أكاديمية كاوست
KAUST ACADEMY

KAUST Academy - Bioinformatics Program 2024

Instructor: Azza Althagafi

Ph.D. in Computer Science (Bioinformatics)

azza.althagafi@kaust.edu.sa

TA: Ibtisam Saleh

ibtisamssaleh@outlook.com

Riyadh, 25-28 Feb, 2024

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

أكاديمية كاوست
KAUST ACADEMY

Course Overview

Course Overview

أكاديمية كاوست
KAUST ACADEMY

Reference Book: Bioinformatics Algorithms, Phillip Compeau and Pavel Pevzner (Link)

Day	Date	Topic
1	Sun 25 Feb	Chapter 1: The Origins of Replication (Part#1)
2	Mon 26 Feb	Chapter 1: The Origins of Replication (Part#2) Introduction to coding (Python)
3	Tues 27 Feb	Chapter 3: Genome Assembly
4	Wed 28 Feb	Exam (9-12) Chapter 5: Sequence Alignment (1-5)

Course Materials and Policies

أكاديمية كاوست
KAUST ACADEMY

- **Day 1 (25 Feb) – Day 4 (28 Feb):**
  - All day 8 am - 5 pm
  - Theoretical part
  - Pratical part (Coding): Using Rosalind

  - All the slieds will be uploaded in: http://tinyurl.com/27chtrbb

Course Material

أكاديمية كاوست
KAUST ACADEMY

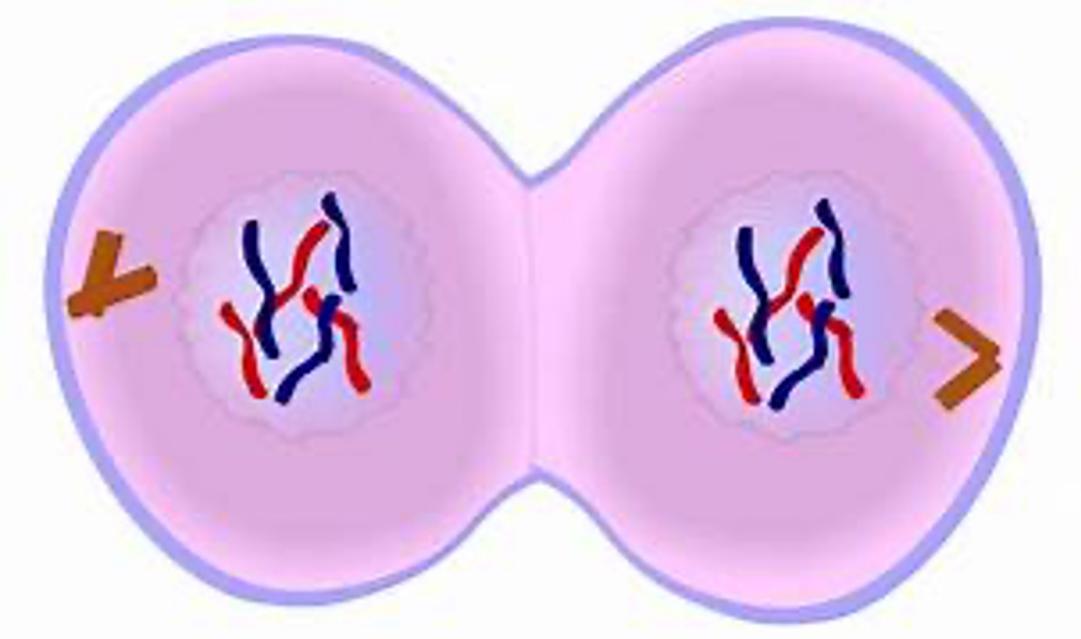
Where in a Genome Does DNA Replication Begin?

•A Journey of a Thousand Miles…

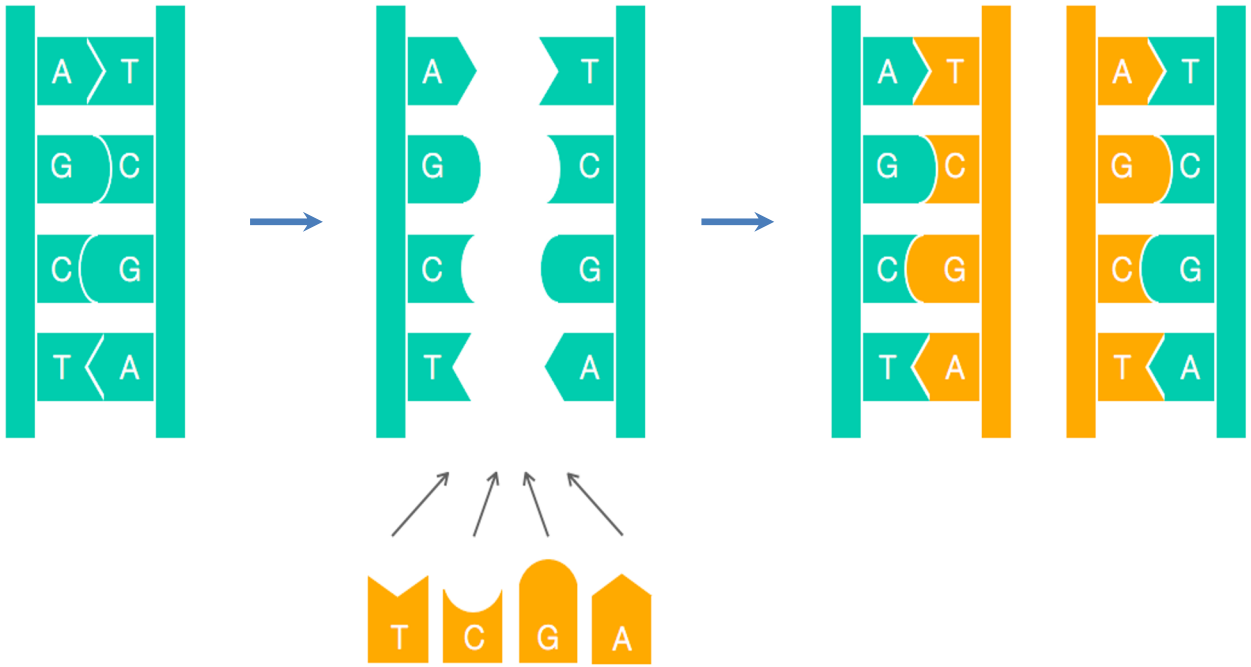
The majority of credit for these slides are attributed to Pavel and Phillip @ *Bioinformatics Algorithms: an Active Learning Approach*

.

Genome replication is one of the most important tasks carried out in the cell!
Before a Cell Divides, it Must Replicate its Genome



The two strands of the parent DNA molecule unwind during replication, and then each parent strand acts as a template for the synthesis of a new strand.
As a result, the replication process begins with a pair of complementary strands of DNA and ends with two pairs of complementary strands.



Although this figure successfully models DNA replication on a simple level, the details of replication turned out to be much more intricate than Watson and Crick imagined
An astounding amount of molecular logistics is required to ensure DNA replication.

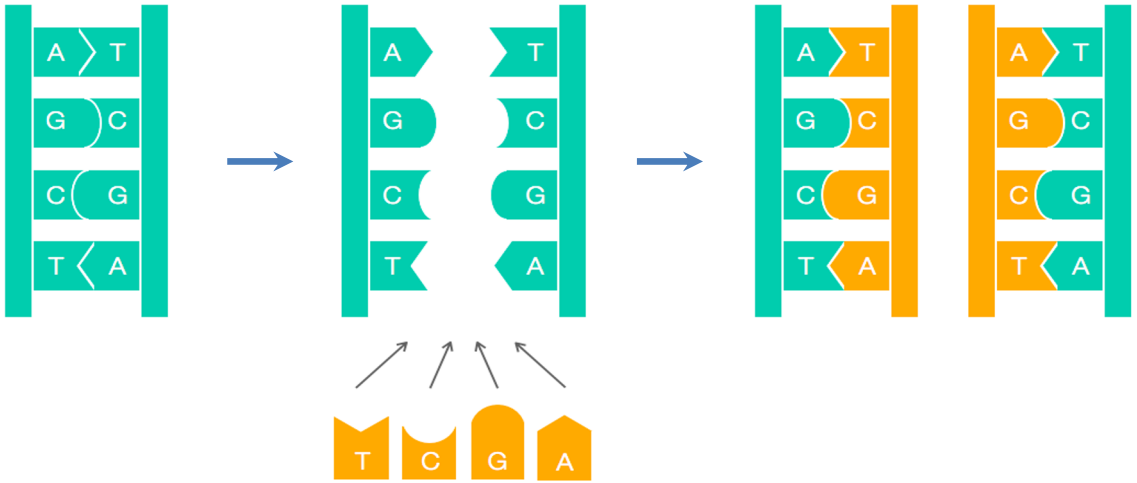


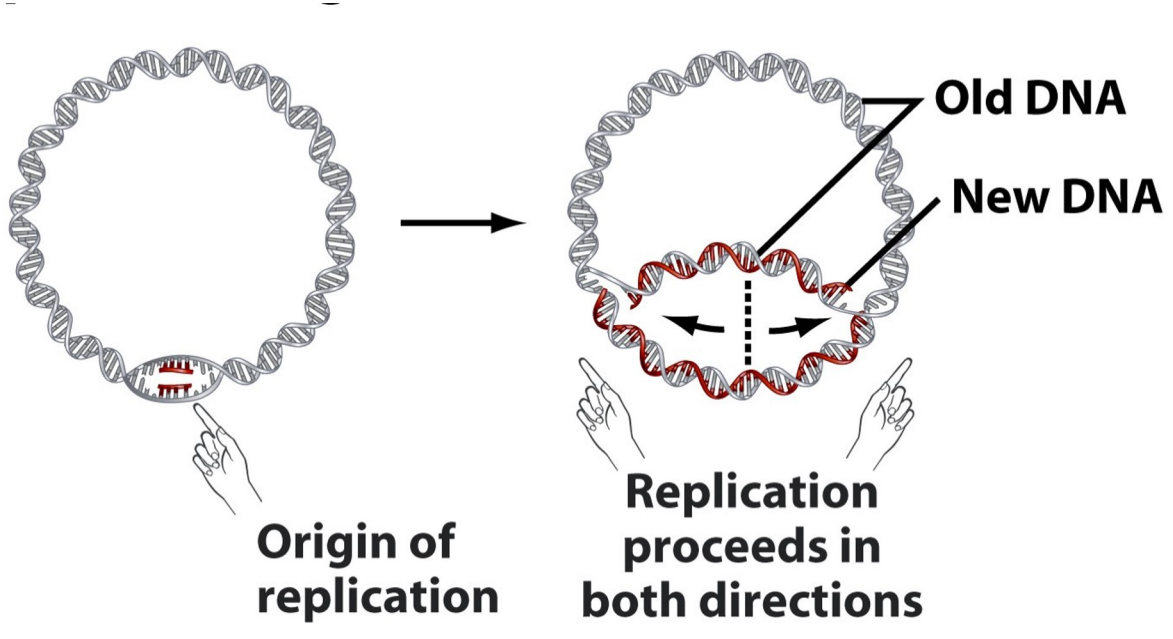
Figure: A naive view of replication. Nucleotides adenine (A) and thymine (T) are complements of each other, as are cytosine (C) and guanine (G). Complementary nucleotides bind to each other in DNA.

At first glance, a computer scientist might not imagine that these details have any computational relevance.

To mimic the process in the above figure algorithmically, we only need to take a string representing the genome and return a copy of it!

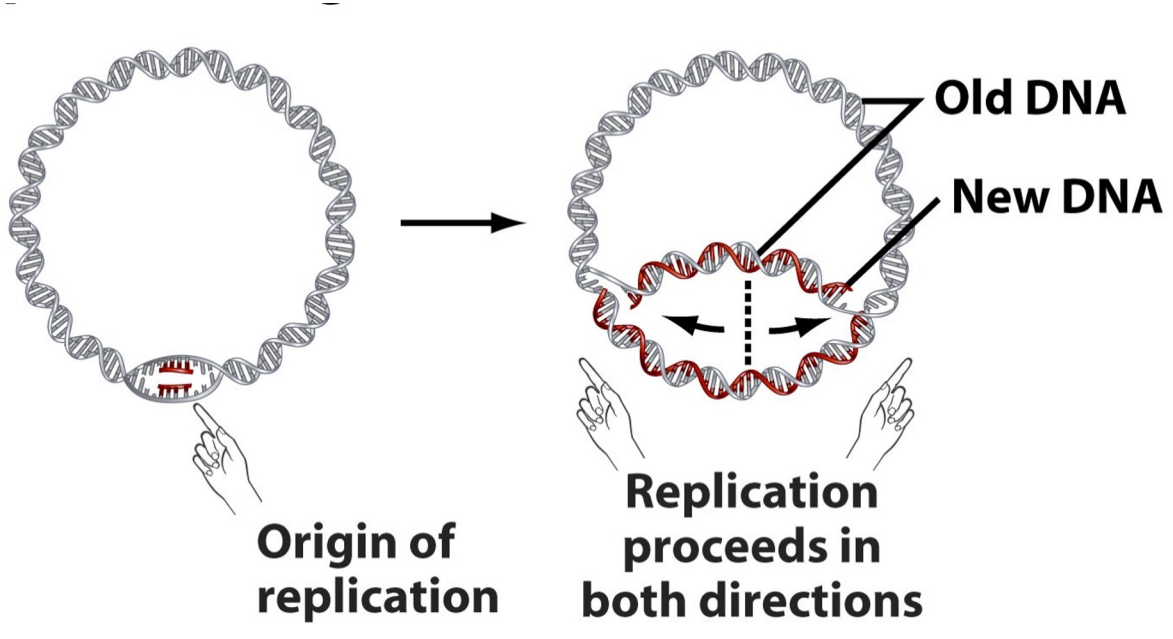
Where in a genome does it all begin?

Replication begins in a region called the replication origin (*oriC*)



- Locating ori is important for understanding cell replication and has implications for biomedical problems.
- Gene therapy methods use viral vectors, genetically engineered mini-genomes that can penetrate cell walls, to carry artificial genes for various applications.
- Viral vectors have been used in agriculture.
- Gene therapy successfully saved many lives.

Replication begins in a region called the replication origin (*oriC*)



Where in a genome does it all begin?

Outline

- **Search for Hidden Messages in Replication Origin**
  - **What is a Hidden Message in Replication Origin?**
  - Some Hidden Messages are More Surprising than Others
  - Clumps of Hidden Messages
- **From a Biological Insight toward an Algorithm for Finding Replication Origin**
  - Asymmetry of Replication
  - Why would a computer scientist care about assymetry of replication?
  - Skew Diagrams
  - Finding Frequent Words with Mismatches
  - Open Problems

Finding Origin of Replication

In the following problem, we assume that a genome has a single *ori* and is represented as a **DNA string**, or a string of nucleotides from the four-letter alphabet {A, C, G, T}.

Finding *oriC* Problem: Finding *oriC* in a genome.

- **Input.** A genome.
- **Output.** The location of *oriC* in the genome.

OK – let's cut out this DNA fragment. Can the genome replicate without it?

Biologists might delete various short segments from the genome in an effort to find a segment whose deletion stops replication

This is not a computational problem!

Computer scientists shake their heads and demand more information before they can even start thinking

Finding Origin of Replication

In the following problem, we assume that a genome has a single *ori* and is represented as a **DNA string**, or a string of nucleotides from the four-letter alphabet {A, C, G, T}.

Finding *oriC* Problem: Finding *oriC* in a genome.

- **Input.** A genome.
- **Output.** The location of *oriC* in the genome.

OK – let's cut out this DNA fragment.
Can the genome replicate without it?

Why should biologists care what computer scientists think?

This is not a computational problem!

Computational methods are now the only realistic way to answer many questions in modern biology

Finding Origin of Replication

أكاديمية كاوست
KAUST ACADEMY

Finding *oriC* Problem: Finding *oriC* in a genome.

- **Input.** A genome.
- **Output.** The location of *oriC* in the genome.

First, these methods are much faster than experimental approaches.
Second, the results of many experiments cannot be interpreted without computational analysis.
In particular, existing experimental approaches to *ori* prediction are rather time consuming. As a result, *ori* has only been experimentally located in a handful of species.
Thus, we would like to design a computational approach to find *ori* so that biologists are free to spend their time and money on other tasks.

How Does the Cell Know to Begin Replication in Short *oriC*?

أكاديمية كاوست
KAUST ACADEMY

In this chapter, we will focus on the relatively easy case of finding *ori* in bacterial genomes, most of which consist of a single circular chromosome. Research has shown that the region of the bacterial genome encoding *ori* is typically a few hundred nucleotides long. Our plan is to begin with a bacterium in which *ori* is known, and then determine what makes this genomic region special in order to design a computational approach for finding *ori* in other bacteria.

Replication origin of *Vibrio cholerae* (≈500 nucleotides):

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

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

How Does the Cell Know to Begin Replication in Short *oriC*?

أكاديمية كاوست
KAUST ACADEMY

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

How does the bacterial cell know to begin replication exactly in this short region within the much larger *Vibrio cholerae* chromosome, which consists of 1,108,250 nucleotides?

There must be some "hidden message" in the *ori* region ordering the cell to begin replication here.

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

You can think of the *DnaA* box as a message within the DNA sequence telling the *DnaA* protein: "bind here!"

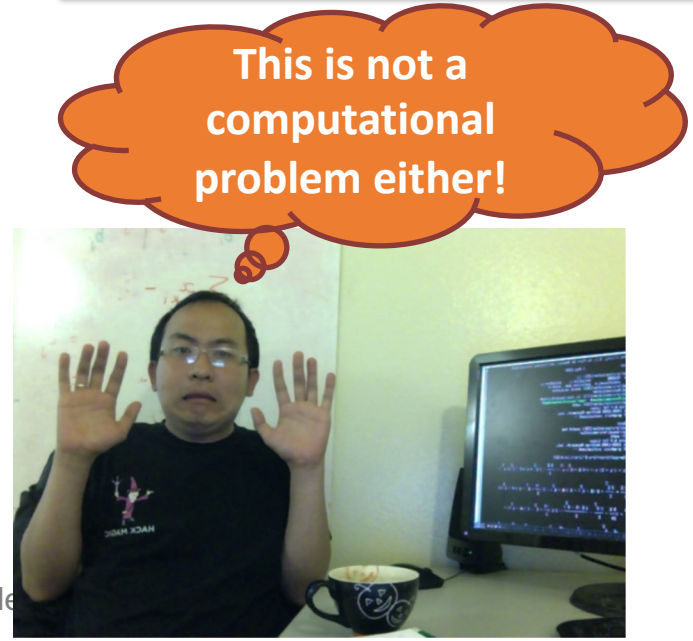
The question is how to find this hidden message without knowing what it looks like in advance—can you find it? In other words, can you find something that stands out in *ori*? This discussion motivates the following problem.

The Hidden Message Problem

STOP and Think: Does the Hidden Message Problem represent a clearly stated computational problem?

Hidden Message Problem. Finding a hidden message in a string.

- **Input.** A string *Text* (representing replication origin).
- **Output.** A hidden message in *Text*.



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

The *ori* region of *Vibrio cholerae* is currently just as puzzling as the parchment discovered by William Legrand in Edgar Allan Poe's story "The Gold-Bug". Written on the parchment was

“The Gold-Bug” Problem


```
53++!305))6*;4826)4+.)4+);806*;4
8!8`60))85;]8*:+*8!83(88)5*!;46(
;88*96*?;8)*+(;485);5*!2:*+(;495
6*2(5*4)8`8*;4069285);)6!8)4++;1
(+9;48081;8:8+1;48!85;4)485!5288
06*81(+9;48;(88;4(+?34;48)4+;161
;:188;+?;
```

A secret message left by pirates
(“The Gold-Bug” by Edgar Allan Poe)

“The Gold-Bug” Problem


```
53++!305))6*;4826)4+.)4+);806*;4
8!8`60))85;]8*:+*8!83(88)5*!;46(
;88*96*?;8)*+(;485);5*!2:*+(;495
6*2(5*4)8`8*;4069285);)6!8)4++;1
(+9;48081;8:8+1;48!85;4)485!5288
06*81(+9;48;(88;4(+?34;48)4+;161
;:188;+?;
```

Upon seeing the parchment, the narrator remarks, "Were all the jewels of Golconda awaiting me upon my solution of this enigma, I am quite sure that I should be unable to earn them

Why is “;48” so Frequent?

Hint: The message is in English

```
53++!305))6*;4826)4+.)4+);806*48
!8`60))85;]8*:+*8!83(88)5*!46(88
*96*?;8)*+(;485);5*!2:*+(;4956*2
(5*4)8`8*;4069285);)6!8)4++;1(+9
;48081;8:8+1;48!85;4)485
528806*81(+9;48;(88;4(+?34;48)4+
;161;:188;+?;
```

“THE” is the Most Frequent English Word

53++!305))6*THE26)4+.)4+)806*THE
!8`60))85;]8*:+*8!83(88)5*!;46(;
88*96*?;8)*+(THE5);5*!2:*+(;4956
*2(5*4)8`8*;4069285);)6!8)4++;1(
+9THE081;8:8+1THE!85;4)485!52880
6*81(+9THE;(88;4(+?34THE)4+;161;
:188;+?;

Could you Complete Decoding the Message?

أكاديمية كاوست
KAUST ACADEMY

53++!305))6*THE26)H+.)H+)806*THE
!E`60))E5;]E*:+*E!E3(EE)5*!TH6(T
EE*96*?;E)*+(THE5)T5*!2:*+(TH956
*2(5*H)E`E*TH0692E5)T)6!E)H++T1(
+9THE0E1TE:E+1THE!E5T4)HE5!52880
6*E1(+9THET(EETH(+?34THE)H+T161T
:1EET+?T

Operating under the assumption that DNA is a language of its own, let's borrow Legrand's method and see if we can find any surprisingly frequent "words" within the *ori* of *Vibrio cholerae*

The Hidden Message Problem Revisited

أكاديمية كاوست
KAUST ACADEMY

Hidden Message Problem. Finding a hidden message in a string.

- **Input.** A string *Text* (representing *oriC*).
- **Output.** A hidden message in *Text*.

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

Hint: For various biological signals, certain words appear surprisingly frequently in small regions of the genome.

AATTT is a surprisingly frequent 5-mer in:

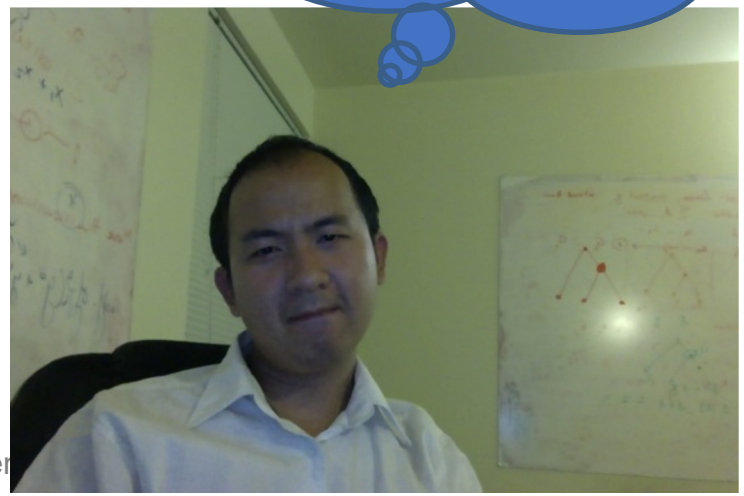
ACAAATTTGCATAATTTCGGGAAATTTCCT

The Frequent Words Problem

Frequent Words Problem. Finding most frequent *k*-mers in a string.

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

This is better, but where is the definition of "a most frequent *k*-mer?"

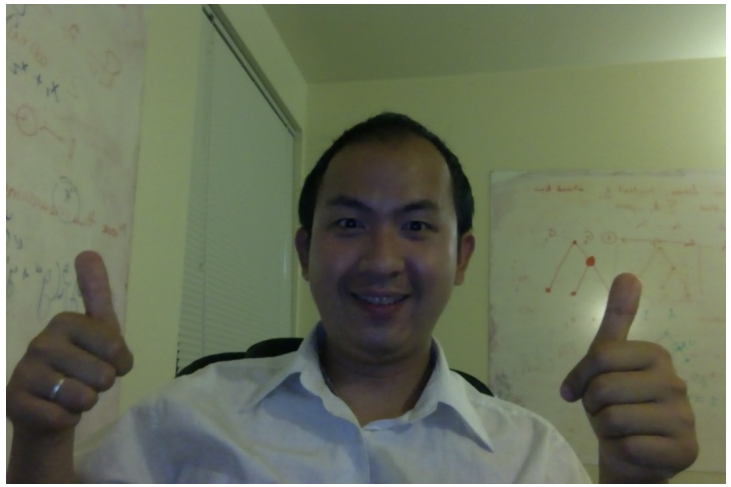


We will use the term k-mer to refer to a string of length k and define Count(Text, Pattern) as the number of times that a k-mer Pattern appears as a substring of Text.

The Frequent Words Problem

Frequent Words Problem. Finding most frequent *k*-mers in a string.

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



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

AATTT is a most frequent 5-mer in:

ACA**AATTT**GCAT**AATTT**CGGGA**AATTT**CCT

Count(ACAACTATGCATACTATCGGGAACTATCCT, ACTAT) = 3

Does the Frequent Words Problem Make Sense to Biologists?

Frequent Words Problem. Finding most frequent *k*-mers in a string.

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

Replication is performed by **DNA polymerase** and the initiation of replication is mediated by a protein called ***DnaA***.

DnaA binds to short (typically 9 nucleotides long) segments within the replication origin known as a ***DnaA* box**.

A *DnaA* box is a hidden message telling *DnaA*: "**bind here!**" And *DnaA* wants to see multiple *DnaA* boxes.

What is the Runtime of Your Algorithm?

Frequent Words Problem. Finding most frequent *k*-mers in a string.

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

- $|Text|^2 \cdot k$
- $4^k + |Text| \cdot k$
- $|Text| \cdot k \cdot \log(|Text|)$
- $|Text|$

???

You will later see how a **naive and slow** algorithm with $|Text|^2 \cdot k$ runtime can be turned into a **fast** algorithm with $|Text|$ runtime

($|Text|$ stands for the length of string ***Text***)

Code Challenge:
Implement **PatternCount()**


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

Let's refresh our foundational knowledge of Python.

Python

- Python Syntax
- String and Console Output
- Conditionals and control flow
- Functions
- Lists and Dictionaries

www.codecademy.com

Code Challenge:
Implement **PatternCount()**

أكاديمية كاوست
KAUST ACADEMY

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

Code Challenge:
Implement **PatternCount()**


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

```python
import sys

# Please do not remove package declarations because these are used by the autograder.

# Insert your PatternCount function here, along with any subroutines you need
def pattern_count(text: str, pattern: str) -> int:
        count = 0
        overlap = len(text) - len(pattern) + 1
        for i in range(overlap):
            start = i
            end = i + len(pattern)
            if text[start:end] == pattern:
                count += 1

        return count
```

The Frequent Words Problem

We say that *Pattern* is a **most frequent *k*-mer** in *Text* if it maximizes *Count*(*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.

The Frequent Words Problem

A straightforward algorithm for finding the most frequent k-mers in a string Text checks all k-mers appearing in this string (there are |Text| − k + 1 such k-mers) and then computes how many times each k-mer appears in Text.

To implement this algorithm, called FrequentWords(), we will need to generate an array Count, where Count(i) stores Count(Text, Pattern) for Pattern = Text(i, k)

Text	**A**	**C**	**T**	G	**A**	**C**	**T**	C	C	C	A	C	C	C	C
COUNT	**2**	1	1	1	**2**	1	1	3	1	1	1	3	3		

Figure: The array *Count* for *Text* = ACTGACTCCCACCCC and *k* = 3. For example, *Count*(0) = *Count*(4) = 2 because ACT (shown in boldface) appears twice in *Text*.

The Frequent Words Problem


```
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
```

STOP and Think: Take another look at the pseudocode for **FrequentWords()**. How fast is this algorithm?

The Frequent Words Problem


```
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
```

Although **FrequentWords()** finds most frequent *k*-mers, it is not very efficient. Each call to **PatternCount**(*Text*, *Pattern*) checks whether the *k*-mer *Pattern* appears in position 0 of *Text*, position 1 of *Text*, and so on. Since each *k*-mer requires $|Text| - k + 1$ such checks, each one requiring as many as *k* comparisons, the overall number of steps of **PatternCount**(*Text*, *Pattern*) is $(|Text| - k + 1) \cdot k$.
Furthermore, **FrequentWords()** must call **PatternCount()** $|Text| - k + 1$ times (once for each *k*-mer of *Text*), so that its overall number of steps is $(|Text| - k + 1) \cdot (|Text| - k + 1) \cdot k$. To simplify the matter, computer scientists often say that the runtime of **FrequentWords()** has an upper bound of $|Text|^2 \cdot k$ steps and refer to the **complexity** of this algorithm as $O(|Text|^2 \cdot k)$.

A Faster Frequent Words Approach

If you were to solve the Frequent Words Problem by hand for a small example, you would probably form a table like the one in the figure below for *Text* equal to "ACGTTTCACGTTTTACGG" and *k* equal to 3.

You would slide a length-*k* window *Text*, and if the current *k*-mer substring of *text* does not occur in the table, then you would create a new entry for it.

Otherwise, you would add 1 to the entry corresponding to the current *k*-mer substring of *Text*. We call this table the **frequency table** for *Text* and *k*.

ACG	CGT	GTT	TTT	TTC	TCA	CAC	TTA	TAC	CGG
3	2	2	3	1	1	1	1	1	1

A Faster Frequent Words Approach


```
FrequencyTable(Text, k)
    freqMap ← empty map
    n ← |Text|
    for i ← 0 to n - k
        Pattern ← Text(i, k)
        if freqMap[Pattern] doesn't exist
            freqMap[Pattern]← 1
        else
           freqMap[pattern] ←freqMap[pattern]+1
    return freqMap
```

```
BetterFrequentWords(Text, k)
    FrequentPatterns ← an array of strings of length 0
    freqMap ← FrequencyTable(Text, k)
    max ← MaxMap(freqMap)
    for all strings Pattern in freqMap
        if freqMap[pattern] = max
            append Pattern to frequentPatterns
    return frequentPatterns
```

Code Challenge:

Solve the Frequent Words Problem.

أكاديمية كاوست
KAUST ACADEMY

ROSALIND

Code Challenge:

Solve the Frequent Words Problem.


```python
# Please do not remove package declarations because these are used by the autograder.
def pattern_count(text: str, pattern: str) -> int:
        count = 0
        overlap = len(text) - len(pattern) + 1
        for i in range(overlap):
            start = i
            end = i + len(pattern)
            if text[start:end] == pattern:
                count += 1

        return count
# Insert your frequent_words function here, along with any subroutines you need
def frequent_words(text: str, k: int) -> list[str]:
    """Find the most frequent k-mers in a given text."""
    frequent_patterns = set()
    count = {}

    # iterate through DNA Text and count kmers
    for i in range(len(text) - k):
        pattern = text[i : i + k]
        # add pattern to dictionary
        # key is i, start position of kmer
        # value is count, using PatternCount function from 1A
        count[i] = pattern_count(text, pattern)

        # find maximum count value in dictionary
        max_count = max(count.values())

    # iterate through text again, if count at that position is max count, slice that kmer and add to set
    for position in range(len(text) - k):
        if count[position] == max_count:
            frequent_patterns.add(text[position : position + k])

    return frequent_patterns
```

Frequent Words in *Vibrio cholerae*

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

The table above reveals the most frequent *k*-mers in the *ori* region from *Vibrio cholerae*, along with the number of times that each *k*-mer occurs.

STOP and Think: Do any of the counts in the above table seem surprisingly large?

Frequent Words in *Vibrio cholerae*

-For example, the 9-mer **ATGATCAAG** appears three times in the *ori* region of *Vibrio cholerae*—is it surprising?

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaag**ATGATCAAG**agaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcatgtt
tccttaaccctctatttttacggaaga**ATGATCAAG**ctgctgctcttgatcatcgtttc

We highlight a most frequent 9-mer instead of using some other value of *k* because experiments have revealed that bacterial *DnaA* boxes are usually nine nucleotides long.

Outline

- **Search for Hidden Messages in Replication Origin**
  - What is a Hidden Message in Replication Origin?
  - **Some Hidden Messages are More Surprising than Others**
  - Clumps of Hidden Messages
- **From a Biological Insight toward an Algorithm for Finding Replication Origin**
  - Asymmetry of Replication
  - Why would a computer scientist care about assymetry of replication?
  - Skew Diagrams
  - Finding Frequent Words with Mismatches
  - Open Problems

oriC of *Vibrio cholerae*

```
atcaatgatcaacgtaagcttctaagcatgatcaaggtgctcacacagtttatccacaacctgagtgg
atgacatcaagataggtcgttgtatctccttcctctcgtactctcatgaccacggaaagatgatcaag
agaggatgatttcttggccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagc
gccatattgcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgttctgttt
atcttgttttgactgagacttgttaggatagacggtttttcatcactgactagccaaagccttactct
gcctgacatcgaccgtaaattgataatgaatttacatgcttccgcgacgatttacctcttgatcatcg
atccgattgaagatcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatca
tgtttccttaaccctctatttttacggaagaatgatcaagctgctgctcttgatcatcgtttc
```

Too Many Frequent Words – Which One is a Hidden Message?

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaacctgagtgg
atgacatcaagataggtcgttgtatctccttcctctcgtactctcatgaccacggaaag**ATGATCAAG**
agaggatgatttcttggccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagc
gccatattgcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgttctgttt
atcttgttttgactgagacttgttaggatagacggtttttcatcactgactagccaaagccttactct
gcctgacatcgaccgtaaattgataatgaatttacatgcttccgcgacgatttacct**CTTGATCAT**cg
atccgattgaagatcttcaattgttaattctcttgcctcgactcatagccatgatgagct**CTTGATCA**
Tgtttccttaaccctctatttttacggaaga**ATGATCAAG**ctgctgct**CTTGATCAT**cgtttc

Most frequent 9-mers in this *oriC* (all appear 3 times):
ATGATCAAG, **CTTGATCAT**, **TCTTGGATCA**, **CTCTTGATC**

Is it **STATISTICALLY** surprising to find a 9-mer appearing **3 or more** times within ≈ 500 nucleotides?

Hidden Message Found!

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaacctgagtgg
atgacatcaagataggtcgttgtatctccttcctctcgtactctcatgaccacggaaag**ATGATCAAG**
agaggatgatttcttggccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagc
gccatattgcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgttctgttt
atcttgttttgactgagacttgttaggatagacggtttttcatcactgactagccaaagccttactct
gcctgacatcgaccgtaaattgataatgaatttacatgcttccgcgacgatttacct**CTTGATCAT**cg
atccgattgaagatcttcaattgttaattctcttgcctcgactcatagccatgatgagct**CTTGATCA**
Tgtttccttaaccctctatttttacggaaga**ATGATCAAG**ctgctgct**CTTGATCAT**cgtttc

ATGATCAAG

| | | | | | | | | |

TACTAGTTC

are **reverse complements** and likely ***DnaA*** boxes (***DnaA*** does not care what strand to bind to)

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

Complementary strand on a **template strand**

أكاديمية كاوست
KAUST ACADEMY

- The figure below shows a template strand **AGTCGCATAGT** and its complementary strand **ACTATGCGACT**. The beginning and end of a DNA strand are denoted 5’ (pronounced “five prime”) and 3’ (pronoun

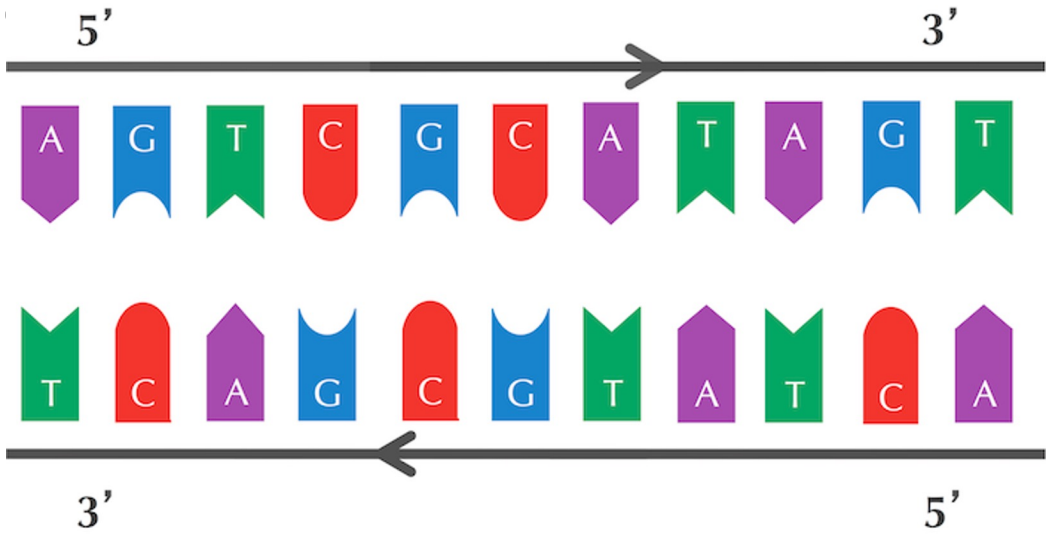
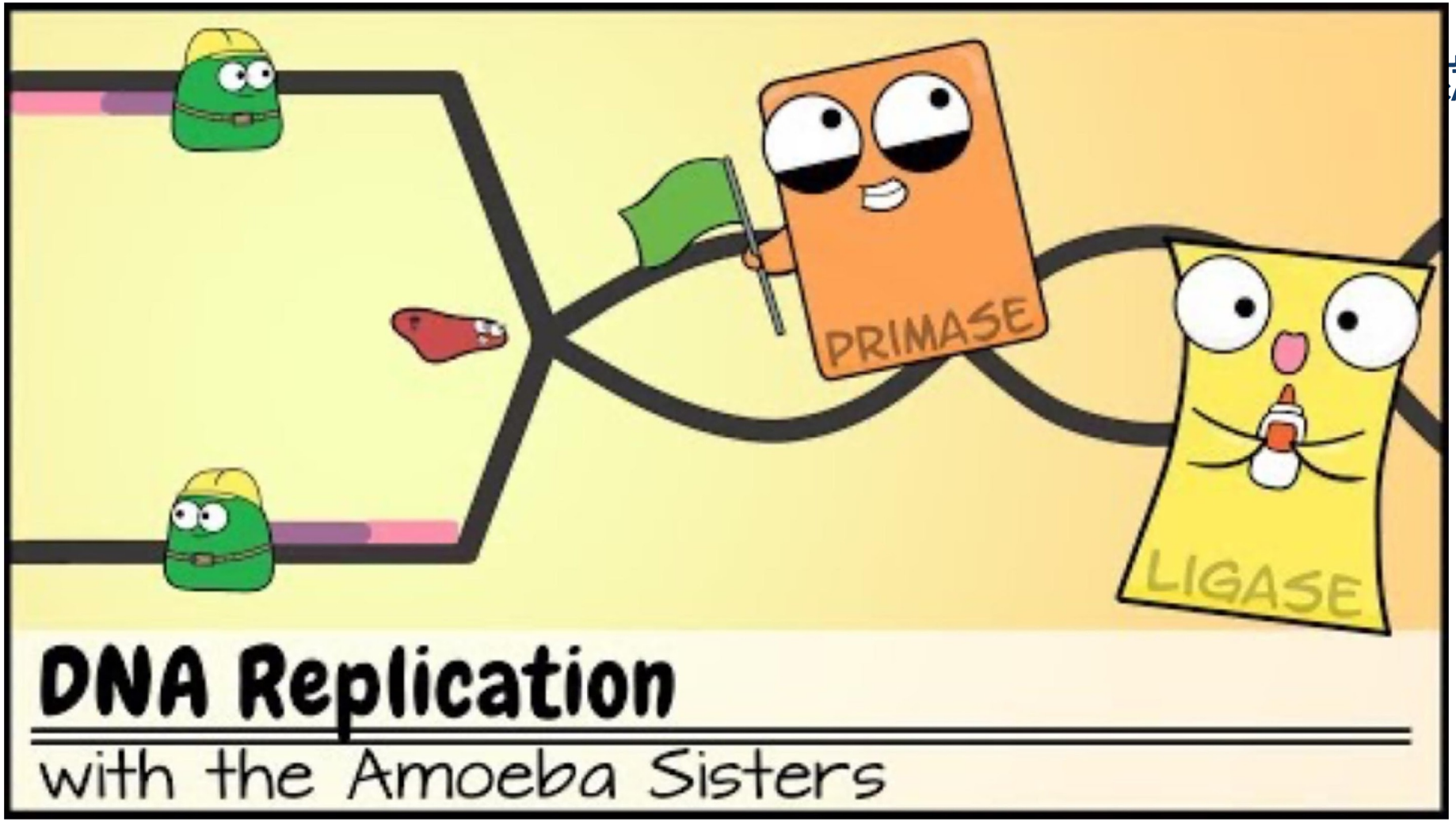


Figure: Complementary strands run in opposite directions. Each strand is read in the 5' → 3' direction.

You may think that we have made a mistake


PRIMASE
LIGASE
DNA Replication
with the Amoeba Sisters

Code Challenge:

Solve the Reverse Complement Problem.

أكاديمية كاوست
KAUST ACADEMY

Reverse Complement Problem: Find the reverse complement of a DNA string.
Input: A DNA string Pattern.
Output: $Pattern_{rc}$, the reverse complement of Pattern.

Code Challenge:

Solve the Reverse Complement Problem.

أكاديمية كاوست
KAUST ACADEMY

Reverse Complement Problem: Find the reverse complement of a DNA string.
Input: A DNA string Pattern.
Output: $Pattern_{rc}$, the reverse complement of Pattern.

```python
import sys

# Please do not remove package declarations because these are used by the autograder.

# Insert your reverse_complement function here, along with any subroutines you need
def reverse_complement(Text: str) -> str:
    """Calculate the reverse complement of a DNA pattern."""
    complement = []
    for i in Text:
        if i == "A":
            complement.append("T")
        elif i == "T":
            complement.append("A")
        elif i == "G":
            complement.append("C")
        elif i == "C":
            complement.append("G")
        else:
            print("Not ACTG!")

    reverse_complement = list(reversed(complement))

    return "".join(reverse_complement)

```

Exercise Break

أكاديمية كاوست
KAUST ACADEMY

Return a space-separated list of starting positions (in increasing order) where **CTTGATCAT** appears as a substring in the *Vibrio cholerae* genome.

Can we Now Find Hidden Messages in *Thermotoga petrophila?*

أكاديمية كاوست
KAUST ACADEMY

```
aactctatacctccttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaactaa
aatggtaggtttggtggtaggttttgtgtacattttgtagtatctgatttttaattacataccgta
tattgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaacctaccaccaaac
tctgtattgaccattttaggacaacttcagggtggtaggtttctgaagctctcatcaatagactat
tttagtctttacaaacaatattaccgttcagattcaagattctacaacgctgttttaatgggcgtt
gcagaaaacttaccacctaaaatccagtatccaagccgatttcagagaaacctaccacttacctac
cacttacctaccacccgggtggtaagttgcagacattattaaaaacctcatcagaagcttgttcaa
aaatttcaatactcgaaacctaccacctgcgtcccctattatttactactactaataatagcagta
taattgatctgaaaagaggtggtaaaaaa
```

No single occurrence of **ATGATCAAG** or **CTTGATCAT** from *Vibrio Cholerae*!!!

Applying the Frequent Words Problem to this replication origin:
AACCTACCA, ACCTACCAC, GGTAGGTTT, TGGTAGGTT, AAACCTACC, CCTACCACC

Different genomes ⍰ different hidden messages (*DnaA* boxes)

Hidden Messages in *Thermotoga petrophila*

aactctatacctccttttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaactaa
aatggtaggtttGGTGGTAGGttttgtgtacattttgtagtatctgatttttaattacataccgta
tattgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaaCCTACCACCaaac
tctgtattgaccattttaggacaacttcagGGTGGTAGGtttctgaagctctcatcaatagactat
tttagtctttacaaacaatattaccgttcagattcaagattctacaacgctgttttaatgggcgtt
gcagaaaacttaccacctaaaatccagtatccaagccgatttcagagaaacctaccacttacctac
cactaCCTACCACCcgggtggtaagttgcagacattattaaaaacctcatcagaagcttgttcaa
aaatttcaatactcgaaaCCTACCACCtgcgtcccctattatttactactactaataatagcagta
taattgatctgaaaagaggtggtaaaaaa

Ori-Finder software confirms that

CCTACCACC

| | | | | | | | | | are candidate hidden messages.

GGATGGTGG

We learned how to find hidden messages **IF *oriC* is given.** But we have no clue **WHERE** *oriC* is located in a (long) genome.

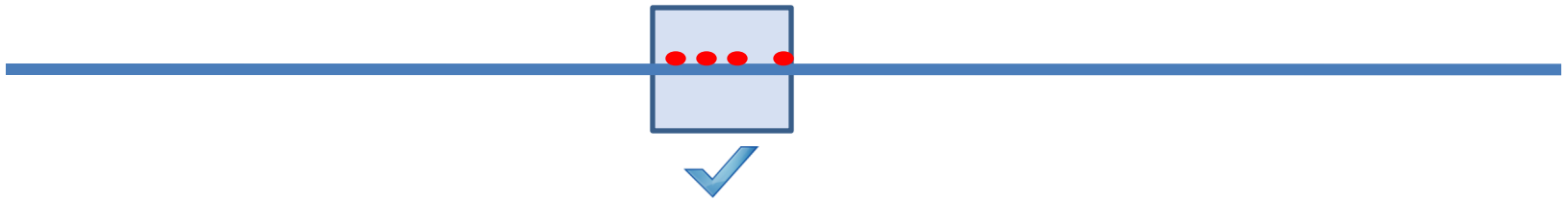
Outline

- **Search for Hidden Messages in Replication Origin**
  - What is a Hidden Message in Replication Origin?
  - Some Hidden Messages are More Surprising than Others
  - **Clumps of Hidden Messages**
- **From a Biological Insight toward an Algorithm for Finding Replication Origin**
  - Asymmetry of Replication
  - Why would a computer scientist care about assymetry of replication?
  - Skew Diagrams
  - Finding Frequent Words with Mismatches
  - Open Problems

Finding Replication Origin

Our strategy **BEFORE**: given a previously **known** *oriC* (a 500-nucleotide window), find **frequent words** (clumps) in *oriC* as candidate *DnaA* boxes.

replication origin → **frequent words**



Finding Replication Origin

أكاديمية كاوست
KAUST ACADEMY

Our strategy **BEFORE**: given previously **known** *oriC* (a 500-nucleotide window), find **frequent words** (clumps) in *oriC* as candidate *DnaA* boxes.

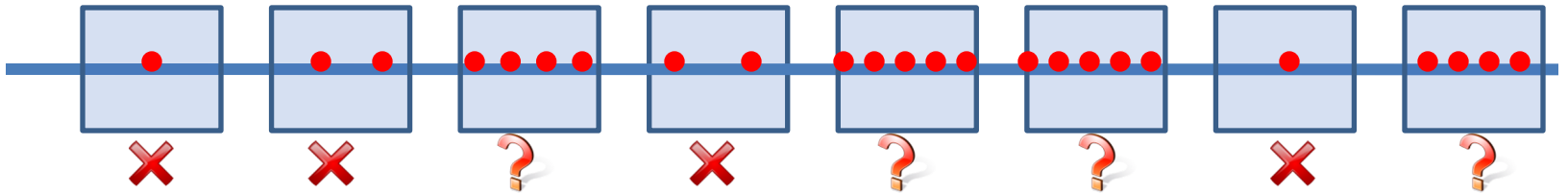
replication origin → **frequent words**

But what if the position of the replication origin within a genome is **unknown**!

Finding Replication Origin

Our strategy **BEFORE**: given previously **known** *oriC* (a 500-nucleotide window), find **frequent words** (clumps) in *oriC* as candidate *DnaA* boxes.

replication origin → **frequent words**



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

frequent words → **replication origin**

What is a Clump?

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

Clump Finding Problem. Find patterns forming clumps in a string.

- **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*.

There exist **1904** *different* 9-mers forming (500,3)-clumps in *E. coli* genome. **It is absolutely unclear which of them point to the replication origin…**

The Clump Finding Problem

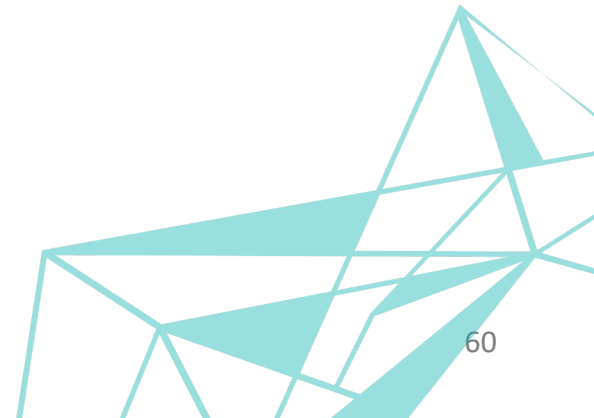

```
FindClumps(Text, k, L, t)
    Patterns ← an array of strings of length 0
    n ← |Text|
    for every integer i between 0 and n - L
        Window ← Text(i, L)
        freqMap ← FrequencyTable(Window, k)
        for every key s in freqMap
            if freqMap[s] ≥ t
                append s to Patterns
    remove duplicates from Patterns
    return Patterns
```

Code Challenge:

Solve the Clump Finding Problem

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*.



Exercise Break

How many *different* 9-mers form (500,3)-clumps in the *E. coli* genome? (In other words, do not count a 9-mer more than once.)