Where does DNA Replication Begin?

continued from last time...

After a couple of false starts, we continue on our quest to develop an algorithm for finding the origin of replication, OriC, locus in a DNA sequence.
Let's take a closer look at the biology

Recall DNA Strands have Directions:

The two strands run in opposite directions (from 5' to 3'):
Blue Strand Clockwise,
Green Strand Counter-Clockwise
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 continues until the center of termination, terC, is reached.

But it doesn’t exactly 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 along the (3’ → 5’) direction.
Replication progresses in one 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 boundary of the separation between single-stranded and double-stranded DNA is called the replication fork. Eventually, this separation exposes a significantly large single-stranded DNA on 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.

These short partial copies are called *Okazaki fragments* and they lie along the *Lagging* half-strand of the replication.
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

- 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 *deamination*.
- Deamination tends to convert C nucleotides into T nucleotides.

\[
\begin{align*}
\text{Cytosine (C)} & \quad \text{Uracil (U)} \\
\end{align*}
\]
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 deamination over multiple generations (replications)

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

Fewer Cs -->

\[
\begin{align*}
5' & \ldots \text{CAACCTACCCACCAACTCTGTATTGACCA} | \text{TTTTAGGACACCTCAGGGTGTTAGGTTTC} \ldots 3' \\
3' & \ldots \text{GTGGATGTTTGGAGCATACAGGTTG} | \text{AAAATCCTGTTGAAGTCCACCATCCAAAG} \ldots 5'
\end{align*}
\]

<-- fewer Cs
**Let's look for evidence**

Recall *Thermotoga Petrophila*, from last lecture (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:

<table>
<thead>
<tr>
<th>base</th>
<th>Total</th>
<th>Forward</th>
<th>Reverse</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>427419</td>
<td>207901</td>
<td>219518</td>
<td>-11617</td>
</tr>
<tr>
<td>G</td>
<td>413241</td>
<td>211607</td>
<td>201634</td>
<td>9973</td>
</tr>
<tr>
<td>A</td>
<td>491488</td>
<td>247525</td>
<td>243963</td>
<td>3562</td>
</tr>
<tr>
<td>T</td>
<td>491363</td>
<td>244722</td>
<td>246641</td>
<td>-1919</td>
</tr>
</tbody>
</table>

The *Lagging strand* in the forward direction corresponds to exposed Cs, while Gs in the reverse direction correspond to Cs of the *Lagging strand*. Thus, the Lagging strands have $9973 + 11617 = 21590$ fewer Cs than the Leading strands.
Code for reading sequences from last time

def loadFasta(filename):
    """ Parses a classically formatted and possibly
    compressed FASTA file into a list of headers
    and fragment sequences for each sequence contained"""
    if (filename.endswith(".gz")):
        fp = gzip.open(filename, 'rb')
    else:
        fp = open(filename, 'rb')
    # split at headers
    data = fp.read().split('>
)
    fp.close()
    # ignore whatever appears before the 1st header
    data.pop(0)
    headers = []
    sequences = []
    for sequence in data:
        lines = sequence.split('n')
        headers.append(lines.pop(0))
        # add an extra "+" to make string "1-referenced"
        sequences.append('+' + ' .join(lines))

    return (headers, sequences)
Code for reading sequences from last time

import os
import numpy
import pandas

header, seq = loadFasta("data/ThermotogaPetrophila.fa")

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

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

CP000702.1 Thermotoga petrophila RKU-1, complete genome
1823511 bases +AGTTGGACGAAGTTCTGATCCCTACAGA ... TCAATGTTATAAAATACCGTGCAAAAAC
Counting base occurrences in large strings

Here's a somewhat standard approach to counting characters in a string.

```python
def getStatsV1(sequence, start):
    halflen = len(sequence)//2
    terC = start + halflen
    # handle genome's circular nature
    if (terC > len(sequence)):
        terC = terC - len(sequence) + 1
    total = { base: 0 for base in "ACGT" }
    forwardCount = { base: 0 for base in "ACGT" }
    reverseCount = { base: 0 for base in "ACGT" }
    for position in xrange(1,len(sequence)):
        base = sequence[position]
        total[base] += 1
        if (terC > start):
            if position >= start and position < terC:
                forwardCount[base] += 1
            else:
                reverseCount[base] += 1
        else:
            if position >= start or position < terC:
                forwardCount[base] += 1
            else:
                reverseCount[base] += 1
    return {key: (total[key], forwardCount[key], reverseCount[key]) for key in total.iterkeys()}
```
Another way to count

This version makes four passes, one for each base, but moves the dictionary overhead outside of the linear scan.

```python
def getStatsV2(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):
            forwardCount = sequence[start:terC].count(base)
            reverseCount = total - forwardCount
        else:
            reverseCount = sequence[terC:start].count(base)
            forwardCount = total - reverseCount
        stats[base] = (total, forwardCount, reverseCount)
    return stats
```
Let's compare counting approaches

How much difference do you expect? Why do we care?

```python
for getStats in [getStatsV1, getStatsV2]:
    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)
    %timeit getStats(seq[0], oriCStart+oriOffset)
    print
```

C:  427419  207901  219518   -11617
G:  413241  211607  201634     9973
A:  491488  247525  243963     3562
T:  491363  244723  246640    -1917
1 loop, best of 3: 517 ms per loop

C:  427419  207901  219518   -11617
G:  413241  211607  201634     9973
A:  491488  247525  243963     3562
T:  491363  244723  246640    -1917
10 loops, best of 3: 53.3 ms per loop
One more contender

Python provides an optimized library called "numpy" for processing vectorized data. Our sequence can be considered a vector of bases.

```python
import numpy

def getStatsV3(sequence, start):
    halflen = len(sequence)//2
    terC = start + halflen
    # handle genome's circular nature
    if (terC > len(sequence)):
        terC = terC - len(sequence) + 1
    genome = numpy.fromstring(sequence, dtype="uint8")
    total = numpy.bincount(genome)
    if (terC > start):
        forwardCount = numpy.bincount(genome[start:terC])
        reverseCount = total - forwardCount
    else:
        reverseCount = numpy.bincount(match[terC:start])
        forwardCount = total - reverseCount
    return {b: (total[ord(b)], forwardCount[ord(b)], reverseCount[ord(b)]) for b in "ACGT"}
```
Verify and time it

```python
answer = getStatsV3(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)
print
%timeit getStatsV2(seq[0], oriCStart+oriOffset)
%timeit getStatsV3(seq[0], oriCStart+oriOffset)

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

10 loops, best of 3: 36.8 ms per loop
100 loops, best of 3: 14.7 ms per loop
```
A New approach for finding OriC

So let's sample the genome looking for positions where the #G - #C is maximally skewed.
Counting with cumulative sums

We’ll use a vectorized cumulative sum method to compute counts in the G-C skew genome wide. Given an input vector, V, of length N. S = V.cumsum() returns:

\[ S_i = \sum_{j=0}^{i} V_j \]

Cumulative sums can be used to compute counts over any interval, \( Count_{ij} = S_j - S_i \). Example:

```python
v = numpy.array(numpy.random.random(20) < 0.25, dtype="int8")
s = numpy.concatenate(([0], v.cumsum()))
print v
print s
```

```
[0 1 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0]
[0 0 1 1 1 2 3 3 4 5 6 7 7 7 7 7 7 7 7 7 7 7 7 7]
5
```
def GCSkew(sequence):
    half = len(sequence)//2
    full = len(sequence)
    genome = numpy.fromstring(sequence+sequence, dtype='uint8')
    matchC = numpy.concatenate(([0], numpy.array(genome == ord('C'), dtype='int8').cumsum()))
    matchG = numpy.concatenate(([0], numpy.array(genome == ord('G'), dtype='int8').cumsum()))
    matchGC = matchG - matchC
    skew = matchGC[half:half+full]-matchGC[0:full]+matchGC[full-half:2*full-half]-matchGC[full:2*full]
    return skew

Let's test it function on the short sequence: CATGGGCATCGGCCATACGCC

```python
import matplotlib
import matplotlib.pyplot as plt
test = "+CATGGGCATCGGCCATACGCC"
y = GCSkew(test)
plt.figure(num=None, figsize=(24, 8), dpi=100)
plt.ylim([-10,10])
plt.xticks(range(len(test)), [c for c in test])
result = plt.plot(range(len(y)), y)
```
Finding the genome-wide GC skew
Now let's revisit our genome

```python
# Run on Thermotoga Petrophila
y = GCSkew(seq[0])
N = len(y)
plt.figure(num=None, figsize=(24, 7), dpi=100)
plt.axvline(oriCStart+oriOffset, color="r", linestyle='--')
result = plt.plot(range(0,N,1000), y[0:N:1000])
```
Now on the original Colera genome

header, seq = loadFasta("data/VibrioCholerae.fa")
oricStart = 151887
y = GCSkew(seq[0])
N = len(y)
plt.figure(num=None, figsize=(24, 7), dpi=100)
plt.axvline(oricStart, color="r", linestyle='--')
result = plt.plot(range(0, N, 1000), y[0:N:1000])
A 3rd "test" genome

header, seq = loadFasta("data/EscherichiaColi.fa")

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

CP003289.1 Escherichia coli 0104:H4 str. 2011C-3493, complete genome
5273097 bases +CATTATCGACTTTTGTTTCAGTGAGTCC ... GTCAACAATCATGATTTTAC6CCTTAGT

CP003291.1 Escherichia coli 0104:H4 str. 2011C-3493 plasmid pAA-EA11, complete sequence
74217 bases +GCCTCGCAAAAACATTGCTCTATTCATGCA ... TTCTGACCGTCCTGATTTCTGTATATATA

CP003290.1 Escherichia coli 0104:H4 str. 2011C-3493 plasmid pESBL-EA11, complete sequence
88544 bases +GTTGGGATGACGCCAGACCAACCTCAAAT ... CGCCTGGTGCCAGTTCTGTATTTTATTT

CP003292.1 Escherichia coli 0104:H4 str. 2011C-3493 plasmid pG-EA11, complete sequence
1549 bases +CTAGCTGAAAAACCTTGGAGTTAGCAAG ... TGTGGCGCTGTGTTGCGGATCAGCAATT
Plot the G-C skew

```python
shift = '+*seq[0][1:1000000]+seq[0][1:1000000]
y = GCSkew(shift)
oriCGuess = y.argmax()
N = len(y)
plt.figure(num=None, figsize=(24, 7), dpi=100)
plt.axvline(oriCGuess, color='r', linestyle='--')
result = plt.plot(range(0, N, 1000), y[0:N:1000])
```
Did we found the OriC region of E. Coli?

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

```
aatgatgatagctcaaaagatccggataaaacatggtgattgcctgcataacgcgta
tgaaatggctgtaagccgccctggctttactcaactttgtcgcctgagaagacc
tgggtcctgggtataaaaaagagatctatatttttagagatcgtgtttgtgtctc
ttttaaggatgcctgcctgtggataaacaaggtatccggcttttaagatcaacaacctgg
aaaggatcattaactcttgatagtctcggatcctgggtattaaaaagaagatctatttattttagagatctgctgatctctattaggatcgcactgccctgtggataacaaggatccggcttttaagatcaacaacctgg
aaaggatcattaactcttgatagtctcggatcctgggtattaaaaagaagatctatttattttagagatctgctgatctctattaggatcgcactgccctgtggataacaaggatccggcttttaagatcaacaacctgg
```

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

**What now?**
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 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.

```
ATGATCAAG
gtgctcacacagtttatccacaac
cggaaagATGATCAAGagaggatgatttcttgcccatatcgcaatgaatctttgtgacctt
gtccttccatgacatctccagcccatattgcgctggccaaaggtgacggagcggatt
acgaagaACATGATCAAgctgctttgtttctgttatattttttgactgagacttgttagga
tagacggttttttctcactagcactagccaaagcctttactcttgccctgacatcgacgaattatgataatgtttcagtttccgagcagattttacct
CTTGATCATcgaatccagtgaagatctccattgctgctgctgacctatcctgagcagctgctgctgct
```

How would you approach this problem?
Finally, the \textit{DnaA} Boxes of \textit{E. Coli}

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

\begin{verbatim}
aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctgcataacgcgga
tgaaatgattgaagccgccgcggctgatatctaactattttgtgctgcttgagaagacc
tgggatctgggtattaaagaaagatctatttatatttagagatctgtttatgatcttc
tatagagatgcactgccctgGTAATGGGGAAcaaggatccggcttttaaagatcaacacctgg
aaaggatcattataacttggaatctgggtgatccctgactccgttataagctggatcagaatga
gggTTATAACAAtactcaaaaactgaaacacagtgtttcTTGTAATAATccggatttgac
cagctccctggagagTTATCCACAGtagatgtcagcatctgtataacttatatttgagtaaa
ctaaccacagatccagccattctctctgtccggatcttccgaatgtctgatcaagatgt
tgatcttctagtg
\end{verbatim}
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.

Two key concepts:

- Algorithms must be correct-- give the expected answer for any valid input
- Many algorithms compute the same function, but some are faster than others

Next time, we will think more about methods for analyzing sequences.