its end (destination)
- Horizontal edges add gaps in v
- Vertical edges add gaps in w
- Diagonal edges are a match
- Notice that we've only included valid diagonal edges for "matches" in our graph
- *An maximum LCS is a path from (ε,ε) to the end of both strings that matches the most bases (a.k.a. a Manhattan tour)*

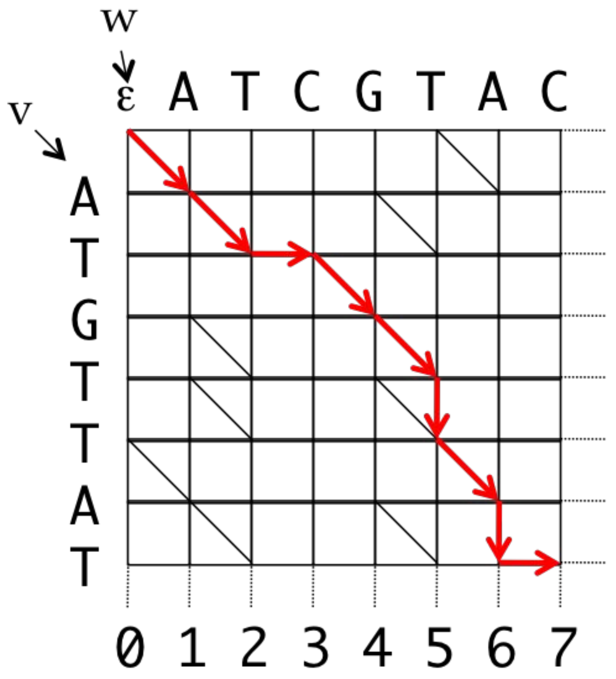


The "Space" of All Alignments

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

```
0 1 2 2 3 4 5 6 7 7
v A T _ G T T A T _
w A T C G T _ A _ C
0 1 2 3 4 5 5 6 6 7
```

- Path:
  (0,0), (1,1), (2,2), (2,3), (3,4), (4,5), (5,5), (6,6), (7,6), (7,7)

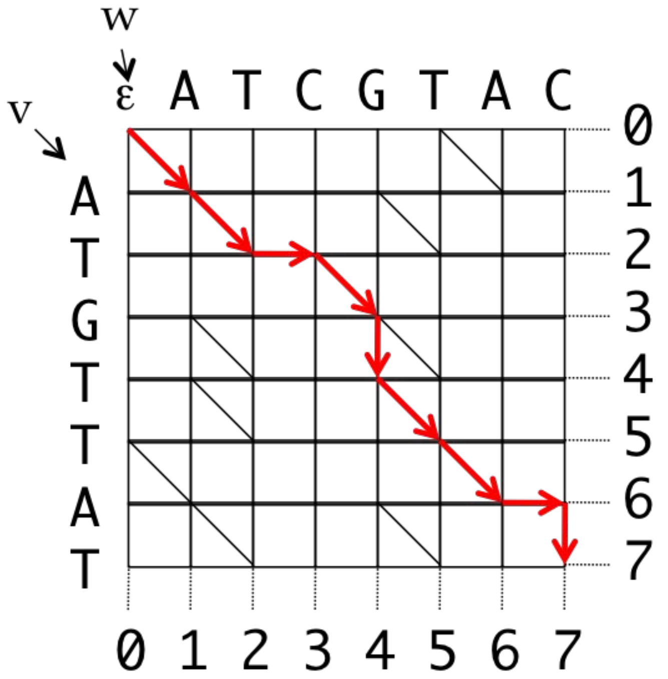


Alternate Alignment

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

```
0 1 2 2 3 4 5 6 6 7
v A T _ G T T A _ T
w A T C G _ T A C _
0 1 2 3 4 4 5 6 7 7
```

- Path:
  (0,0), (1,1), (2,2), (2,3), (3,4), (4,4), (5,5), (6,6), (6,7), (7,7)

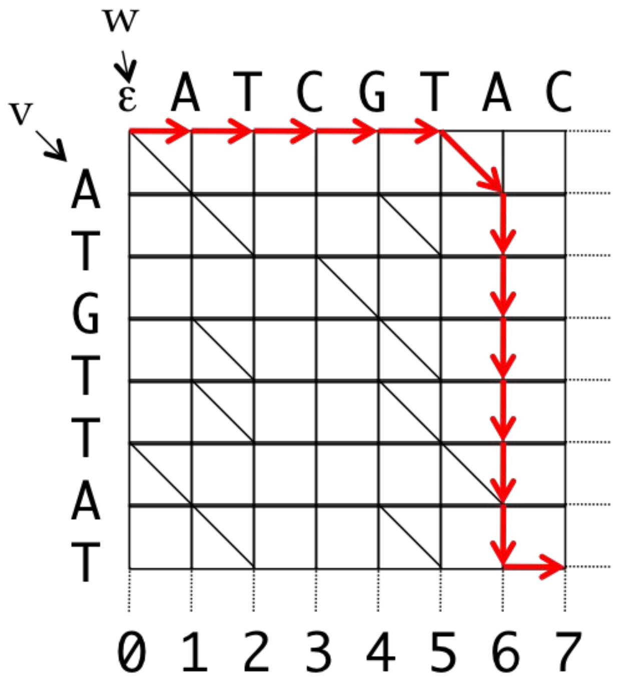


Even Bad Alignments

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

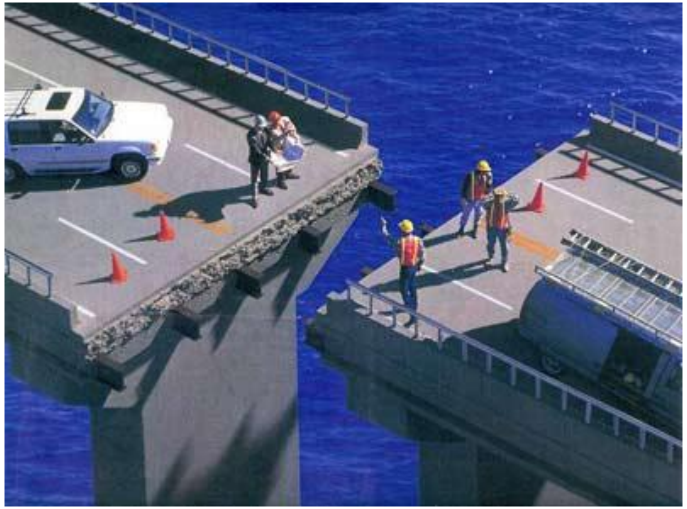
```
0 0 0 0 0 0 1 2 3 4 5 6 7 7
v _ _ _ _ _ _ A T G T T A T _
w A T C G T A _ _ _ _ _ _ C
0 1 2 3 4 5 6 6 6 6 6 6 6 7
```

- Path:
  (0,0), (0,1), (0,2), (0,3), (0,4), (0,5), (1,6), (2,6), (3,6), (4,6), (5,6), (6,6), (7,6), (7,7)



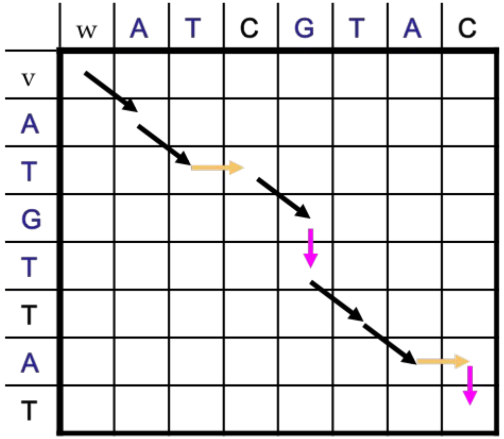
What makes a good alignment?

- Using as many diagonal segments, when they correspond to matches, as possible. Why?
- The end of a good alignment from (j...k) begins with a good alignment from (i..j)
- Same as Manhattan Tourist problem, where the ***sites*** are only on the diagonal streets!
- Set diagonal street weights = 1, and horizontal and vertical weights = 0



LCS: Dynamic Program

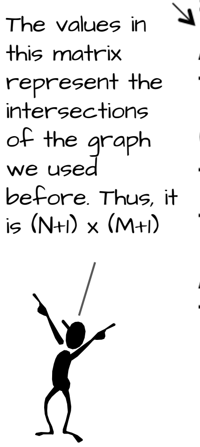
$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_i \quad \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

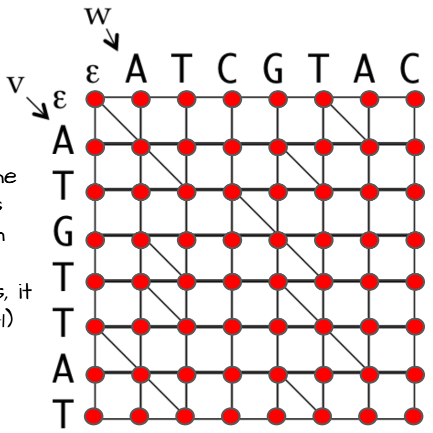


Step 1

Initialize 1st row and 1st column to all zeroes.

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0							
T	0							
G	0							
T	0							
T	0							
A	0							
T	0							

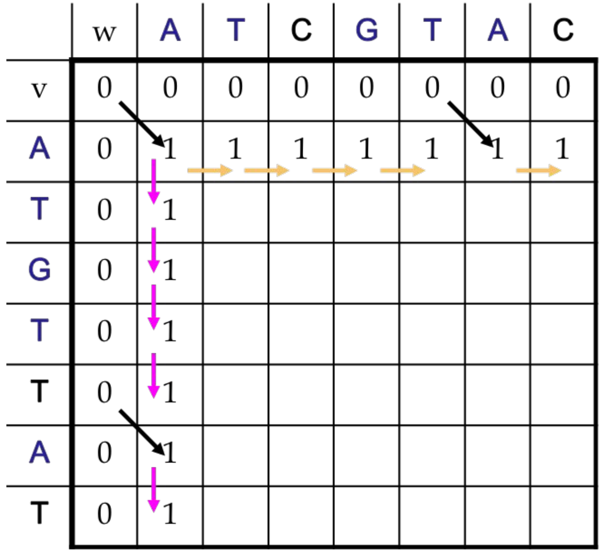

The values in this matrix represent the intersections of the graph we used before. Thus, it is (N+1) x (M+1)



- Note intersections/vertices are cells/entries of this matrix

Step 2

Evaluate recursion for next row and/or next column

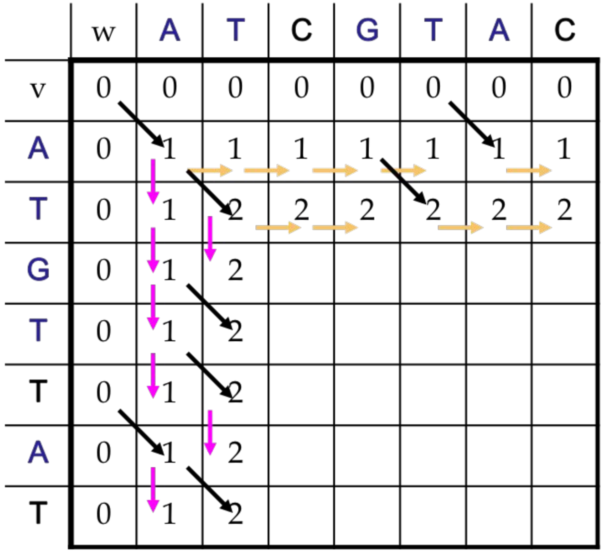


	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1						
G	0	1						
T	0	1						
T	0	1						
A	0	1						
T	0	1						

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_j \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Step 3

Continue recursion for next row and/or next column

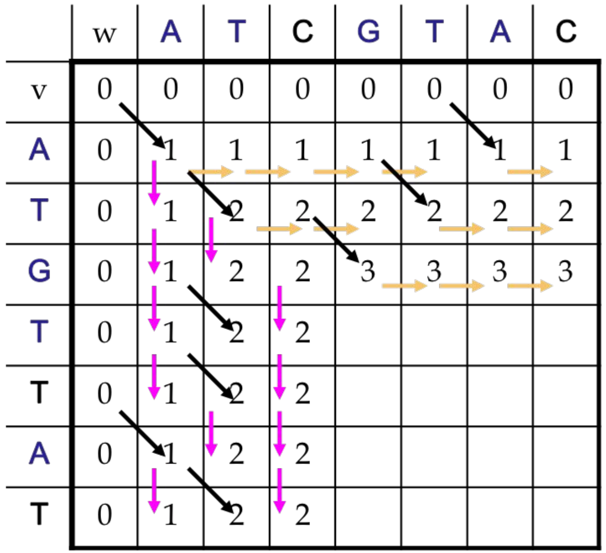


	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2					
T	0	1	2					
T	0	1	2					
A	0	1	2					
T	0	1	2					

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_j \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Step 4

Then one more row and/or column

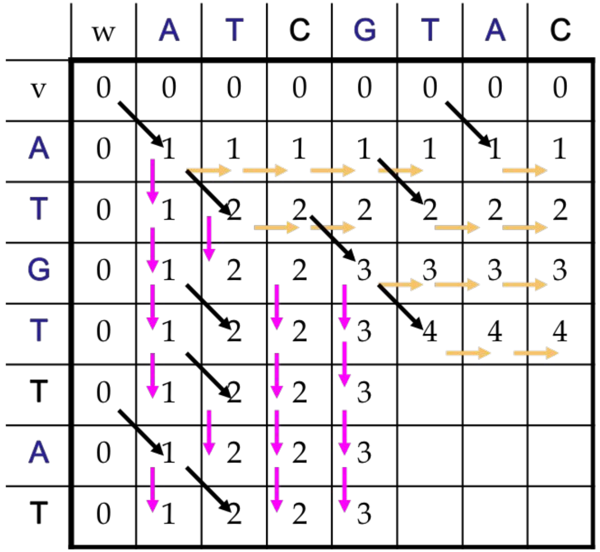


	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2				
T	0	1	2	2				
A	0	1	2	2				
T	0	1	2	2				

$$s_{i,j} = \max\begin{cases} s_{i-1,j-1}+1 & \text{if } v_i = w_j \ \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Step 5

And so on...

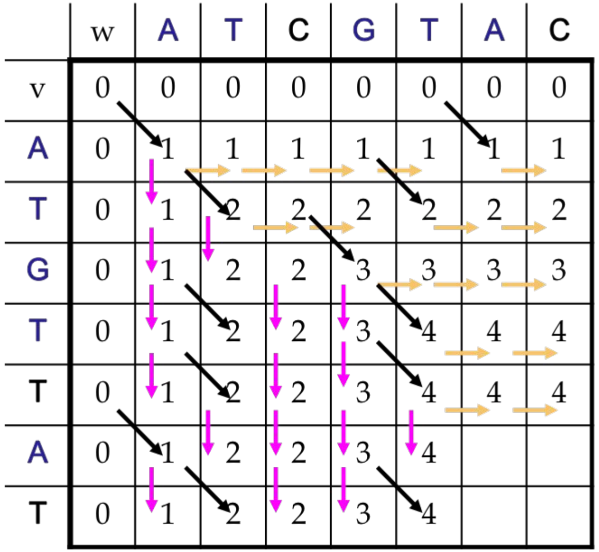


	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3			
A	0	1	2	2	3			
T	0	1	2	2	3			

$$s_{i,j} = \max\begin{cases} s_{i-1,j-1}+1 & \text{if } v_i = w_j \quad \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Step 6

And so on...

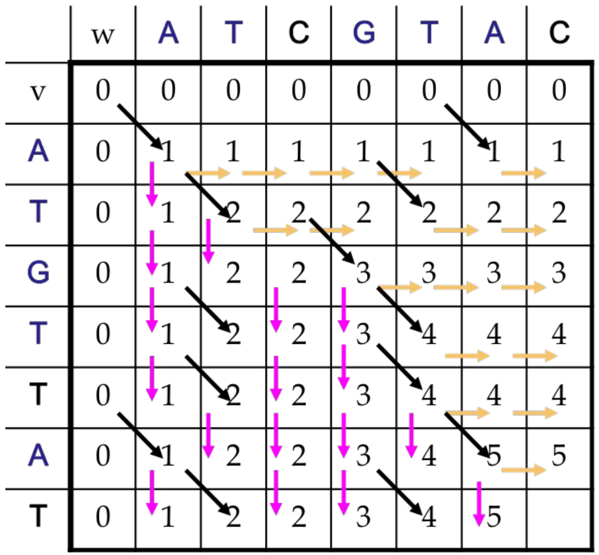


	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3	4	4	4
A	0	1	2	2	3	4		
T	0	1	2	2	3	4		

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_j \quad \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Step 7

Getting closer

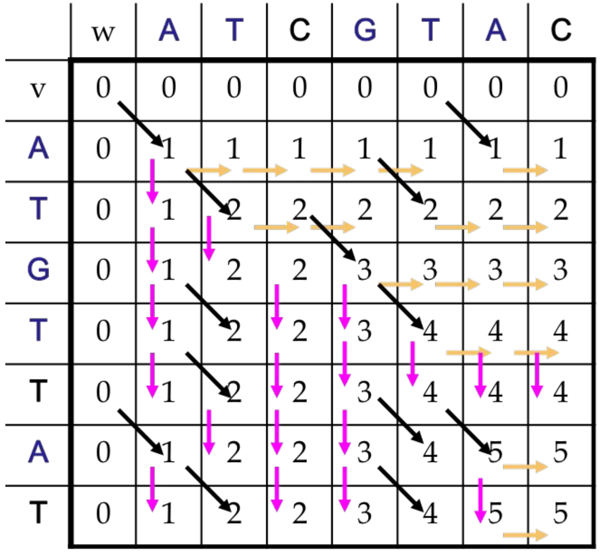


	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3	4	4	4
A	0	1	2	2	3	4	5	5
T	0	1	2	2	3	4	5	

$$s_{i,j} = \max\begin{cases} s_{i-1,j-1}+1 & \text{if } v_i = w_j \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Step 8

Until we reach the last row and column

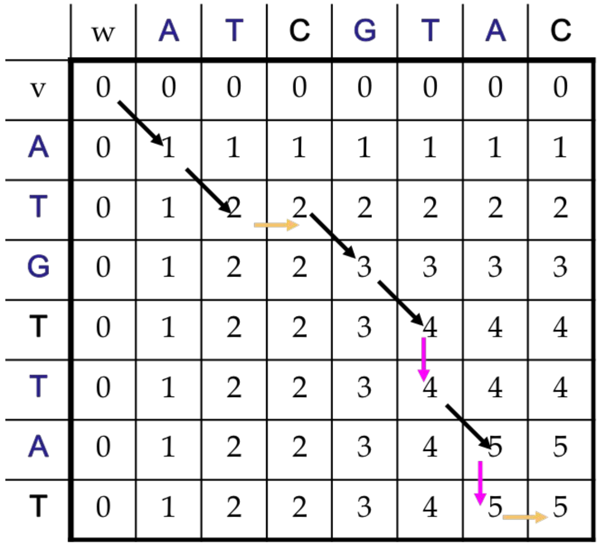


	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3	4	4	4
A	0	1	2	2	3	4	5	5
T	0	1	2	2	3	4	5	5

$$s_{i,j} = \max\begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_j \quad \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Finally

We reach the end, which corresponds to an LCS of length 5



	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3	4	4	4
A	0	1	2	2	3	4	5	5
T	0	1	2	2	3	4	5	5

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & if\ v_i = w_j \ \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

```
w  =  ATCGT_A_C
v  =  AT_GTTAT_
```

len(LCS) = 5

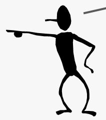
Our answer includes both an optimal score, and a path back to both the LCS and an alignment

LCS Code

Let's see how well the code matches the approach we sketched out…

```python
from numpy import *

def findLCS(v, w):
    score = zeros((len(v)+1,len(w)+1), dtype="int32")
    backt = zeros((len(v)+1,len(w)+1), dtype="int32")
    for i in range(1,len(v)+1):
        for j in range(1,len(w)+1):
            # find best score at each vertex
            if (v[i-1] == w[j-1]):  # test for a match ("diagonal street")
                score[i,j], backt[i,j] = max((score[i-1,j-1]+1,3), (score[i-1,j],1), (score[i,j-1],2))
            else:
                score[i,j], backt[i,j] = max((score[i-1,j],1), (score[i,j-1],2))
    return score, backt

v = "ATGTTAT"
w = "ATCGTAC"
s, b = findLCS(v,w)
for i in range(len(s)):
    print("%10s %-20s    %12s %-20s" % ('' if i else 'score =', str(s[i]), '' if i else 'backtrack =', str(b[i]))
```

The tableau is made one larger than you'd expect because it simplifies the edge cases.

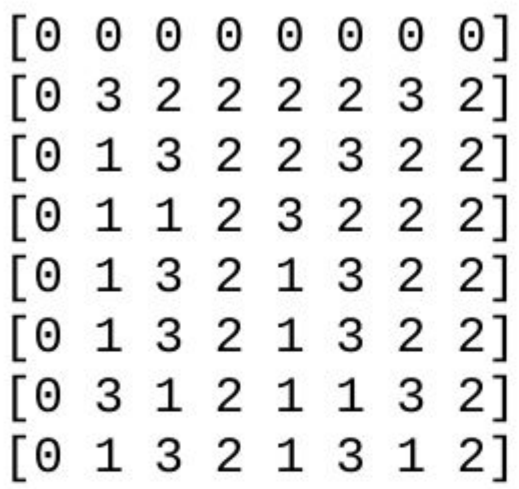


Here is where the "edges" of the graph are considered. The graph is implicit.

```
score = [0 0 0 0 0 0 0 0]        backtrack = [0 0 0 0 0 0 0 0]
        [0 1 1 1 1 1 1 1]                    [0 3 2 2 2 2 3 2]
        [0 1 2 2 2 2 2 2]                    [0 1 3 2 2 3 2 2]
        [0 1 2 2 3 3 3 3]                    [0 1 1 2 3 2 2 2]
        [0 1 2 2 3 4 4 4]                    [0 1 3 2 1 3 2 2]
        [0 1 2 2 3 4 4 4]                    [0 1 3 2 1 3 2 2]
        [0 1 2 2 3 4 5 5]                    [0 3 1 2 1 1 3 2]
        [0 1 2 2 3 4 5 5]                    [0 1 3 2 1 3 1 2]
```

- The same score matrix that we found by hand
- *"backtrack"* keeps track of the "arrow" used, 1 is ↓, 2 is →, 3 is ↘

Backtracking



```
[0 0 0 0 0 0 0 0]
[0 3 2 2 2 2 3 2]
[0 1 3 2 2 3 2 2]
[0 1 1 2 3 2 2 2]
[0 1 3 2 1 3 2 2]
[0 1 3 2 1 3 2 2]
[0 3 1 2 1 1 3 2]
[0 1 3 2 1 3 1 2]
```

Our score table kept track of the longest common subsequence so far. How do we figure out what the subsequence is?

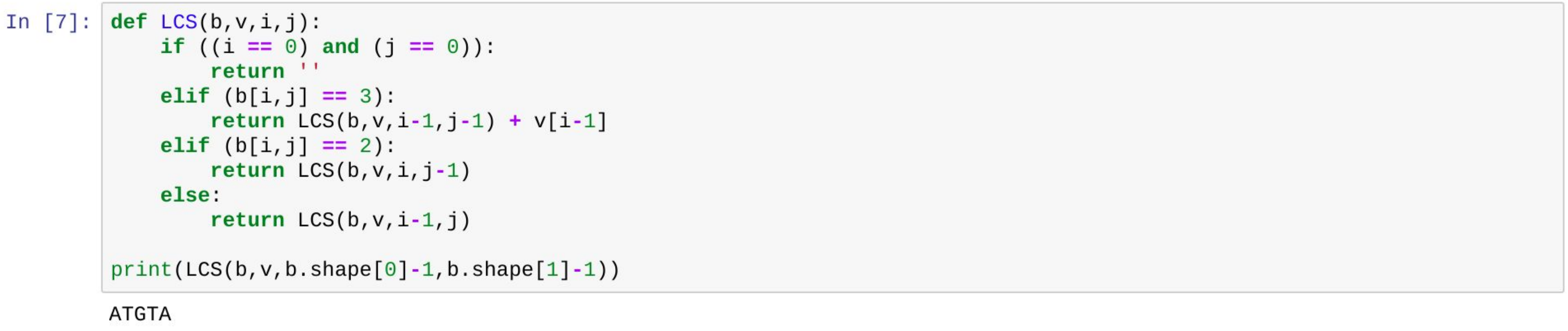
The second "arrow" table kept track of the decisions we made… and we'll use it to backtrack to our answer.

In our example we used arrows {↓, →, ↘}, which were represented in our matrix as {1,2,3} respectively. This numbering is *arbitrary*, *except that it does break ties in our implementation* (matches > w deletions > w insertions).

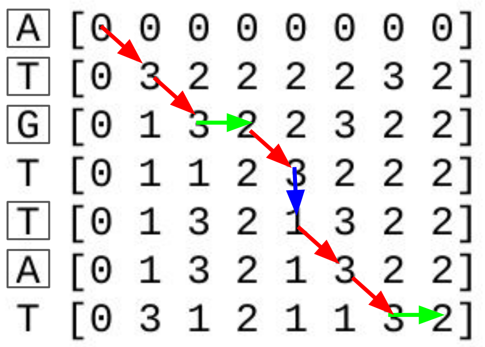
Now we need code that finds a path from the end of our strings to the beginning using our arrow matrix

Code to extract an answer

A simple recursive LCS( ) routine to return along the path of arrows that led to our best score.



```
In [7]: def LCS(b,v,i,j):
            if ((i == 0) and (j == 0)):
                return ''
            elif (b[i,j] == 3):
                return LCS(b,v,i-1,j-1) + v[i-1]
            elif (b[i,j] == 2):
                return LCS(b,v,i,j-1)
            else:
                return LCS(b,v,i-1,j)

        print(LCS(b,v,b.shape[0]-1,b.shape[1]-1))

        ATGTA
```

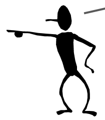


```
A [0 0 0 0 0 0 0 0]
T [0 3 2 2 2 2 3 2]
G [0 1 3 2 2 3 2 2]
T [0 1 1 2 3 2 2 2]
T [0 1 3 2 1 3 2 2]
A [0 1 3 2 1 3 2 2]
T [0 3 1 2 1 1 3 2]
```

But that’s not an alignment

- Technically correct, ATGTA is the LCS, But an alignment accounts for both those letter used in the LCS as well as those skipped

```
w = ATcGT_A_c
v = AT_GTtAt_
```

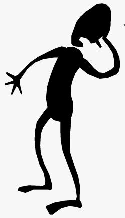


This is a proper alignment of the sequences w and v. It accounts for all letters in both sequences

- Notice that LCS( ) needed only one of *v* or *w* since both contain the LCS
- How might we get an alignment instead of just the LCS

An alignment of v and w

```
In [10]: def Alignment(b,v,w,i,j):
             if ((i == 0) and (j == 0)):
                 return ['','']
             if (b[i,j] == 3):
                 result = Alignment(b,v,w,i-1,j-1)
                 result[0] += v[i-1]
                 result[1] += w[j-1]
                 return result
             if (b[i,j] == 2):
                 result = Alignment(b,v,w,i,j-1)
                 result[0] += "_"
                 result[1] += w[j-1]
                 return result
             if (b[i,j] == 1):
                 result = Alignment(b,v,w,i-1,j)
                 result[0] += v[i-1]
                 result[1] += "_"
                 return result

         align = Alignment(b,v,w,b.shape[0]-1,b.shape[1]-1)
         print("v =", align[0])
         print("w =", align[1])

         v = AT_GTTAT_
         w = ATCG_TA_C
```



Once again, it is a recursive function. That handles one arrow on each call.

From an LCS to an Alignment

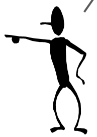
Longest Common Subsequence (LCS) is a special case of alignment

1. Construct a graph
2. Define a recurrence relation
3. Solve it for all paths from (0,0) to (n,m)
4. Used a dynamic program where each step relies only on solutions already computed and saved in our tableau

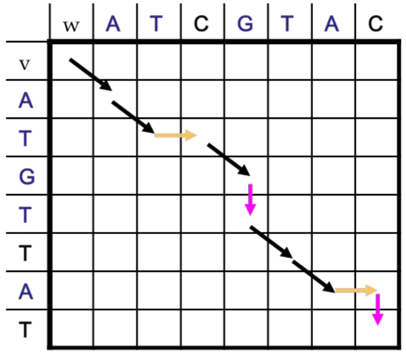
$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \mathit{if}\ v_i = w_i \quad \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

How about alternate recurrence relations?

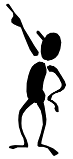
$$S_{ij} = max \begin{cases} S_{i-1,j-1} \boxed{+\,1} & \text{if } v_i = w_j \\ S_{i-1,j-1} & \text{if } v_i \neq w_j \\ S_{i-1,j} - \boxed{2} \\ S_{i,j-1} - \boxed{2} \end{cases}$$



This term allows us to take a few "diagonals", even if the row and column letters don't match.



What if we want to change these reward/penalty values? Perhaps we'd prefer an INDEL over a mismatch



A more "*general purpose*" alignment graph

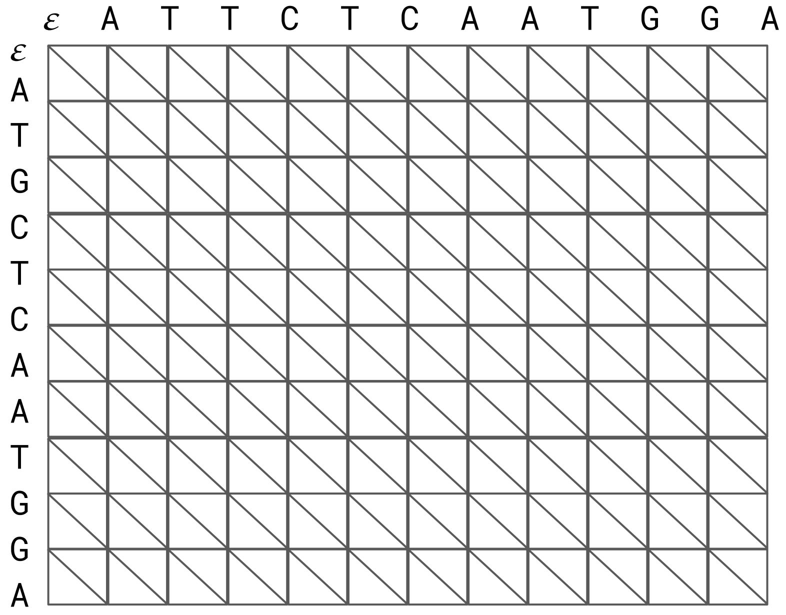
Now consider a more uniform "Manhattan"

There are four ways to reach an intersection

From the north,

From the east,

From a diagonal at every intersection
    with different scores for a
    "match" and a "mismatch"



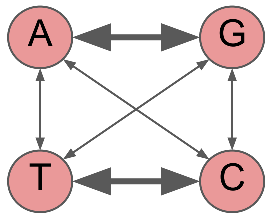
Alignment using a Scoring Matrix

- Rather *edit distance* one can use a table with costs for every symbol aligned to any other
- Scoring matrices allow alignments to consider biological constraints
- Alignments can be thought of as two sequences that differ due to mutations.
- Some types of mutations are more common, or have little or no effect on function, therefore some mismatch penalties, $\delta(v_i, w_j)$, should be less harsh than others.

Example: ***DNA transitions and transversions***

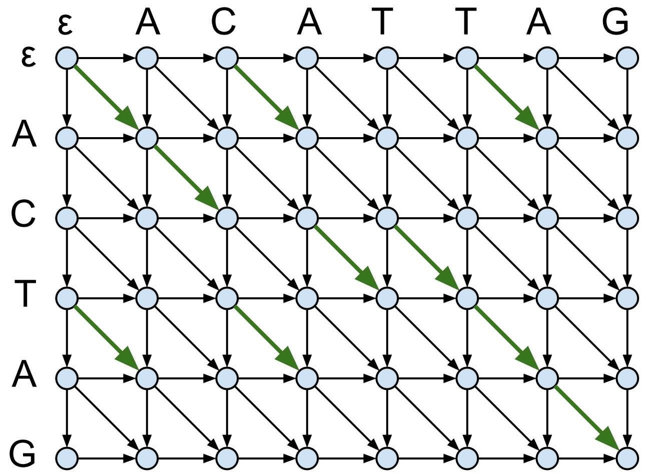
- Like LCS, we want to maximize sequence matches, so each should have a positive score (diagonal of scoring matrix)
- Unlike LCS, we need to allow for occasional mismatches, as well as INDELs.
- The 4 DNA nucleotides come in two types, *purines* (A and G), which have two-rings and *pyrimidines*, (C and T) which have only one.
- Mutations within types are far more common than mutations between types, despite there being twice as many. This higher mutation rate can be encoded as a smaller substitution penalty.
- Insertions and deletions are even less common that any substitution, thus they have even higher penalties.

δ	A	C	G	T	_
A	1	-2	-1	-2	-3
C	-2	1	-2	-1	-3
G	-1	-2	1	-2	-3
T	-2	-1	-2	1	-3
_	-3	-3	-3	-3	



Impact on Alignment

Graph includes all diagonal edges, but many with negative weights

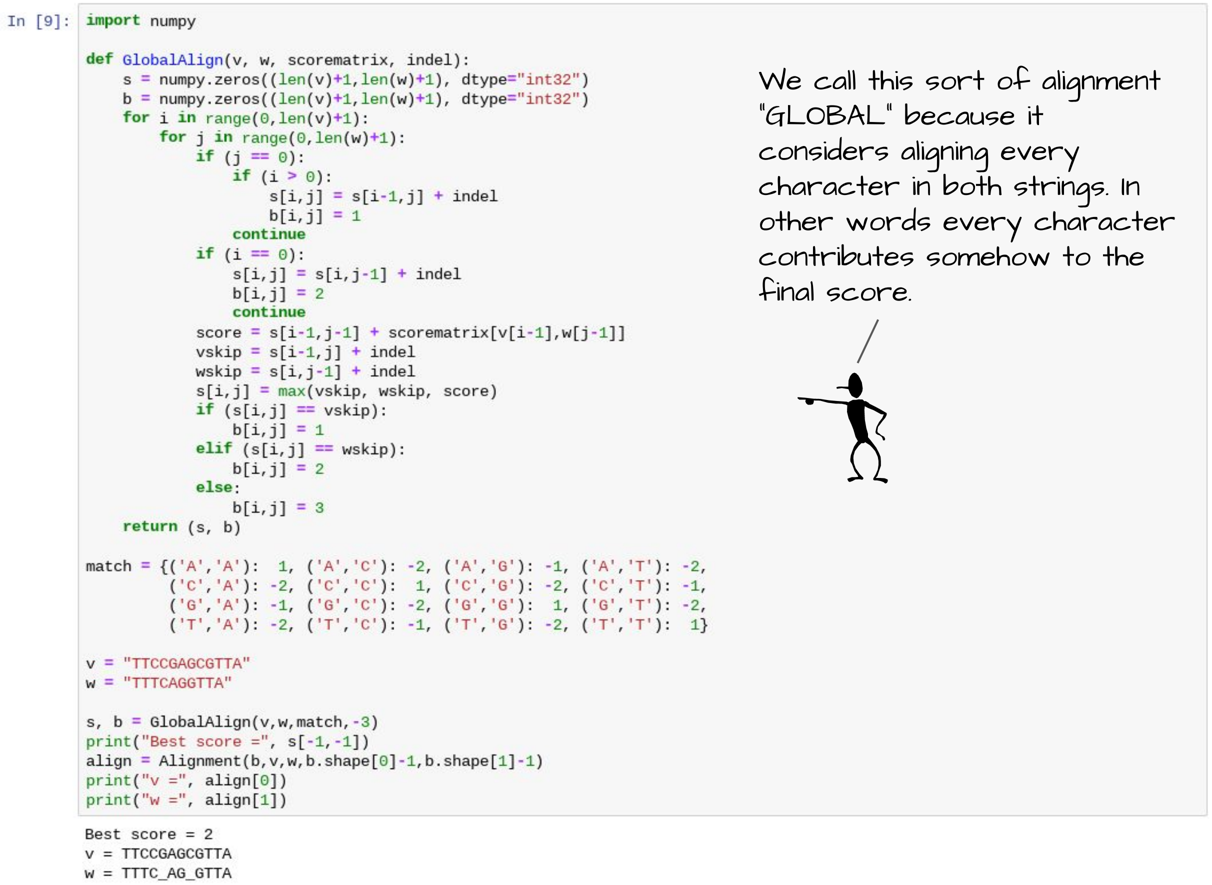


$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, \_) \\ s_{i,j-1} + \delta(\_, w_j) \end{cases}$$

Generalized recurrence relation

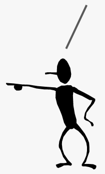
Needleman−Wunsch Alignment Algorithm

Global Alignment with a scoring matrix



```
In [9]: import numpy

        def GlobalAlign(v, w, scorematrix, indel):
            s = numpy.zeros((len(v)+1,len(w)+1), dtype="int32")
            b = numpy.zeros((len(v)+1,len(w)+1), dtype="int32")
            for i in range(0,len(v)+1):
                for j in range(0,len(w)+1):
                    if (j == 0):
                        if (i > 0):
                            s[i,j] = s[i-1,j] + indel
                            b[i,j] = 1
                        continue
                    if (i == 0):
                        s[i,j] = s[i,j-1] + indel
                        b[i,j] = 2
                        continue
                    score = s[i-1,j-1] + scorematrix[v[i-1],w[j-1]]
                    vskip = s[i-1,j] + indel
                    wskip = s[i,j-1] + indel
                    s[i,j] = max(vskip, wskip, score)
                    if (s[i,j] == vskip):
                        b[i,j] = 1
                    elif (s[i,j] == wskip):
                        b[i,j] = 2
                    else:
                        b[i,j] = 3
            return (s, b)

        match = {('A','A'):  1, ('A','C'): -2, ('A','G'): -1, ('A','T'): -2,
                 ('C','A'): -2, ('C','C'):  1, ('C','G'): -2, ('C','T'): -1,
                 ('G','A'): -1, ('G','C'): -2, ('G','G'):  1, ('G','T'): -2,
                 ('T','A'): -2, ('T','C'): -1, ('T','G'): -2, ('T','T'):  1}

        v = "TTCCGAGCGTTA"
        w = "TTTCAGGTTA"

        s, b = GlobalAlign(v,w,match,-3)
        print("Best score =", s[-1,-1])
        align = Alignment(b,v,w,b.shape[0]-1,b.shape[1]-1)
        print("v =", align[0])
        print("w =", align[1])

        Best score = 2
        v = TTCCGAGCGTTA
        w = TTTC_AG_GTTA
```

We call this sort of alignment "GLOBAL" because it considers aligning every character in both strings. In other words every character contributes somehow to the final score.



Next Time

- Global vs. Local alignments
- Affine gap penalties
- Aligning more than
  Two sequences