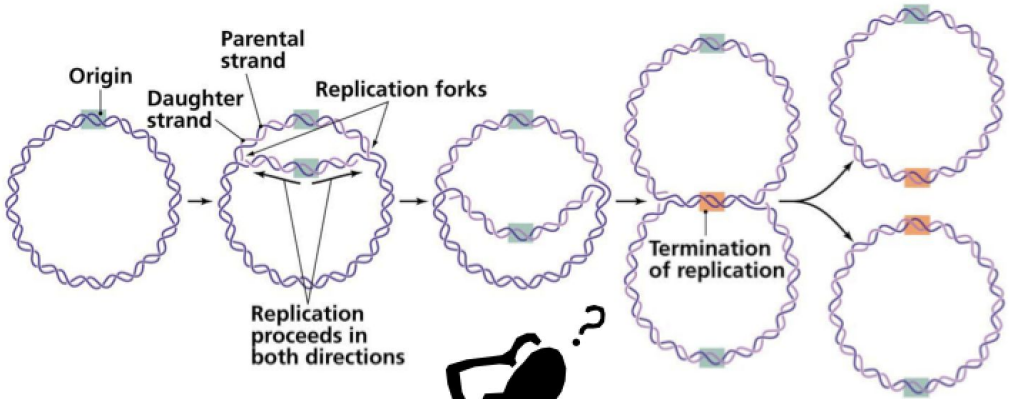
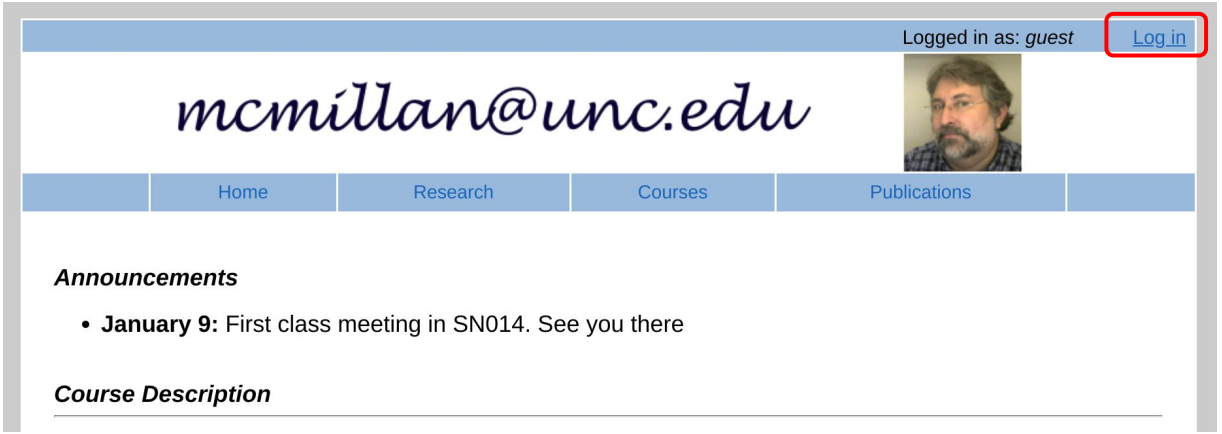
Comp 555 - BioAlgorithms - Spring 2020



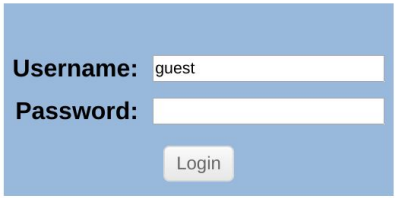
Finding Patterns in DNA

Login to Course Website

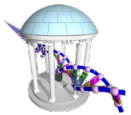
1) Login to your Comp555 account



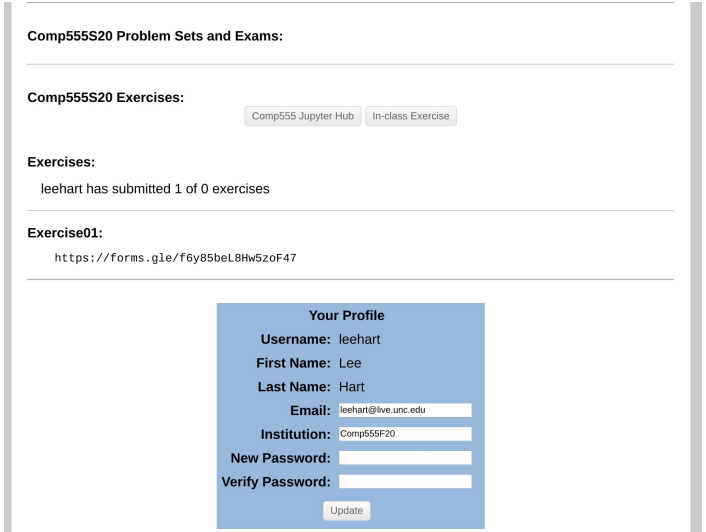
2) Your username is your UNC ONYEN and password is your PID



Next Steps

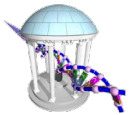


3) Once you are logged in, press "Course" and then a “Setup” button should appear. Press "Setup" and you should see something like:

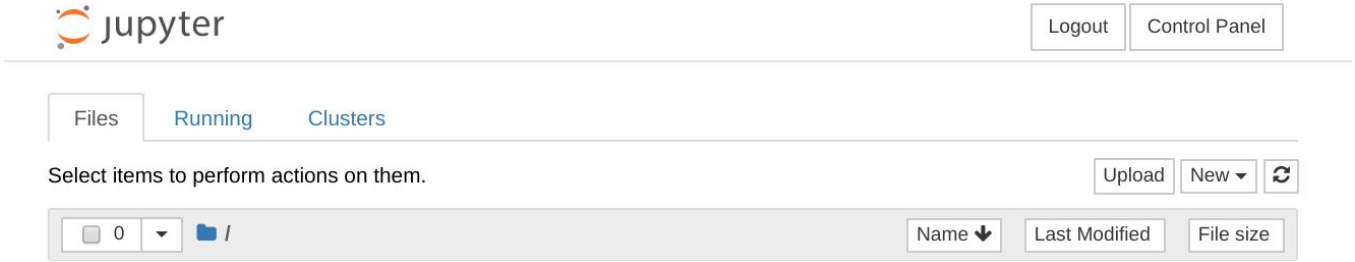


4) Now press the [Comp555 Jupyter Hub] button.
(BTW, you can also change your password here if you want).

Your Own Notebook

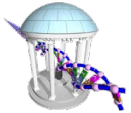


5) You should eventually get to a page like:



6) At this point you should download the Lecture02 "notebook" and "data" from the course website and upload it to your notebook. Run each cell. Go ahead and play for a couple of minutes

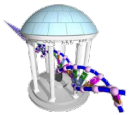
For those without a login...



- Go back to the login page, and click "registered"

- Then enter the following information:
- Once registered a screen will indicate you've been verified; then click "Course" and "Setup" as before.

Is there a pattern for predicting *OriC*?

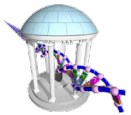


Recall last time that we found within the known *OriC* region of *Vibrio Cholerae* repeated 9-mers at a frequency far higher than we would expect by chance:

```
atcaatgatcaacgtaagcttctaagcATGATCAAGgtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagATGATCAAGagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctCTTGATCATcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctCTTGATCATgtt
tccttaaccctctatttttacggaagaATGATCAAGctgctgctCTTGATCATcgtttc
```

Does this insight generalize?

Let's look at another Bacteria



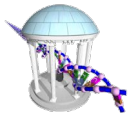
Here's the OriC region of another bacteria *Thermotoga petrophila*

```
aactctatacctccttttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaac
taaaatggtaggtttggtggtaggttttgtgtacattttgtagtatctgatttttaattacat
accgtatattgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaaccta
ccaccaaactctgtattgaccattttaggacaacttcagggtggtaggttctgaagctctca
tcaatagactattttagtctttacaaacaatattaccgttcagattcaagattctacaacgct
gttttaatgggcgttgcagaaaacttaccacctaaaatccagtatccaagccgatttcagaga
aacctaccacttacctaccacttacctaccacccgggtggtaagttgcagacattattaaaaa
cctcatcagaagcttgttcaaaaatttcaatactcgaaacctaccacctgcgtcccctattat
ttactactactaataatagcagtataattgatctgaaaagaggtggtaaaaaa
```

The most frequent 9-mers are: [(ACCTACCAC,5), (GGTAGGTTT,3), (CCTACCACC,3), (AACCTACCA,3), (TGGTAGGTT,3), (AAACCTACC,3)]. There is no occurence of the patterns ATGATCAAG and CTTGATCAT.

Thus, it appears that different genomes have different DnaA box patterns. Let's go back to the drawing board. By the way, the DnaA box pattern of Thermotoga petrophila is: CCTACCACC , GGTGGTAGG

Another Strategy



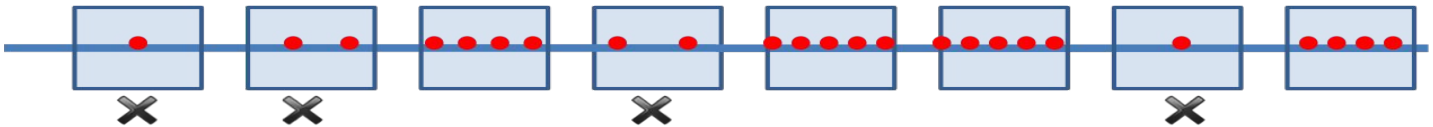
Our previous approach was to find frequent words in oriC region as candidate DnaA boxes, as if

$$\textit{replication origin} \rightarrow \textit{frequent words}$$

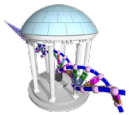
Suppose that we reverse our approach, we use clumps of frequent words to infer the replication origin, testing if

$$\textit{nearby frequent words} \rightarrow \textit{replication origin}$$

We can apply this approach to find candidate DnaA boxes.



What is a Clump?



We have an initition of what is meant by a clump of k-mers, but in order to define an algorithm we will need more precise definitons.

Formal Definition:

A k-mer forms an (L, t)-clump inside Genome if there is a short (length L) interval of Genome in which it appears many (at least t) times.

Clump Finding Problem:

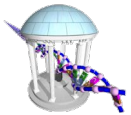
Find patterns that form clumps within a string.

Input: A string and integers k (length of a pattern), L (window length),

and t (number of patterns in a clump).

Output: All k-mers forming (L,t) clumps in the string

Find Locations of k-mer

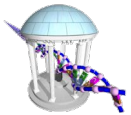


In [10]:
```
def kmerPositions(k, sequence):
    """ returns the position of all k-mers in sequence as a dictionary"""
    kmerPosition = {}
    for i in range(1,len(sequence)-k+1):
        kmer = sequence[i:i+k]
        kmerPosition[kmer] = kmerPosition.get(kmer,[])+[i]
    # combine kmers with their reverse complements
    pairPosition = {}
    for kmer, posList in kmerPosition.items():
        krev = ''.join([{'A':'T','C':'G','G':'C','T':'A'}[base] for base in reversed(kmer)])   # one-liner
        if (kmer < krev):
            pairPosition[kmer] = sorted(posList + kmerPosition.get(krev, []))
        elif (krev < kmer):
            pairPosition[krev] = sorted(kmerPosition.get(krev, []) + posList)
        else:
            pairPosition[kmer] = posList
    return pairPosition
```

Modified k-mer counting function from last time. It now saves a list of *positions*, rather than counts. Consider: `kmerPosition[kmer] = kmerPosition.get(kmer,[])+[i]`

It also merges k-mers with their reverse complements. Position lists are sorted.

Whoa! Let's take a look at that one-liner



A "list comprehension" that forms the reverse-complement of a given sequence

```
krev = ''.join([{'A':'T','C':'G','G':'C','T':'A'}[base] for base in reversed(kmer)])
```

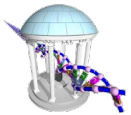
A recipe for building a data structure:

1. Step through each base of a string in reverse … `for base in reversed(kmer)`
2. Convert each base to it's Watson complement … `{'A':'T','C':'G','G':'C','T':'A'}[base]`
3. Glue the resulting character list into a string … `''.join([...])`

What do you think?

Would you rather write a function to hide all of this ugliness?
Is it ugly or concise? There are other ways. We'll discuss them later on.

Next, find Clumps



In [12]:
```
def findClumps(string, k, L, t):
    """ Find clumps of repeated k-mers in string. A clump occurs when t or more k-mers appear
        within a window of size L. A list of (k-mer, position, count) tuples is returned """
    clumps = []
    kmerData = kmerPositions(k, string)
    for kmer, posList in kmerData.items():
        for start in range(len(posList)-t):
            end = start + t - 1
            while ((posList[end] - posList[start]) <= L-k):
                end += 1
                if (end >= len(posList)):
                    break
            if (end - start >= t):
                clumps.append((kmer, posList[start], end - start))
    return clumps
```

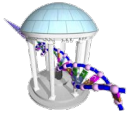
In [13]:
```
clumpList = findClumps(genome, 9, 500, 6)
print(len(clumpList))
print([clumpList[i] for i in range(min(20,len(clumpList)))])
```

```
172
[('ATCAAAAAT', 566252, 6), ('AACCAGAAC', 922082, 13), ('AACCAGAAC', 922088, 12), ('AACCAGAAC', 922094, 11), ('AACCAG
AAC', 922100, 10), ('AACCAGAAC', 922106, 9), ('AACCAGAAC', 922112, 8), ('AACCAGAAC', 922118, 7), ('AACCAGAAC', 92212
4, 6), ('GCAATAACA', 704434, 6), ('ATGTTATTG', 704433, 6), ('AATAACATC', 704432, 6), ('CTCTCTCTC', 798535, 6), ('AAA
TCAAAA', 566247, 7), ('AAATCAAAA', 566254, 6), ('AACAGCAAC', 1073067, 21), ('AACAGCAAC', 1073073, 20), ('AACAGCAAC',
1073079, 19), ('AACAGCAAC', 1073085, 18), ('AACAGCAAC', 1073091, 17)]
```

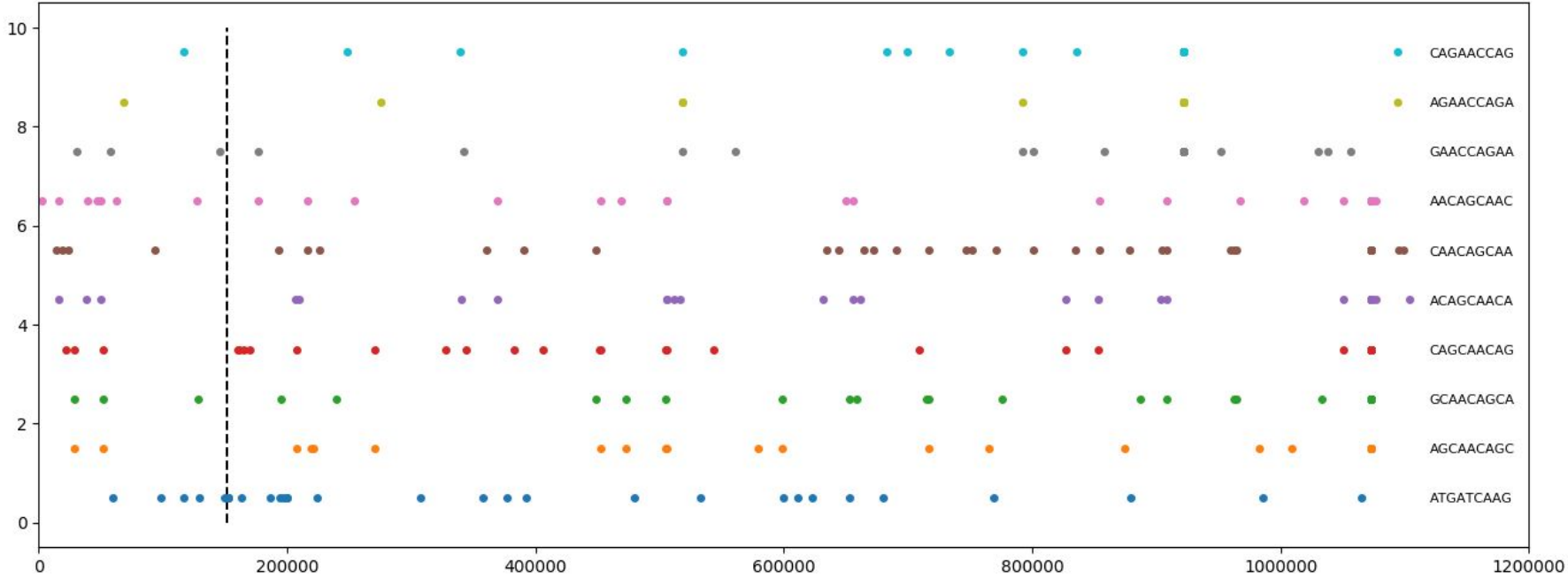
Wow!

There are 172 k-mers that appear in clumps of 6 or more within any 500 base window. That's a lot more than expected. I guess that means that genomes are not that random at all.

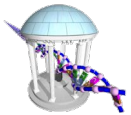
Let’s look at the Top10



These are 9-mers that appear in many (500, 6) clumps.



So Far...

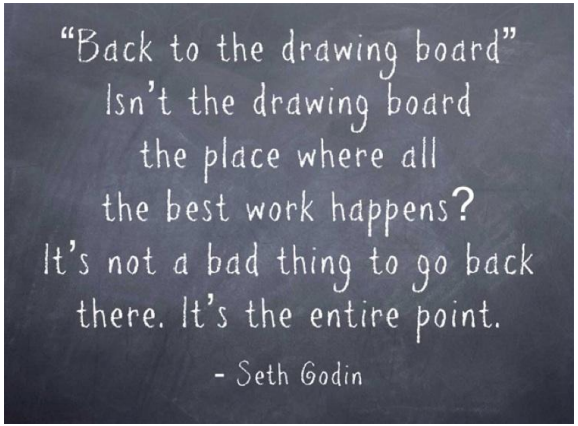


Things have not gone as planned

- **We still don't have a working algorithm for finding OriC**
- **We tried searching for patterns in a known OriC region, but the patterns we found did not generalize to other genomes.**
- **We tried to find clumps of repeated k-mers, but that led to too many hypotheses to follow up on**

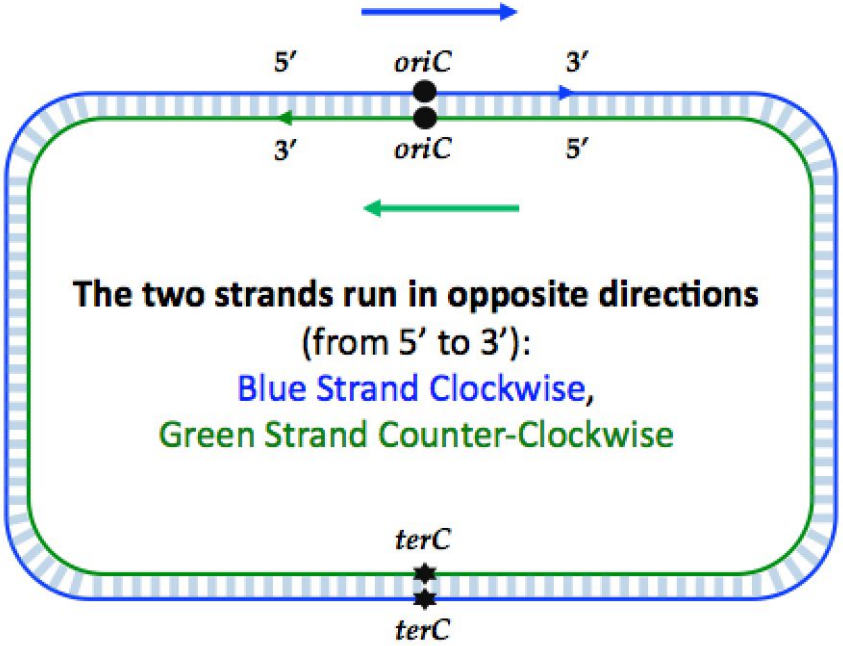
But we won't give up.

Let's see if there are any more biological insights that we might leverage

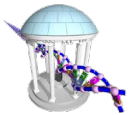


A Closer Look

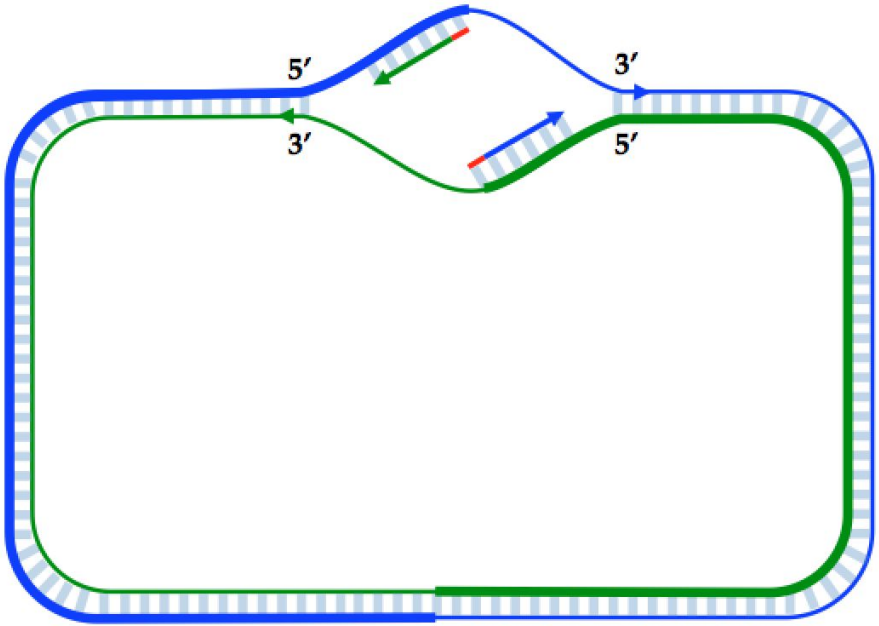
Replication begins at the OriC, but progresses at different rates in the 5’ and 3’ directions.



DNA Polymerases do the copying



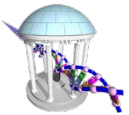
Once the DNA strands are pulled apart the process of replication begins. It proceeds in both directions on both strands and contines until the center of termimination, terC, is reached. But it doesn't progress symmetrically in both directions. DNA polymerases, the proteins which actually copy the strands, operate unidirectionally. They first must attach to specific subsequences, called primers. Once they begin, they copy the attached strands only along the (3' → 5') direction.



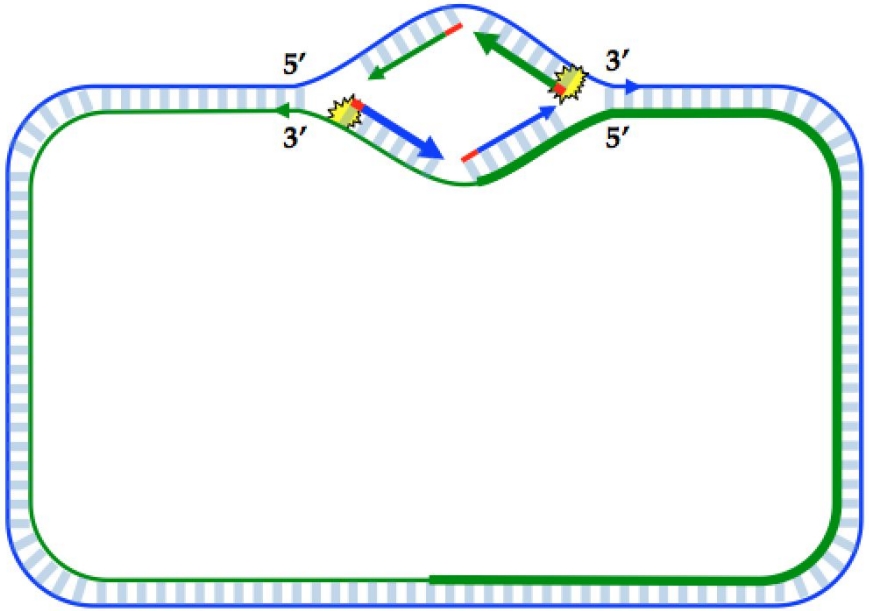
Beginning at the *oriC* locus the DNA molecule is pulled apart and two DNA polymerases, one on each strand begin copying on each strand.

As they progress the DNA separates more. The boundrary is called the ***replication fork***. Eventually, this separation exposes a significantly large single-stranded DNA on the trailing edge of each strand.

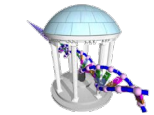
Once the replication fork opens enough...



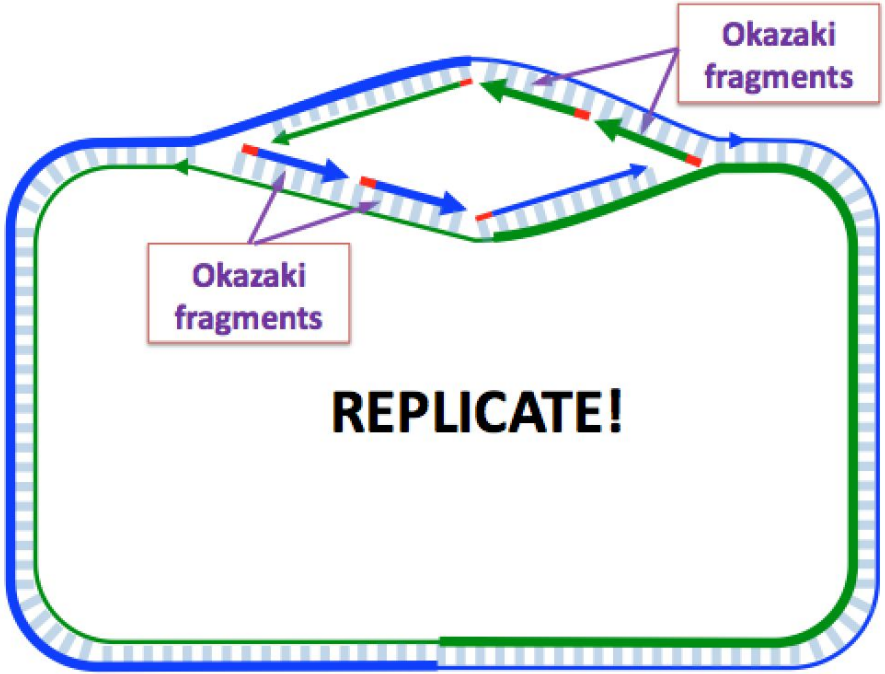
This open region of single-stranded DNA eventually allows a second phase of the replication process to begin. A second DNA polymerase detects a primer sequence, and then start replicating the exposed sequence Ahead of it and works towards the beginning of the previous replication primer. However, this DNA polymerase does not have too far to go.



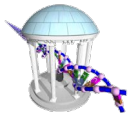
When opened a little more



As the initial, or *Leading*, polymerase continues to copy its half strand more of the complement strand is exposed, which sets off the process over and over again until the termination center is reached.

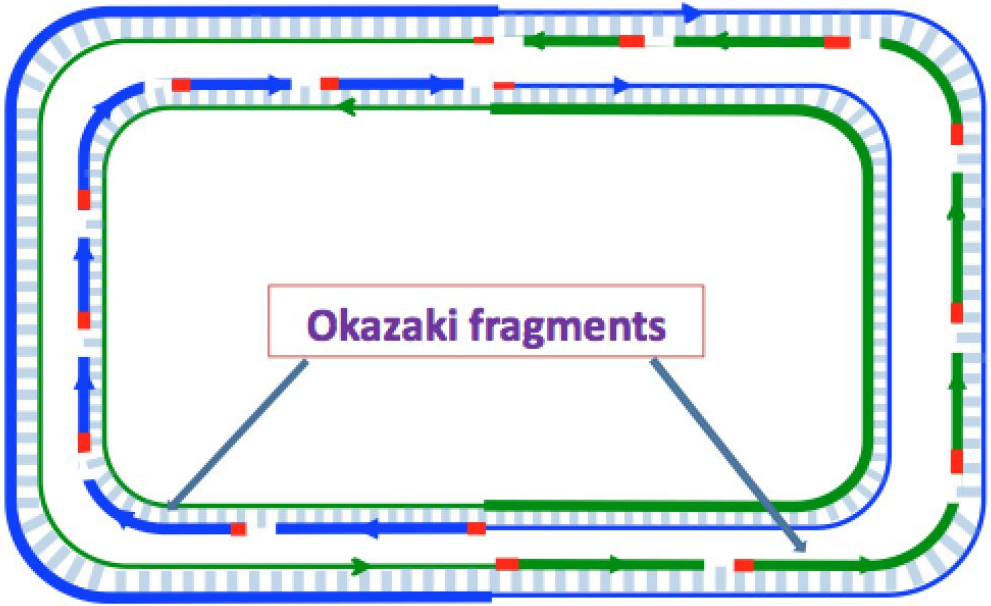


Eventually the whole genome is replicated



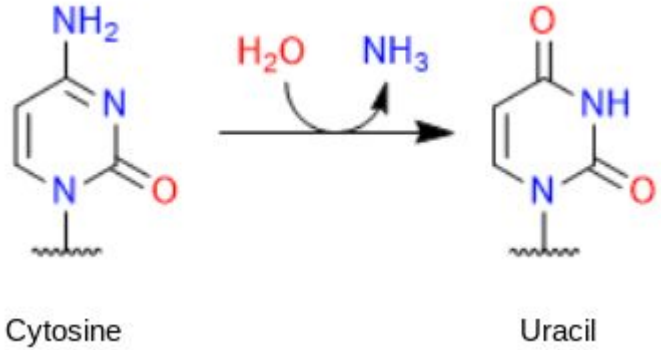
The lengths of Okazaki fragments in prokaryotes and eukaryotes differ. Prokaryotes tend to have longer Okazaki fragments (≈ 2,000 nucleotides long) than eukaryotes (100 to 200 nucleotides long).

Once completed, the adjacent Okazaki fragments are joined by another important protein called a DNA ligase.

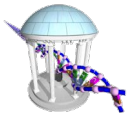


Observations and Implications

- **The leading half strand is copied as a single contiguous piece that progresses at a uniform rate as the DNA separates**
- **The other lagging half strand lies exposed while waiting for the gap to enlarge enough, and until another primer sequence appears so that another DNA polymerase can start**
- **Replication on the lagging half-strand proceeds in a stop-and-go fashion extending by one Okazaki fragment at a time**
- **A DNA repair mechanism then comes along to fix all of the lagging half-strand fragments**
- **What is the downside of leaving single-stranded DNA exposed?**
  - Single-stranded DNA is less stable than double-stranded
  - Single-stranded DNA can potentially mutate when exposed
  - The most common mutation type is called deanimation
  - Deanimation tends to convert C nucelotides into T nucelotides.



Now what?



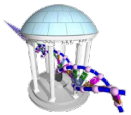
- How might these observations inform a new algorithm for finding OriC?
- When considering the half-strands on either side of a candidate OriC region what would we expect?
- More primer patterns on the lagging side to promote Okazaki fragments
- Which primer do we look for?
- Go back to our k-mer counts from last time?

But whatever the primer pattern is, there should be fewer Cytosines on the lagging side due to deanimation over multiple generations (replications)

Idea: Look for positions that divide the genome such that number of Cs in the suffix, and prefix, reverse complemented, are minimized

```
                                        fewer Cs -->
5'-...CAAACCTACCACCAAACTCTGTATTGACCA|TTTTAGGACAACTTCAGGGTGGTAGGTTTC...-3'
3'-...GTTTGGATGGTGGTTTGAGACATAACTGGT|AAAATCCTGTTGAAGTCCCACCATCCAAAG...-5'
                          <-- fewer Cs
```

Looking for Evidence

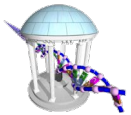


Recall *Thermotoga Petrophila*, (the bacteria whose k-mers did not match the frequent ones that we found in *Vibrio Cholerae*). Let's examine the nucleotide counts on either side of its *OriC* region:

Base	Total	Forward	Reverse	Difference
C	427419	207901	219518	-11617
G	413241	211607	201634	9973
A	491488	247525	243963	3562
T	491363	244722	246641	-1919

The lagging strand in the primary sequence corresponds to exposed Cs in the direction of increasing indices, while Gs in the direction of decreasing indices of the primary sequence correspond to Cs of the lagging strand. Thus, the lagging strands have 9973 + 11617 = 21590 fewer Cs than the leading strands.

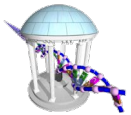
Another Genome



In [19]:
```
header, seq = loadFasta("data/ThermotogaPetrophila.fa")

for i in range(len(header)):
    print(header[i])
    print(len(seq[i])-1, "bases", seq[i][:30], "...", seq[i][-30:])
    print()

oriCStart = 786686
oriOffset = 211          # offset to the middle of OriC

x = seq[0][oriCStart+oriOffset-50:oriCStart+oriOffset+50]
y = ''.join({'A':'T', 'C':'G', 'G':'C', 'T':'A'}[b] for b in x)
print(x[:50],x[50:])
print(y[:50],y[50:])
```

```
CP000702.1 Thermotoga petrophila RKU-1, complete genome
1823511 bases +AGTTGGACGAAGGTTCTGATCCCTACAGA ... TCAATGTTATAATAAATACCGTGCAAAAAC

GGAATTGAATATATGCAAAACAAACCTACCACCAAACTCTGTATTGACCA TTTTAGGACAACTTCAGGGTGGTAGGTTTCTGAAGCTCTCATCAATAGAC
CCTTAACTTATATACGTTTTGTTTGGATGGTGGTTTGAGACATAACTGGT AAAATCCTGTTGAAGTCCCACCATCCAAAGACTTCGAGAGTAGTTATCTG
```

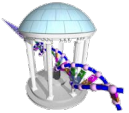
Counting Bases



In [22]:
```
def getStats(sequence, start):
    halflen = len(sequence)//2
    terC = start + halflen
    # handle genome's circular nature
    if (terC > len(sequence)):
        terC = terC - len(sequence) + 1
    stats = {}
    for base in "ACGT":
        total = sequence.count(base)
        if (terC > start):                                  # case 1: ----S========T---->
            forwardCount = sequence[start:terC].count(base)
            reverseCount = total - forwardCount
        else:                                               # case 2: ====T--------S====>
            reverseCount = sequence[terC:start].count(base)
            forwardCount = total - reverseCount
        stats[base] = (total, forwardCount, reverseCount)
    return stats

answer = getStats(seq[0], oriCStart+oriOffset)
for base in "CGAT":
    total, forwardCount, reverseCount = answer[base]
    print("%s: %8d %8d %8d %8d" % (base,total,forwardCount,reverseCount,forwardCount-reverseCount))
```

```
C:   427419   207901   219518   -11617
G:   413241   211607   201634     9973
A:   491488   247525   243963     3562
T:   491363   244723   246640    -1917
```

Genome-wide GC Skew

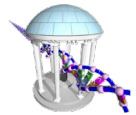


In [43]:
```
def getGCdiff(sequence, start):
    halflen = len(sequence)//2
    terC = start + halflen
    # handle genome's circular nature
    if (terC > len(sequence)):
        terC = terC - len(sequence) + 1
    if (terC > start):                                  # case 1: ----S========T---->
        G = 2*sequence[start:terC].count('G') - sequence.count('G')
        C = 2*sequence[start:terC].count('C') - sequence.count('C')
    else:                                               # case 2: ====T--------S====>
        G = sequence.count('G') - 2*sequence[terC:start].count('G')
        C = sequence.count('C') - 2*sequence[terC:start].count('C')
    return G - C

def GCSkew(genome):
    x = []
    y = []
    for i in range(1,len(genome),500):
        x.append(i)
        y.append(getGCdiff(genome,i))
    return x, y
```

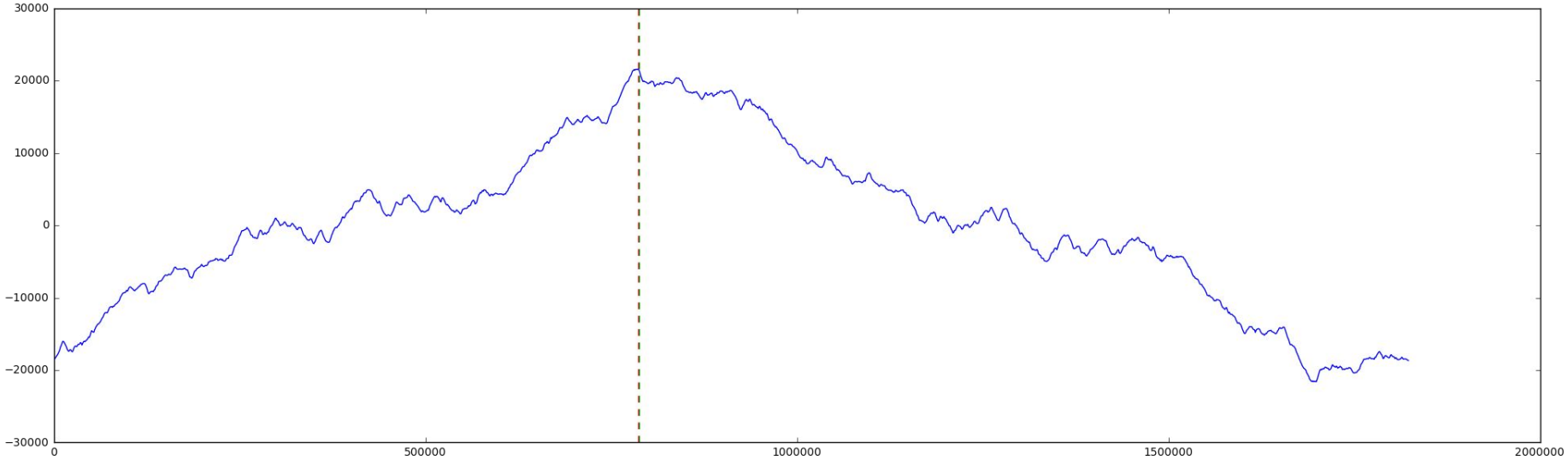
In [44]:
```
x, y = GCSkew(seq[0])

plt.figure(num=None, figsize=(24, 7), dpi=100)
yargmax = y.index(max(y))
plt.axvline(oriCStart+oriOffset, color="r", linestyle='--')
plt.axvline(x[yargmax], color="g", linestyle='--')
result = plt.plot(x, y)
print(x[yargmax], y[yargmax])
```

```
786001 21602
```

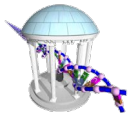
Plot of G-C Skew



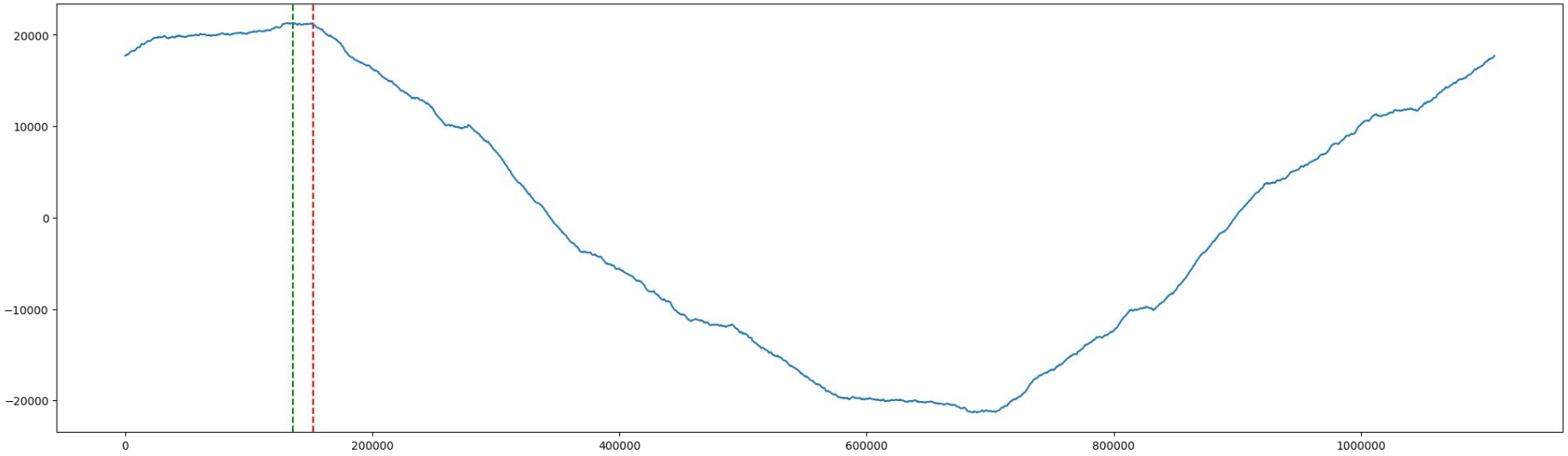
Our prediction of 786001 is slightly off from the reported value of 786686, but we pnly sampled every 500 bases.



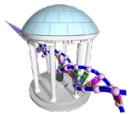
Back to our Original Colera Genome



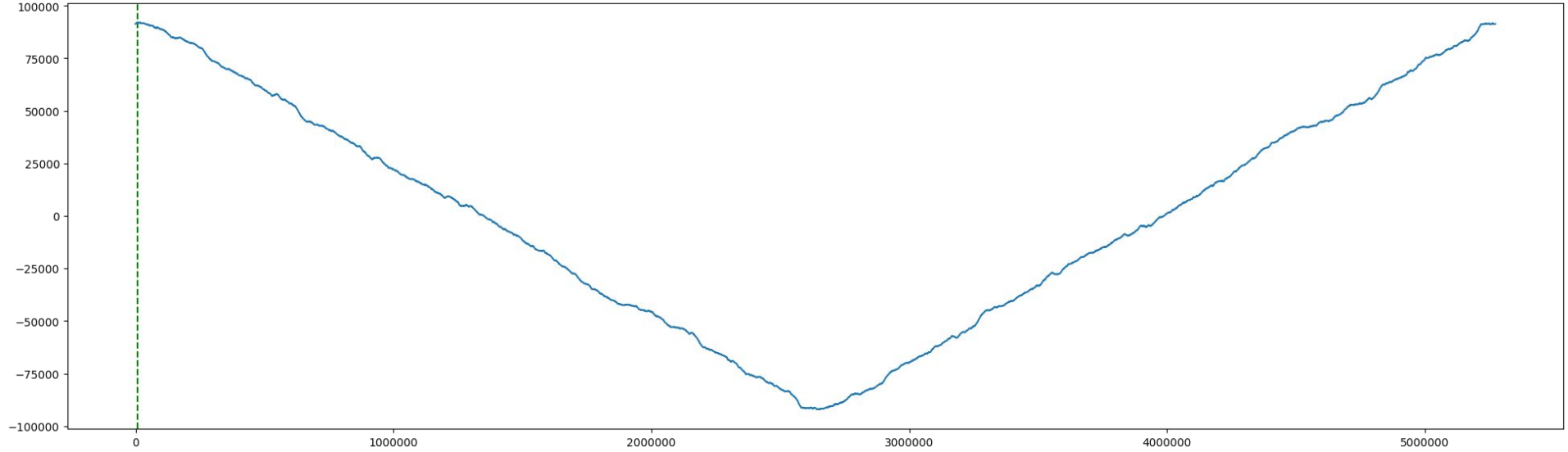
Note here, that the *OriC* estimate is more approximate, due to the relatively "flat" maximum. How might we address this?



A Genome we haven’t seen

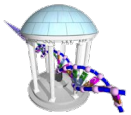


***Escherichia Coli*, or *E. Coli* for short, is a widely studied, and mostly harmless model organism, but with a few pathogenic strains. It has a larger Genome of 5.3 Mbp, and its C-G skew plot looks like:**



With a notable maximum near the beginning of its genome sequence.

Did we found the *OriC* region of *E. Coli*?

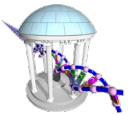


The minimum of the Skew Diagram points to this region in E. coli:

```
aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgcataacgcggta
tgaaaatggattgaagcccgggccgtggattctactcaactttgtcggcttgagaaagacc
tgggatcctgggtattaaaaagaagatctatttatttagagatctgttctattgtgatctc
ttattaggatcgcactgccctgtggataacaaggatccggcttttaagatcaacaacctgg
aaaggatcattaactgtgaatgatcggtgatcctggaccgtataagctgggatcagaatga
ggggttatacacaactcaaaaactgaacaacagttgttctttggataactaccggttgatc
caagcttcctgacagagttatccacagtagatcgcacgatctgtatacttatttgagtaaa
ttaacccacgatcccagccattcttctgccggatcttccggaatgtcgtgatcaagaatgt
tgatcttcagtg
```

But there are NO frequent 9-mers (that appear three or more times) in this region!

DnaA is more forgiving than we imagined



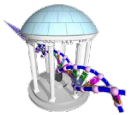
The *OriC* binding sites might not have exactly repeated 9-mers, but instead 9-mers that are very close in their target sequence. The *DnaA* is willing to look over these small differences.

This leads to a new problem:

Frequent Approximate k-mer Matches: Find the most frequent k-mer allowing for a small number of mismatches.

Input: A string Text, and integers k and d
Output: All most frequent k-mers with up to d mismatches in Text

Example: Revisiting Vibrio Cholerae

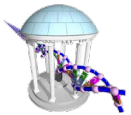


If we allow for just one difference in the 9-mers ATGATCAAG and CTTGATCAT that we found for Vibrio Cholerae, we see a few more potential binding regions pop out.

atca**ATGATCAAC**gtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaag**ATGATCAAG**agaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaag**CATGATCAT**ggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacct**CTTGATCAT**cgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagct**CTTGATCAT**gtt
tccttaaccctctatttttacggaaga**ATGATCAAG**ctgctgct**CTTGATCAT**cgtttc

How would you approach this problem?

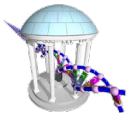
Finally, the *DnaA* Boxes of *E. Coli*



Frequent 9-mers, and their reverse complements, allowing for 1-Mismatch in the inferred *oriC* region of *E. Coli*.

aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgcataacgcggta
tgaaaatggattgaagcccgggccgtggattctactcaactttgtcggcttgagaaagacc
tgggatcctgggtattaaaaagaagatctatttatttagagatctgttctattgtgatctc
ttattaggatcgcactgcccTGTGGATAAcaaggatccggcttttaagatcaacaacctgg
aaaggatcattaactgtgaatgatcggtgatcctggaccgtataagctgggatcagaatga
ggggTTATACACAactcaaaaactgaacaacagttgttcTTTGGATAActaccggttgatc
caagcttcctgacagagTTATCCACAgtagatcgcacgatctgtatacttatttgagtaaa
ttaacccacgatcccagccattcttctgccggatcttccggaatgtcgtgatcaagaatgt
tgatcttcagtg

Summary



The problem of finding the *OriC* region of the genome is really just a toy problem to get us thinking about both biology and algorithms and how they interact.

Algorithms are like experiments. Most often, they don't provide answers, but only evidence to support a hypothesis. Often, they need to be combined until the evidence is unrefutable.

Combining biological insights into algorithms can lead to innovative and complemetary approaches to standard, wet-lab and dry-lab methods.