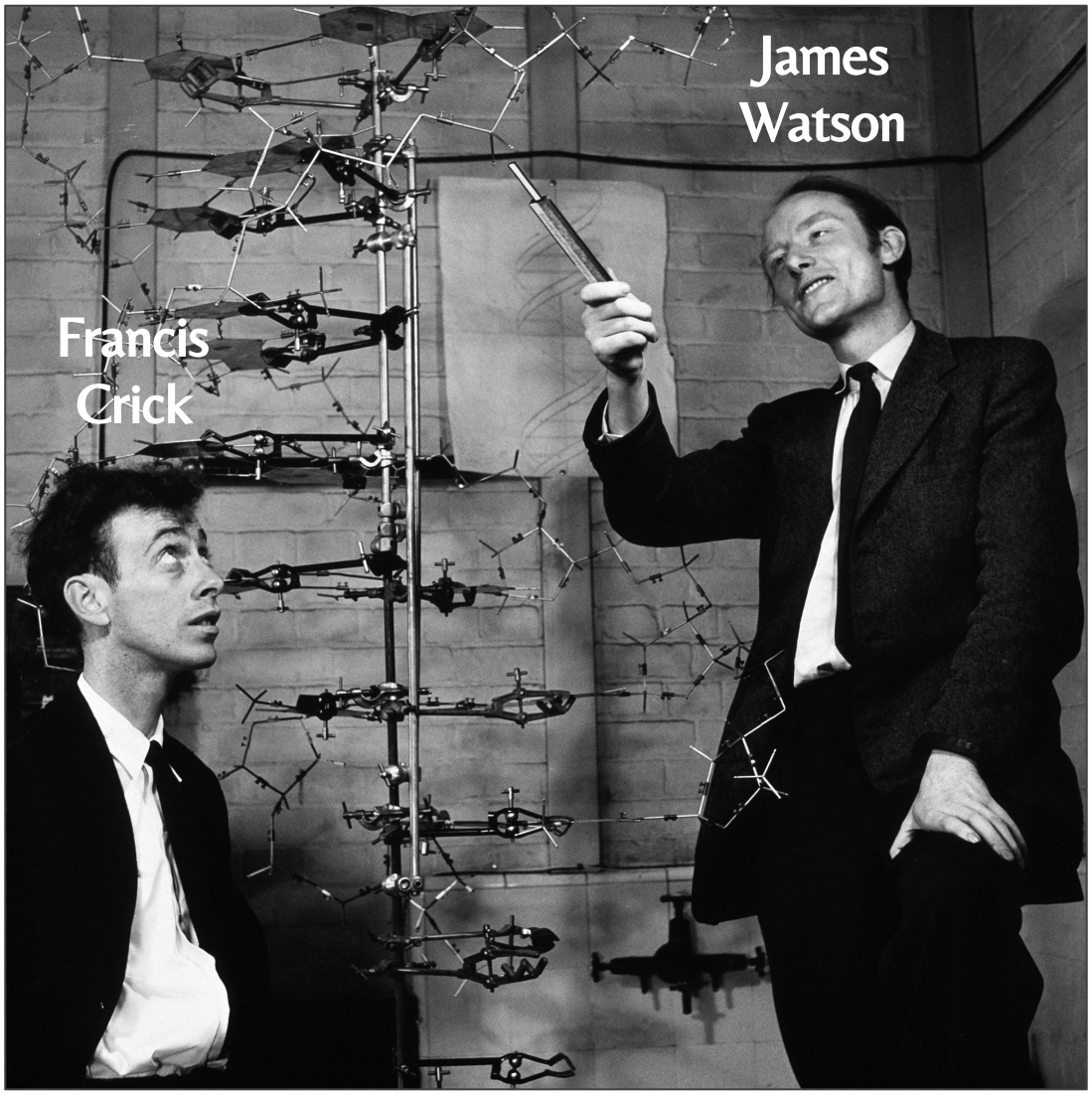
Finding Replication Origins in Bacterial Genomes

Algorithmic Warmup

Outline

- **An Intro to DNA Replication**
- Hidden Messages in the Replication Origin
- Hunting for Frequent Words
- A Faster Frequent Words Approach
- Some Hidden Messages are More Surprising than Others
- An Explosion of Hidden Messages
- Replication Asymmetry Leads Us to the Replication Origin

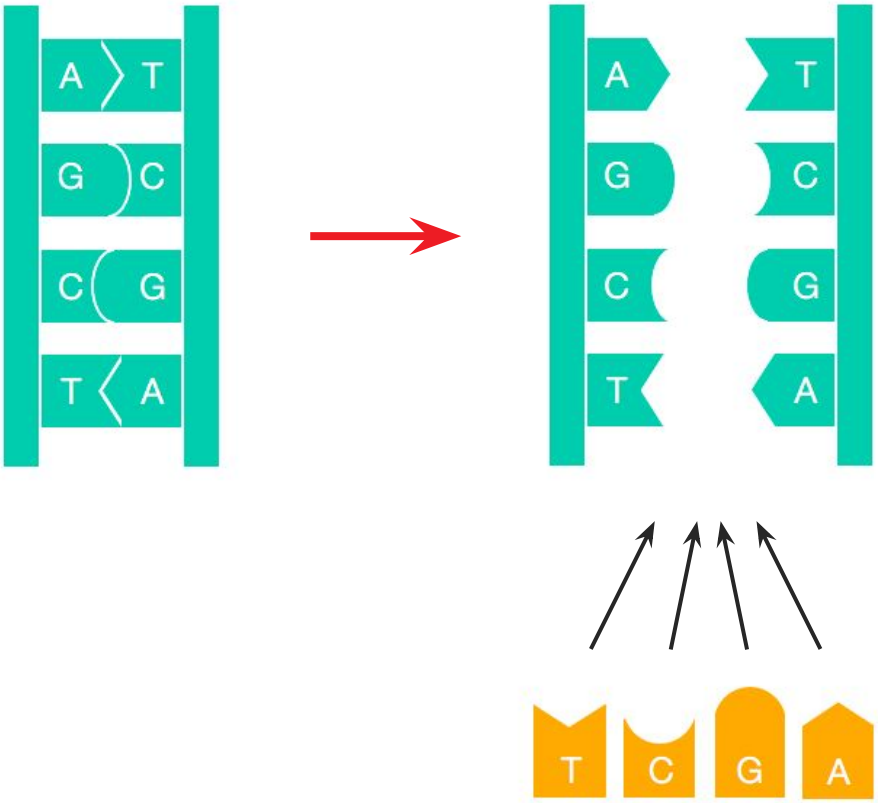
A Prophetic One-Liner (1953)



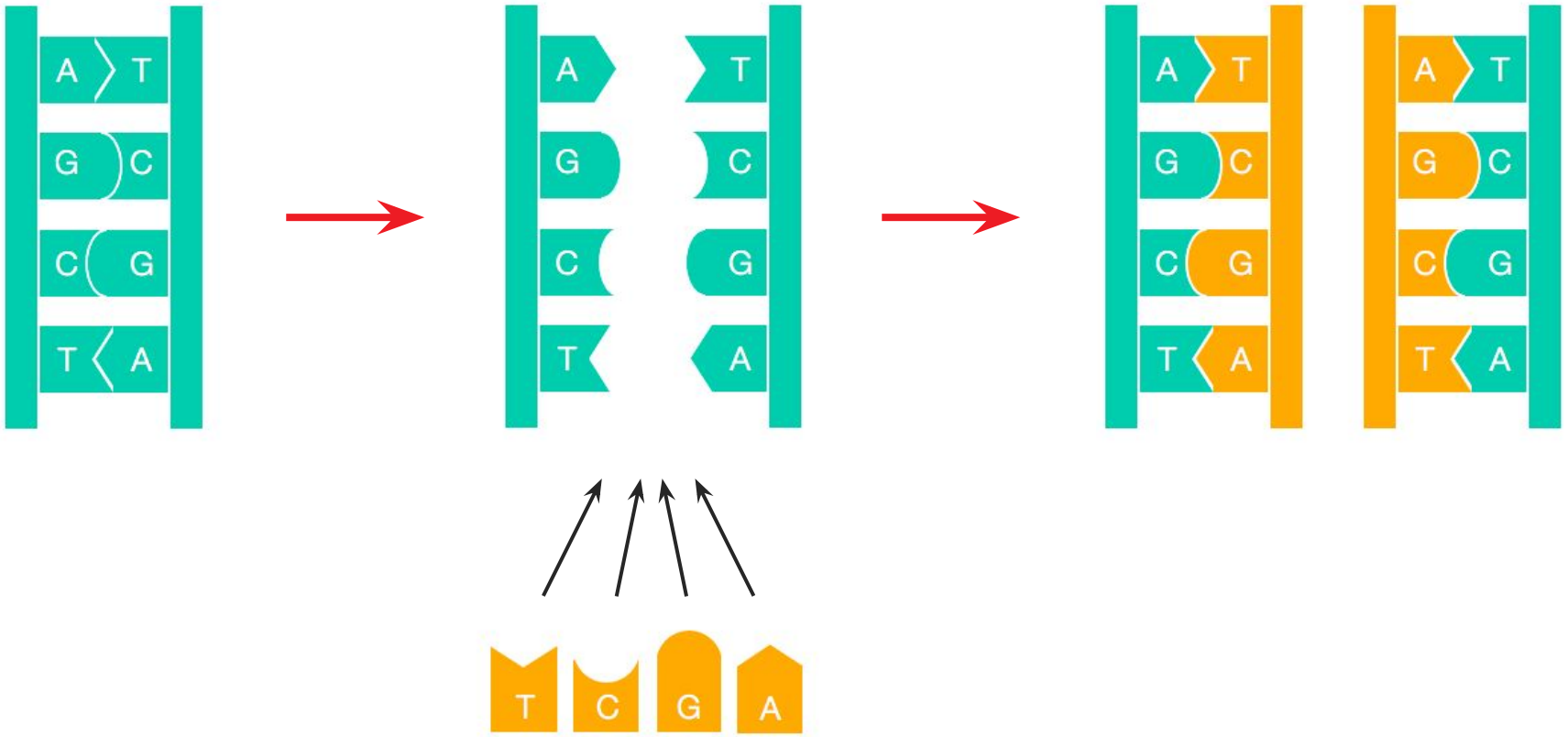
"It has not escaped our notice that the specific pairing we have postulated immediately suggests a possible copying mechanism for the genetic material."

The “Copying Mechanism”

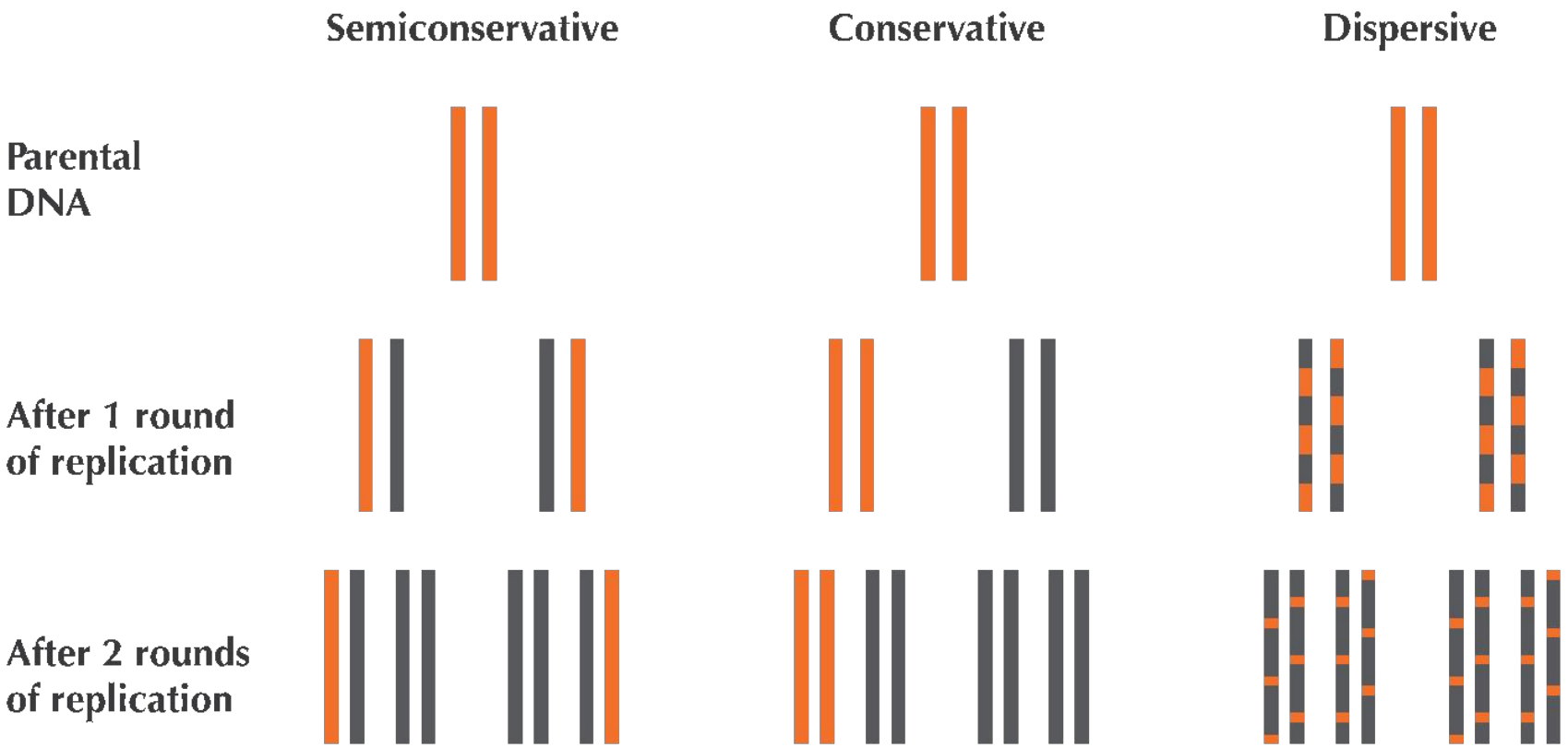
The “Copying Mechanism”



The “Copying Mechanism”



Three Hypotheses for DNA Replication



STOP: Which hypothesis was Watson & Crick's?

The Most Beautiful Experiment in Biology (1958)

Meselson and Stahl's insight: one isotope of nitrogen, Nitrogen-14 (^{14}N), is lighter and more abundant than Nitrogen-15 (^{15}N).

The Most Beautiful Experiment in Biology (1958)

Meselson and Stahl's insight: one isotope of nitrogen, Nitrogen-14 (^{14}N), is lighter and more abundant than Nitrogen-15 (^{15}N).

Meselson and Stahl grew *E. coli* for many rounds of replication in a ^{15}N medium, which caused the bacteria to gain weight as they absorbed the heavier isotope into their DNA. They then transferred the heavy *E. coli* cells to a less dense ^{14}N medium.

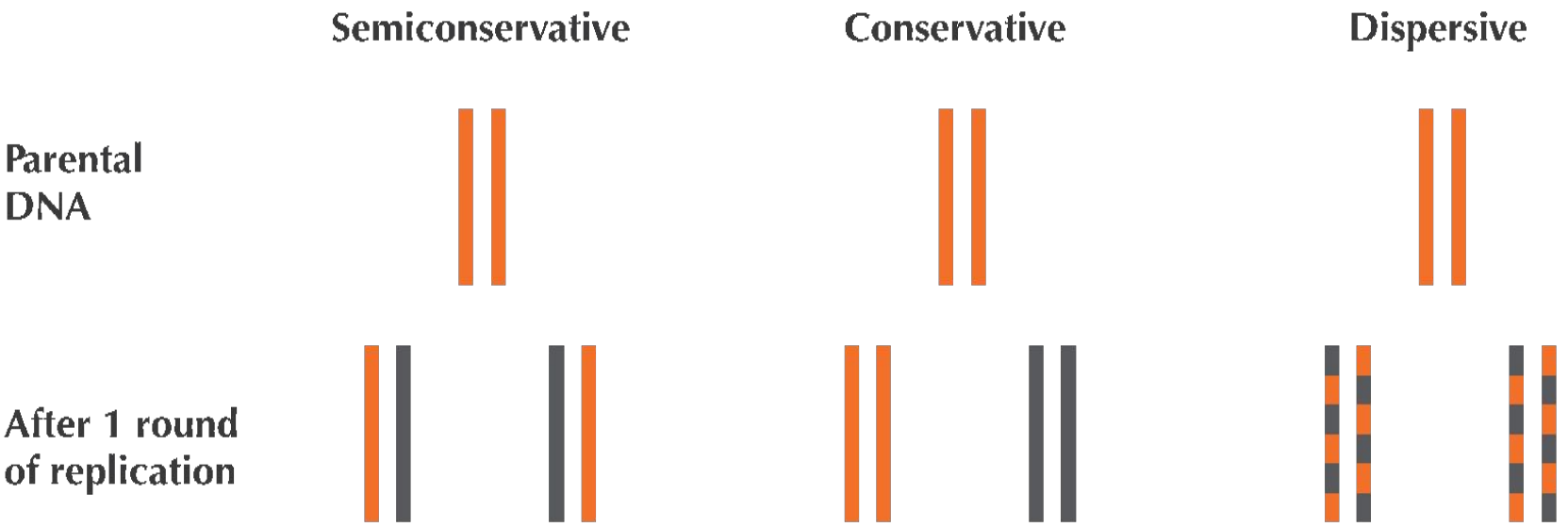
The Most Beautiful Experiment in Biology (1958)

Meselson and Stahl's insight: one isotope of nitrogen, Nitrogen-14 (^{14}N), is lighter and more abundant than Nitrogen-15 (^{15}N).

Meselson and Stahl grew *E. coli* for many rounds of replication in a ^{15}N medium, which caused the bacteria to gain weight as they absorbed the heavier isotope into their DNA. They then transferred the heavy *E. coli* cells to a less dense ^{14}N medium.

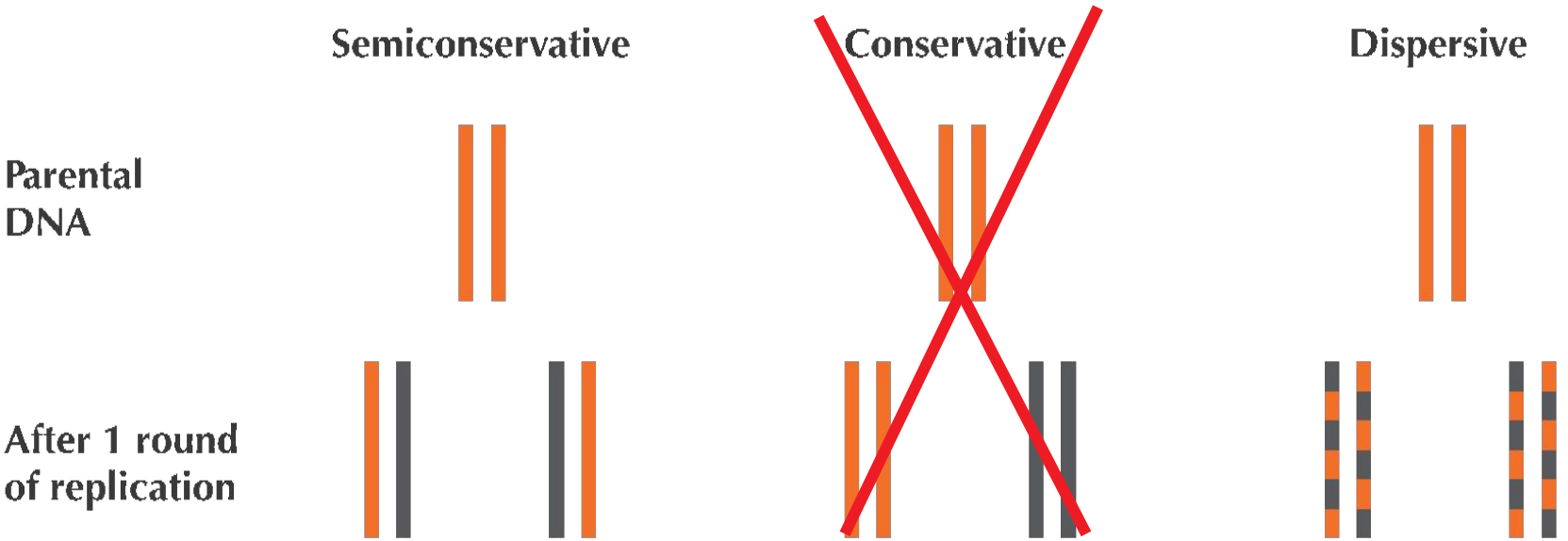
Key Point: any daughter DNA would be lighter!

The Most Beautiful Experiment in Biology (1958)



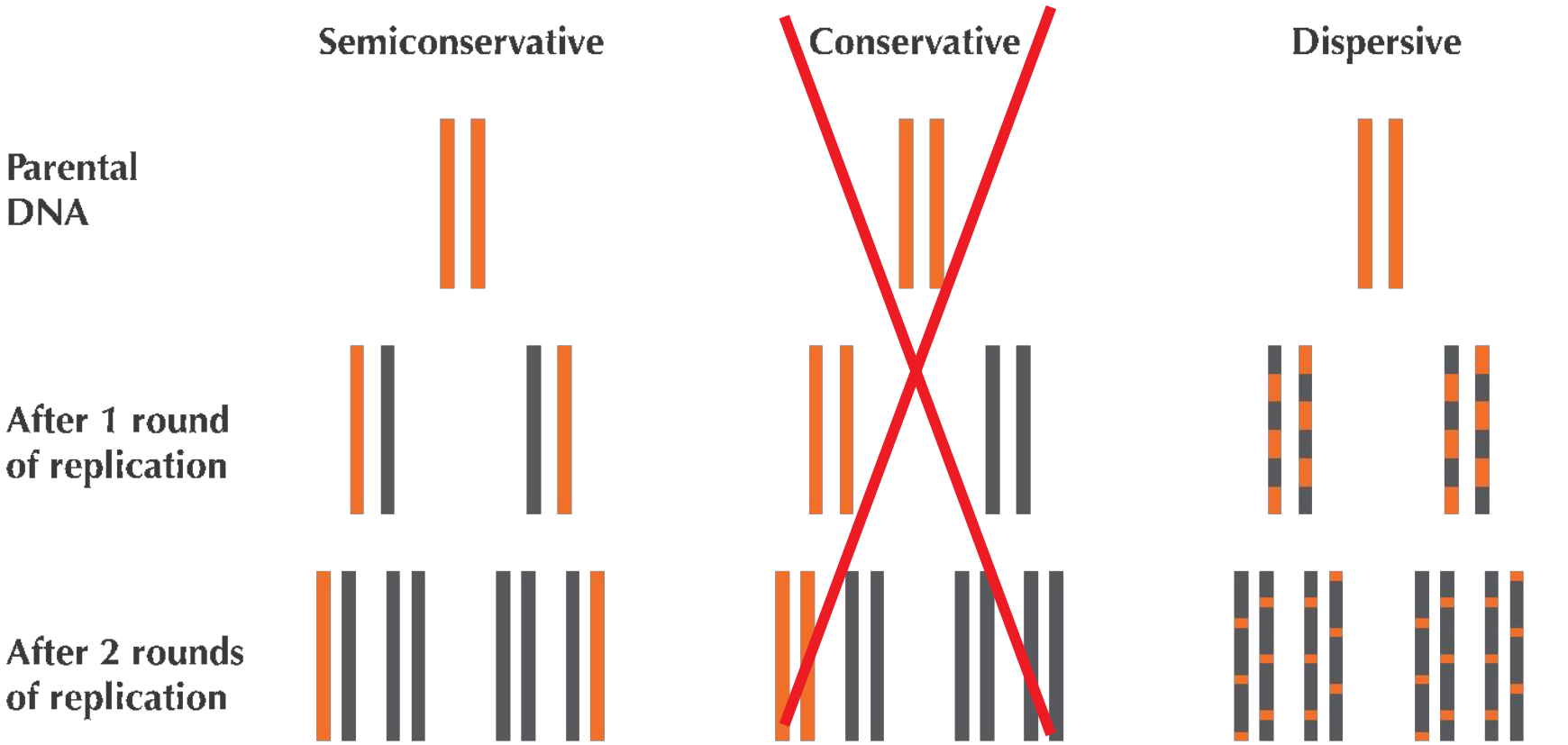
STOP: After one round of replication, Meselson and Stahl spun the DNA in a centrifuge. Why?

The Most Beautiful Experiment in Biology (1958)



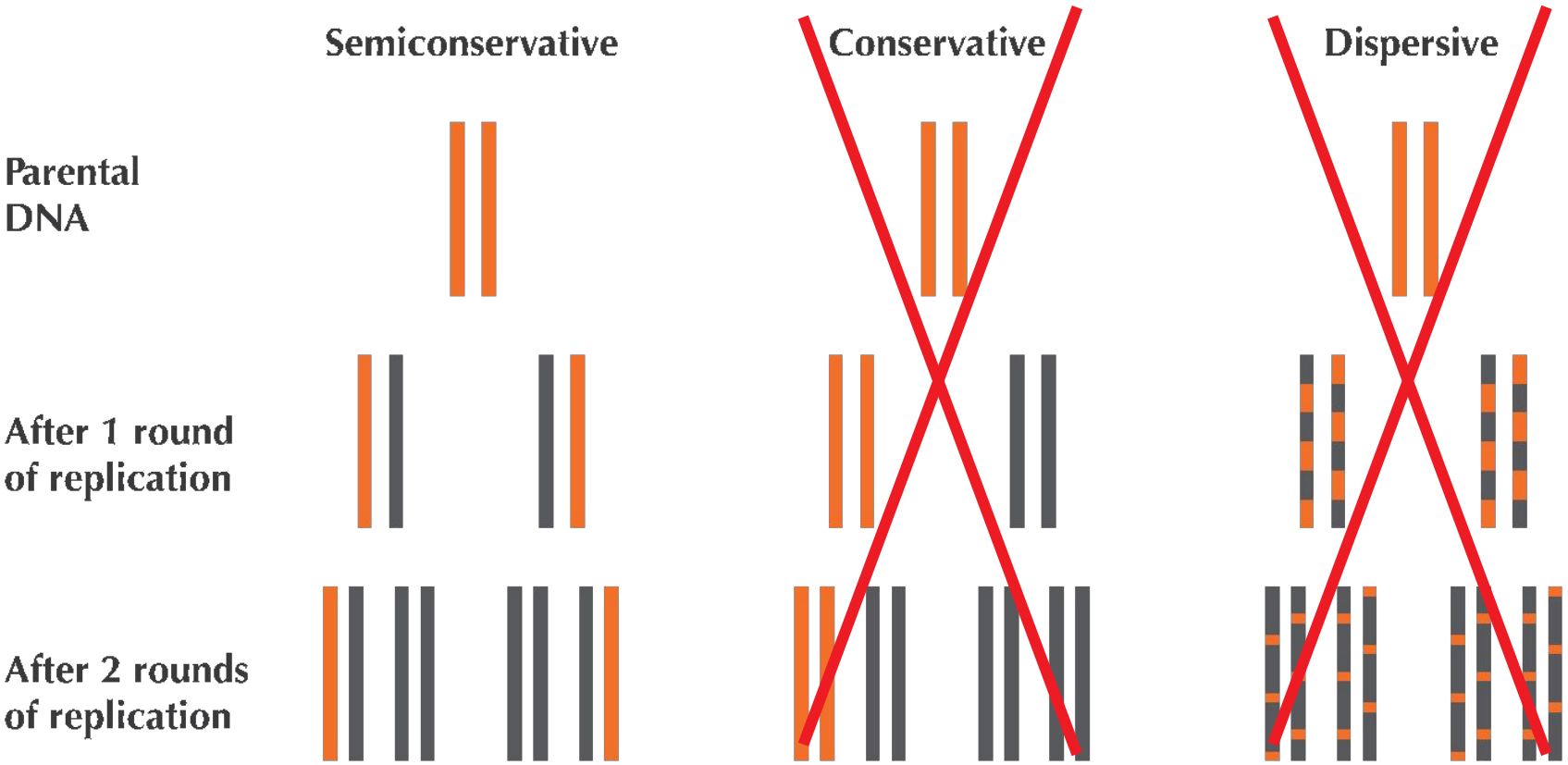
STOP: After one round of replication, Meselson and Stahl spun the DNA in a centrifuge. Why?

The Most Beautiful Experiment in Biology (1958)



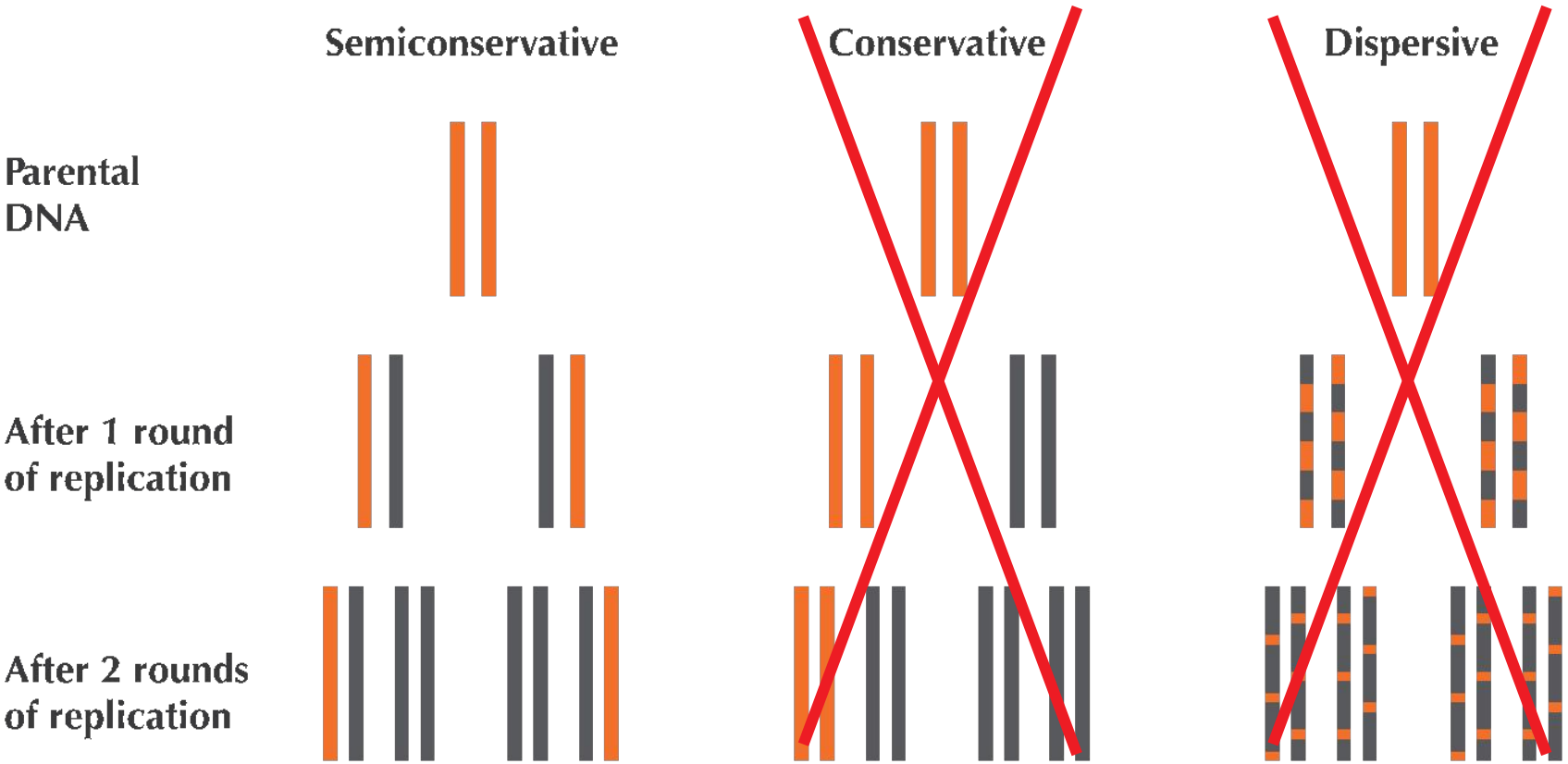
STOP: What would we observe in the centrifuge for the other two models after *two* rounds of replication?

The Most Beautiful Experiment in Biology (1958)



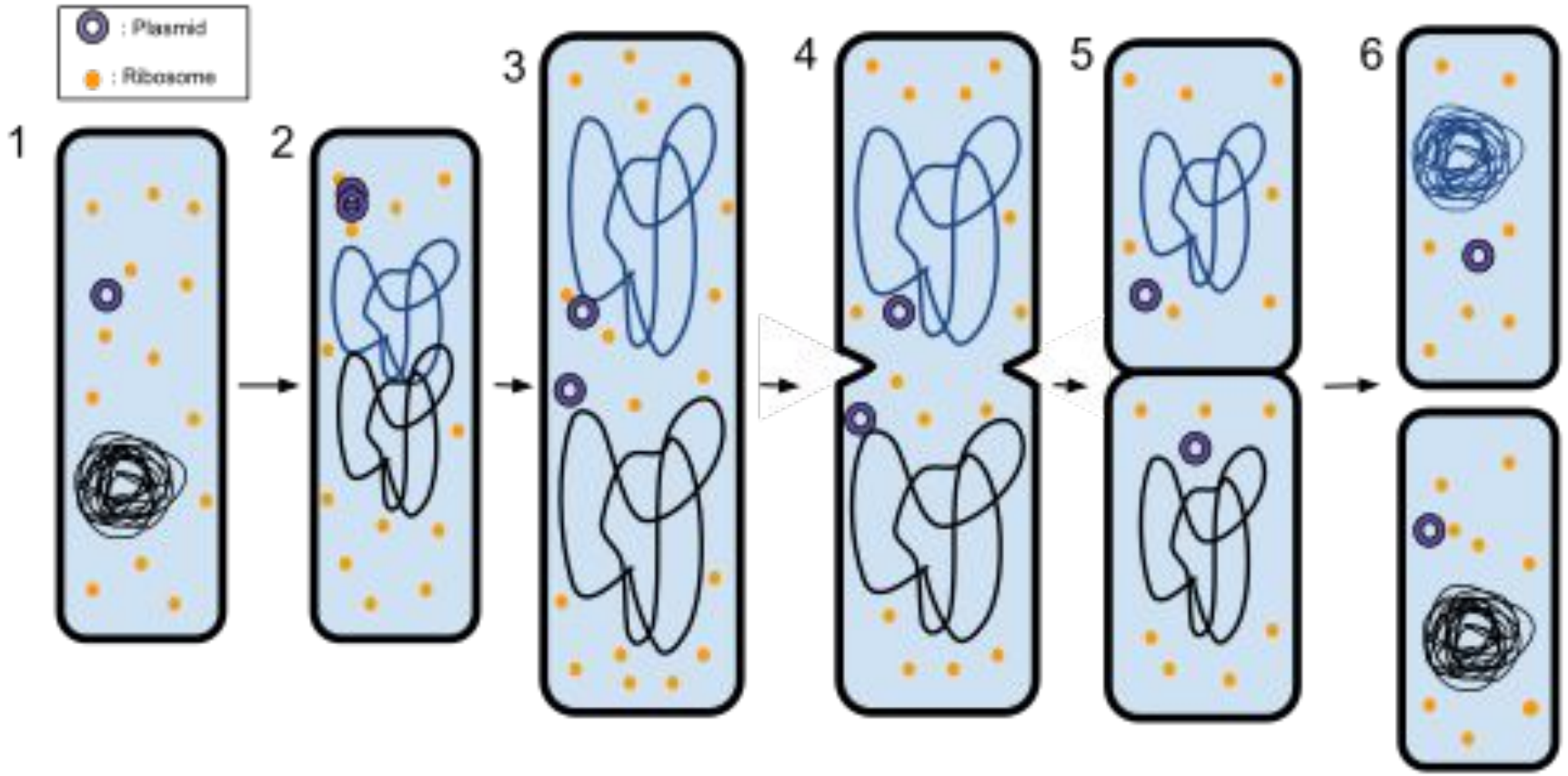
Key Point: After two rounds, the DNA divided into two different densities!

The Most Beautiful Experiment in Biology (1958)



STOP: Does this prove that the semiconservative method must be true?

What a Biologist Sees...



What a Bioinformatician Sees...

String: a contiguous collection of symbols.

What a Bioinformatician Sees...

String: a contiguous collection of symbols.

...ACTGATAACCCAGTATCAGACCAGTATCGAGGACGATACGTA...

DNA String

What a Bioinformatician Sees...

String: a contiguous collection of symbols.

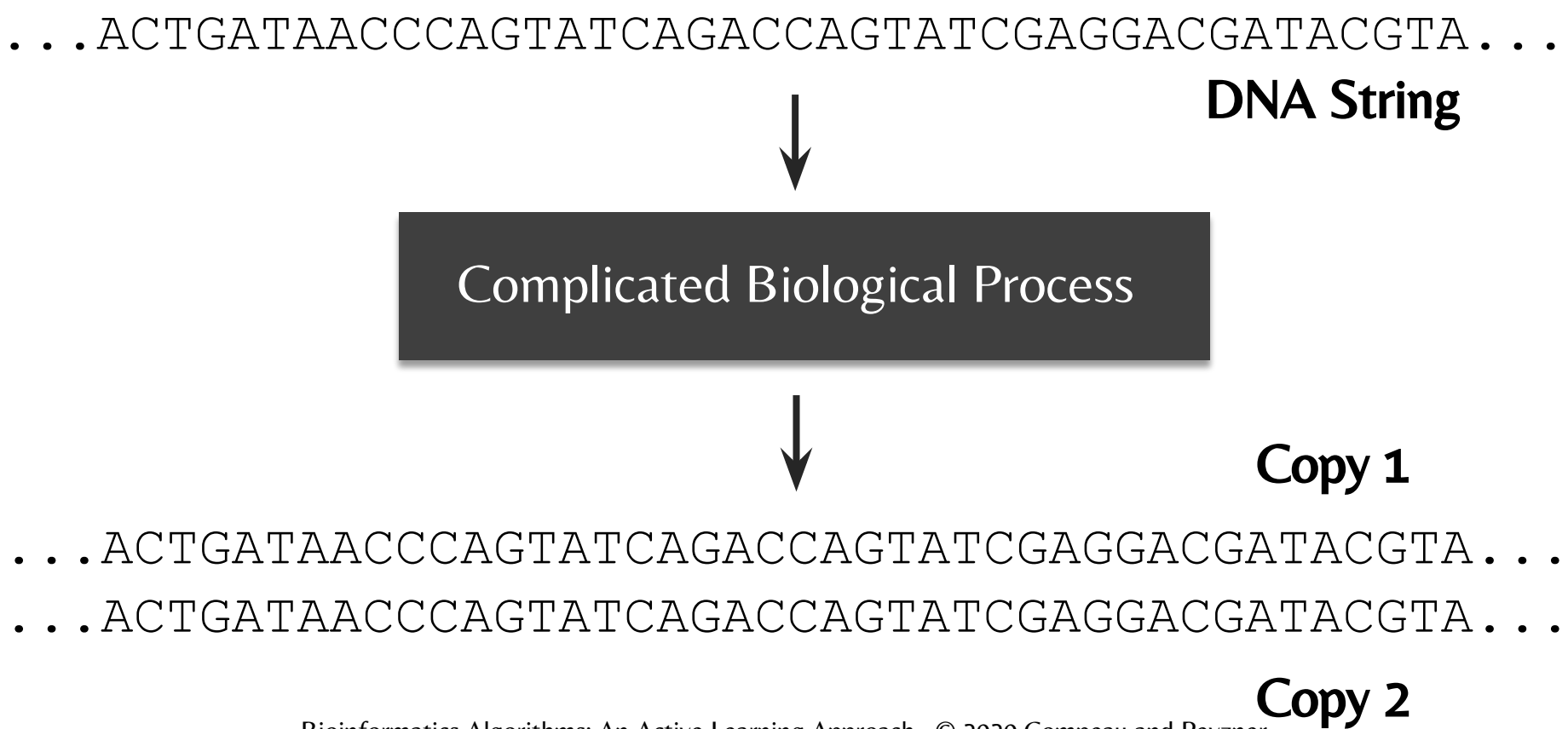
...ACTGATAACCCAGTATCAGACCAGTATCGAGGACGATACGTA...

DNA String

Complicated Biological Process

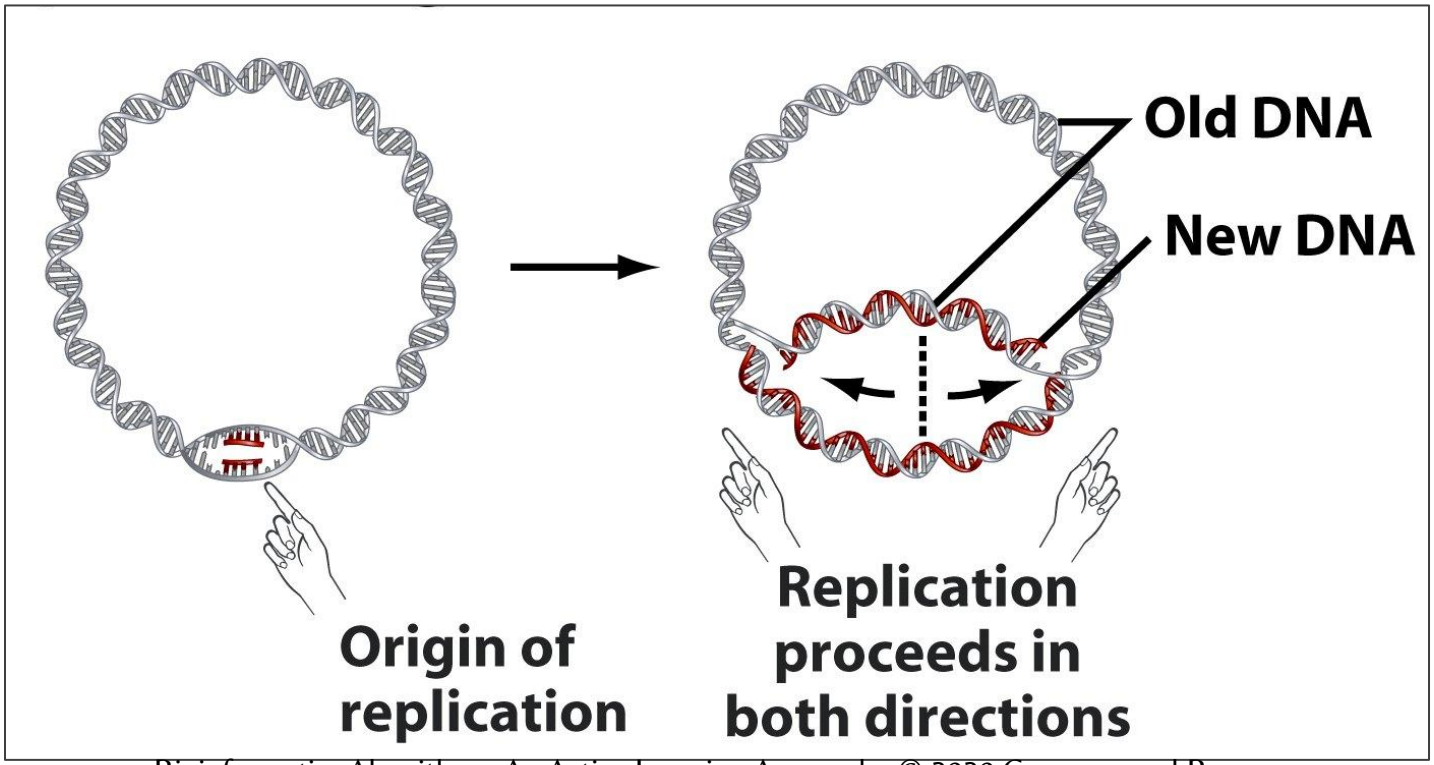
What a Computer Scientist Sees...

String: a contiguous collection of symbols.



Origin of Replication

Replication begins in a region called the **replication origin** (denoted *ori*).



The Finding *ori* Problem

Origin of Replication Problem

- **Input:** A DNA string *genome*.
- **Output:** The location of *ori* in *genome*.

STOP: Is the Hidden Message Problem a computational problem?

Finding the Origin of Replication the wet-way

How can we find *ori* in a genome?

Let's hack out this DNA fragment. Can the genome replicate without it?

Finding the Origin of Replication

How can we find *ori* in a genome?

Let's hack out this DNA fragment. Can the genome replicate without it?

I need more information before I can hack this problem.

Looking for *ori*

Verified *ori* of *Vibrio cholerae*, the bacterium that causes cholera (~500 nucleotides):

```
atcaatgatcaacgtaagcttctaagcatgatcaaggtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagatgatcaagagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcatgtt
tccttaaccctctatttttacggaagaatgatcaagctgctgctcttgatcatcgtttc
```

Looking for *ori*

Verified *ori* of *Vibrio cholerae*, the bacterium that causes cholera (~500 nucleotides):

```
atcaatgatcaacgtaagcttctaagcatgatcaaggtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagatgatcaagagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcatgtt
tccttaaccctctatttttacggaagaatgatcaagctgctgctcttgatcatcgtttc
```

There must be a ***hidden message*** telling the cell to start replication here.

The Hidden Message Problem

Hidden Message Problem

- **Input:** A string *text* (representing *ori*).
- **Output:** A hidden message in *text*.

The Hidden Message Problem

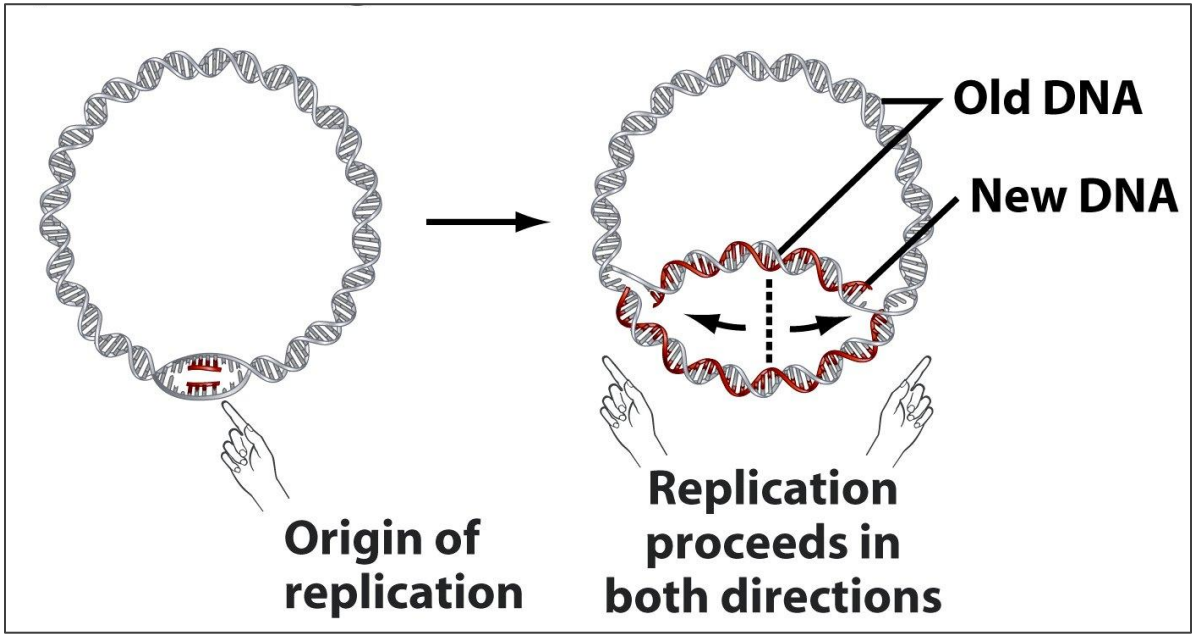
Hidden Message Problem

- **Input:** A string *text* (representing *ori*).
- **Output:** A hidden message in *text*.

STOP: Is the Hidden Message Problem a computational problem?

We Have Two Scientific Problems

1. Given a bacterial genome (~3 Mbp), where is *ori*?



2. Given *ori* (~500 bp), what is the "hidden message" saying that replication should start here?

Outline

- An Intro to DNA Replication
- **Hidden Messages in the Replication Origin**
- Hunting for Frequent Words
- A Faster Frequent Words Approach
- Some Hidden Messages are More Surprising than Others
- An Explosion of Hidden Messages
- Replication Asymmetry Leads Us to the Replication Origin

Hidden Message Problem Revisited

Hidden Message Problem

- **Input:** A string *text* (representing *ori*).
- **Output:** A hidden message in *text*.

The notion of "**hidden message**" is not defined.

Hidden Message Problem Revisited

Hidden Message Problem

- **Input:** A string *text* (representing *ori*).
- **Output:** A hidden message in *text*.

Replication initiation is mediated by a protein called ***DnaA***.

Hidden Message Problem Revisited

Hidden Message Problem

- **Input:** A string *text* (representing *ori*).
- **Output:** A hidden message in *text*.

Replication initiation is mediated by a protein called ***DnaA***.

DnaA binds to a short segment in *ori* known as a ***DnaA box***, a hidden message saying: "*bind here!*"

Hidden Message Problem Revisited

Hidden Message Problem

- **Input:** A string *text* (representing *ori*).
- **Output:** A hidden message in *text*.

Replication initiation is mediated by a protein called ***DnaA***.

DnaA binds to a short segment in *ori* known as a ***DnaA box***, a hidden message saying: "*bind here!*"

Hidden Message Problem Revisited

STOP: Would it make sense for an organism to have multiple *DnaA* boxes, or just one?

Replication initiation is mediated by a protein called ***DnaA***.

DnaA binds to a short segment in *ori* known as a ***DnaA box***, a hidden message saying: "*bind here!*"

Hidden Message Problem Revisited

Answer: Multiple *DnaA* boxes □ higher chance of binding □ higher "fitness"

"Nothing in biology makes sense except in the light of _____."

Theodosius Dobzhansky

Hidden Message Problem Revisited

Answer: Multiple *DnaA* boxes □ higher chance of binding □ higher "fitness"

"Nothing in biology makes sense except in the light of evolution."

Theodosius Dobzhansky

Outline

- An Intro to DNA Replication
- Hidden Messages in the Replication Origin
- **Hunting for Frequent Words**
- A Faster Frequent Words Approach
- Some Hidden Messages are More Surprising than Others
- An Explosion of Hidden Messages
- Replication Asymmetry Leads Us to the Replication Origin

Counting Words

```
atcaatgatcaacgtaagcttctaagcatgatcaaggtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagatgatcaagagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcatgtt
tccttaaccctctatttttacggaagaatgatcaagctgctgctcttgatcatcgtttc
```

We are looking for surprisingly frequent **substrings** (contiguous strings appearing within) this *ori*.

Counting Words

```
atcaatgatcaacgtaagcttctaagcatgatcaaggtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagatgatcaagagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcatgtt
tccttaaccctctatttttacggaagaatgatcaagctgctgctcttgatcatcgtttc
```

We are looking for surprisingly frequent **substrings** (contiguous strings appearing within) this *ori*.

First: let's count how often a *given* substring occurs.

Counting Words Problem

Substring Counting Problem

- **Input:** A string *pattern* and a longer string *text*.
- **Output:** The number of times *pattern* occurs in *text*.

Counting Words Problem

Substring Counting Problem

- **Input:** A string *pattern* and a longer string *text*.
- **Output:** The number of times *pattern* occurs in *text*.

STOP: How many times does ATA occur in CGATATATCCATAG?

Counting Words Problem

Substring Counting Problem

- **Input:** A string *pattern* and a longer string *text*.
- **Output:** The number of times *pattern* occurs in *text*.

STOP: How many times does ATA occur in CGATATATCCATAG?

Answer: It can be 2 or 3. For this application, we will go with 3; that is, we count overlaps.

Substring Indexing

Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)

Substring Indexing

Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)

Substring Indexing

Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
text	C	G	A	T	A	T	A	T	C	C	A	T	A	G

The notation we use for this substring of *text* is:

text[7, 10]

That's weird ... why not *text*[7, 9]?!?

Substring Indexing

Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)

STOP: How would we refer to this substring?

Substring Indexing

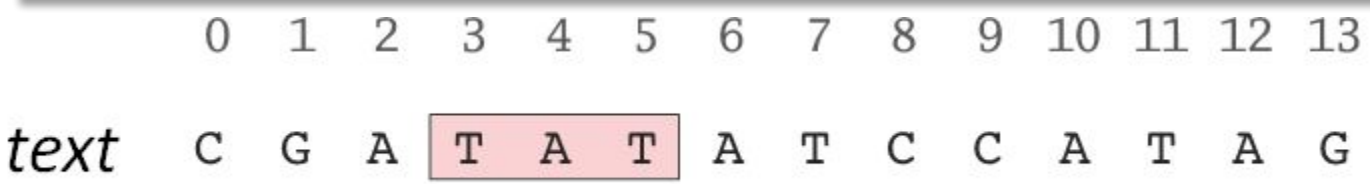
Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)

STOP: How would we refer to this substring?

Answer: *text*[0, 3]

Substring Indexing

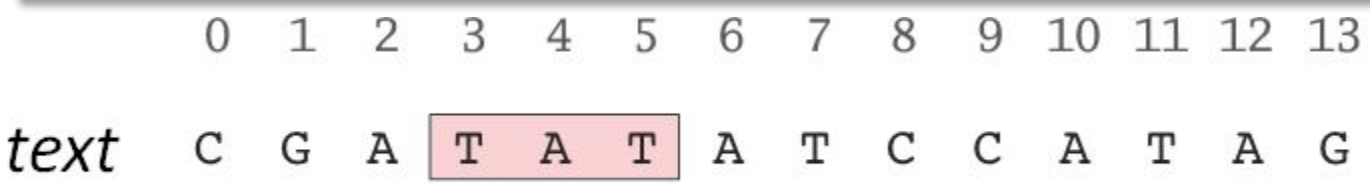
Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)



STOP: What about this substring?

Substring Indexing

Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)

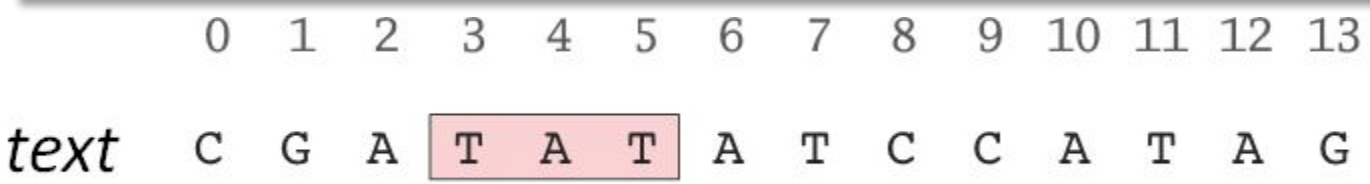


STOP: What about this substring?

Answer: *text*[3, 6]

Substring Indexing

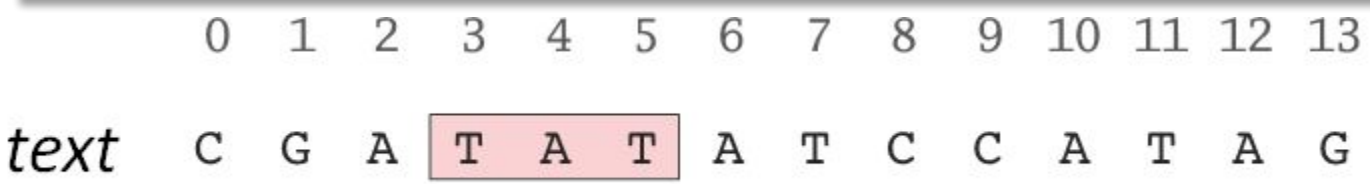
Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)



STOP: What do you notice?

Substring Indexing

Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)

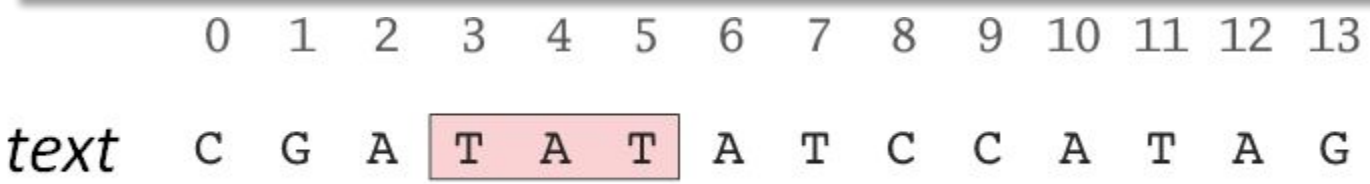


STOP: What do you notice?

Answer: We can easily get the *length* of the substring by subtracting the lower index from the upper index. (Here, 6 - 3 = 3.)

Substring Indexing

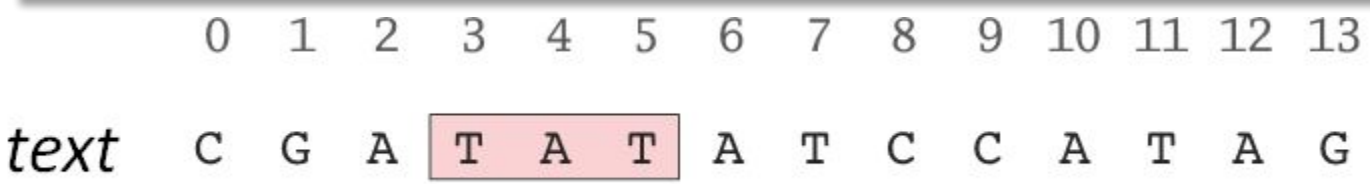
Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)



STOP: How would we refer to the substring of *text* of length *k* starting at position *i*?

Substring Indexing

Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)

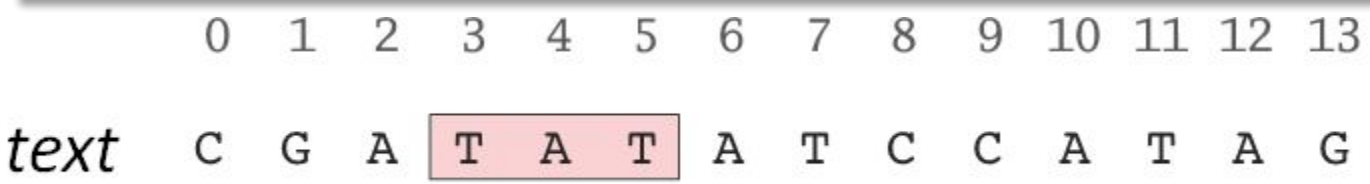


STOP: How would we refer to the substring of *text* of length *k* starting at position *i*?

Answer: *text*[*i*, *i*+*k*]. This will be very useful!

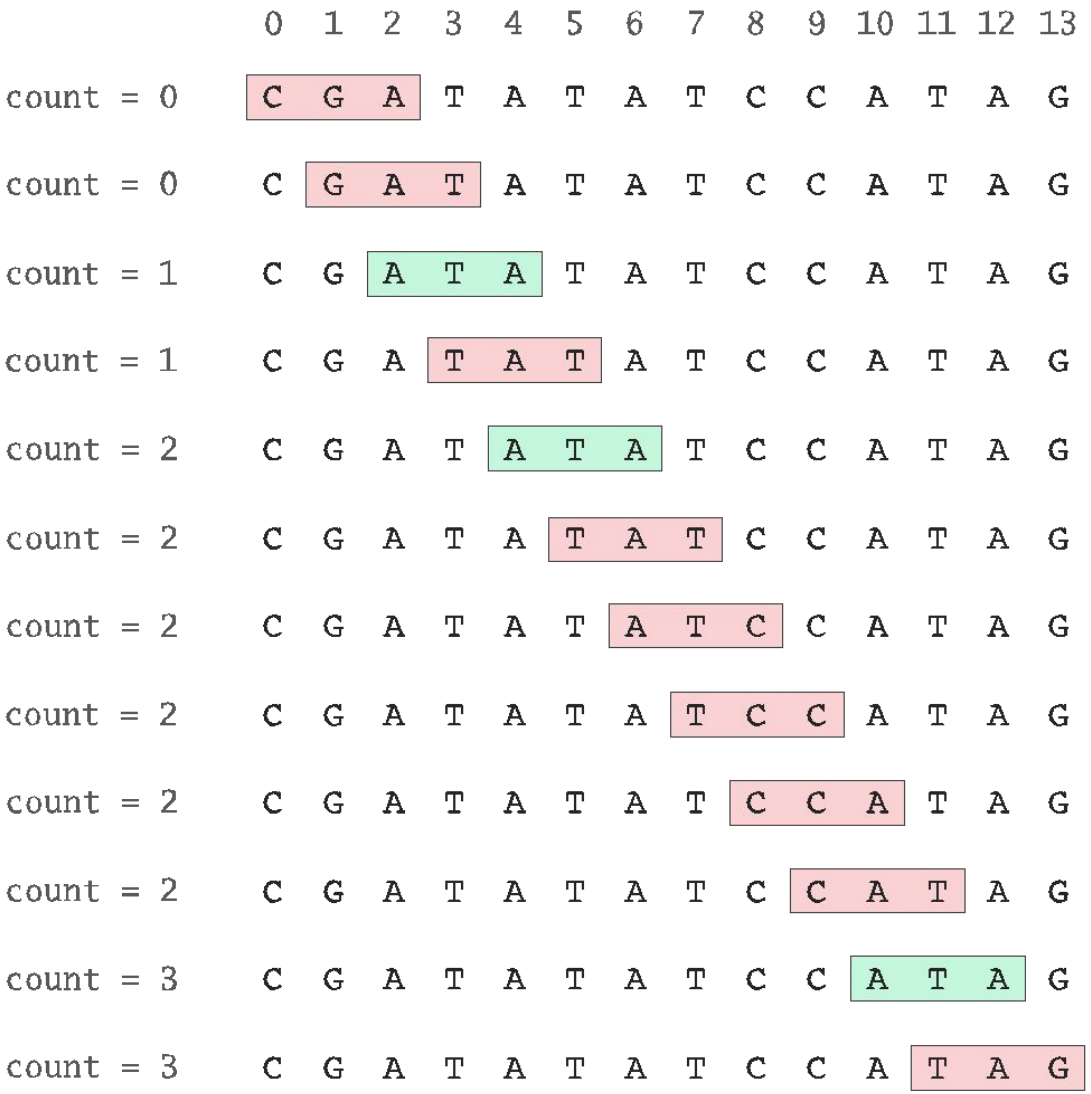
Substring Indexing

Key Point: We think of a string as just an array of symbols. (So it should be 0-indexed.)



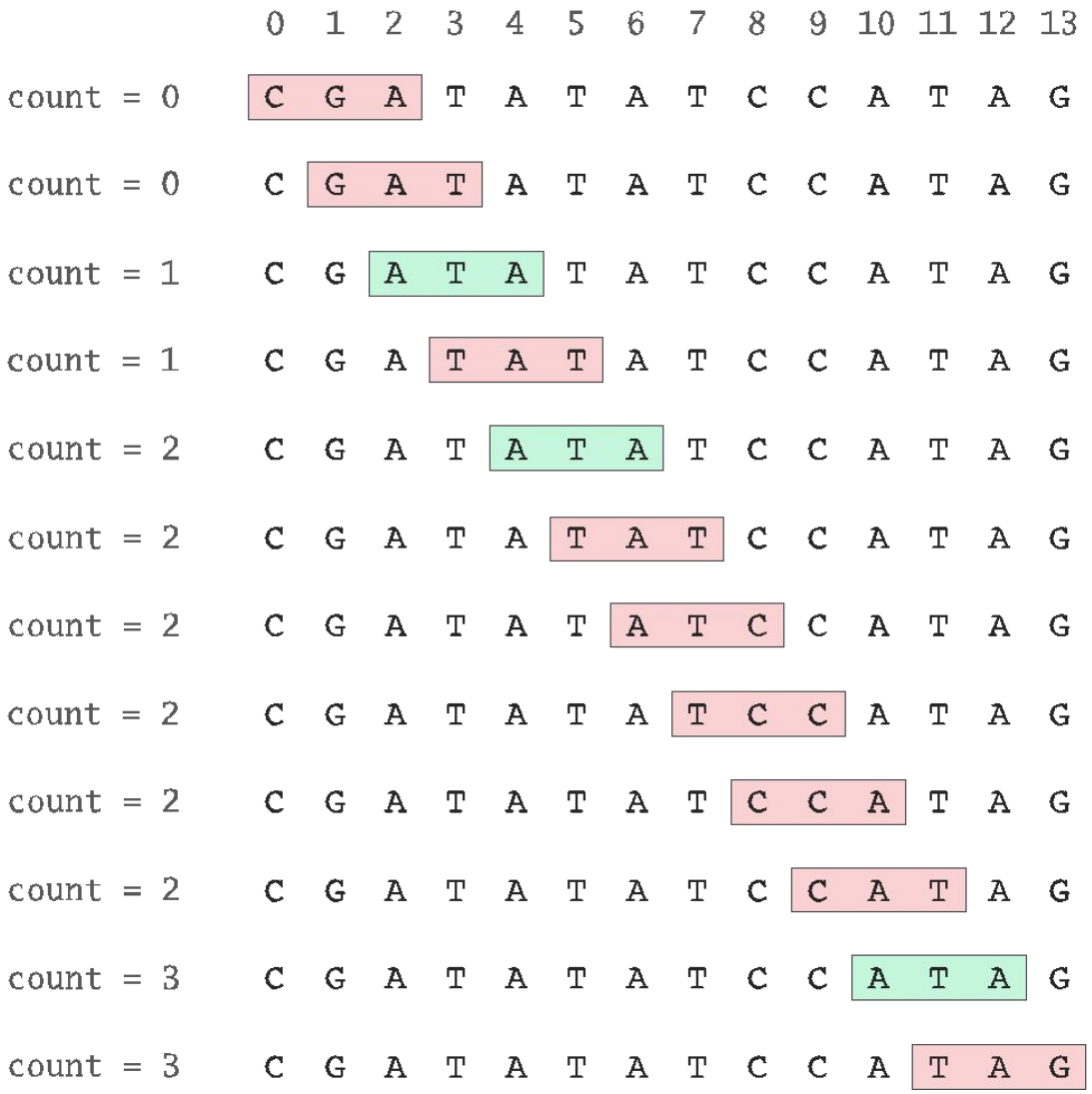
Note: We use the same notation for "subarrays" if we want to refer to a contiguous collection of values in an array.

Our Idea for Counting Patterns



Our Idea for Counting Patterns

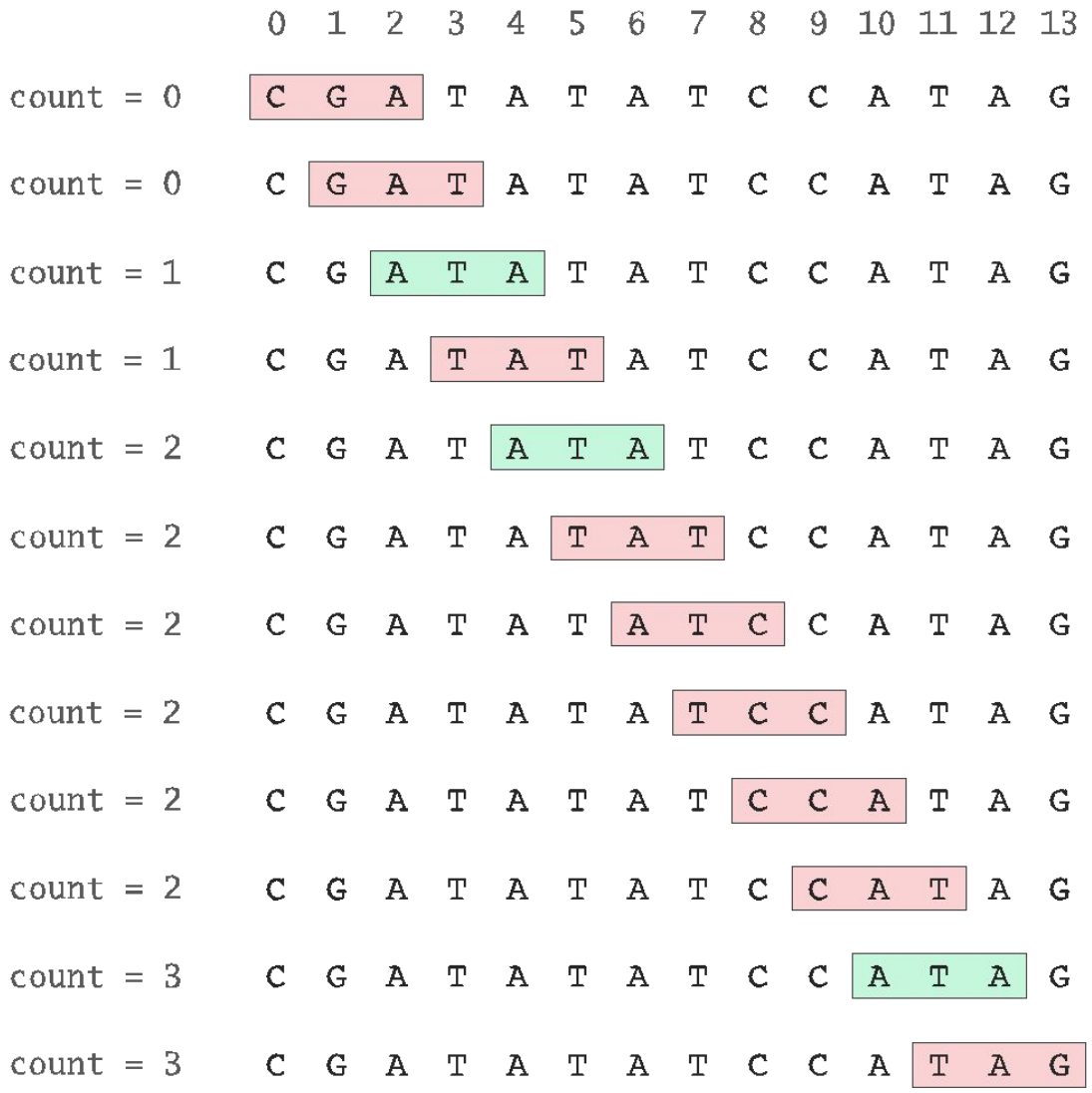
STOP: How many substrings of length *k* are there in a string of length *n*?



Our Idea for Counting Patterns

STOP: How many substrings of length *k* are there in a string of length *n*?

Exercise: Try writing pseudocode to count pattern occurrences.



Pattern Counting

```
PatternCount(pattern, text)
    count ← 0
    k ← len(pattern)
    n ← len(text)
    for every integer i between 0 and n – k
        if text[i, i+k] = pattern
            count ← count + 1
    return count
```

len(): A (typically built-in) function determining the length (number of symbols) in a string; also works for counting elements in an array.

The Frequent Words Problem

***k*-mer:** A string of length k.

The Frequent Words Problem

***k*-mer:** A string of length *k*.

A *k*-mer *pattern* is a **most frequent *k*-mer** in a string if no other *k*-mer is more frequent than *pattern*.

The Frequent Words Problem

***k*-mer:** A string of length *k*.

A *k*-mer *pattern* is a **most frequent *k*-mer** in a string if no other *k*-mer is more frequent than *pattern*.

Frequent Words Problem

- **Input:** A string *text* and an integer *k*.
- **Output:** All **most frequent *k*-mers** in *text*.

The Frequent Words Problem

***k*-mer:** A string of length *k*.

A *k*-mer *pattern* is a **most frequent *k*-mer** in a string if no other *k*-mer is more frequent than *pattern*.

Frequent Words Problem

- **Input:** A string *text* and an integer *k*.
- **Output:** All most frequent *k*-mers in *text*.

STOP: Now is this problem clearly stated?

Solving the Frequent Words Problem

Frequent Words Problem

- **Input:** A string *text* and an integer *k*.
- **Output:** All most frequent *k*-mers in *text*.

```
atcaatgatcaacgtaagcttctaagcatgatcaaggtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagatgatcaagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcatgtt
tccttaaccctctatttttacggaagaatgatcaagctgctgctcttgatcatcgtttc
```

Solving the Frequent Words Problem

Frequent Words Problem

- **Input:** A string *text* and an integer *k*.
- **Output:** All most frequent *k*-mers in *text*.

Example: If *text* = `ACGTTTCACGTTTTACGG` and *k* = 3, then the most frequent words are `ACG` and `TTT` (both occur three times).

Solving the Frequent Words Problem

Frequent Words Problem

- **Input:** A string *text* and an integer *k*.
- **Output:** All most frequent *k*-mers in *text*.

Exercise: How might we solve this problem with an array? What subroutines would you find useful?

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTTTCACGTTTTACGG and *k* = 3.

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTTTCACGTTTTACGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count																

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = **ACG**TTTC**ACG**TTTT**ACG**G and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3															

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTTTCACGTTTTACGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2														

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTTTCACGTTTTACGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2													

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACG**TTT**CACG**TTTT**ACGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3												

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGT**TTC**ACGTTTTACGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3	1											

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTT**TCA**CGTTTTACGG and *k* = 3.

i	0	1	2	3	4	**5**	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3	1	**1**										

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTTTCACGTTTTACGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3	1	1	1									

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = **ACG**TTTC**ACG**TTTT**ACG**G and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3	1	1	1	3								

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTTTCACGTTTTACGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3	1	1	1	3	2							

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTTTCACGTTTTACGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3	1	1	1	3	2	2						

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTTTCACGTTTTACGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3	1	1	1	3	2	2	3					

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACG**TTT**CACG**TTTT**ACGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3	1	1	1	3	2	2	3	3				

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTTTCACGTT**TTA**CGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3	1	1	1	3	2	2	3	3	1			

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTTTCACGTTT**TAC**GG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	**13**	14	15
count	3	2	2	3	1	1	1	3	2	2	3	3	1	**1**		

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = **ACG**TTTC**ACG**TTTT**ACG**G and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3	1	1	1	3	2	2	3	3	1	1	3	1

One Frequent Words Solution

1. Create an array *count* of length len(*text*) - *k* + 1.
2. For each *i*, set *count*[*i*] equal to the number of times *text*[*i*, *i*+*k*] appears in *text*.
3. Take *k*-mers having the maximum values of *count*[*i*].

Example: *text* = ACGTTTCACGTTTTACGG and *k* = 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	3	2	2	3	1	1	1	3	2	2	3	3	1	1	3	1

Solving the Frequent Words Problem

```
FrequentWords(text, k)
    freqPatterns ← an array of strings of length 0
    n ← Len(text)
    count ← array of integers of length n - k + 1
    for every integer i between 0 and n – k
        pattern ← text[i, i+k]
        count[i] ← PatternCount(pattern, text)
    max ← MaxArray(count)
    for every integer i between 0 and n - k
        if count[i] = max
            pattern ← text[i, i+k]
            freqPatterns ← Append(freqPatterns, pattern)     freqPatterns ←
RemoveDuplicates(freqPatterns)
    return freqPatterns
```

PatternCount: our pattern counting function from before

MaxArray: take maximum value in an array *a*

RemoveDuplicates: remove duplicates from list *patterns*

Solving the Frequent Words Problem

```
FrequentWords(text, k)
    freqPatterns □ an array of strings of length 0
    n □ Len(text)
    count □ array of integers of length n - k + 1
    for every integer i between 0 and n – k
        pattern □ text[i, i+k]
        count[i] □ PatternCount(pattern, text)
    max □ MaxArray(count)
    for every integer i between 0 and n - k
        if count[i] = max
            pattern □ text[i, i+k]
            freqPatterns □ Append(freqPatterns, pattern)     freqPatterns □
RemoveDuplicates(freqPatterns)
    return freqPatterns
```

STOP: This algorithm is *inefficient*; why? How could we make it better?

Outline

- An Intro to DNA Replication
- Hidden Messages in the Replication Origin
- Hunting for Frequent Words
- **A Faster Frequent Words Approach**
- Some Hidden Messages are More Surprising than Others
- An Explosion of Hidden Messages
- Replication Asymmetry Leads Us to the Replication Origin

Arrays/Slices Store Lists of Variables

H	i		T	h	e	r	e	!
0	1	2	3	4	5	6	7	8

1	1	2	3	5	8	13	21	34	55	89
0	1	2	3	4	5	6	7	8	9	10

"ACG"	"TTA"	"GAG"	"CCT"	"TAA"	"GGG"	"CAT"
0	1	2	3	4	5	6

What if the Indices Aren't Integers?

Would make things easier when finding frequent words ...

Pattern	count
"AA"	17
"AC"	4
"CG"	15
"GA"	23
"GG"	3
"GT"	30
"TA"	18
"TG"	2
"TT"	24

What if the Indices Aren't Integers?

Map/Dictionary: An association of **keys** with **values** .

Pattern	count
"AA"	17
"AC"	4
"CG"	15
"GA"	23
"GG"	3
"GT"	30
"TA"	18
"TG"	2
"TT"	24

What if the Indices Aren't Integers?

Map/Dictionary: An association of **keys** with **values** .

We use a variable (say, *freq*) to refer to the map.

Pattern	count
"AA"	17
"AC"	4
"CG"	15
"GA"	23
"GG"	3
"GT"	30
"TA"	18
"TG"	2
"TT"	24

What if the Indices Aren't Integers?

Map/Dictionary: An association of **keys** with **values** .

We use a variable (say, *freq*) to refer to the map.

Value access is like arrays: *freq*["GT"]

Pattern	count
"AA"	17
"AC"	4
"CG"	15
"GA"	23
"GG"	3
"GT"	30
"TA"	18
"TG"	2
"TT"	24

Note that not every 2-mer is a key…

Map/Dictionary: An association of **keys** with **values** .

We use a variable (say, *freq*) to refer to the map.

Value access is like arrays: *freq*["GT"]

Pattern	count
"AA"	17
"AC"	4
"CG"	15
"GA"	23
"GG"	3
"GT"	30
"TA"	18
"TG"	2
"TT"	24

Rewriting Frequent Words Pseudocode

```
BetterFrequentWords(text, k)
    freqPatterns ← an empty array
    freqMap ← empty map
    n ← Len(text)
    for every integer i between 0 and n - k
        pattern ← text[i, i+k]
        if freqMap[pattern] doesn't exist
            freqMap[pattern] = 1
        else
            freqMap[pattern] ← freqMap[pattern] + 1
    maxCount ← MaxMap(freqMap)
    for all strings pattern in freqMap
        if freqMap[pattern] = maxCount
            freqPatterns ← Append(freqPatterns, pattern)
    return freqPatterns
```

Note: We don't need **RemoveDuplicates()** or **Count()** !
And this is *much* faster!

Shortening BetterFrequentWords()

```
BetterFrequentWords(text, k)
    freqPatterns ← an array of strings of length 0
    freqMap ← an empty map
    n ← Len(text)
    for every integer i between 0 and n - k
        pattern ← text[i, i+k]
        if freqMap[pattern] doesn't exist
            freqMap[pattern] = 1
        else
            freqMap[pattern] ← freqMap[pattern] + 1
    max ← MaxMap(freqMap)
    for all strings pattern in freqMap
        if freqMap[pattern] = max
            freqPatterns ← Append(freqPatterns, pattern)
    return freqPatterns
```

Subroutine time! We can shorten the code in **red**.

Shortening BetterFrequentWords()

```
BetterFrequentWords(text, k)
    freqPatterns ← an empty array of strings
    freqMap ← FrequencyMap(text, k)
    maxCount ← MaxValue(freqMap)
    for all strings pattern in freqMap
        if freqMap[pattern] = maxCount
            append pattern to freqPatterns
    return freqPatterns
```

```
FrequencyMap(text, k)
    freqMap ← an empty map
    n ← Len(text)
    for every integer i between 0 and n - k
        pattern ← text[i, i+k]
        if freqMap[pattern] doesn't exist
            freqMap[pattern] = 1
        else
            freqMap[pattern] ← freqMap[pattern] + 1
    return freqMap
```

Outline

- An Intro to DNA Replication
- Hidden Messages in the Replication Origin
- Hunting for Frequent Words
- A Faster Frequent Words Approach
- **Some Hidden Messages are More Surprising than Others**
- An Explosion of Hidden Messages
- Replication Asymmetry Leads Us to the Replication Origin

Returning to *ori* of *Vibrio cholerae*

```
atcaatgatcaacgtaagcttctaagcatgatcaaggtgctcacacagtttatccacaacctgagtgga
tgacatcaagataggtcgttgtatctccttcctctcgtactctcatgaccacggaaagatgatcaagag
aggatgatttcttggccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagcgcc
atattgcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgttctgtttatct
tgttttgactgagacttgttaggatagacggtttttcatcactgactagccaaagccttactctgcctg
acatcgaccgtaaattgataatgaatttacatgcttccgcgacgatttacctcttgatcatcgatccga
ttgaagatcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcatgtttcc
ttaaccctctatttttacggaagaatgatcaagctgctgctcttgatcatcgtttc
```

k	3	4	5	6	7	8	9
count	25	12	8	8	5	4	3
***k*-mers**	t ga	at ga	gat ca t gat c	t gat ca	at gat ca	at gat caa	at gat caag ct t gat cat t ct t gat ca ct ct t gat c

Returning to *ori* of *Vibrio cholerae*

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaacctgagtgga
tgacatcaagataggtcgttgtatctccttcctctcgtactctcatgaccacggaaag**ATGATCAAG**ag
aggatgatttcttggccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagcgcc
atattgcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgttctgtttatct
tgttttgactgagacttgttaggatagacggtttttcatcactgactagccaaagccttactctgcctg
acatcgaccgtaaattgataatgaatttacatgcttccgcgacgatttac**CTCTTGATCAT**cgatccga
ttgaagatcttcaattgttaattctcttgcctcgactcatagccatgatgag**CTCTTGATCAT**gtttcc
ttaaccctctatttttacggaaga**ATGATCAAG**ctgctg**CTCTTGATCAT**cgtttc

Most frequent 9-mers in this *ori* (all appear 3 times):
ATGATCAAG, **CTTGATCAT**, **T**CTTGGATCA,
CTCTTGATC

Returning to *ori* of *Vibrio cholerae*

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaacctgagtgga
tgacatcaagataggtcgttgtatctccttcctctcgtactctcatgaccacggaaag**ATGATCAAG**ag
aggatgatttcttggccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagcgcc
atattgcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgttctgtttatct
tgttttgactgagacttgttaggatagacggtttttcatcactgactagccaaagccttactctgcctg
acatcgaccgtaaattgataatgaatttacatgcttccgcgacgattta**CTCTTGATCAT**cgatccga
ttgaagatcttcaattgttaattctcttgcctcgactcatagccatgatga**CTCTTGATCAT**gtttcc
ttaaccctctatttttacggaaga**ATGATCAAG**ctgctg**CTCTTGATCAT**cgtttc

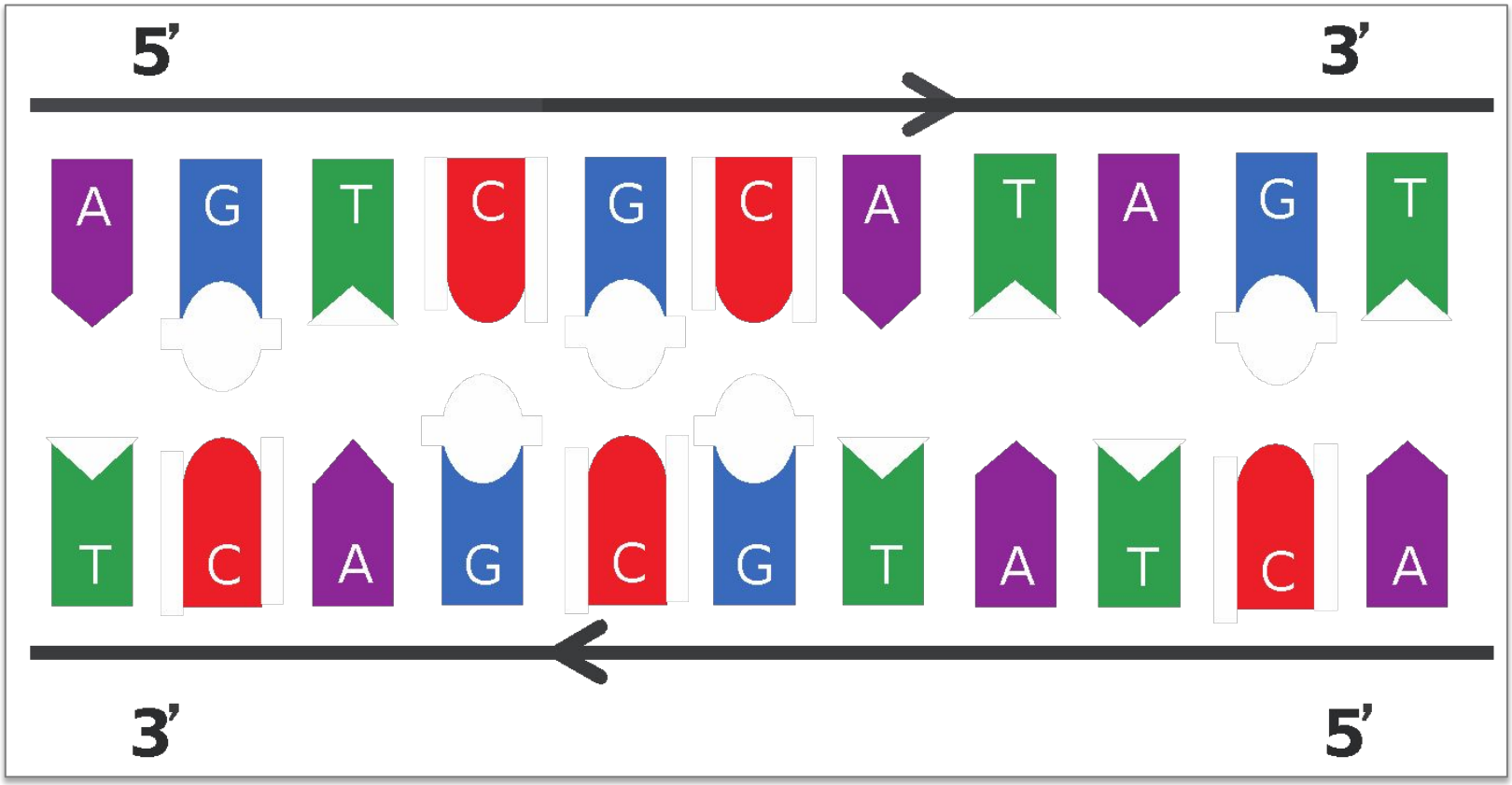
Most frequent 9-mers in this *ori* (all appear 3 times):

ATGATCAAG, **CTTGATCAT**, **T**CTTGGATCA, **CT**CTTGATC

STOP: Now what do you see?

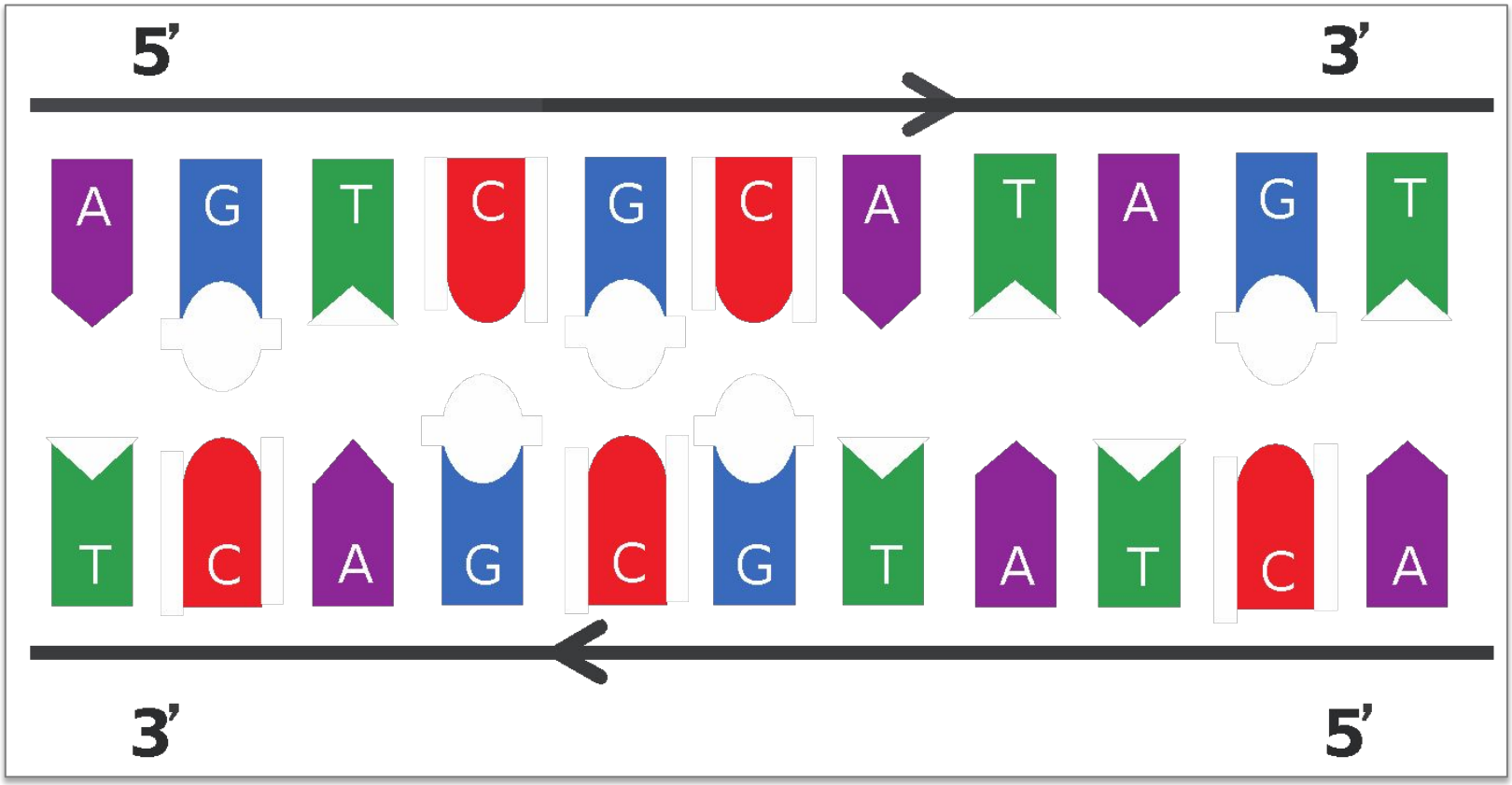
Complementarity of DNA

DNA is double-stranded, and the two strands are **reverse complements** of each other.



Complementarity of DNA

The reverse complement of `AGTCGCATAGT` is `ACTATGCGACT`.



Reverse Complement Problem

Reverse Complement Problem

- **Input:** A DNA string *text*.
- **Output:** The reverse complement of *text*.

Reverse Complement Problem

Reverse Complement Problem

- **Input:** A DNA string *text*.
- **Output:** The reverse complement of *text*.

STOP: Try to write the shortest possible pseudocode function solving this problem.

Reverse Complement Problem

Reverse Complement Problem

- **Input:** A DNA string *text*.
- **Output:** The reverse complement of *text*.

STOP: Try to write the shortest possible pseudocode function solving this problem.

```
ReverseComplement(text)
    x ← Reverse(text)
    y ← Complement(x)
    return y
```

Or in One Line …

Reverse Complement Problem

- **Input:** A DNA string *text*.
- **Output:** The reverse complement of *text*.

STOP: Try to write the shortest possible pseudocode function solving this problem.

```
ReverseComplement(text)
    return Reverse(Complement(text))
```

Or in One Line …

Reverse Complement Problem

- **Input:** A DNA string *text*.
- **Output:** The reverse complement of *text*.

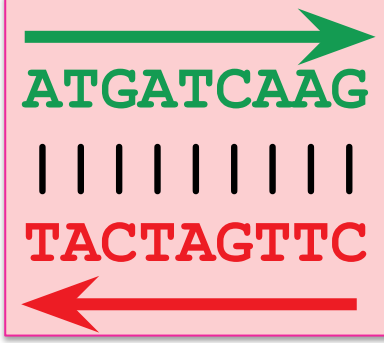
STOP: Try to write the shortest possible pseudocode function solving this problem.

```
ReverseComplement(text)
    return Reverse(Complement(text))
```

We split our work into two easy pieces! More later…

Hidden Message Found!

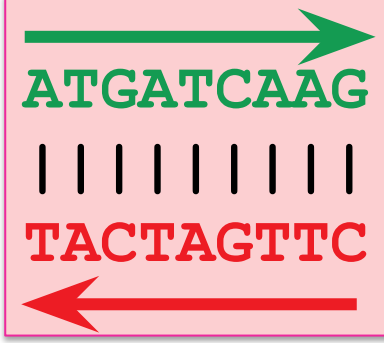
atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaacctgagtgga
tgacatcaagataggtcgttgtatctccttcctctcgtactctcatgaccacggaaag**ATGATCAAG**ag
aggatgatttcttggccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagcgcc
atattgcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgttctgtttatct
tgttttgactgagacttgttaggatagacggtttttcatcactgactagccaaagccttactctgcctg
acatcgaccgtaaattgataatgaatttacatgcttccgcgacgatttacct**CTTGATCAT**cgatccga
ttgaagatcttcaattgttaattctcttgcctcgactcatagccatgatgagct**CTTGATCAT**gtttcc
ttaaccctctatttttacggaaga**ATGATCAAG**ctgctgct**CTTGATCAT**cgtttc



are *reverse complements* and likely *DnaA* boxes (*DnaA* does not know which strand it binds to).

Hidden Message Found!

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaacctgagtgga
tgacatcaagataggtcgttgtatctccttcctctcgtactctcatgaccacggaaag**ATGATCAAG**ag
aggatgatttcttggccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagcgcc
atattgcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgttctgtttatct
tgttttgactgagacttgttaggatagacggtttttcatcactgactagccaaagccttactctgcctg
acatcgaccgtaaattgataatgaatttacatgcttccgcgacgatttacc**CTTGATCAT**cgatccga
ttgaagatcttcaattgttaattctcttgcctcgactcatagccatgatgagc**CTTGATCAT**gtttcc
ttaaccctctatttttacggaaga**ATGATCAAG**ctgctgct**CTTGATCAT**cgtttc



are *reverse complements* and likely *DnaA* boxes (*DnaA* does not know which strand it binds to).

It is **VERY SURPRISING** to find a 9-mer appearing **6 or more** times (with reverse complements) within ≈ 500 nucleotides.

Homework problems

1. What is the probability of finding a specific 9-mer 6-times in a random 500 nucleotide sequence?

2. What is the probability of finding a specific k-mer m-times in an n-mer nucleotide sequence?

Looking for other Hidden Messages?

STOP: Now that we know the "hidden message" in *Vibrio cholerae*, how would we look for a hidden message starting replication in *other* bacteria?

Hidden Messages in *T. petrophila*?

```
aactctatacctccttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaactaaa
atggtaggtttggtggtaggttttgtgtacattttgtagtatctgatttttaattacataccgtata
ttgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaacctaccaccaaactct
gtattgaccattttaggacaacttcagggtggtaggtttctgaagctctcatcaatagactatttta
gtctttacaaacaatattaccgttcagattcaagattctacaacgctgttttaatgggcgttgcaga
aaacttaccacctaaaatccagtatccaagccgatttcagagaaacctaccacttacctaccactta
cctaccacccgggtggtaagttgcagacattattaaaaacctcatcagaagcttgttcaaaaatttc
aatactcgaaacctaccacctgcgtcccctattatttactactactaataatagcagtataattgat
ctgaaaagaggtggtaaaaaa
```

Not one occurrence of **ATGATCAAG** or **CTTGATCAT**!

Hidden Messages in *T. petrophila*?

```
aactctatacctccttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaactaaa
atggtaggtttggtggtaggttttgtgtacattttgtagtatctgatttttaattacataccgtata
ttgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaacctaccaccaaactct
gtattgaccattttaggacaacttcagggtggtaggtttctgaagctctcatcaatagactatttta
gtctttacaaacaatattaccgttcagattcaagattctacaacgctgttttaatgggcgttgcaga
aaacttaccacctaaaatccagtatccaagccgatttcagagaaacctaccacttacctaccactta
cctaccacccgggtggtaagttgcagacattattaaaaacctcatcagaagcttgttcaaaaatttc
aatactcgaaacctaccacctgcgtcccctattatttactactactaataatagcagtataattgat
ctgaaaagaggtggtaaaaaa
```

Not one occurrence of **ATGATCAAG** or **CTTGATCAT**!

Applying Frequent Words Problem to this *ori*:

AACCTACCA, ACCTACCAC, GGTAGGTTT
TGGTAGGTT, AAACCTACC, CCTACCACC

Hidden Messages in *T. petrophila*?

```
aactctatacctccttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaactaaa
atggtaggtttggtggtaggttttgtgtacattttgtagtatctgatttttaattacataccgtata
ttgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaacctaccaccaaactct
gtattgaccattttaggacaacttcagggtggtaggtttctgaagctctcatcaatagactatttta
gtctttacaaacaatattaccgttcagattcaagattctacaacgctgttttaatgggcgttgcaga
aaacttaccacctaaaatccagtatccaagccgatttcagagaaacctaccacttacctaccactta
cctaccacccgggtggtaagttgcagacattattaaaaacctcatcagaagcttgttcaaaaatttc
aatactcgaaacctaccacctgcgtcccctattatttactactactaataatagcagtataattgat
ctgaaaagaggtggtaaaaaa
```

Different genomes □ different hidden messages

Applying Frequent Words Problem to this *ori*:

AACCTACCA, ACCTACCAC, GGTAGGTTT

TGGTAGGTT, AAACCTACC, CCTACCACC

Hidden Messages in *Thermotoga petrophila*

aactctatacctccttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaactaaa
atggtaggtttGGTGGTAGGttttgtgtacattttgtagtatctgatttttaattacataccgtata
ttgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaaCCTACCACCaaactct
gtattgaccattttaggacaacttcagGGTGGTAGGtttctgaagctctcatcaatagactatttta
gtctttacaaacaatattaccgttcagattcaagattctacaacgctgttttaatgggcgttgcaga
aaacttaccacctaaaatccagtatccaagccgatttcagagaaacctaccacttacctaccactta
CCTACCACCcgggtggtaagttgcagacattattaaaaacctcatcagaagcttgttcaaaaatttc
aatactcgaaaCCTACCACCtgcgtcccctattatttactactactaataatagcagtataattgat
ctgaaaagaggtggtaaaaaa

CCTACCACC
| | | | | | | | |
GGATGGTGG

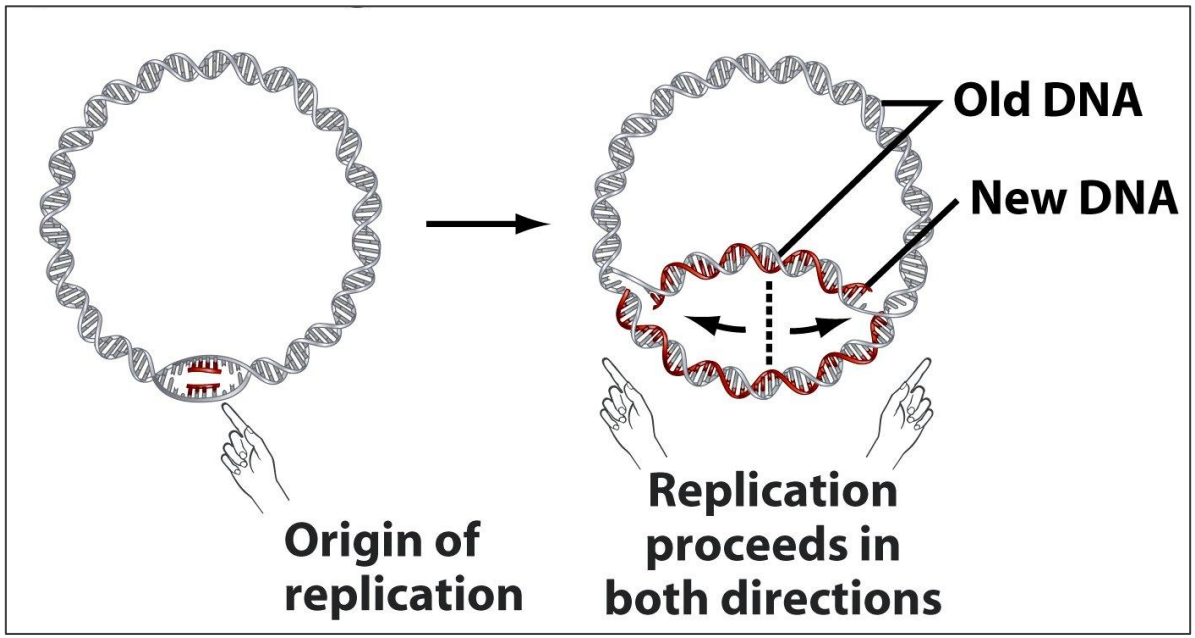
are candidate hidden messages.

Outline

- An Intro to DNA Replication
- Hidden Messages in the Replication Origin
- Hunting for Frequent Words
- A Faster Frequent Words Approach
- Some Hidden Messages are More Surprising than Others
- **An Explosion of Hidden Messages**
- Replication Asymmetry Leads Us to the Replication Origin

Returning to "Problem 1"

We have found hidden messages if *ori* is given. But we still don't know how to find *ori* in a (long) genome.



Bacteria with Unknown *ori*

STOP: Now that we know that "hidden messages" may differ, how could we look for *ori* in a newly sequenced bacterial genome?

Finding *ori* Computationally

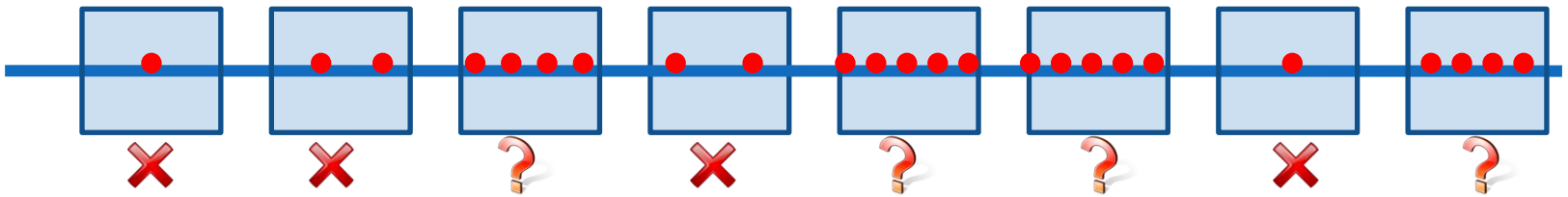
OLD strategy: given a previously **known** *ori* (500 nucleotide window), find **frequent words** (clumps) in *ori* as candidate *DnaA* boxes.

replication origin → **frequent words**

Finding *ori* Computationally

OLD strategy: given a previously **known** *ori* (500 nucleotide window), find **frequent words** (clumps) in *ori* as candidate *DnaA* boxes.

replication origin → **frequent words**

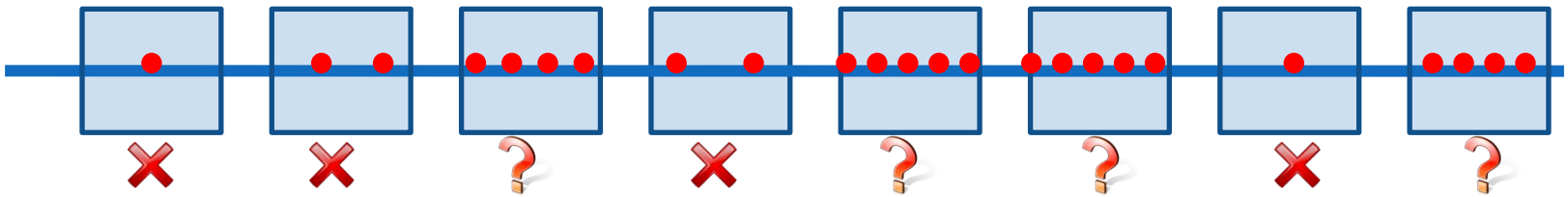


NEW strategy: find frequent words in **ALL** windows within a (3 million nucleotide) genome. Windows with **clumps** of frequent words are candidate replication origins.

frequent words → **replication origin**

Finding *ori* Computationally

Exercise: Define a computational problem modeling our new strategy.



NEW strategy: find frequent words in **ALL** windows within a (3 million nucleotide) genome. Windows with **clumps** of frequent words are candidate replication origins.

frequent words → **replication origin**

Defining and Hunting for "Clumps"

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.

Defining and Hunting for "Clumps"

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

- **Input:** A string *Genome* 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 *Genome*.

Defining and Hunting for "Clumps"

```
FindClumps(text, k, L, t,H)
    patterns  an array of strings of length 0
    n  Len(text)
    for every integer i between 0 and n – L
        window  text[i, i + L]
        freqMap  FrequencyMap(window, k)
        for every string pattern in freqMap
            if freqMap[pattern] >= t
                patterns  Append(patterns, pattern)
    patterns  RemoveDuplicates(patterns)
    return patterns
```

Note: A complicated function can be made easier by using subroutines as building blocks.

Defining and Hunting for "Clumps"

```
FindClumps(text, k, L, t)
    patterns ← an array of strings of length 0
    n ← Len(text)
    for every integer i between 0 and n – L
        window ← text[i, i + L]
        freqMap ← FrequencyMap(window, k)
        for every string pattern in freqMap
            if freqMap[pattern] >= t
                patterns ← Append(patterns, pattern)
    patterns ← RemoveDuplicates(patterns)
    return patterns
```

STOP : Why is looking for clumps in bacterial genomes as a source of hidden messages destined to fail?

What's the Issue?

Genomes have *many* **repeats** , some more useful than others. Alu in humans is ~300 bp long and occurs (with some changes) 1 million times.

Nonrandom Distribution of Alu Elements in Genes of Various Functional Categories: Insight from Analysis of Human Chromosomes 21 and 22

Deepak Grover ✉, Partha P. Majumder, Chandrika B. Rao, Samir K. Brahmachari, Mitali Mukerji

Molecular Biology and Evolution, Volume 20, Issue 9, September 2003, Pages 1420–1424,
https://doi.org/10.1093/molbev/msg153
Published: 01 September 2003 **Article history** ▾

What's the Issue?

Genomes have *many* **repeats** , some more useful than others. Alu in humans is ~300 bp long and occurs (with some changes) 1 million times.

In *E. coli*, **1900+** *different* 9-mers form (500,3)-clumps. It is unclear which ones point to *ori*...

We Are Back Where We Started...

Let's hack out this DNA fragment. Can the genome replicate without it?

I need more information before I can hack this problem.

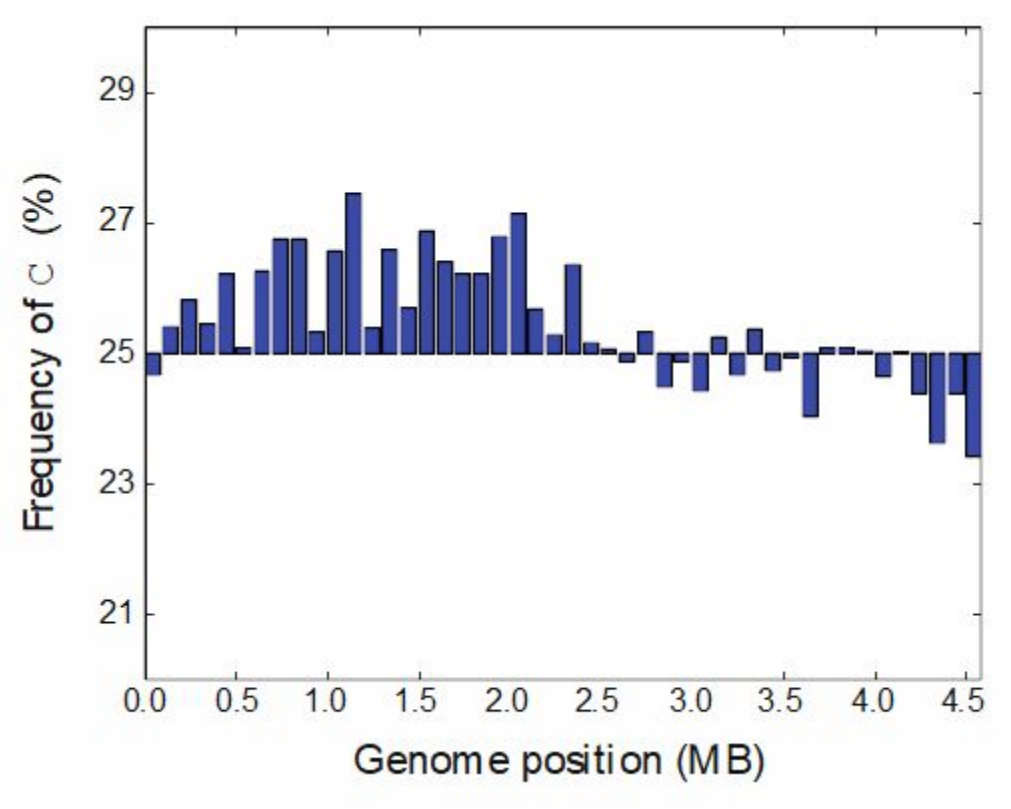
Outline

- An Intro to DNA Replication
- Hidden Messages in the Replication Origin
- Hunting for Frequent Words
- A Faster Frequent Words Approach
- Some Hidden Messages are More Surprising than Others
- An Explosion of Hidden Messages
- **Replication Asymmetry Leads Us to the Replication Origin**

A Surprising Pattern in Nucleotide Counts

Let's run a very simple computational analysis: take frequency of each nucleotide in 100,000 nucleotide windows of *E. coli* (verified *ori*).

Why would there be more C on half the genome?

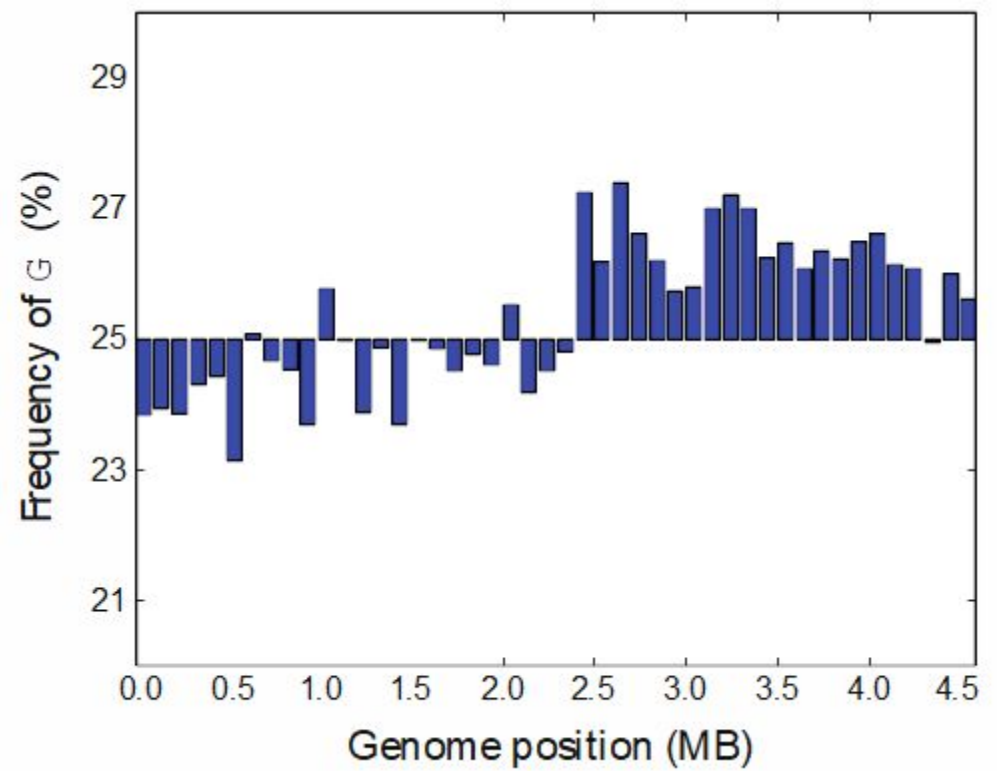


ori ***ter***

A Surprising Pattern in Nucleotide Counts

Let's run a very simple computational analysis: take frequency of each nucleotide in 100,000 nucleotide windows of *E. coli* (verified *ori*).

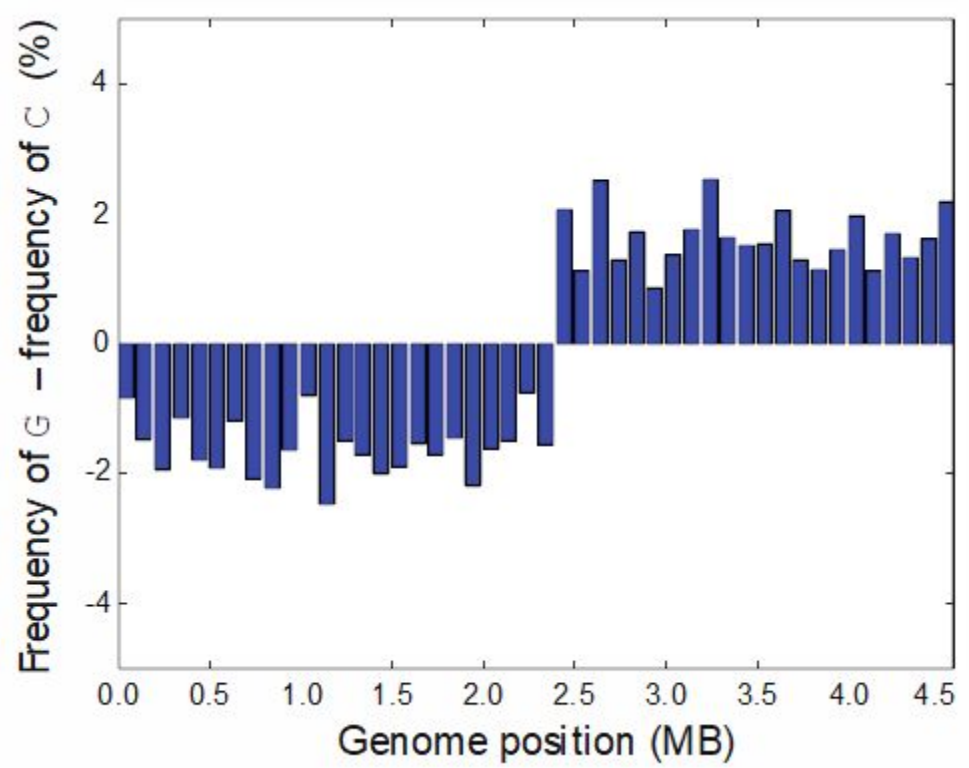
And why would the story be opposite when we count G's?



ori ***ter***

Taking Difference in G – C

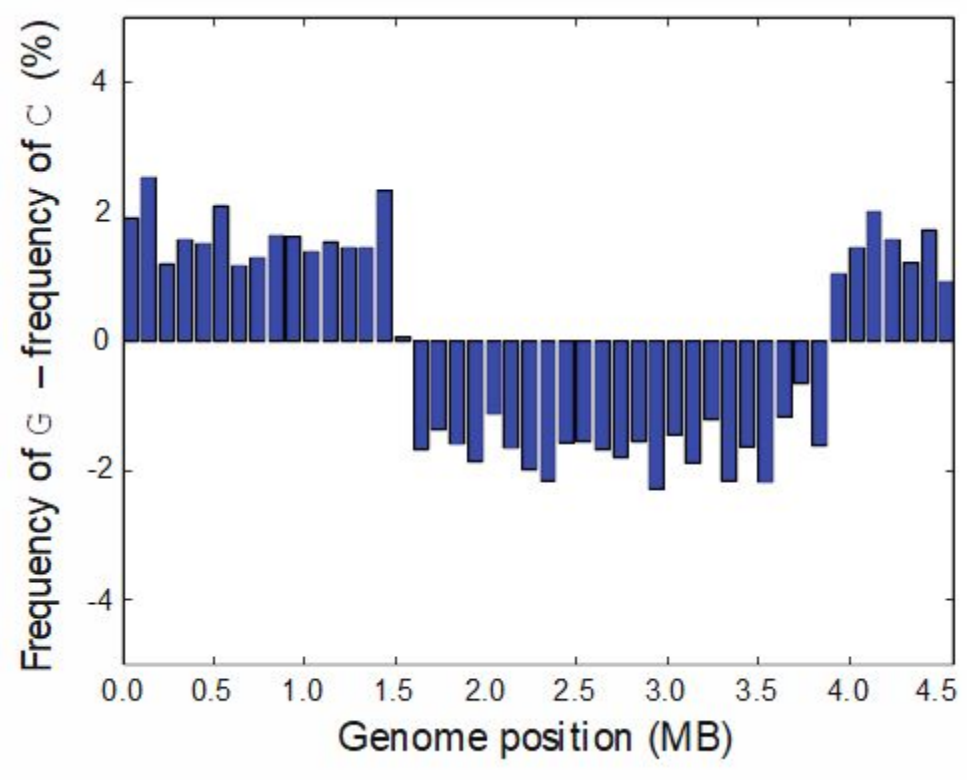
The pattern is even more stark if we take the *difference* between the frequency of G and the frequency of C ...



ori ***ter***

Taking Difference in G – C

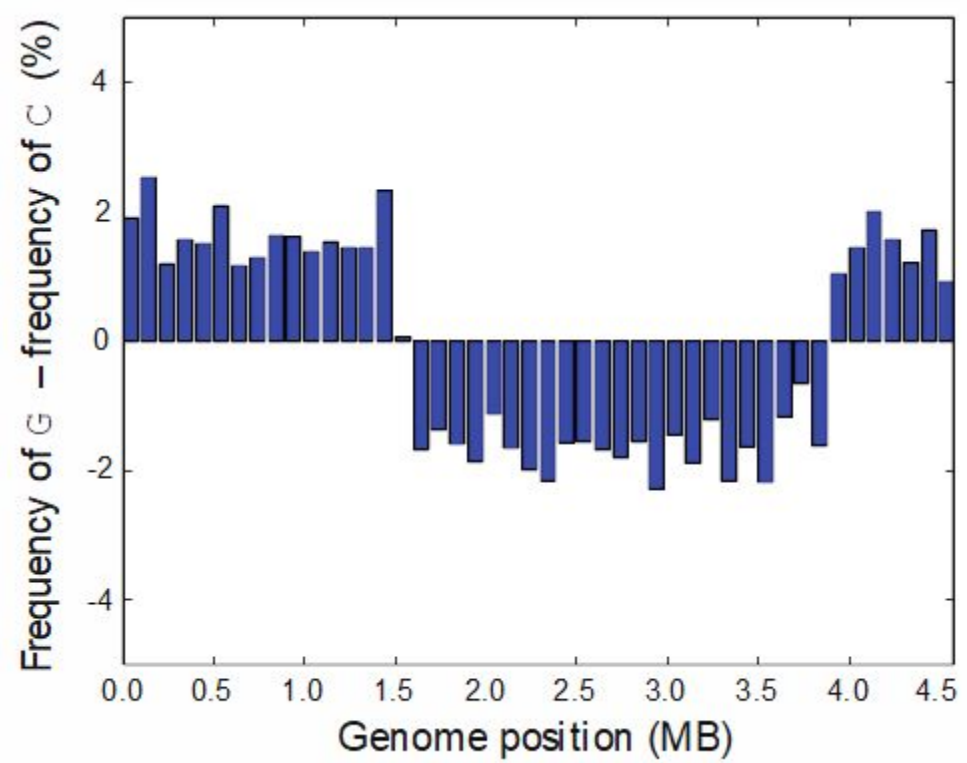
And the pattern is still there even if we didn't know where *ori* was and start counting at some arbitrary spot.



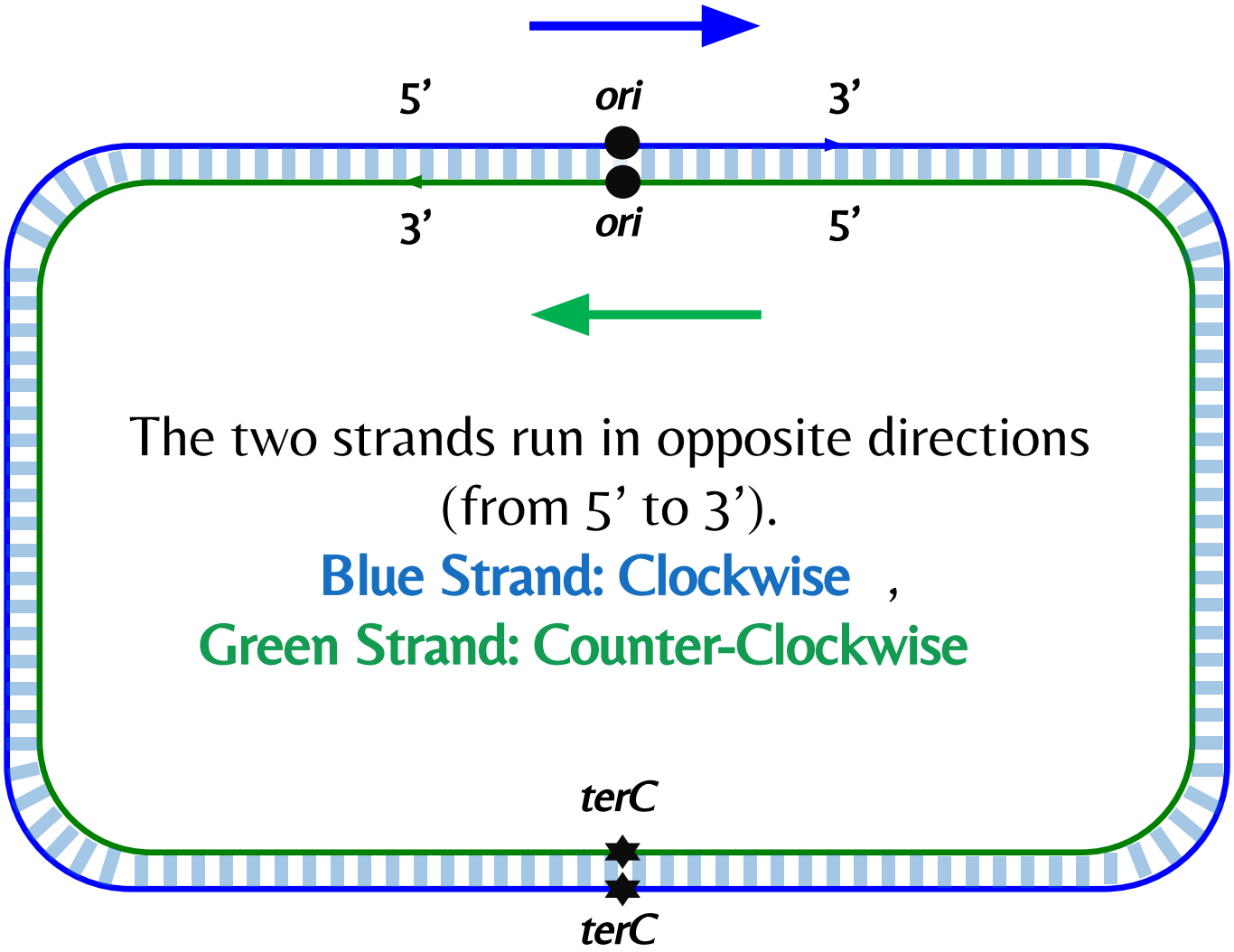
Taking Difference in G – C

And the pattern is still there even if we didn't know where *ori* was and start counting at some arbitrary spot.

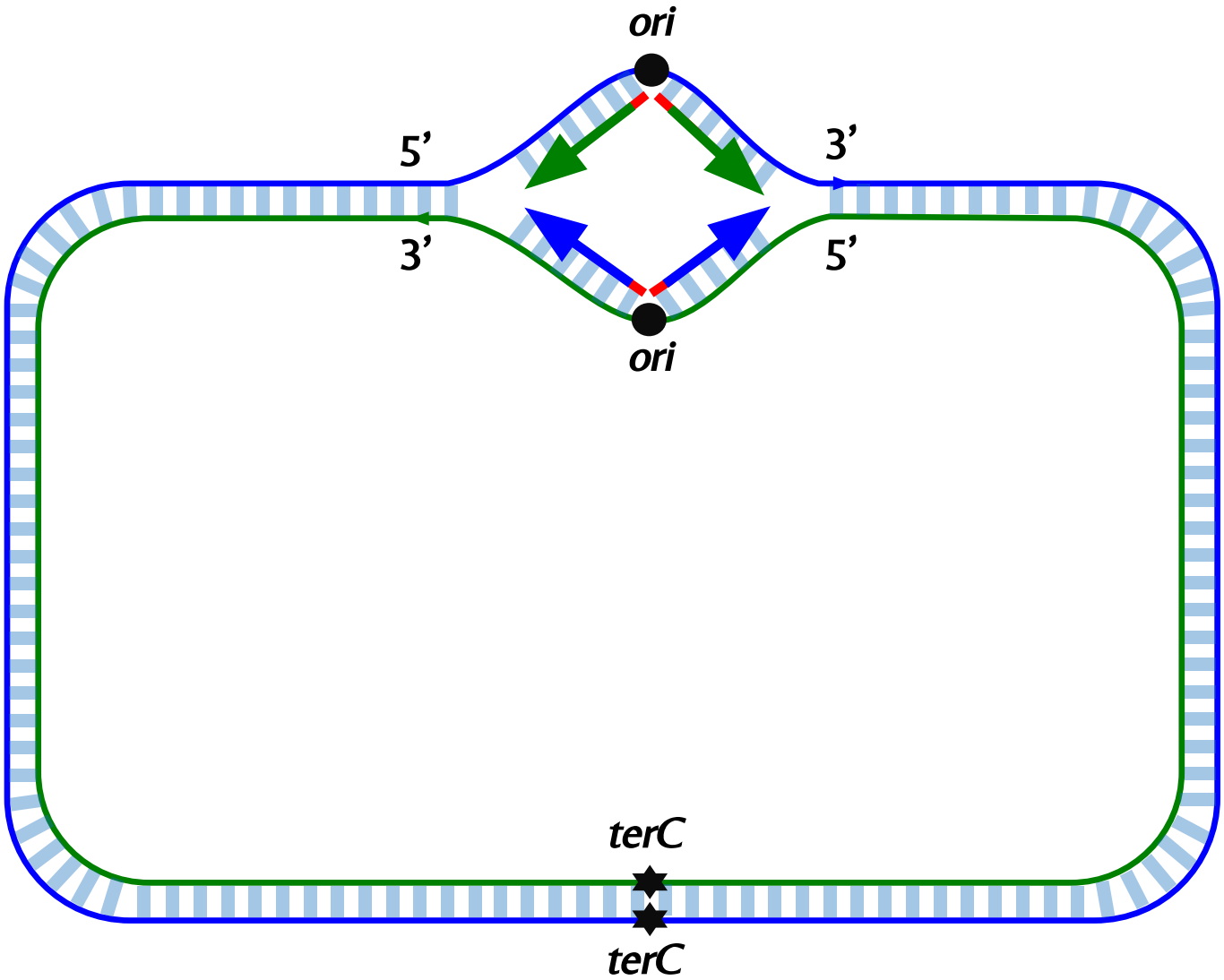
Let's learn more about replication in the hope of finding an answer...



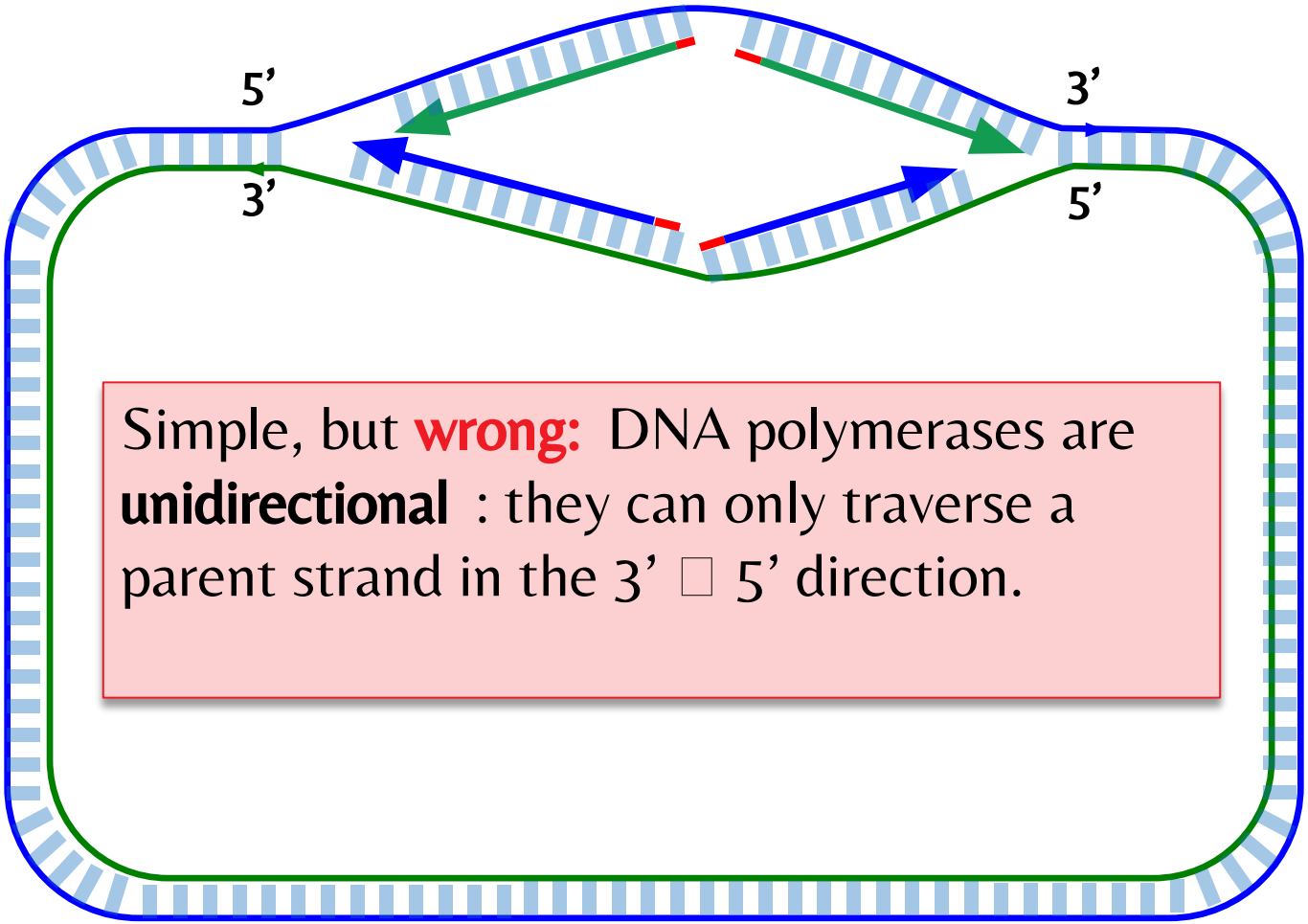
DNA Strands Have Directions



Four DNA Polymerases Can Do the Job

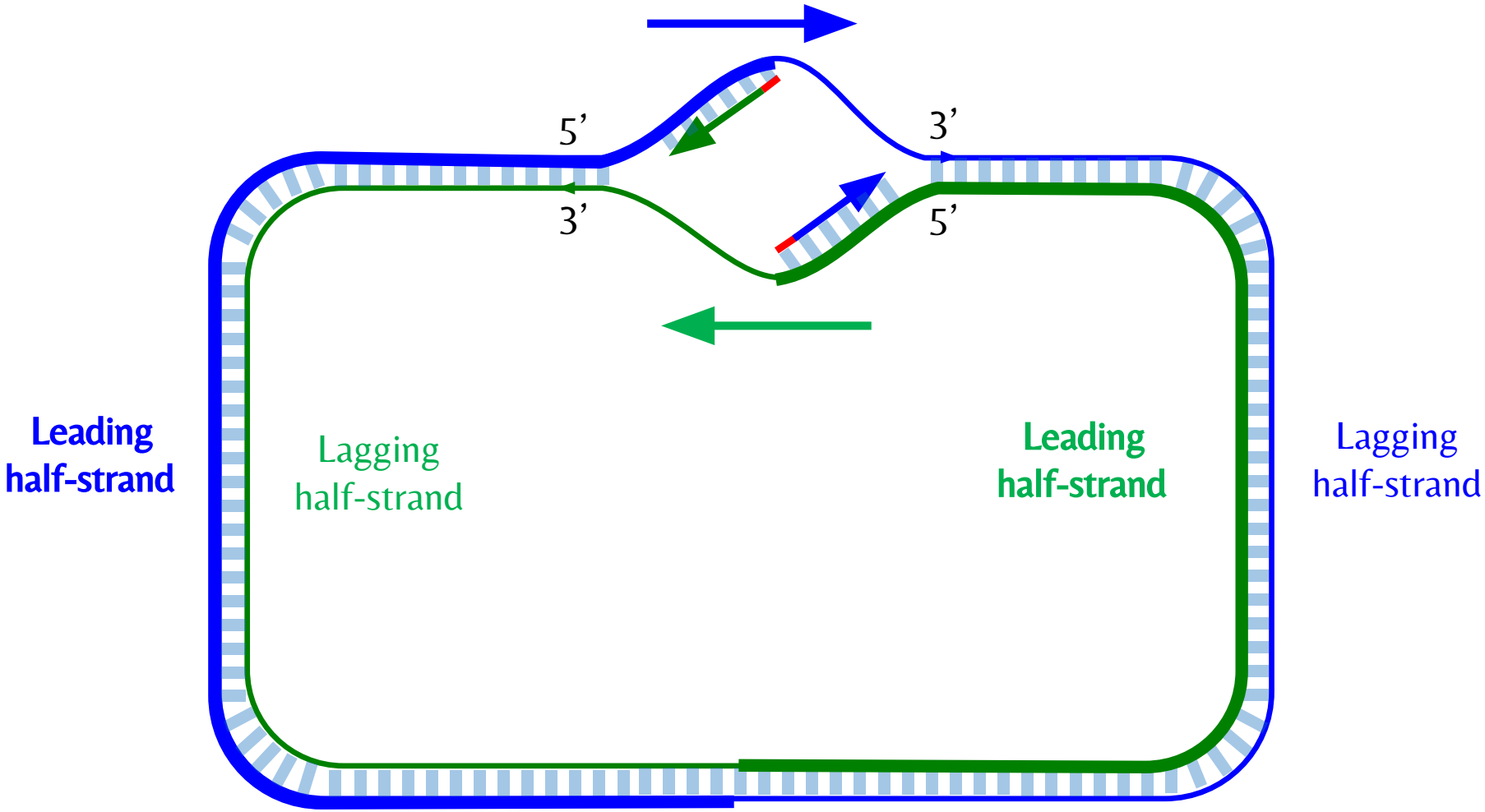


Continue as Replication Fork Enlarges



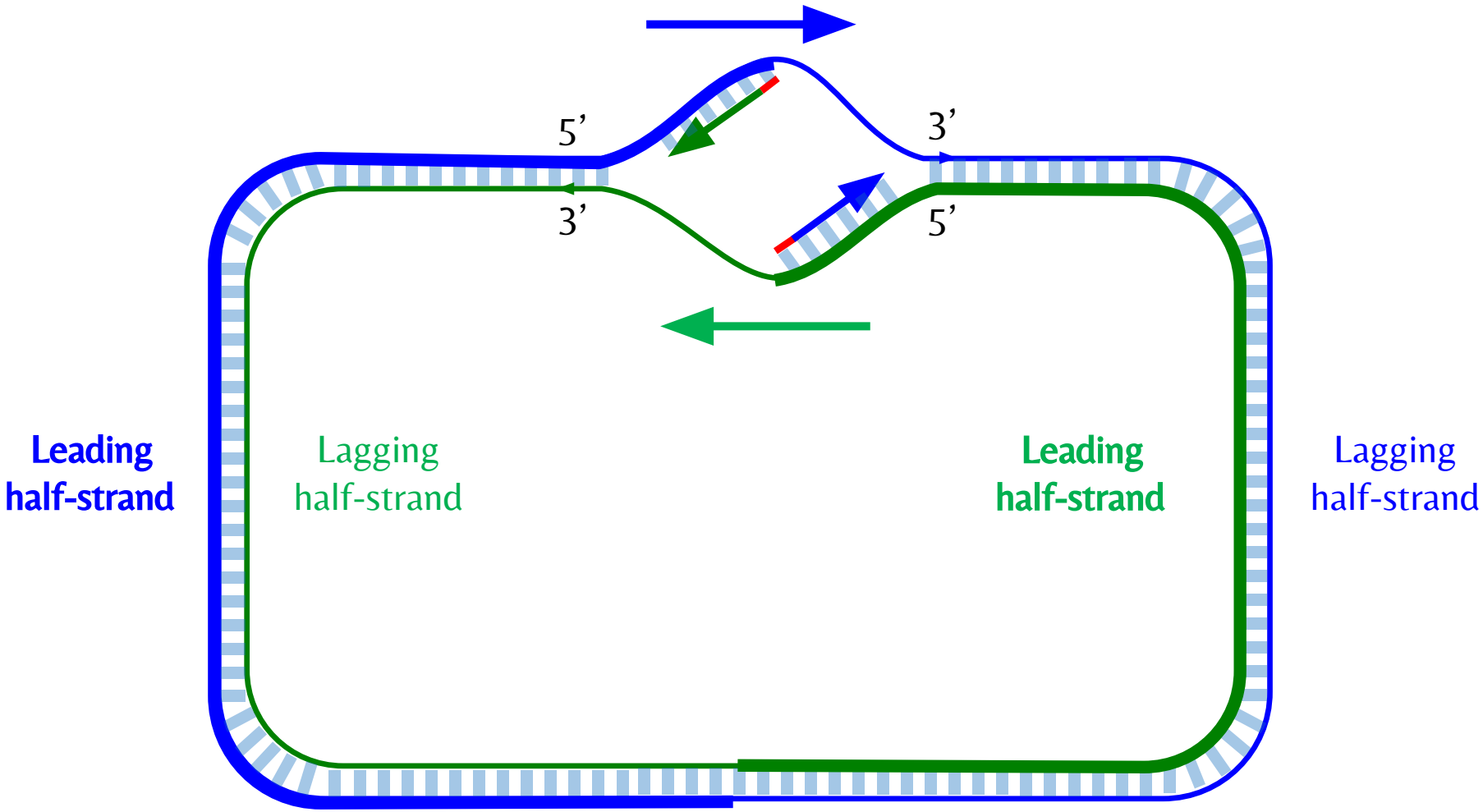
Simple, but **wrong:** DNA polymerases are **unidirectional** : they can only traverse a parent strand in the 3’ □ 5’ direction.

If you Were a **UNIDIRECTIONAL** DNA Polymerase, how Would you Replicate a Genome?



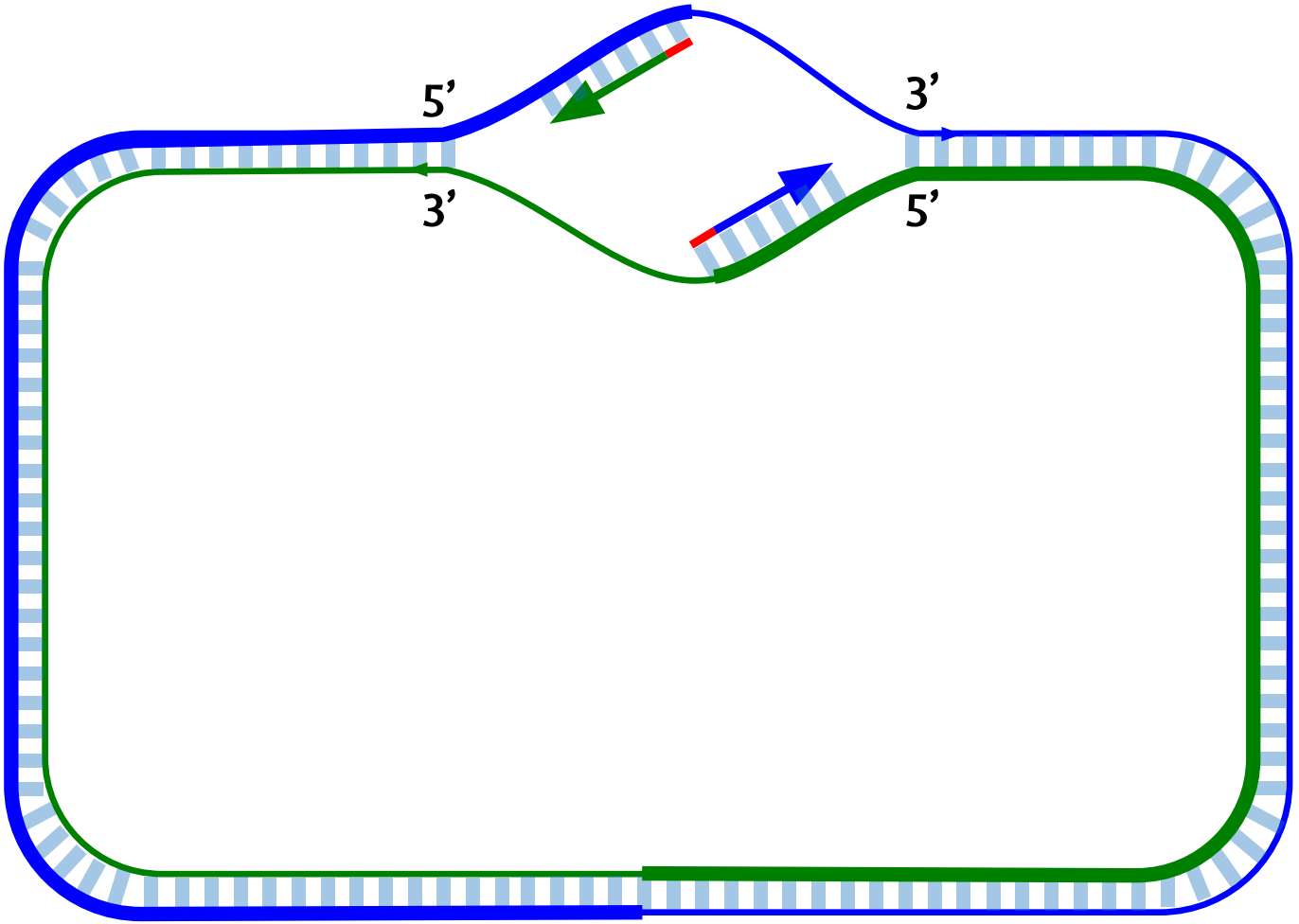
Big problem replicating lagging half-strands (thin lines) .

If you Were a **UNIDIRECTIONAL** DNA Polymerase, how Would you Replicate a Genome?

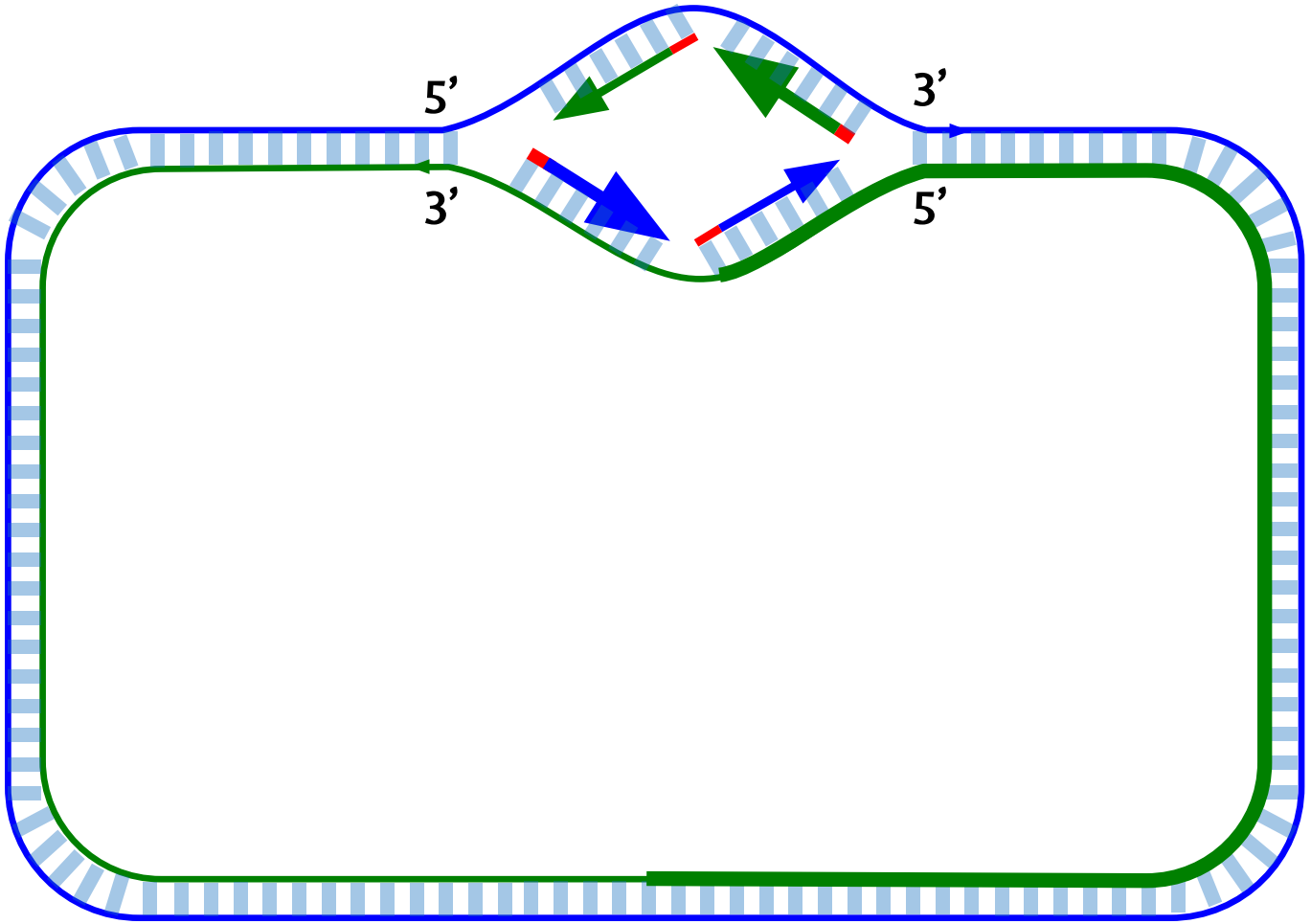


Note: Leading/lagging half-strands are *complementary*.

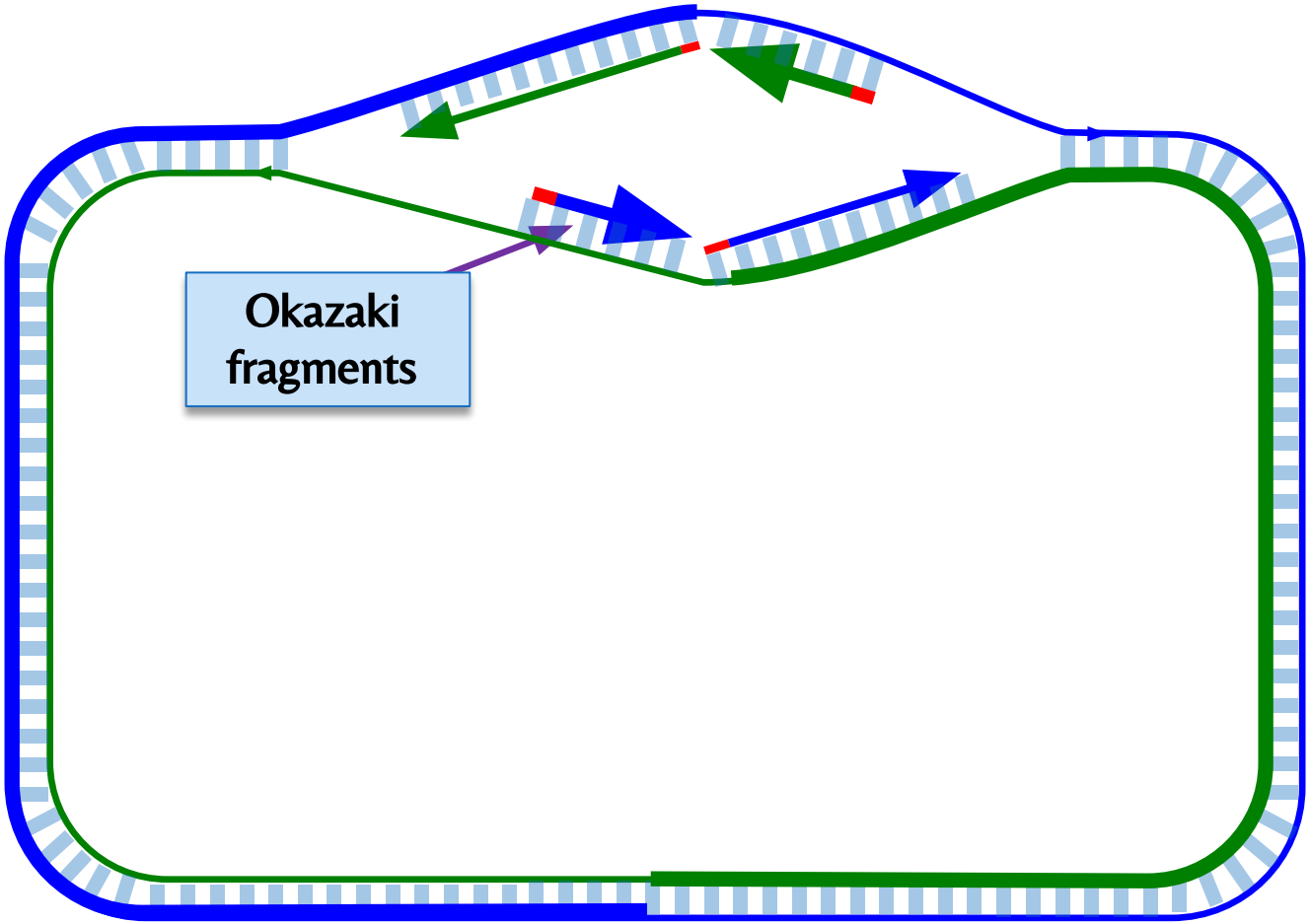
Wait until the Fork Opens and ...



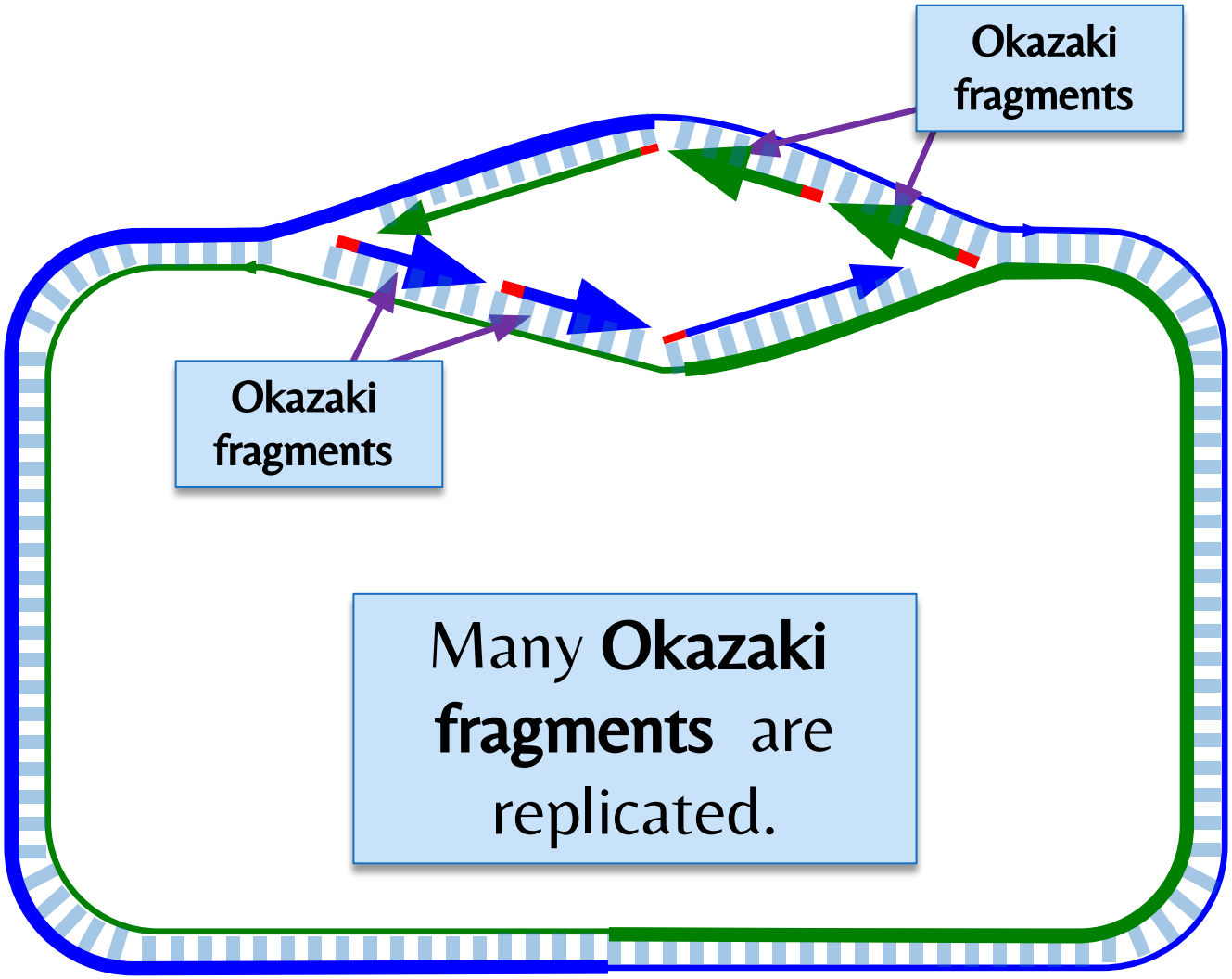
Wait until the Fork Opens and Replicate



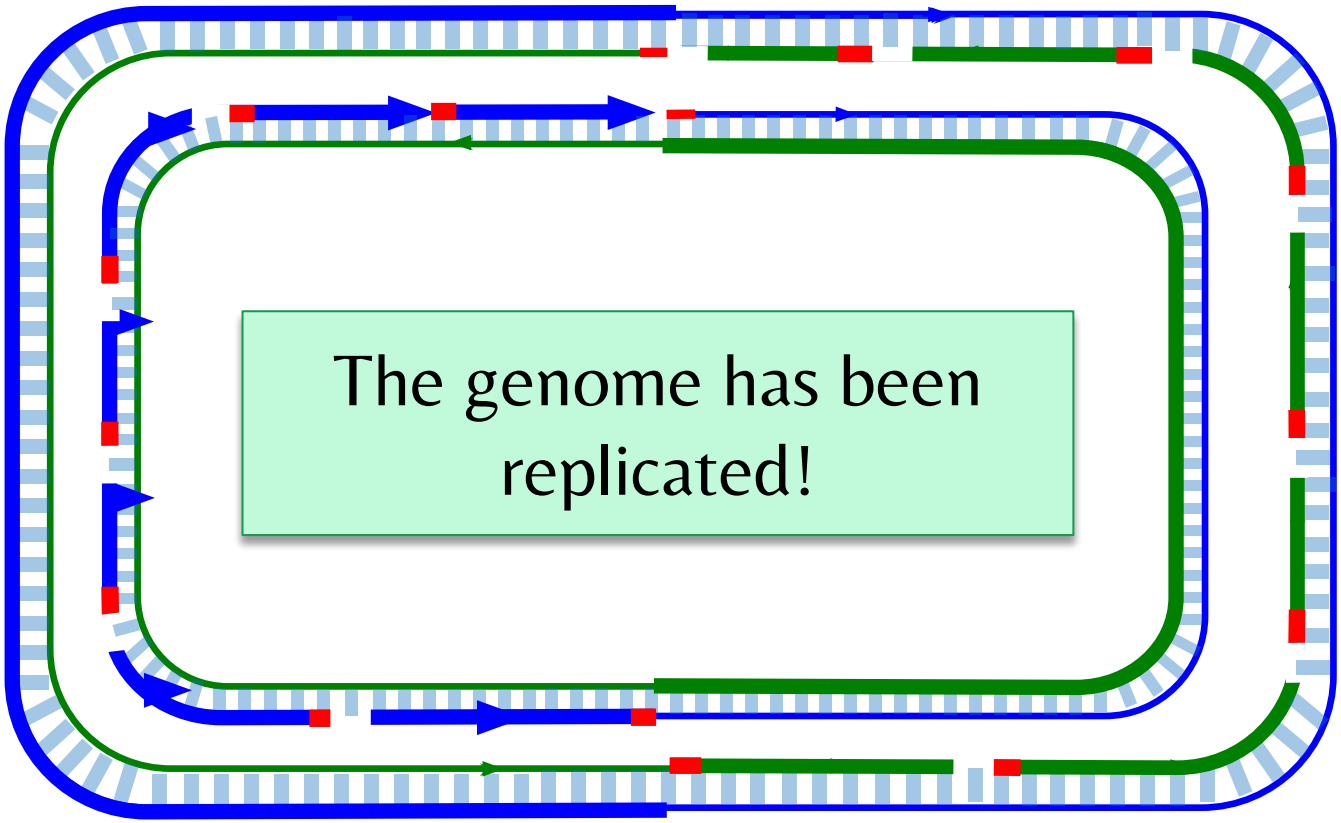
Iterate this Process



Iterate this Process



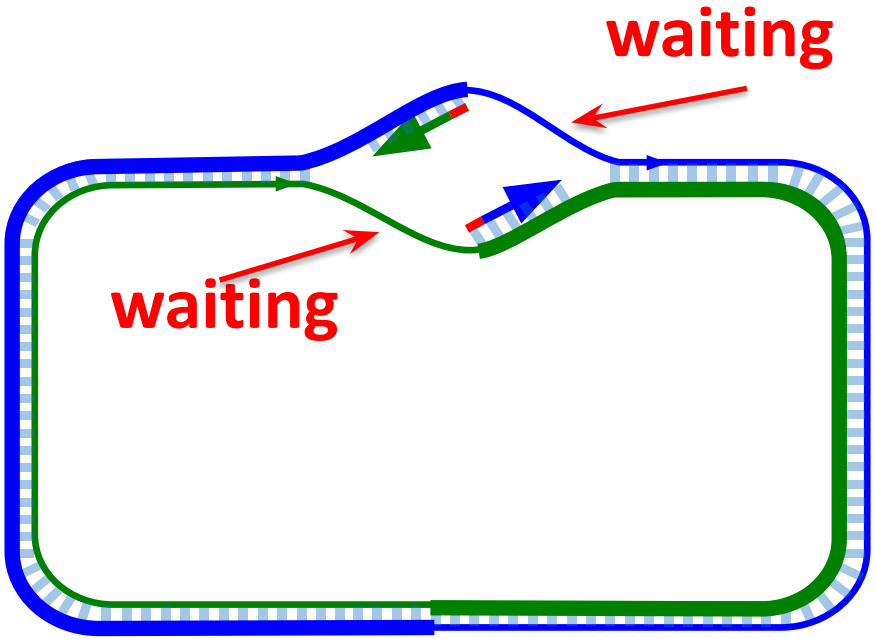
Okazaki Fragments Must Be Ligated



Different Lifestyles of Half-strands

The **leading half-strand** lives a **double-stranded** life most of the time.

The **lagging half-strand** spends a large portion of its life **single-stranded** , **waiting** to be replicated.

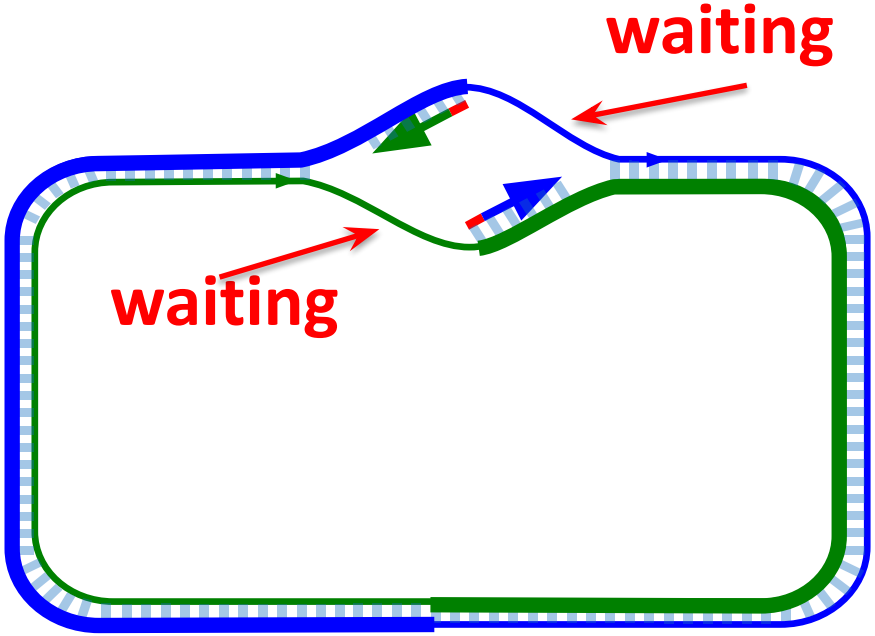


Different Lifestyles of Half-strands

The **leading half-strand** lives a **double-stranded** life most of the time.

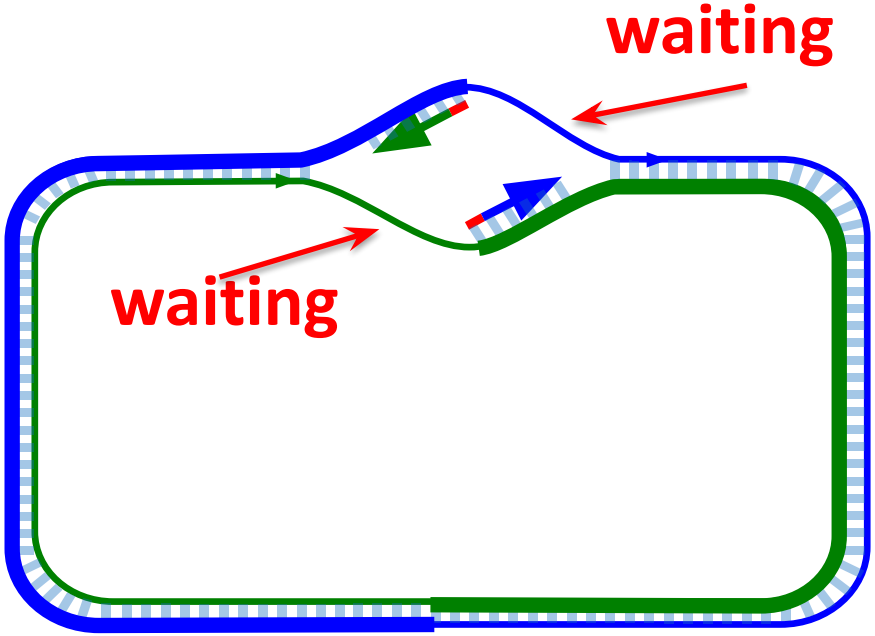
The **lagging half-strand** spends a large portion of its life **single-stranded** , **waiting** to be replicated.

But why would a computer scientist care?



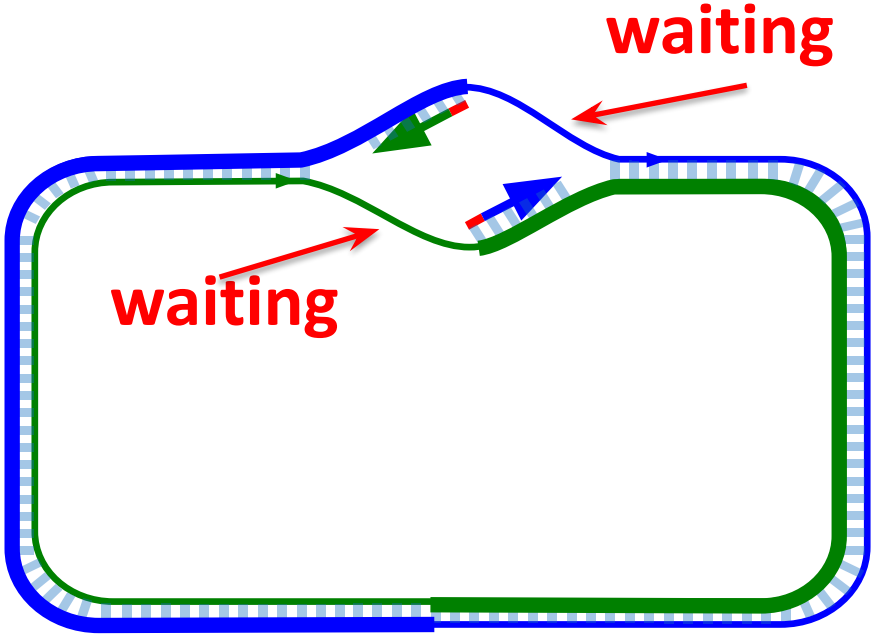
Asymmetry of Replication Affects Nucleotide Frequencies

Single-stranded DNA has a much higher mutation rate than double-stranded DNA.



Asymmetry of Replication Affects Nucleotide Frequencies

Single-stranded DNA has a much higher mutation rate than double-stranded DNA.



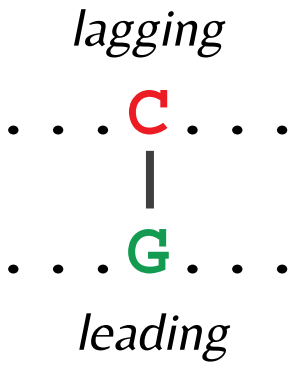
Thus, if one nucleotide has a greater mutation rate, then we should observe its **shortage** on the lagging half-strand, since it is more often single-stranded!

Deamination is the Answer

Cytosine (C) rapidly mutates into thymine (T) through **deamination** ; deamination rates rise 100-fold when DNA is single-stranded!

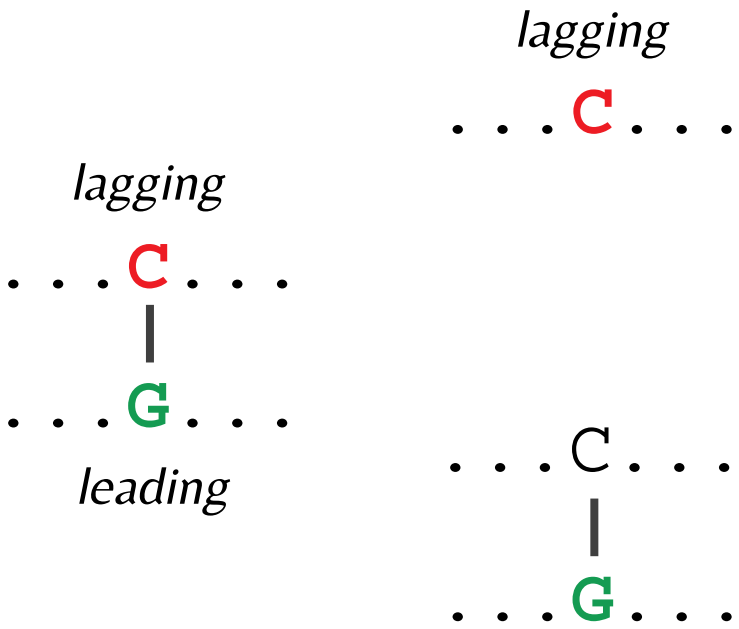
Deamination is the Answer

Cytosine (C) rapidly mutates into thymine (T) through **deamination** ; deamination rates rise 100-fold when DNA is single-stranded!



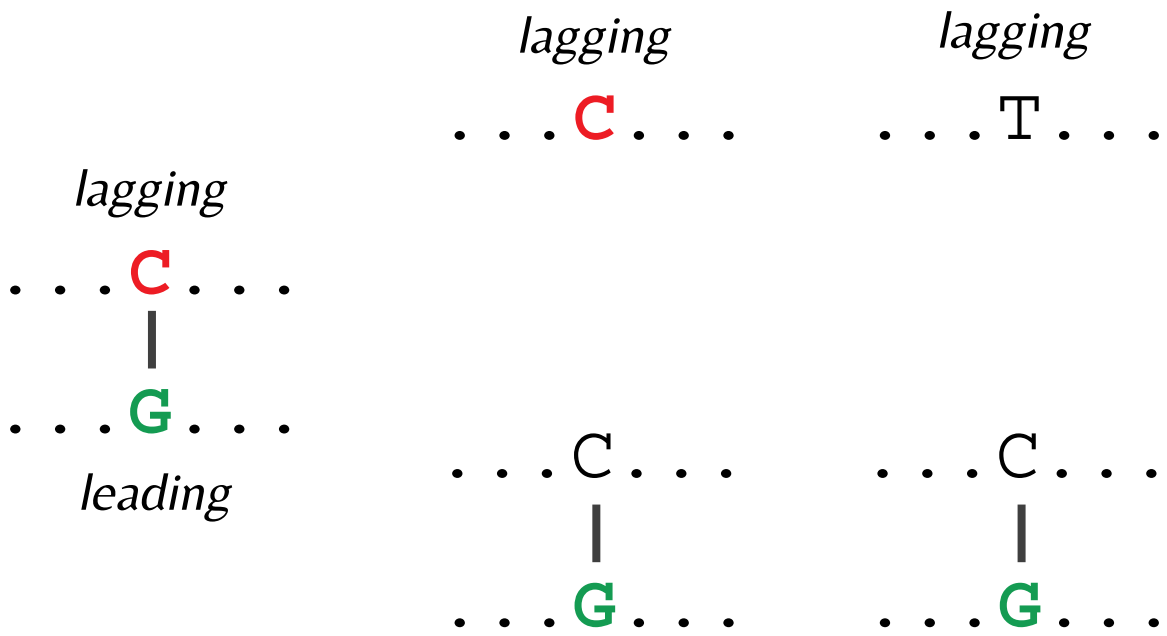
Deamination is the Answer

Cytosine (C) rapidly mutates into thymine (T) through **deamination** ; deamination rates rise 100-fold when DNA is single-stranded!



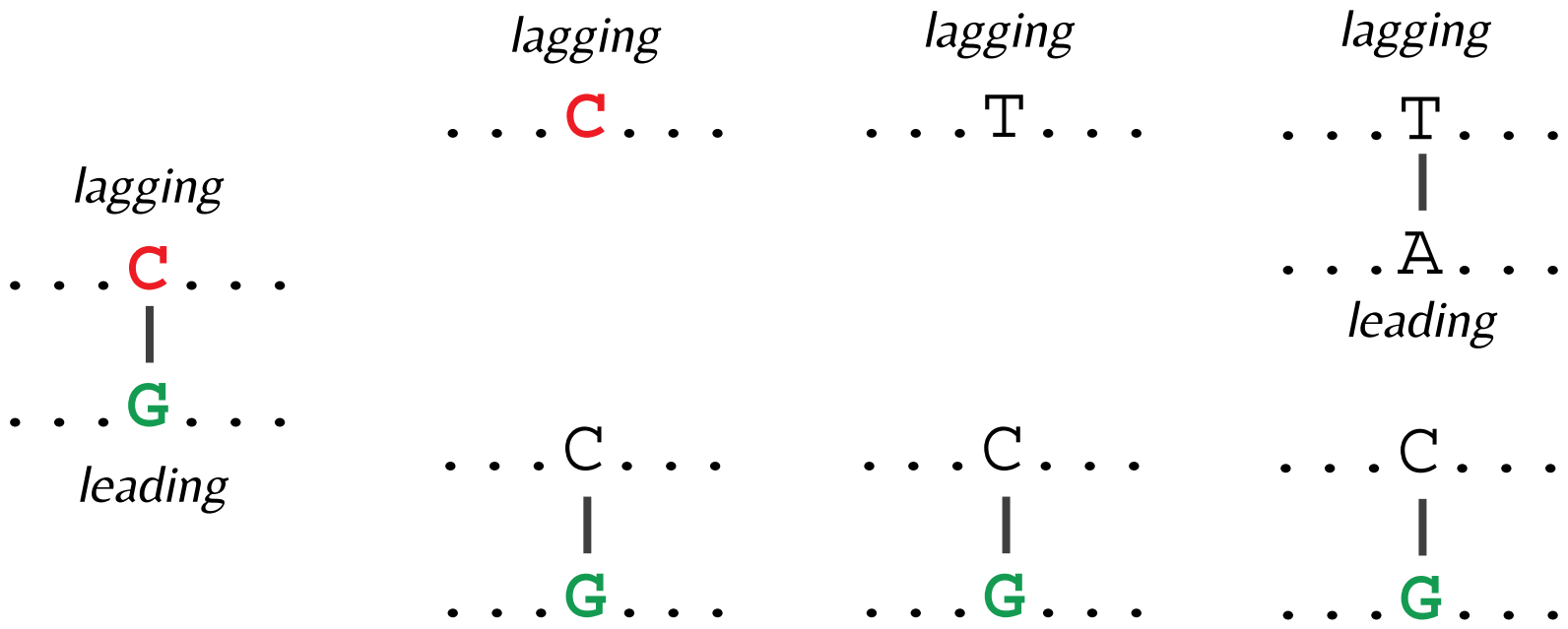
Deamination is the Answer

Cytosine (C) rapidly mutates into thymine (T) through **deamination** ; deamination rates rise 100-fold when DNA is single-stranded!

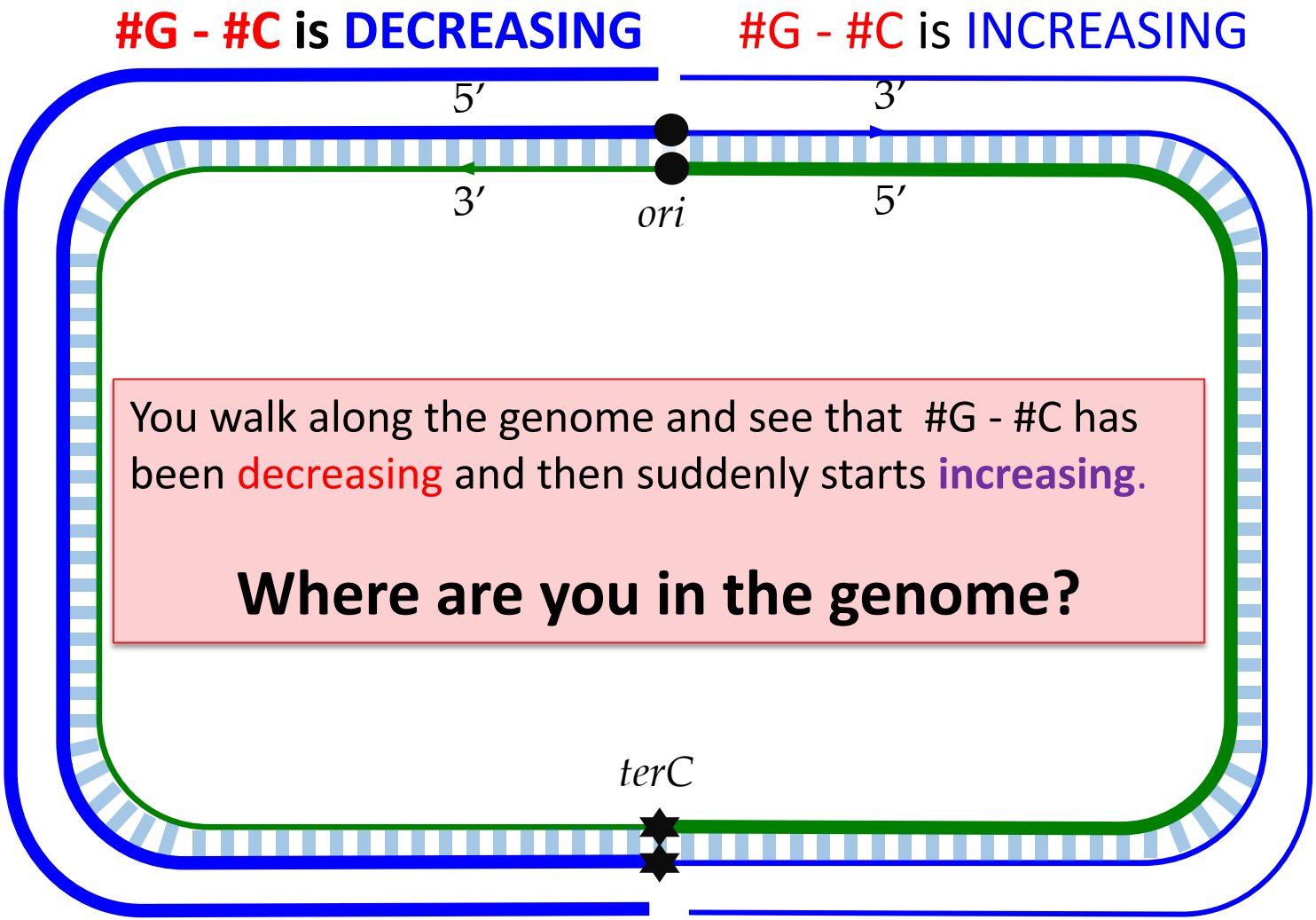


Deamination is the Answer

Cytosine (C) rapidly mutates into thymine (T) through **deamination** ; deamination rates rise 100-fold when DNA is single-stranded!



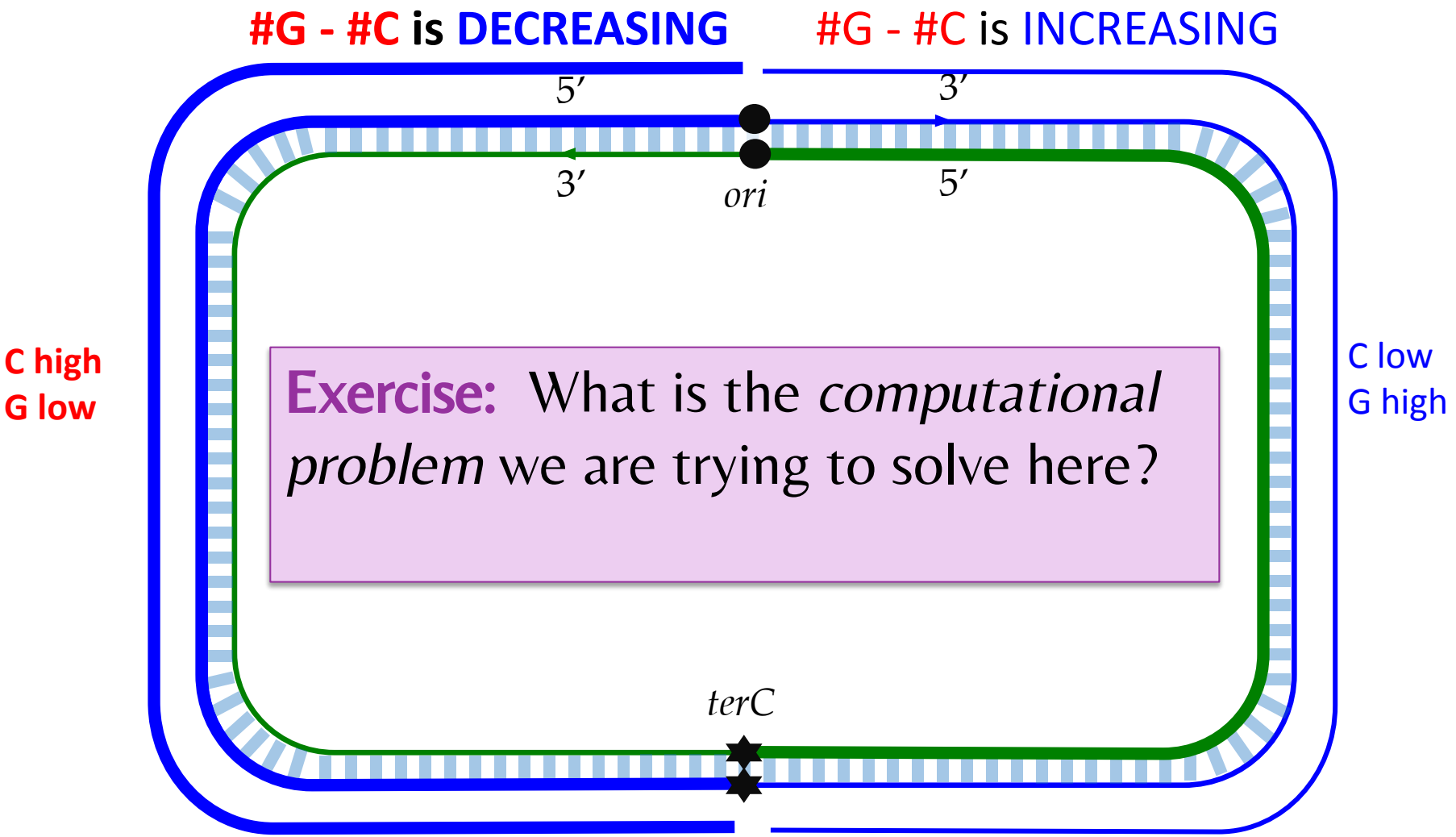
Take a Walk Along the Genome



C high/G low → **#G - #C is DECREASING as we walk along the LEADING half-strand**

C low/G high → #G - #C is INCREASING as we walk along the LAGGING half-strand

Take a Walk Along the Genome



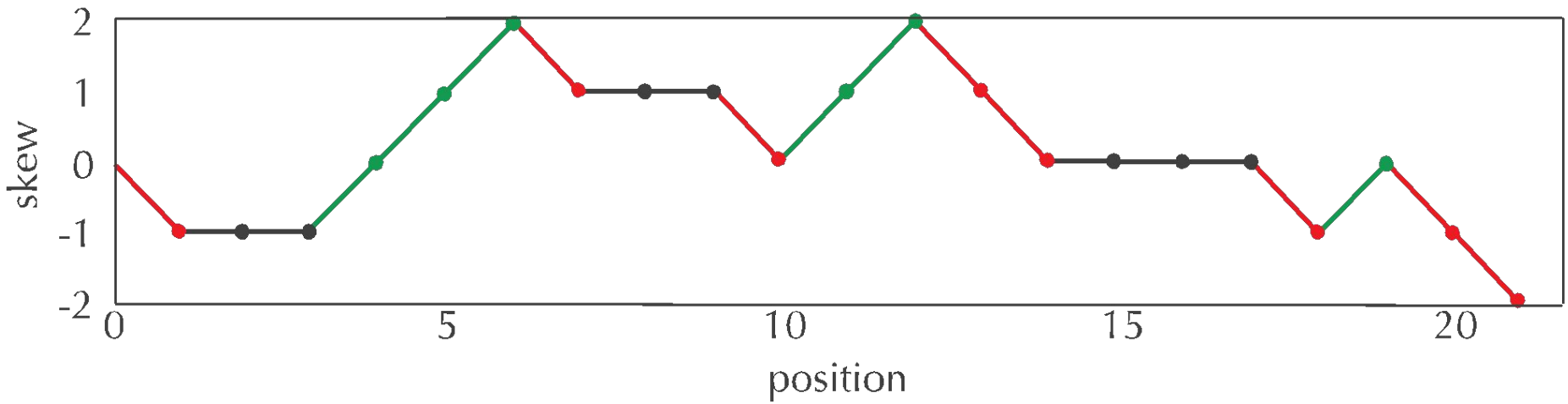
C high/G low → **#G - #C is DECREASING as we walk along the LEADING half-strand**

C low/G high → #G - #C is INCREASING as we walk along the LAGGING half-strand

Skew Array/Diagram

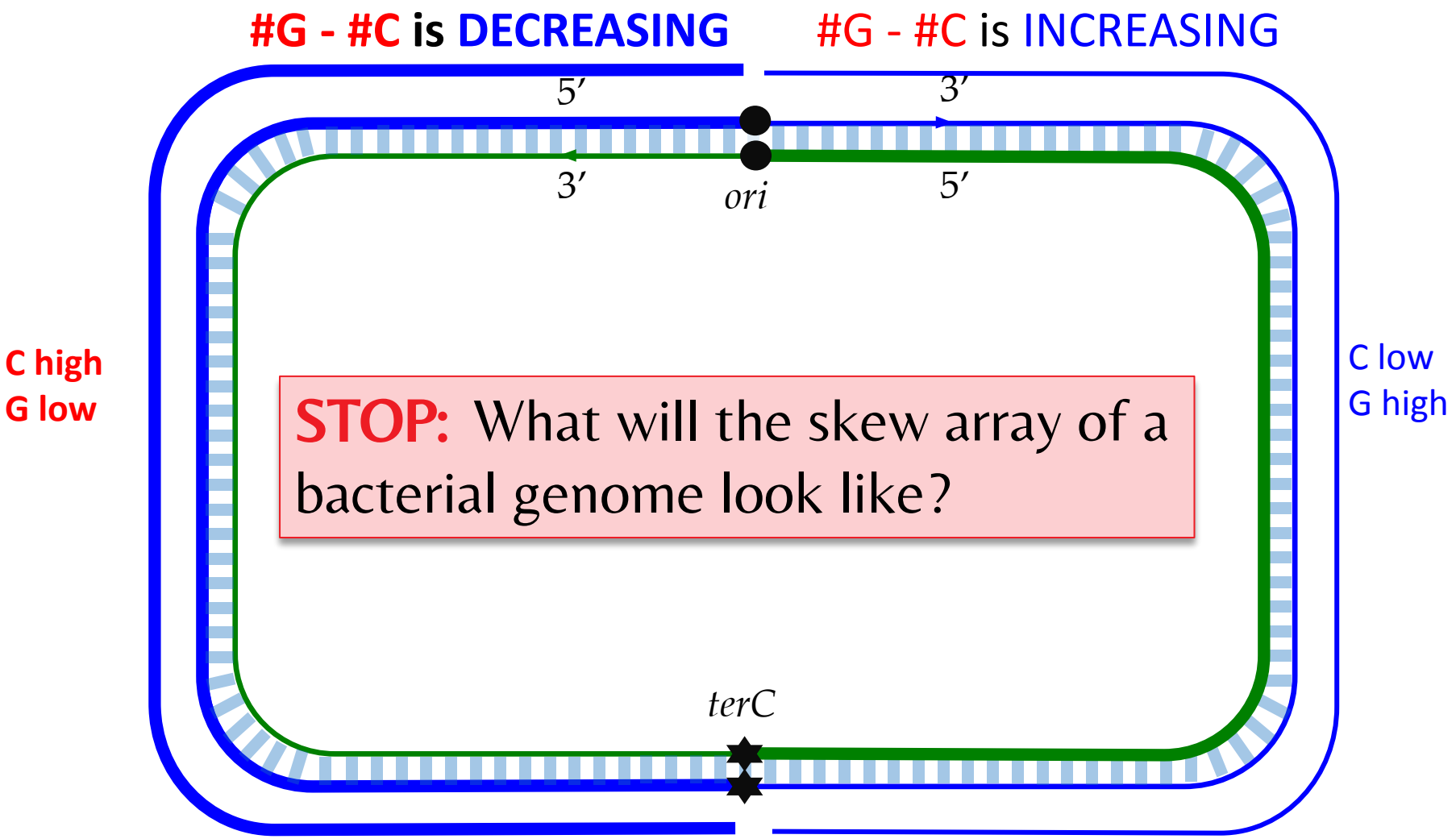
Skew array : *Skew*[*k*] = #G - #C for the **first *k* nucleotides** of *Genome*.

Skew diagram : Plot *Skew*[*k*] against *k*.

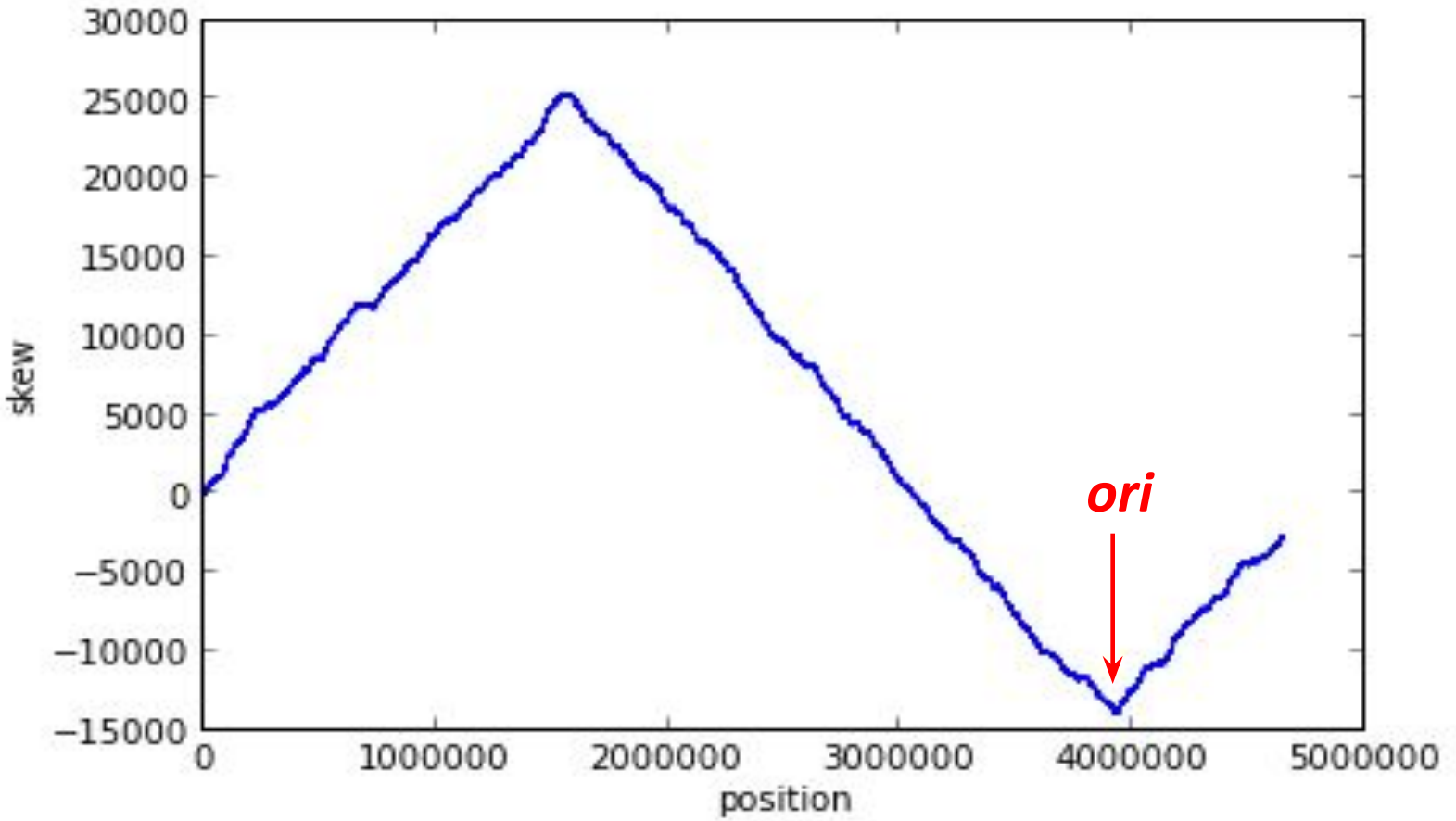


CATGGGCATCGGCCATACGCC

Skew Array/Diagram



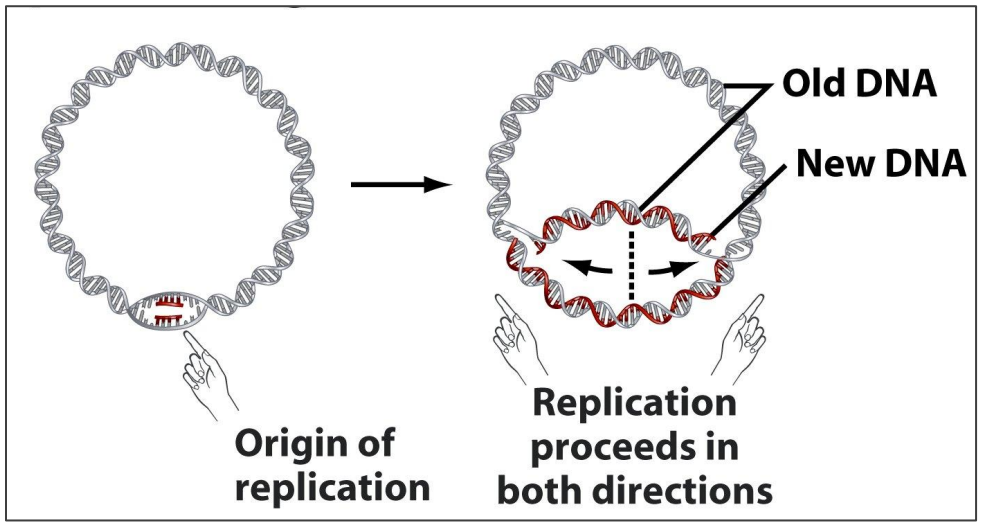
Skew Diagram of *E. Coli*



You walk along the genome and see that #G - #C have been decreasing and then suddenly starts increasing . Where are you in the genome?

We Have Now "Solved" Question 1!

Given a bacterial genome (~3 Mbp), where is *ori*?



Minimum Skew Problem

- **Input:** A DNA string *genome*.
- **Output:** The min value of *Skew*[*k*] for *genome*.

We Have Now "Solved" Question 1!

Given a bacterial genome (~3 Mbp), where is *ori*?

Nucleic Acids Res. 1998 May 15;26(10):2286-90.

Analyzing genomes with cumulative skew diagrams.

Grigoriev A[1].

⊕ **Author information**

Abstract

A novel method of cumulative diagrams shows that the nucleotide composition of a microbial chromosome changes at two points separated by about a half of its length. These points coincide with sites of replication origin and terminus for all bacteria where such sites are known. The leading strand is found to contain more guanine than cytosine residues. This fact is used to predict origin and terminus locations in other bacterial and archaeal genomes. Local changes, visible as diagram distortions, may represent recent genome rearrangements, as demonstrated for two strains of Escherichia coli . Analysis of the diagrams of viral and mitochondrial genomes suggests a link between the base composition bias and the time spent by DNA in a single stranded state during replication.

PMID: 9580676 PMCID: PMC147580

We Found the Replication Origin in *E. Coli* **BUT**…

```
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 of *E. coli*!

We Found the Replication Origin in *E. Coli* **BUT**…

```
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 of *E. coli*!

STOP: Any ideas? Should we give up?

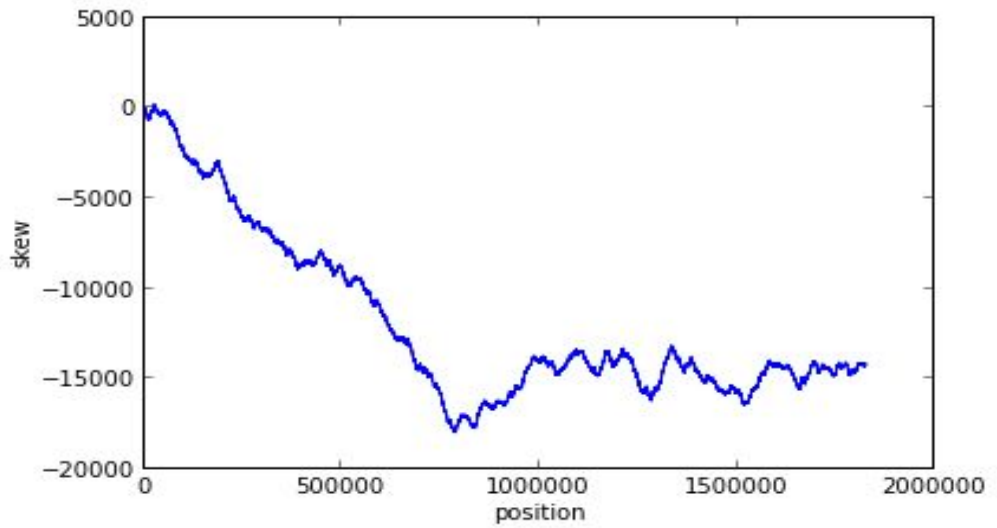
Accounting for Point Mutations

aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgcataacgcggta
tgaaaatggattgaagcccgggccgtggattctactcaactttgtcggcttgagaaagacc
tgggatcctgggtattaaaaagaagatctatttatttagagatctgttctattgtgatctc
ttattaggatcgcactgcccTGTGGATAAcaaggatccggcttttaagatcaacaacctgg
aaaggatcattaactgtgaatgatcggtgatcctggaccgtataagctgggatcagaatga
ggggTTATACACAactcaaaaactgaacaacagttgttcTTTGGATAACtaccggttgatc
caagcttcctgacagagTTATCCACAgtagatcgcacgatctgtatacttatttgagtaaa
ttaacccacgatcccagccattcttctgccggatcttccggaatgtcgtgatcaagaatgt
tgatcttcagtg

Frequent 9-mers (with 1 Mismatch and Reverse Complements) in putative *ori* of *E. coli*

Complications

- Some bacteria have fewer *DnaA* boxes.
- Terminus of replication is often not located directly opposite to *ori*.
- The skew diagram is often more complex than in the case of *E. coli*.



Skew diagram of *T. petrophila*

Moral?

This division is not an appropriate view of how biology (or science in general) can and should operate in the 21st Century.

Homework problem

Take a random sequence with equal number of A, T,G,C and then make then undergo replications and then assume that 50% of all C in the lagging strand are getting mutated.

Ori

Ex. ATGCATGCATGCATGCATGCATGCATGC

Lab: Psuedocode for an OriFinder

http://tubic.tju.edu.cn/Ori-Finder/