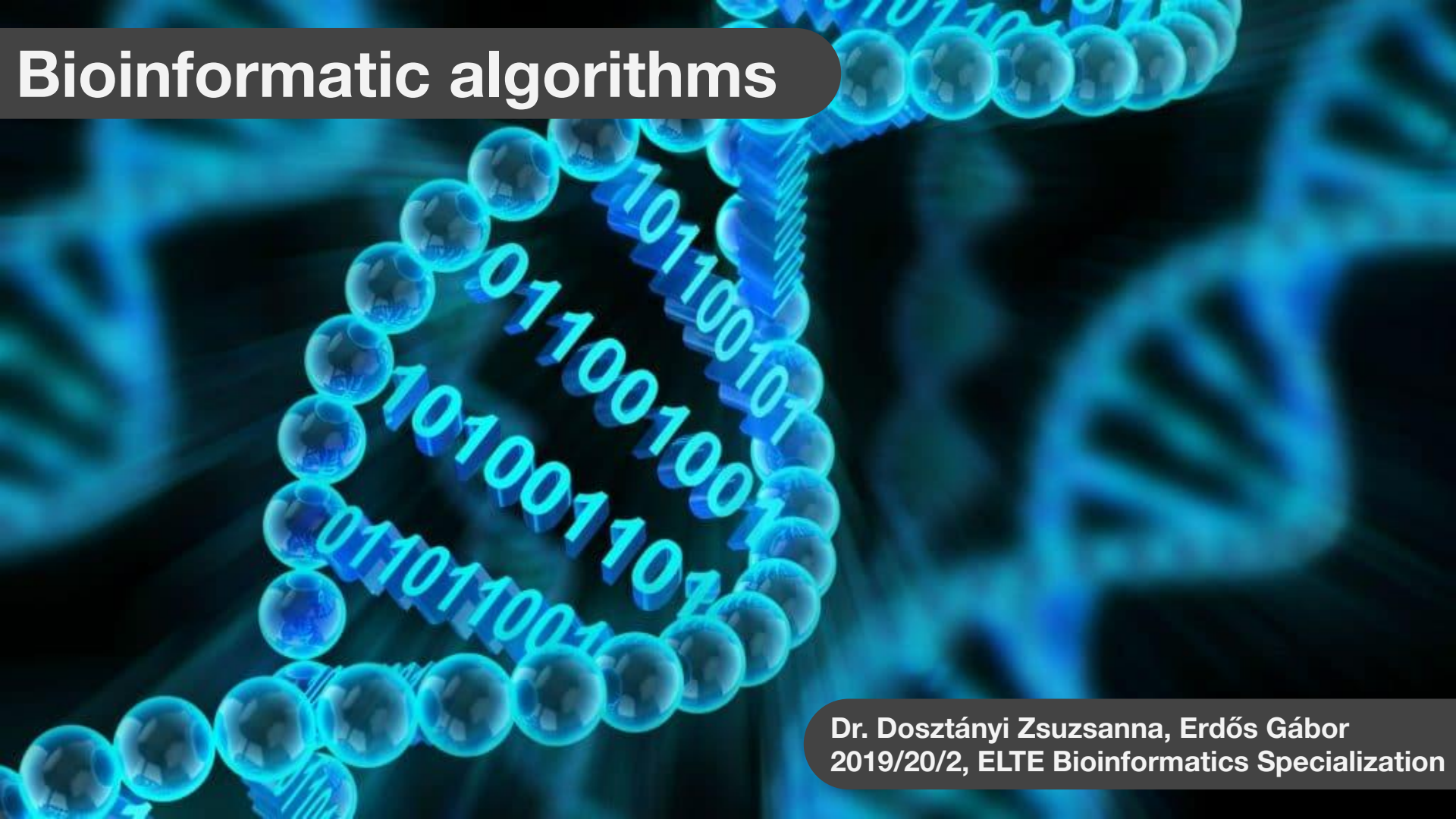
Bioinformatic algorithms



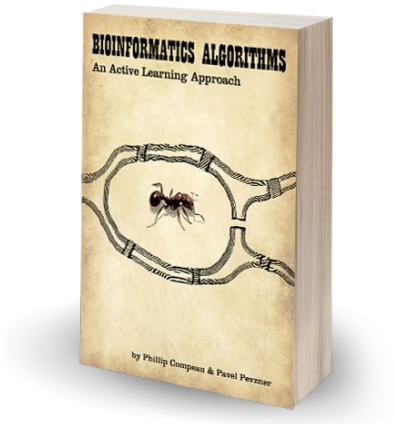
Dr. Dosztányi Zsuzsanna, Erdős Gábor
2019/20/2, ELTE Bioinformatics Specialization

Bioinformatics algorithms

- Thursday, 09:00-12:30
  - Southern building, 5th floor, Department of Biochemistry, meeting room
- Requirements
  - To be present in classes
  - 80% of homework exercises
  - Small test in the beginning of the lesson
- Graded at the end of the semester
- Consultation
  - e.gabor90@gmail.com
  - Southern building, 5th floor, Department of Biochemistry, 5.213

Bioinformatic algorithms

- Recommended literature
  - Bioinformatics Algorithms - An Active Learning Approach; Phillip Compeau & Pavel Pevzner
  - Coursera course with same title
- Topics
  - Motif search
  - Genome sequencing
  - Evolutionary trees
  - Clustering
  - Mass Spectrometry



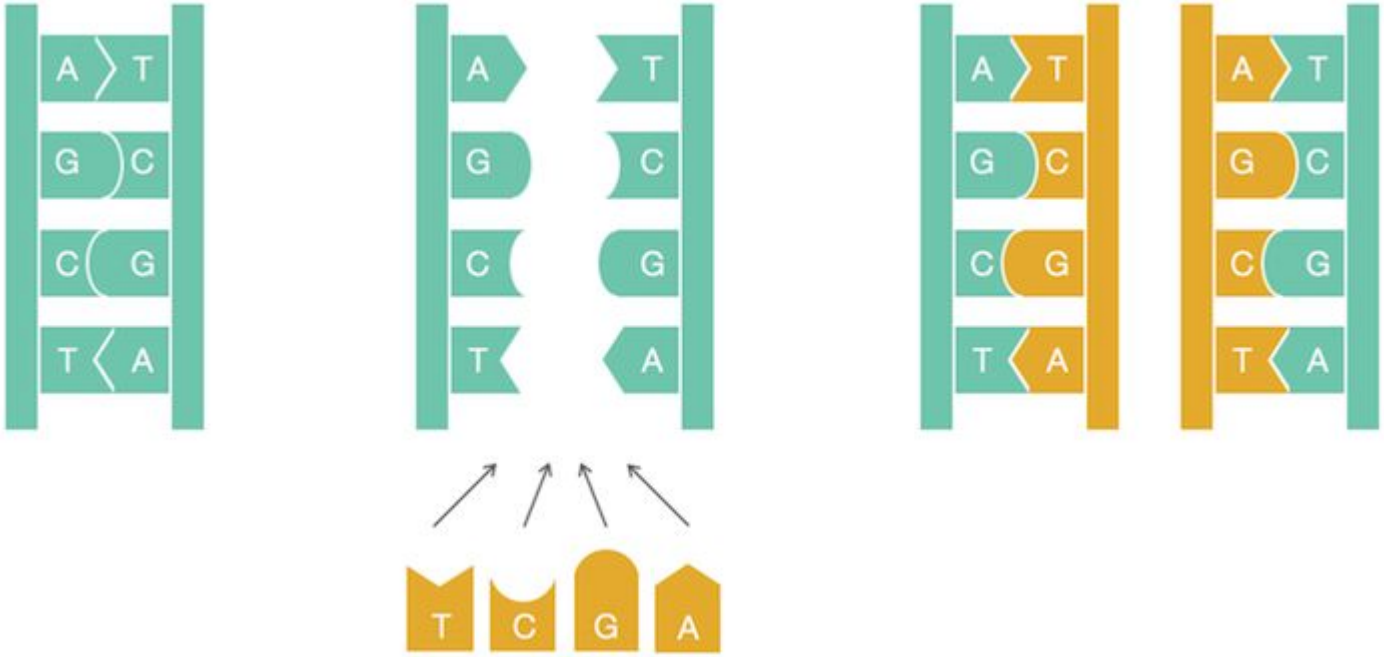
Aims of the course

- Gain further insight into programming
- Familiarize yourself with bioinformatic algorithms
- Algorithms in general
- How can we solve biological problems with a computer?
  - Phrase the problem
  - Find or come up with a correct algorithm
  - Find correct data types or structures
- What is considered a correct algorithm?
  - Giving a solution is not (always) enough!
    - Running times
  - Testing
    - debugger
    - print()

Resources

- http://gerdos.web.elte.hu/edu/bioinformatics_algorithms/
- https://github.com/gerdos/teaching/
- You can use any IDE or coding environment
  - We recommend PyCharm (PRO version is available for university students for free)
  - You can use Jupyter notebooks, however we advise against it
- We recommend you to use version control (like git)
  - VC is out of the scope of this class
  - It is EXTREMELY important for programming

Genome replication



Finding OriC

The reality is a bit more complicated...
...even for bacteria

- Replication begins in a genomic region called the **replication origin** (denoted *ori*)
- carried out by molecular copy machines called **DNA polymerases**

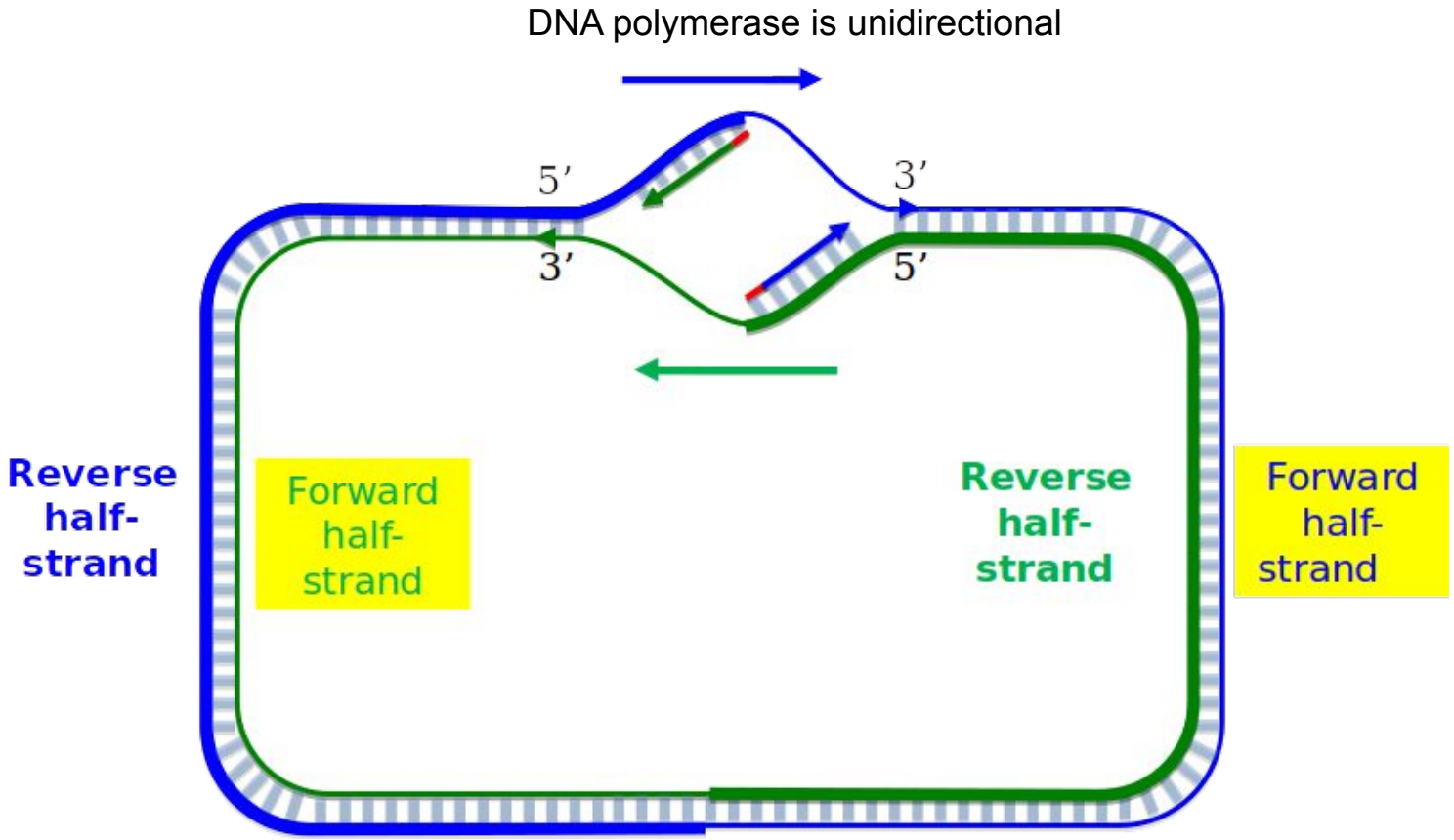
Experimentally : How would you locate *ori* using experimental approaches?

ori has only been experimentally located in a handful of species

- Experiments are time-consuming
- Interpretation of results often require computational approaches

Let's see if we can use computational approaches to help experimental biologists !

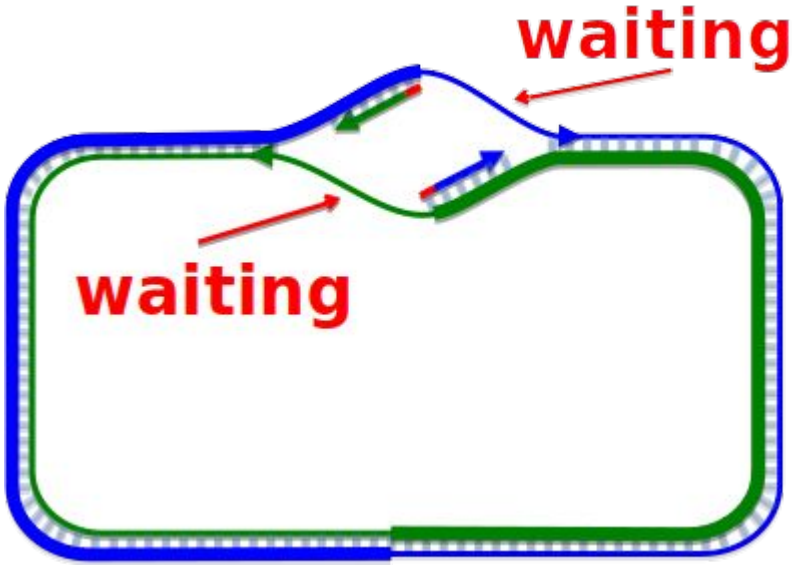
The asymmetry of DNA replication



Different Lifestyles of Reverse and Forward Half-Strands

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

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

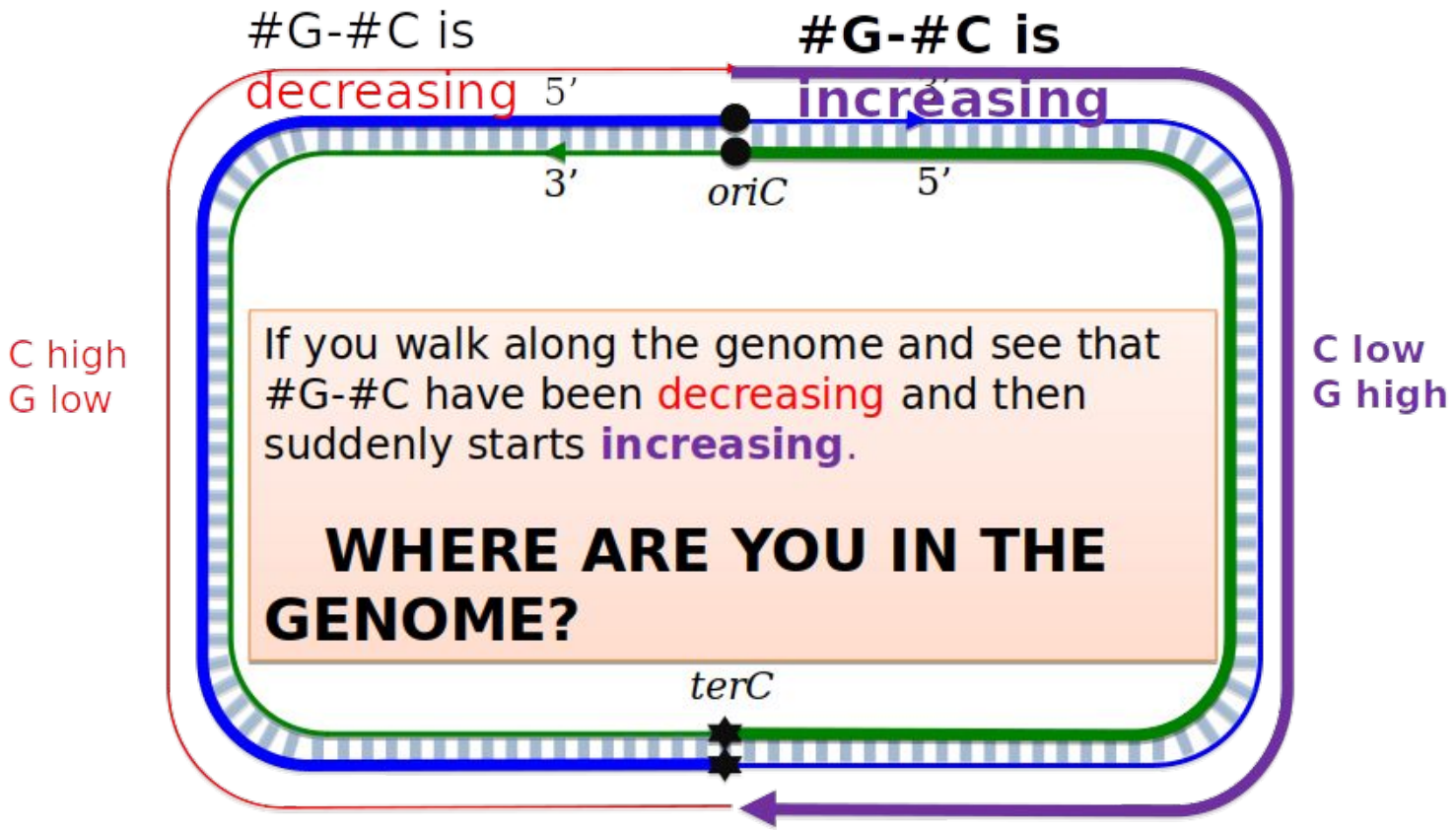


Single-stranded DNA has a much higher mutation rate than double-stranded DNA.

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

Forward half-strand (single-stranded life): **shortage of C, normal G**

Reverse half-strand (double-stranded life): **shortage of G, normal C**



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

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

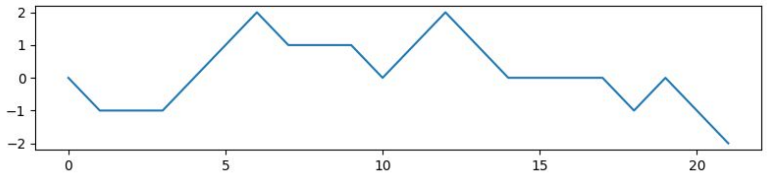
Genome skew problem

Problem: Given a full genome in FASTA format find the approximate location of the ORI

Sample Input:

CATGGGCATCGGCCATACGCC

Sample Output:



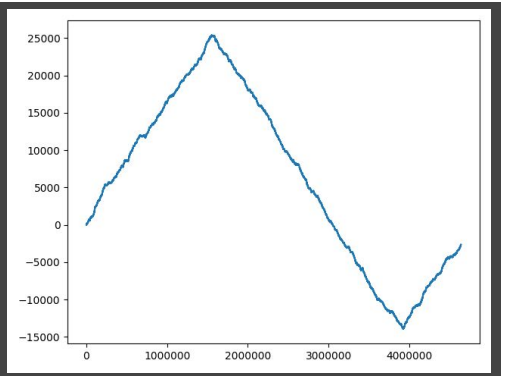
Find the ORI region in the *E. coli* genome!

```
import matplotlib.pyplot as plt

def read_fasta(file_location):
    sequence = ''
    with open(file_location) as fn:
        for line in fn:
            if not line.startswith(">"):
                sequence += line.strip()
    return sequence


def skew(genome):
    skew_list, cntr = [], 0
    for nuc in genome:
        cntr += 1 if nuc == 'G' else -1 if nuc == 'C' else 0
        skew_list.append(cntr)
    return skew_list

e_coli_genom = "data/ecoli.fasta"
genome = read_fasta(e_coli_genom)
plt.plot(skew(genome))
plt.show()
```



Where is the ORI?

```
skew_list = skew(genome)
min_skew = min(skew_list)

######
print(skew_list.index(min_skew))
######
print([n for n, i in enumerate(skew_list) if i==min_skew])
######
positions, min_value = [], float("inf")
for n, i in enumerate(skew_list):
  if i < min_value:
      pos = [n]
      min_value = i
  elif i == min_value:
      positions.append(n)
print(positions)
```

1

2

3

Consider this three methods.
What is the difference between them?

- Do they yield the same results?
- What about their running time?

Save an 500 nucleotide long region from the skew minimum to a file

Let's jump to another genome for now

Download the ORI region of *Vibrio cholerae*

Hidden messages

The initiation of replication is mediated by ***DnaA***, a protein that binds to a short segment within the *ori* known as a ***DnaA* box**

Certain proteins can only bind to DNA if a specific string of nucleotides is present, and if there are more occurrences of the string, then it is more likely that binding will successfully occur. (It is also less likely that a mutation will disrupt the binding process.)

Let's find “words” that are more frequent than others

For example, **ACTAT** is a surprisingly frequent substring of ACA**ACTAT**GCAT**ACTAT**CGGGA**ACTAT**CCT.

First count the occurances of a given word (substring) in a string

Pseudocode

```
PatternCount(Text, Pattern)
        count ← 0
        for i ← 0 to |Text| - |Pattern|
            if Text(i, |Pattern|) = Pattern
               count ← count + 1
        return count
```

Finding a pattern in a string

Pattern Matching Problem: *Find all occurrences of a pattern in a string.*

- **Input**: Strings *Pattern* and *Genome*.
- **Output**: All starting positions in *Genome* where *Pattern* appears as a substring.

Do you know any libraries that might ease this problem?

```
atcaatgatcaacgtaagcttctaagcATGATCAAGgtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagATGATCAAGagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcatgtt
tccttaaccctctatttttacggaagaATGATCAAGctgctgctcttgatcatcgtttc
```

The Frequent Words Problem

We say that *Pattern* is a **most frequent *k*-mer** in *Text* if it maximizes *PatternCount*(*Text*, *Pattern*) among all *k*-mers. You can see that **ACTAT** is a most frequent 5-mer of ACA**ACTAT**GCAT**ACTAT**CGGGA**ACTAT**CCT, and **ATA** is a most frequent 3-mer of CG**ATATA**TCC**ATA**G.

Can a string have multiple most frequent *k*-mers?

```
FrequentWords(Text, k)
        FrequentPatterns ← an empty set
        for i ← 0 to |Text| - k
            Pattern ← the k-mer Text(i, k)
            Count(i) ← PatternCount(Text, Pattern)
        maxCount ← maximum value in array Count
        for i ← 0 to |Text| - k
            if Count(i) = maxCount
                 add Text(i, k) to FrequentPatterns
         remove duplicates from FrequentPatterns
         return FrequentPatterns
```

```
atcaatgatcaacgtaagcttctaagcATGATCAAGgtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagATGATCAAGagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcatgtt
tccttaaccctctatttttacggaagaATGATCAAGctgctgctcttgatcatcgtttc
```

How efficient is this algorithm?

The Frequent Words Problem

Lets see the most frequent k-mers in the region of ORI in the *Vibrio cholerae* genome

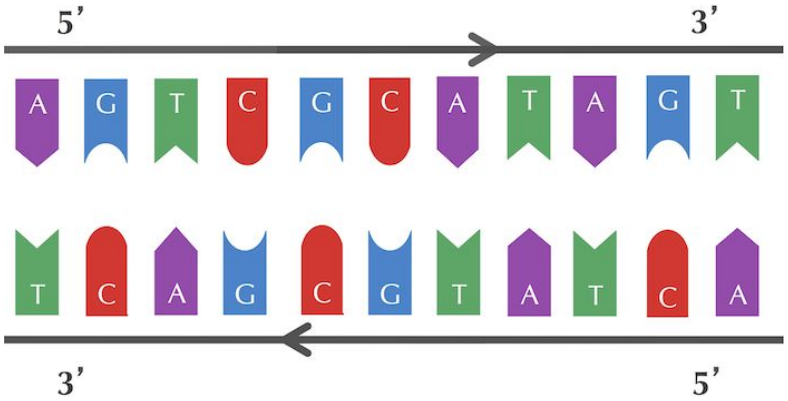
k	3	4	5	6	7	8	9
count	25	12	8	8	5	4	3
k-mers	tga	atga	gatca	tgatca	atgatca	atgatcaa	atgatcaag
			tgatc				cttgatcat
							tcttgatca
							ctcttgatc

What are the expected probabilities of the kmers? See any unusual occurrences?

Can you spot anything interesting among the odd kmers?

atgatcaag is the reverse complementary strand of **cttgatcat**

Reverse complementary problem



Given a nucleotide *p*, we denote its complementary nucleotide as *p**. The **reverse complement** of a string *Pattern* = $p_1 \dots p_n$ is the string $Pattern_{rc} = p_n^* \dots p_1^*$ formed by taking the complement of each nucleotide in *Pattern*, then reversing the resulting string.

atcaatgatcaacgtaagcttctaagcATGATCAAGgtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagATGATCAAGagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctCTTGATCATcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctCTTGATCATgtt
tccttaaccctctatttttacggaagaATGATCAAGctgctgctCTTGATCATcgtttc

The four most frequent 9-mers in *ori* of *Vibrio cholerae*, ATGATCAAG and **CTTGATCAT** are reverse complements of each other, resulting in the six total occurrences

Back to *E. coli*

Can you find frequent (appears more than 3 times) 9-mers in the *E. coli* ORI region?

If yes what is the most frequent 9-mer?
If not why?

Before we give up, let's examine the ori of Vibrio cholerae one more time. You may have noticed that in addition to the three occurrences of **ATGATCAAG** and three occurrences of its reverse complement **CTTGATCAT**, the *Vibrio cholerae ori* contains additional occurrences of **ATGATCAA**C and **CATGATCAT**, which differ from **ATGATCAAG** and **CTTGATCAT** in only a single nucleotide

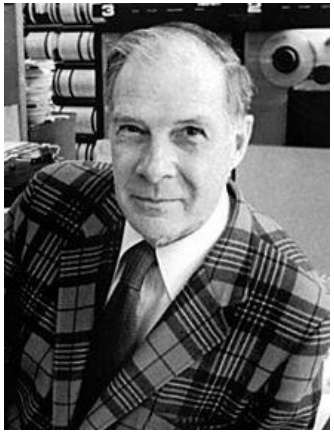
atcaATGATCAACgtaagcttctaagcATGATCAAGgtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagATGATCAAGagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagCATGATCATggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctCTTGATCATcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctCTTGATCATgtt
tccttaaccctctatttttacggaagaATGATCAAGctgctgctCTTGATCATcgtttc

Hamming distance

We say that position *i* in *k*-mers *p*1 … *pk* and *q*1 … *qk* is a **mismatch** if *pi* ≠ *qi*. For example, CG**A**A**T** and CG**G**A**C** have two mismatches. The number of mismatches between strings *p* and *q* is called the **Hamming distance** between these strings and is denoted *HammingDistance*(*p*, *q*).

Hamming Distance Problem: *Compute the Hamming distance between two strings.*

- **Input**: Two strings of equal length.
- **Output**: The Hamming distance between these strings.



Richard Hamming

Approximate Patterns

We say that a *k*-mer *Pattern* appears as a substring of *Text* with at most *d* mismatches if there is some *k*-mer substring *Pattern'* of *Text* having *d* or fewer mismatches with *Pattern*, i.e., *HammingDistance*(*Pattern*, *Pattern'*) ≤ *d*.

Approximate Pattern Matching Problem: *Find all approximate occurrences of a pattern in a string.*

- ***Input***: *Strings Pattern and Text along with an integer d.*
- ***Output***: *All starting positions where Pattern appears as a substring of Text with at most d mismatches.*

Approximate Pattern Count Problem: Computing $Count_d(Text, Pattern)$ simply requires us to compute the Hamming distance between *Pattern* and every *k*-mer substring of *Text*, which is achieved by the following pseudocode.

```
ApproximatePatternCount(Text, Pattern, d)
        count ← 0
        for i ← 0 to |Text| - |Pattern|
            Pattern' ← Text(i , |Pattern|)
           if HammingDistance(Pattern, Pattern') ≤ d
                count ← count + 1
        return count
```

Frequent words with mismatches

A **most frequent *k*-mer with up to *d* mismatches** in *Text* is a string *Pattern* maximizing $Count_d(Text, Pattern)$ among all *k*-mers. Note that *Pattern* does not need to actually appear as a substring of *Text*; for example, **AAAAA** is the most frequent 5-mer with 1 mismatch in **AA**C**AA**GCTG**A**T**AAA**C**A**TTT**AAA**G**A**G, even though it does not appear exactly in this string.

Frequent Words with Mismatches Problem: *Find the most frequent k-mers with mismatches in a string*.

- **Input**: A string *Text* as well as integers *k* and *d*. (You may assume $k \leq 12$ and $d \leq 3$.)
- **Output**: All most frequent *k*-mers with up to *d* mismatches in *Text*.

In order to be able to do this, the easiest (but by far not the fastest) solution is to generate all possible k-mers.

Do you know any way to create all k-mers?

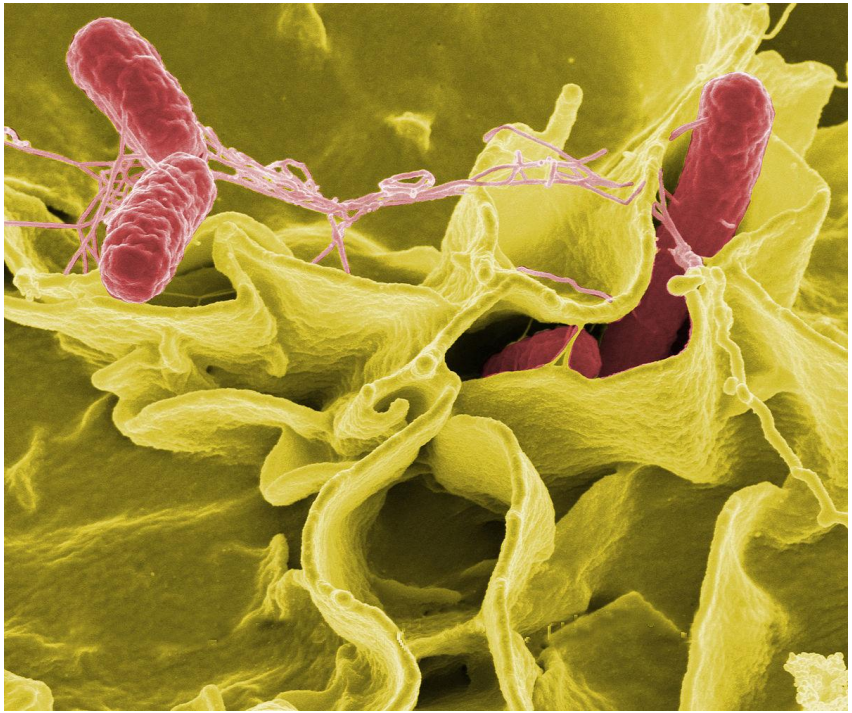
There is a handy library in python called itertools

```
itertools.product(['A', 'C', 'G', 'T'], repeat=9)
```

Any you think of any idea, that might help us make this algorithm more efficient?

Salmonella typhimurium

Salmonella typhimurium is a close relative of E. coli that causes typhoid fever and foodborne illness. After having learned what DnaA boxes look like in E. coli, how would you look for DnaA boxes in *Salmonella typhimurium*?



Clump finding problem

Let's change our computational focus: instead of finding clumps of a specific k-mer, let's try to find every k-mer that forms a clump in the genome.

We defined a *k*-mer as a "clump" if it appears many times within a short interval of the genome. More formally, given integers *L* and *t*, a *k*-mer *Pattern* forms an **(*L*, *t*)-clump** inside a (longer) string *Genome* if there is an interval of *Genome* of length *L* in which this *k*-mer appears at least *t* times.

For example, **TGCA** forms a (25,3)-clump in the following *Genome*:

gatcagcataagggtccC**TGCA**A**TGCA**TGACAAGCC**TGCA**GTtgttttac

Clump Finding Problem: *Find patterns forming clumps in a string.*
Input: A string *Genome*, and integers *k*, *L*, and *t*.
Output: All distinct *k*-mers forming (*L*, *t*)-clumps in *Genome*.

Can we solve this with our current methods?
How efficient it is?