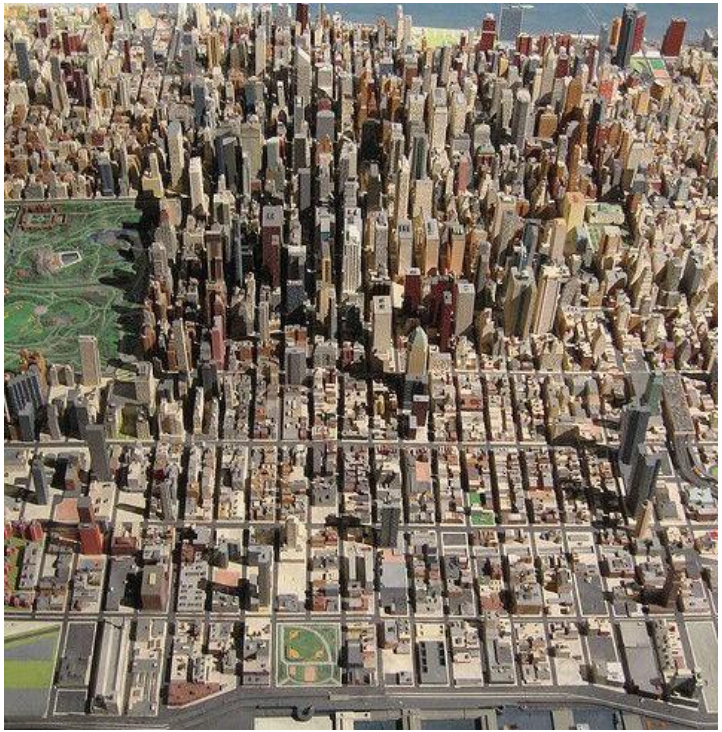
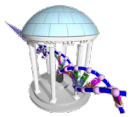
Comp 555 - BioAlgorithms - Spring 2018



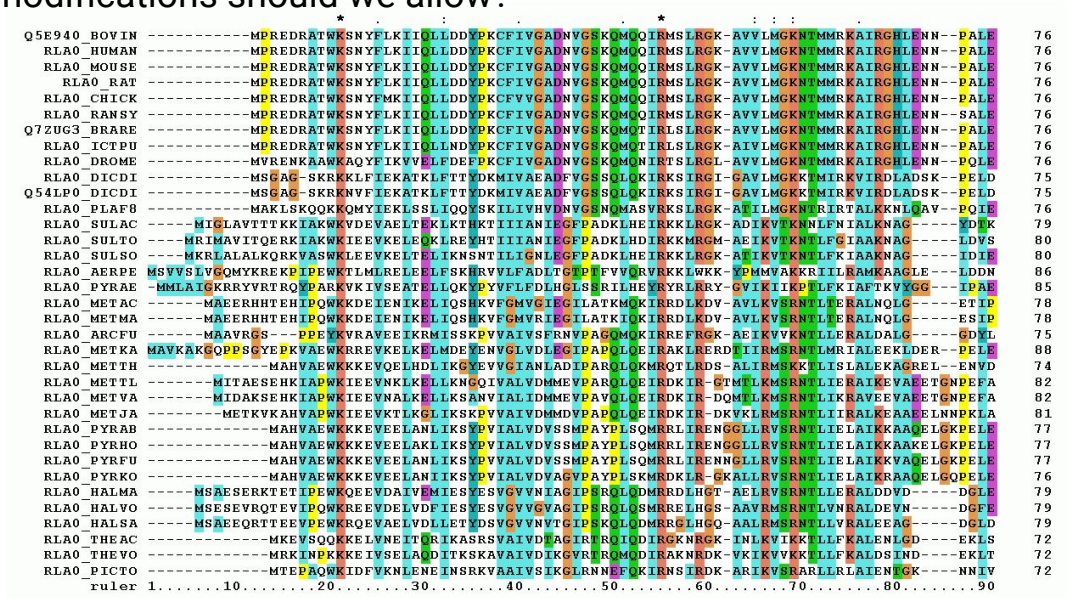
- Relating sequence alignment to our Manhattan Tour Problem
- Problem Set #3
- Problem Set #4 coming soon...

Sequence Alignment

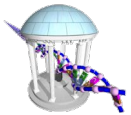
A Biological Dynamic Programming Problem



- How to measure the similarity between a pair of nucleotide or amino acid sequences
- When Motif-Searching we used Hamming distance as a measure of sequence similarity
- Is Hamming distance the best measure?
- How can we distinguish matches that occur by chance from slightly modified patterns?
- What sorts of modifications should we allow?



Best Sequence Matches

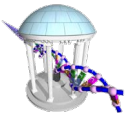


- Depends on how you define *Best*
- Consider the two DNA sequences *v* and *w* :

```
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:

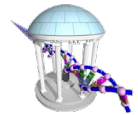
```
v: _TAGACAAT
w: AGAGACA_T
   110000010 = 3
```

Another one:

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

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

Edit Distance

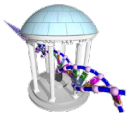


- Vladimir Levenshtein introduced the notion of an “edit distance” between two strings as the minimum number of elementary operations (insertions, deletions, and substitutions) to transform one string into the other in 1965.
- $d_L(v,w)$ = Minimum number of elementary operations to transform $v \rightarrow w$
- Computing Hamming distance is a trivial task
- Computing edit distance is less trivial

Vladimir Levenshtein

1935 - 2017

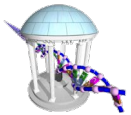
Edit Distance: Example



```
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)

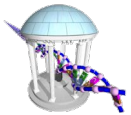


```
TGCATAT → ATCCGAT in 4 steps


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 $v \rightarrow w$, one can generate $w \rightarrow v$, 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

Longest Common Subsequence



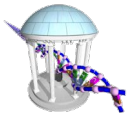
- A special case of edit distance where no *substitutions* are allowed
- A subsequence need not be contiguous, but the symbol order must be preserved
  Ex. If v = ATTGCTA then AGCA and TTTA are subsequences of v, but TGTT and ACGA are not
- All substrings of *v* are subsequences, but not vice versa
- The edit distance, d_L, is related to the length of the LCS, *s*, by the following relationship:

$$d_L(u,w) = len(v) + len(w) - 2s(u,w)$$

Example:

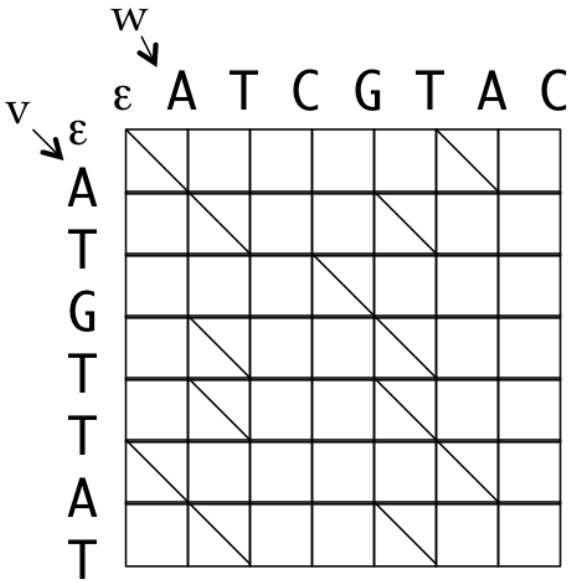
```
ANUNCLEIKE
UNCBEATDUKE

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

LCS as a Dynamic Program

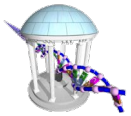


There are similarites between the LCS and MTP

- 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 appear in our graph



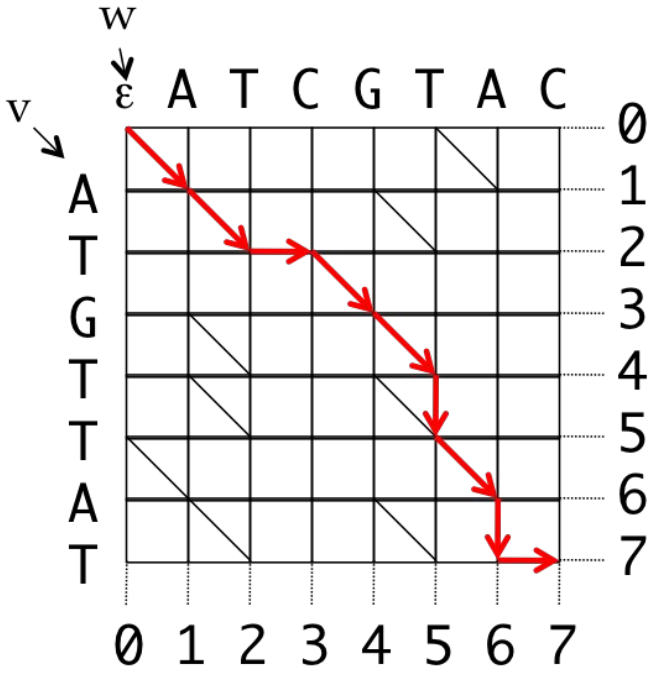
Various Alignments



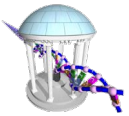
- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

```
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)



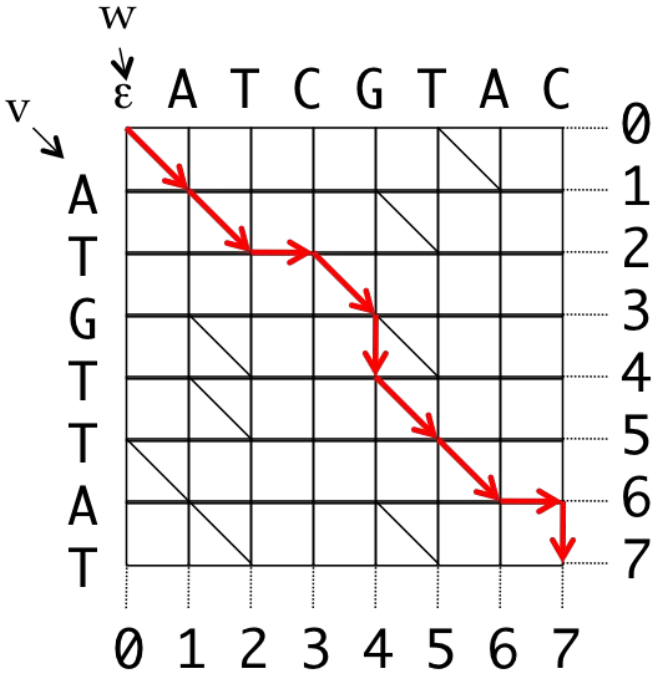
Alternate Alignment



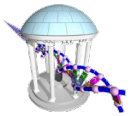
- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

```
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)



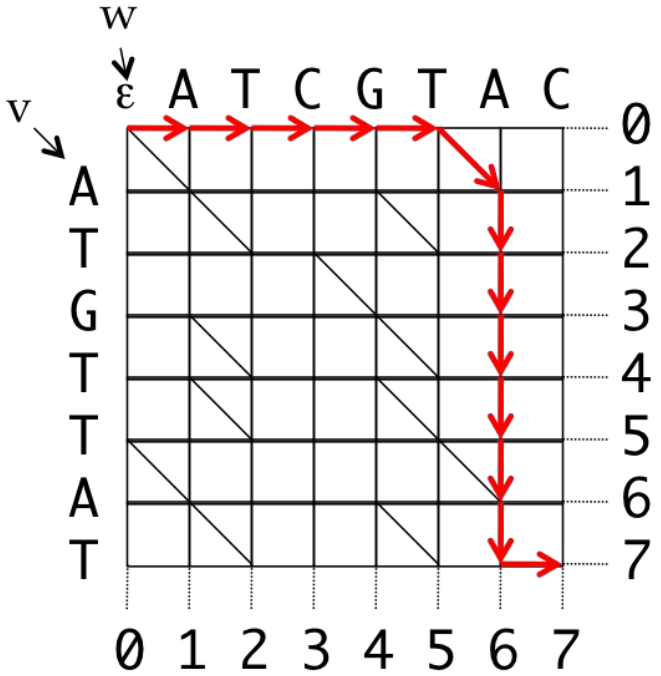
Even Bad Alignments



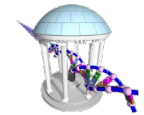
- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

	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	7	

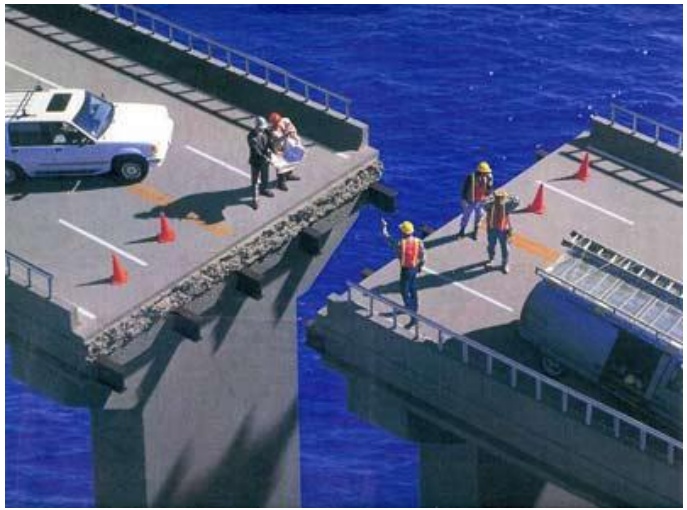
- Path:
  (0,0), (0,1), (0,2), (0,3), (0,4), (0,5), (1,6),
  (2,6), (3,6), (4,6), (5,6), (6,6), (7,6), (7,7)



What makes a good alignment?

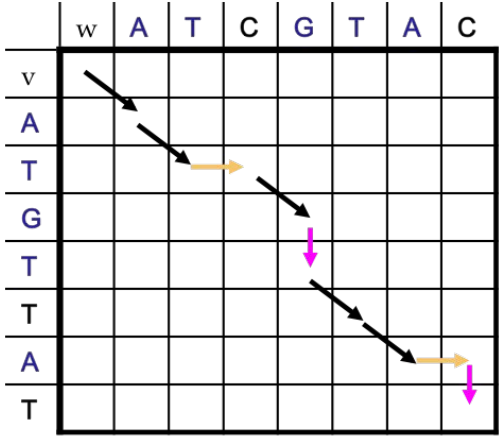


- Using as many diagonal segments (matches) as possible. Why?
- The end of a good alignment from (j...k) begins with a good alignment from (i..j)
- Same as Manhattan Tourist problem, where the ***sites*** are only on the diagonal streets!
- Set diagonal street weights = 1, and horizontal and vertical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

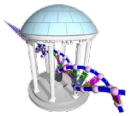


LCS: Dynamic Program

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



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							

- 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						

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

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					

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

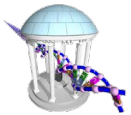
Step 4

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				

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

Step 5

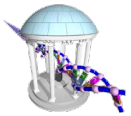


And so on...

	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			

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

Step 6

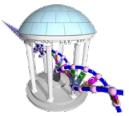


And so on...

	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		

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

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	

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

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

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

Finally

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

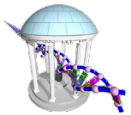
	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3	4	4	4
A	0	1	2	2	3	4	5	5
T	0	1	2	2	3	4	5	5

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

W = ATCGT-A-C
V = AT-GTTAT-

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

LCS Code



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

In [1]:
```
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]):
                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])))
```

```
     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 arrows that we used

Backtracking

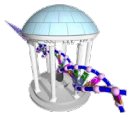
Our score table kept track of the longest common subsequence so far. How do we figure out what the subsequence is?

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

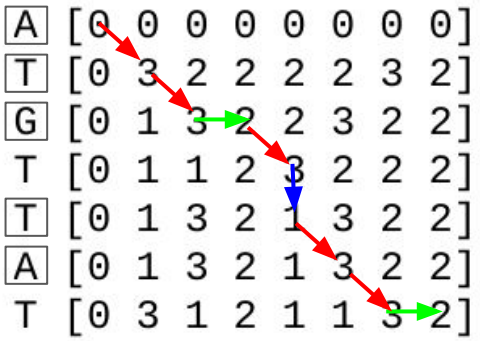
Code to extract an answer



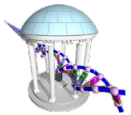
A simple recursive routine to return along the path of arrows that led to our best score.

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
```



But that’s not an alignment

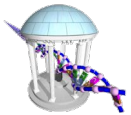


- Technically correct, ATGTA is the LCS

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

- Notice that we only need one of *v* or *w* since both contain the LCS
- But we would like to get more than just the LCS
- For example, the corresponding alignment.

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
```

Next Time

- Convert LCS to a general purpose sequence aligner
- Scoring matrices
- Global vs. Local alignments
- Affine gap penalites