Finding Hidden Messages in DNA

Phillip Compeau & Pavel Pevzner

http://bioinformaticsalgorithms.org

Active Learning Publishers

Welcome!

Thank you for joining us! As you explore this book, you will find a number of active learning components that help you learn the material at your own pace.

1. **CODE CHALLENGES** ask you to implement the algorithms that you will encounter (in any programming language you like). These code challenges are hosted in the "Bioinformatics Textbook Track" location on Rosalind (http://rosalind.info), a website that will automatically test your implementations.

2. **CHARGING STATIONS** provide additional insights on implementing the algorithms you encounter. However, we suggest trying to solve a Code Challenge before you visit a Charging Station.

3. **EXERCISE BREAKS** offer "just in time" assessments testing your understanding of a topic before moving to the next one.

4. **STOP and Think** questions invite you to slow down and contemplate the current material before continuing to the next topic.

5. **DETOURS** provide extra content that didn't quite fit in the main text.

6. **FINAL CHALLENGES** ask you to apply what you have learned to real experimental datasets.

This textbook powers our popular online course on Coursera. We encourage you to sign up for a session and learn this material while interacting with thousands of other talented students from around the world. You can also find lecture videos and PowerPoint slides at the textbook website, http://bioinformaticsalgorithms.org.

Active Learning Publishers
9768 Claiborne Square
La Jolla, CA 92037

To my family. — P. C.

To my parents. — P. P.

Contents

List of Code Challenges

About the Textbook

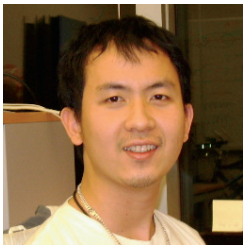
Meet the Authors

PHILLIP COMPEAU is an Assistant Teaching Professor in the Computational Biology Department at Carnegie Mellon University. He is a former postdoctoral researcher in the Department of Computer Science & Engineering at the University of California, San Diego, where he received a Ph. D. in mathematics. He is passionate about the future of both offline and online education, having cofounded Rosalind with Nikolay Vyahhi in 2012. A retired tennis player, he dreams of one day going pro in golf.

PAVEL PEVZNER is Ronald R. Taylor Professor of Computer Science at the University of California, San Diego. He holds a Ph. D. from Moscow Institute of Physics and Technology, Russia. He is a Howard Hughes Medical Institute Professor (2006), an Association for Computing Machinery Fellow (2010), and an International Society for Computational Biology Fellow (2012). He has also authored the textbooks *Computational Molecular Biology: An Algorithmic Approach* (2000) and *An Introduction to Bioinformatics Algorithms* (2004) (jointly with Neil Jones).

Meet the Development Team

OLGA BOTVINNIK is a Ph.D. candidate in Bioinformatics and Systems Biology at the University of California, San Diego. She holds an M.S. in Bioinformatics from University of California, Santa Cruz and B.S. degrees in Mathematics and Biological Engineering from the Massachusetts Institute of Technology. Her research interests are data visualization and single-cell transcriptomics. She enjoys yoga, photography, and playing cello.



SON PHAM is a postdoctoral researcher at Salk Institute in La Jolla, California. He holds a Ph. D. in Computer Science and Engineering from the University of California, San Diego and an M.S. in Applied Mathematics from St. Petersburg State University, Russia. His research interests include graph theory, genome assembly, comparative genomics, and neuroscience. Besides research, he enjoys walking meditation, gardening, and trying to catch the big one.

NIKOLAY VYAHHI coordinates the M.S. Program in Bioinformatics in the Academic University of St. Petersburg, Russian Academy of Sciences. In 2012, he cofounded the Rosalind online bioinformatics education project. He recently founded the Bioinformatics Institute in St. Petersburg as well as Stepic, a company focusing on content delivery for online education.

KAI ZHANG is a Ph. D. candidate in Bioinformatics and Systems Biology at the University of California, San Diego. He holds an M.S. in Molecular Biology from Xiamen University, China. His research interests include epigenetics, gene regulatory networks, and machine learning algorithms. Besides research, Kai likes basketball and music.

Acknowledgments

This textbook was greatly improved by the efforts of a large number of individuals, to whom we owe a debt of gratitude.

The development team (Olga Botvinnik, Son Pham, Nikolay Vyahhi, and Kai Zhang), as well as Laurence Bernstein and Ksenia Krasheninnikova, implemented coding challenges and exercises, rendered figures, helped typeset the text, and offered insightful feedback on the manuscript.

Glenn Tesler provided thorough chapter reviews and even implemented some software to catch errors in the early version of the manuscript!

Robin Betz, Petar Ivanov, James Jensen, and Yu Lin provided insightful comments on the manuscript in its early stages. David Robinson was kind enough to copy edit a few chapters and palliate our punctuation maladies.

Randall Christopher brought to life our ideas for illustrations in addition to the textbook cover.

Andrey Grigoriev gave advice on the content of Chapter 1. Martin Tompa helped us develop the narrative in Chapter 2 by suggesting that we analyze latent tuberculosis infection.

Nikolay Vyahhi led a team composed of Andrey Balandin, Artem Suschev, Aleksey Kladov, and Kirill Shikhanov, who worked hard to support an online, interactive version of this textbook used in our online course on Coursera.

Our students on Coursera, especially Mark Mammel and Dmitry Kuzminov, found hundreds of typos in our preliminary manuscript.

Mikhail Gelfand, Uri Keich, Hosein Mohimani, Son Pham, and Glenn Tesler advised us on some of the book's "Open Problems" and led Massive Open Online Research projects (MOORs) in the first session of our online course.

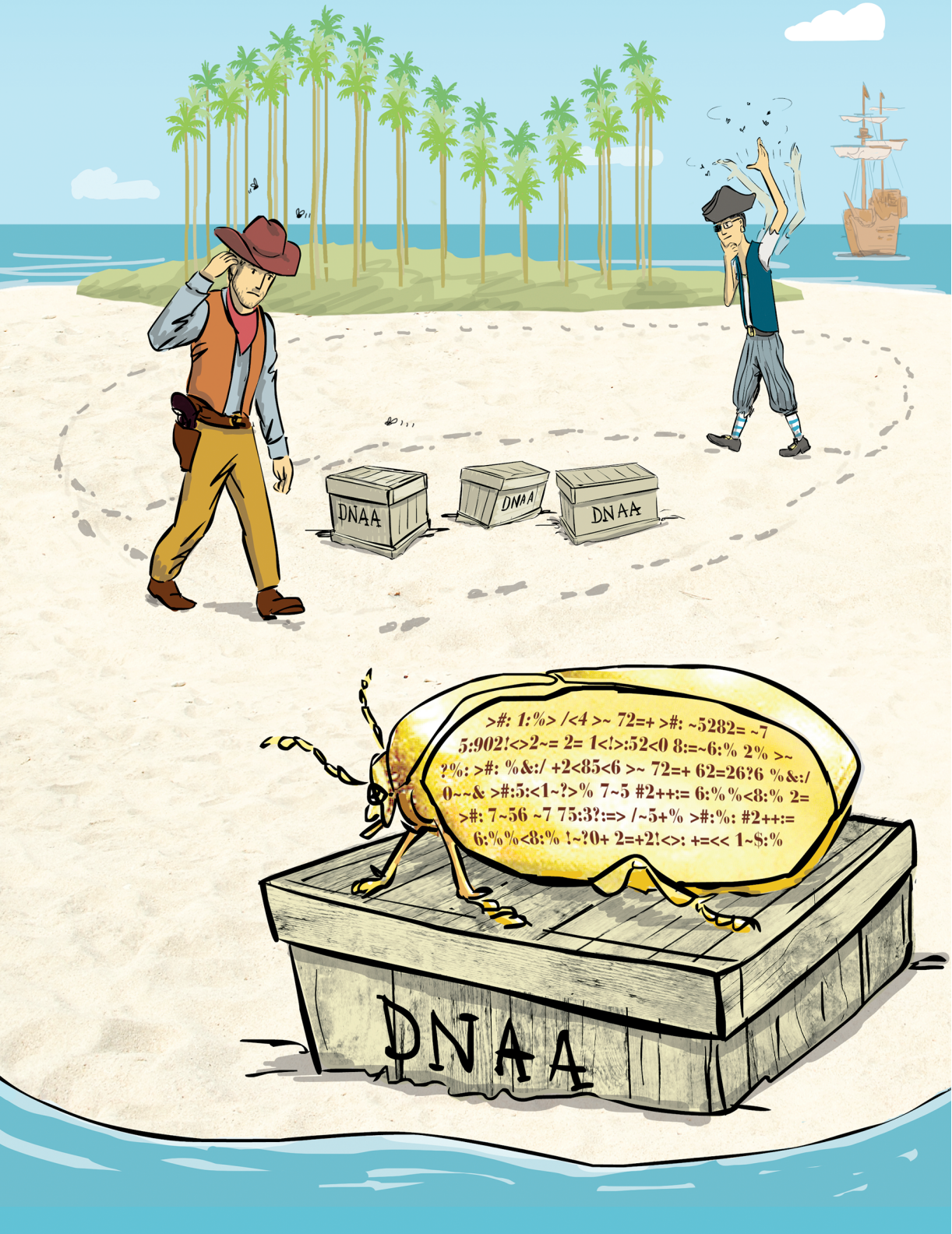
Howard Hughes Medical Institute and the Russian Ministry of Education and Science generously gave their support for the development of the online courses based on this textbook. The Bioinformatics and Systems Biology Program and the Computer Science & Engineering Department at the University of California, San Diego provided additional support.

Finally, our families gracefully endured the many long days and nights that we spent poring over manuscripts, and they helped us preserve our sanity along the way.

P. C. and P. P.
San Diego
July 2015

Where in the Genome Does DNA Replication Begin?

Algorithmic Warmup



A Journey of a Thousand Miles...

Genome replication is one of the most important tasks carried out in the cell. Before a cell can divide, it must first replicate its genome so that each of the two daughter cells inherits its own copy. In 1953, James Watson and Francis Crick completed their landmark paper on the DNA double helix with a now-famous phrase:

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

They conjectured that 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, as shown in Figure 1.1.

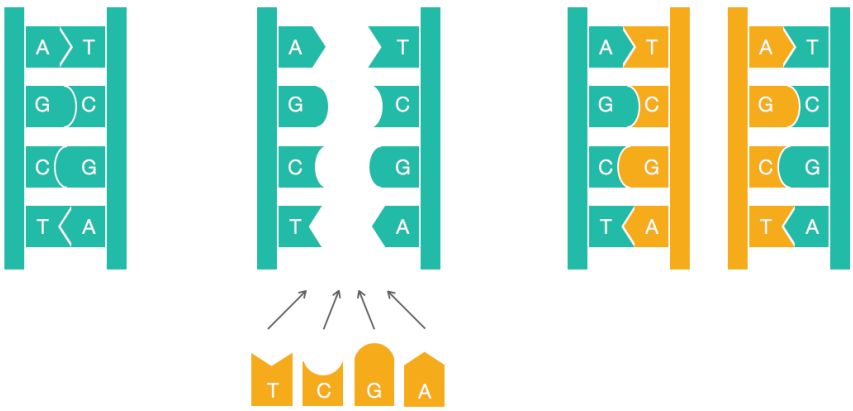


FIGURE 1.1 A naive view of DNA replication. Nucleotides adenine (A) and thymine (T) are complements of each other, as are cytosine (C) and guanine (G). Complementary nucleotides bind to each other in DNA.

Although Figure 1.1 models DNA replication on a simple level, the details of replication turned out to be much more intricate than Watson and Crick imagined; as we will see, an astounding amount of molecular logistics is required to ensure DNA replication.

At first glance, a computer scientist might not imagine that these details have any computational relevance. To mimic the process in Figure 1.1 algorithmically, we only need to take a string representing the genome and return a copy of it! Yet if we take

the time to review the underlying biological process, we will be rewarded with new algorithmic insights into analyzing replication.

Replication begins in a genomic region called the **replication origin** (denoted *oriC*) and is performed by molecular copy machines called **DNA polymerases**. Locating *oriC* presents an important task not only for understanding how cells replicate but also for various biomedical problems. For example, some gene therapy methods use genetically engineered mini-genomes, which are called **viral vectors** because they are able to penetrate cell walls (just like real viruses). Viral vectors carrying artificial genes have been used in agriculture, such as to engineer frost-resistant tomatoes and pesticide-resistant corn. In 1990, gene therapy was first successfully performed on humans when it saved the life of a four-year-old girl suffering from Severe Combined Immunodeficiency Disorder; the girl had been so vulnerable to infections that she was forced to live in a sterile environment.

The idea of gene therapy is to intentionally infect a patient who lacks a crucial gene with a viral vector containing an artificial gene that encodes a therapeutic protein. Once inside the cell, the vector replicates and eventually produces many copies of the therapeutic protein, which in turn treats the patient's disease. To ensure that the vector actually replicates inside the cell, biologists must know where *oriC* is in the vector's genome and ensure that the genetic manipulations that they perform do not affect it.

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

Finding Origin of Replication Problem:

Input: A DNA string *Genome*.
Output: The location of *oriC* in *Genome*.

STOP and Think: Does this biological problem represent a clearly stated computational problem?

Although the Finding Origin of Replication Problem asks a legitimate biological question, it does not present a well-defined computational problem. Indeed, biologists would immediately plan an experiment to locate *oriC*: for example, they might delete various short segments from the genome in an effort to find a segment whose deletion

stops replication. Computer scientists, on the other hand, would shake their heads and demand more information before they can even start thinking about the problem.

Why should biologists care what computer scientists think? Computational methods are now the only realistic way to answer many questions in modern biology. 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 *oriC* prediction are rather time consuming. As a result, *oriC* has only been experimentally located in a handful of species. Thus, we would like to design a computational approach to find *oriC* so that biologists are free to spend their time and money on other tasks.

Hidden Messages in the Replication Origin

DnaA boxes

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

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

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 *oriC* 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 *oriC* 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 *oriC*? This discussion motivates the following problem.

Hidden Message Problem:
Find a "hidden message" in the replication origin.

Input: A string *Text* (representing the replication origin of a genome).
Output: A hidden message in *Text*.

STOP and Think: Does this problem represent a clearly stated computational problem?

Hidden messages in "The Gold-Bug"

Although the Hidden Message Problem poses a legitimate intuitive question, it again makes absolutely no sense to a computer scientist because the notion of a "hidden message" is not precisely defined. The *oriC* 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 following:

```
53++!305))6*;4826)4+.)4+);806*;48!8‘60))85;1+(;:+*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;+?;
```

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". Legrand retorts, "It may well be doubted whether human ingenuity can construct an enigma of the kind which human ingenuity may not, by proper application, resolve". He reasons that the three consecutive symbols "**;48**" appear with surprising frequency on the parchment:

```
53++!305))6*;48826)4+.)4+);806*;48!8‘60))85;1+(;:+*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;+?;
```

Legrand had already deduced that the pirates spoke English; he therefore assumed that the high frequency of ";48" implied that it encodes the most frequent English word, "THE". Substituting each symbol, Legrand had a slightly easier text to decipher, which would eventually lead him to the buried treasure. Can you decode this message too?

```
53++!305))6*THE26)H+.)H+)TE06*THE!E‘60))E5T1+(T:+*E
!E3(EE)5*!TH6(TEE*96*?TE)*+(THE5)T5*!2:*+(TH956*2(5
*-H)E‘E*TH0692E5)T)6!E)H++T1(+9THE0E1TE:E+1THE!E5TH
)HE5!52EE06*E1(+9THET(EETH(+?3HTHE)H+T161T:1EET+?T
```

Counting words

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 *oriC* of *Vibrio cholerae*. We have added reason to look for frequent words in the *oriC* because for various biological processes, certain nucleotide strings often appear surprisingly often in small regions of the genome. For example, **ACTAT** is a surprisingly frequent substring of

ACA**ACTAT**GCAT**ACTAT**CGGGA**ACTAT**CCT.

We 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*. Following the above example,

COUNT(ACA**ACTAT**GCAT**ACTAT**CGGGA**ACTAT**CCT, **ACTAT**) $= 3$.

Note that COUNT(CG**ATATA**TCC**ATA**G, **ATA**) is equal to 3 (not 2) since we should account for overlapping occurrences of *Pattern* in *Text*.

To compute COUNT(*Text*, *Pattern*), our plan is to "slide a window" down *Text*, checking whether each k-mer substring of *Text* matches *Pattern*. We will therefore refer to the k-mer starting at position i of *Text* as $Text(i, k)$. Throughout this book, we will often use **0-based indexing**, meaning that we count starting at 0 instead of 1. In this case, *Text* begins at position 0 and ends at position $|Text| - 1$ ($|Text|$ denotes the number of symbols in *Text*). For example, if $Text =$ GACCATACTG, then $Text(4, 3) =$ ATA. Note that the last k-mer of *Text* begins at position $|Text| - k$, e.g., the last 3-mer of GACCATACTG starts at position $10 - 3 = 7$. This discussion results in the following pseudocode for computing COUNT(*Text*, *Pattern*).

```
PatternCount(Text, Pattern)
    count ← 0
    for i ← 0 to |Text| − |Pattern|
        if Text(i, |Pattern|) = Pattern
            count ← 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 for *Text* = ACA**ACTAT**GCAT**ACTAT**CGGGA**ACTAT**CCT, and **ATA** is a most frequent 3-mer for *Text* = CG**ATATA**TCC**ATA**G.

STOP and Think: Can a string have multiple most frequent *k*-mers?

We now have a rigorously defined computational problem.

Frequent Words Problem:
Find the most frequent k-mers in a string.

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

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) (see Figure 1.2).

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 1.2 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* at positions 0 and 4.

```
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: How fast is **FrequentWords**?

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 $\mathcal{O}(|Text|^2 \cdot k)$ (see **DETOUR: Big-O Notation**).

PAGE 52

CHARGING STATION (The Frequency Array): If $|Text|$ and k are small, as is the case when looking for *DnaA* boxes in the typical bacterial *oriC*, then an algorithm with running time of $\mathcal{O}(|Text|^2 \cdot k)$ is perfectly acceptable. But once we find some new biological application requiring us to solve the Frequent Words Problem for a very long $Text$, we will quickly run into trouble. Check out this Charging Station to learn about solving the Frequent Words Problem using a frequency array, a data structure that will also help us solve new coding challenges later in the chapter.

Frequent words in Vibrio cholerae

Figure 1.3 reveals the most frequent *k*-mers in the *oriC* region from *Vibrio cholerae*.

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

FIGURE 1.3 The most frequent *k*-mers in the *oriC* region of *Vibrio cholerae* for *k* from 3 to 9, along with the number of times that each *k*-mer occurs.

STOP and Think: Do any of the counts in Figure 1.3 seem surprisingly large?

For example, the 9-mer **ATGATCAAG** appears three times in the *oriC* 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. The probability that there exists a 9-mer appearing three or more times in a randomly generated DNA string of length 500 is approximately 1/1300 (see **DETOUR: Probabilities of Patterns in a String**). In fact, there are four different 9-mers repeated three or more times in this region: **ATGATCAAG**, CTTGATCAT, TCTTGATCA, and CTCTTGATC.

PAGE 52

The low likelihood of witnessing even one repeated 9-mer in the *oriC* region of *Vibrio cholerae* leads us to the working hypothesis that one of these four 9-mers may represent a potential *DnaA* box that, when appearing multiple times in a short region, jump-starts replication. But which one?

STOP and Think: Is any one of the four most frequent 9-mers in the *oriC* of *Vibrio cholerae* "more surprising" than the others?

Some Hidden Messages are More Surprising than Others

Recall that nucleotides **A** and **T** are complements of each other, as are **C** and **G**. Having one strand of DNA and a supply of "free floating" nucleotides as shown in Figure 1.1, one can imagine the synthesis of a **complementary strand** on a **template strand**. This model of replication was confirmed by Meselson and Stahl in 1958 (see **DETOUR: The Most Beautiful Experiment in Biology**). Figure 1.4 shows a template strand **AGTCGCATAGT** and its complementary strand **ACTATGCGACT**.

PAGE 57

At this point, you may think that we have made a mistake, since the complementary strand in Figure 1.4 reads out **TCAGCGTATCA** from left to right rather than **ACTATGCGACT**. We have not: each DNA strand has a direction, and the complementary strand runs in the opposite direction to the template strand, as shown by the arrows in Figure 1.4. Each strand is read in the 5′ → 3′ direction (see **DETOUR: Directionality of DNA Strands** to learn why biologists refer to the beginning and end of a strand of DNA using the terms 5′ and 3′).

PAGE 59

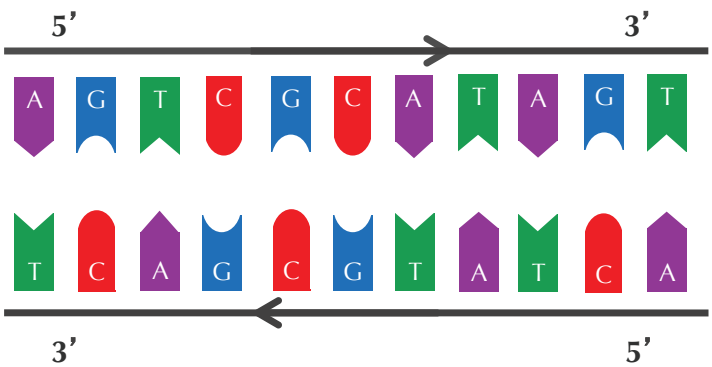


FIGURE 1.4 Complementary strands run in opposite directions.

Given a nucleotide p, we denote its complementary nucleotide as $\overline{p}$. The **reverse complement** of a string $Pattern = p_1 \cdots p_n$ is the string $\overline{Pattern} = \overline{p_n} \cdots \overline{p_1}$ formed by taking the complement of each nucleotide in *Pattern*, then reversing the resulting string. We will need the solution to the following problem throughout this chapter.

Reverse Complement Problem:
Find the reverse complement of a DNA string.

Input: A DNA string *Pattern*.
Output: $\overline{Pattern}$, the reverse complement of *Pattern*.

STOP and Think: Look again at the four most frequent 9-mers in the *oriC* region of *Vibrio cholerae* from Figure 1.3. Now do you notice anything surprising?

Interestingly, among the four most frequent 9-mers in the *oriC* region of *Vibrio cholerae*, **ATGATCAAG** and **CTTGATCAT** are reverse complements of each other, resulting in the following six occurrences of these strings.

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaag**ATGATCAAG**agaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacct**CTTGATCAT**cgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagct**CTTGATCAT**gtt
tccttaaccctctatttttacggaaga**ATGATCAAG**ctgctgct**CTTGATCAT**cgtttc

Finding a 9-mer that appears six times (either as itself or as its reverse complement) in a DNA string of length 500 is far more surprising than finding a 9-mer that appears three times (as itself). This observation leads us to the working hypothesis that **ATGATCAAG** and its reverse complement **CTTGATCAT** indeed represent *DnaA* boxes in *Vibrio cholerae*. This computational conclusion makes sense biologically because the *DnaA* protein that binds to *DnaA* boxes and initiates replication does not care which of the two strands it binds to. Thus, for our purposes, both **ATGATCAAG** and **CTTGATCAT** represent *DnaA* boxes.

However, before concluding that we have found the *DnaA* box of *Vibrio cholerae*, the careful bioinformatician should check if there are other short regions in the *Vibrio cholerae* genome exhibiting multiple occurrences of **ATGATCAAG** (or **CTTGATCAT**). After all, maybe these strings occur as repeats throughout the entire *Vibrio cholerae* genome, rather than just in the *oriC* region. To this end, we need to solve the following problem.

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.

After solving the Pattern Matching Problem, we discover that **ATGATCAAG** appears 17 times in the following positions of the *Vibrio cholerae* genome:

116556, 149355, **151913**, **152013**, **152394**, 186189, 194276, 200076, 224527, 307692, 479770, 610980, 653338, 679985, 768828, 878903, 985368

With the exception of the three occurrences of **ATGATCAAG** in *oriC* at starting positions **151913**, **152013**, and **152394**, no other instances of **ATGATCAAG** form **clumps**, i.e., appear close to each other in a small region of the genome. You may check that the same conclusion is reached when searching for **CTTGATCAT**. We now have strong statistical evidence that **ATGATCAAG**/**CTTGATCAT** may represent the hidden message to *DnaA* to start replication.

STOP and Think: Can we conclude that **ATGATCAAG**/**CTTGATCAT** also represents a *DnaA* box in other bacterial genomes?

An Explosion of Hidden Messages

Looking for hidden messages in multiple genomes

We should not jump to the conclusion that **ATGATCAAG**/**CTTGATCAT** is a hidden message for all bacterial genomes without first checking whether it even appears in known *oriC* regions from other bacteria. After all, maybe the clumping effect of **ATGATCAAG**/**CTTGATCAT** in the *oriC* region of *Vibrio cholerae* is simply a statistical fluke that has nothing to do with replication. Or maybe different bacteria have different *DnaA* boxes . . .

Let's check the proposed *oriC* region of *Thermotoga petrophila*, a bacterium that thrives in extremely hot environments; its name derives from its discovery in the water beneath oil reservoirs, where temperatures can exceed 80° Celsius.

```
aactctatacctccttttttgtcgaatttgtgtgatttatagagaaaatcttattaactga
aactaaaatggtaggtttggtggtaggttttgtgtacattttgtagtatctgatttttaa
ttacataccgtatattgtattaaattgacgaacaattgcatggaattgaatatatgcaaa
acaaacctaccaccaaactctgtattgaccattttaggacaacttcagggtggtaggttt
ctgaagctctcatcaatagactattttagtctttacaaacaatattaccgttcagattca
agattctacaacgctgttttaatgggcgttgcagaaaacttaccacctaaaatccagtat
ccaagccgatttcagagaaacctaccacttacctaccacttacctaccacccgggtggta
agttgcagacattattaaaaacctcatcagaagcttgttcaaaaatttcaatactcgaaa
cctaccacctgcgtcccctattatttactactactaataatagcagtataattgatctga
```

This region does not contain a single occurrence of **ATGATCAAG** or **CTTGATCAT**! Thus, different bacteria may use different *DnaA* boxes as "hidden messages" to the *DnaA* protein.

Application of the Frequent Words Problem to the *oriC* region above reveals that the following six 9-mers appear in this region three or more times:

AACCTACCA	AAACCTACC	ACCTACCAC
CCTACCACC	GGTAGGTTT	TGGTAGGTT

Something peculiar must be happening because it is extremely unlikely that six different 9-mers will occur so frequently within the same short region in a random string. We will cheat a little and consult with Ori-Finder, a software tool for finding replication origins in DNA sequences. This software chooses **CCTACCACC** (along with its reverse complement **GGTGGTAGG**) as a working hypothesis for the *DnaA* box in *Thermotoga petrophila*. Together, these two complementary 9-mers appear five times in the replication origin:

```
aactctatacctccttttttgtcgaatttgtgtgatttatagagaaaatcttattaactga
aactaaaatggtaggtttGGTGGTAGGttttgtgtacattttgtagtatctgatttttaa
ttacataccgtatattgtattaaattgacgaacaattgcatggaattgaatatatgcaaa
acaaaCCTACCACCaaactctgtattgaccattttaggacaacttcagGGTGGTAGGttt
ctgaagctctcatcaatagactattttagtctttacaaacaatattaccgttcagattca
agattctacaacgctgttttaatgggcgttgcagaaaacttaccacctaaaatccagtat
ccaagccgatttcagagaaacctaccacttacctaccacttaCCTACCACCcgggtggta
agttgcagacattattaaaaacctcatcagaagcttgttcaaaaatttcaatactcgaaa
CCTACCACCtgcgtcccctattatttactactactaataatagcagtataattgatctga
```

The Clump Finding Problem

Now imagine that you are trying to find *oriC* in a newly sequenced bacterial genome. Searching for "clumps" of **ATGATCAAG**/**CTTGATCAT** or **CCTACCACC**/**GGTGGTAGG** is unlikely to help, since this new genome may use a completely different hidden message! Before we lose all hope, 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. Hopefully, the locations of these clumps will shed light on the location of *oriC*.

Our plan is to slide a window of fixed length L along the genome, looking for a region where a k-mer appears several times in short succession. The parameter value $L = 500$ reflects the typical length of *oriC* in bacterial genomes.

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. (This definition assumes that the k-mer *completely* fits within the interval.) For example, **TGCA** forms a $(25, 3)$-clump in the following *Genome*:

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

From our previous examples of *oriC* regions, **ATGATCAAG** forms a $(500, 3)$-clump in the *Vibrio cholerae* genome, and **CCTACCACC** forms a $(500, 3)$-clump in the *Thermotoga petrophila* genome. We are now ready to formulate the following 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*.

CHARGING STATION (Solving the Clump Finding Problem): You can solve the Clump Finding Problem by simply applying your algorithm for the Frequent Words Problem to each window of length L in *Genome*. However, if your algorithm for the Frequent Words Problem is not very efficient, then such an approach may be impractical. For example, recall that FREQUENTWORDS has $\mathcal{O}(L^2 \cdot k)$ running time. Applying this algorithm to each window of length L in *Genome* will result in an algorithm with $\mathcal{O}(L^2 \cdot k \cdot |Genome|)$ running time. Moreover, even if we use a faster algorithm for the Frequent Words Problem (like the one described when we introduce a frequency array on page 39), the running time remains high when we try to analyze a bacterial — let alone human — genome. Check out this Charging Station to learn about a more efficient approach for solving the Clump Finding Problem.

Let's look for clumps in the *Escherichia coli* (*E. coli*) genome, the workhorse of bacterial genomics. We find hundreds of different 9-mers forming $(500, 3)$-clumps in the *E. coli* genome, and it is absolutely unclear which of these 9-mers might represent a *DnaA* box in the bacterium's *oriC* region.

STOP and Think: Should we give up? If not, what would you do now?

At this point, an unseasoned researcher might give up, since it appears that we do not have enough information to locate *oriC* in *E. coli*. But a fearless veteran bioinformatician would try to learn more about the details of replication in the hope that they provide new algorithmic insights into finding *oriC*.

The Simplest Way to Replicate DNA

We are now ready to discuss the replication process in more detail. As illustrated in Figure 1.5 (top), the two complementary DNA strands running in opposite directions around a circular chromosome unravel, starting at *oriC*. As the strands unwind, they create two **replication forks**, which expand in both directions around the chromosome until the strands completely separate at the **replication terminus** (denoted *terC*). The replication terminus is located roughly opposite to *oriC* in the chromosome.

An important thing to know about replication is that a DNA polymerase does not wait for the two parent strands to completely separate before initiating replication; instead, it starts copying *while* the strands are unraveling. Thus, just four DNA polymerases, each responsible for one half-strand, can all start at *oriC* and replicate the entire chromosome. To start replication, a DNA polymerase needs a **primer**, a short complementary segment (shown in red in Figure 1.5) that binds to the parent strand and jump starts the DNA polymerase. After the strands start separating, each of the four DNA polymerases starts replication by adding nucleotides, beginning with the primer and proceeding around the chromosome from *oriC* to *terC* in either the clockwise or counterclockwise direction. When all four DNA polymerases have reached *terC*, the chromosome's DNA will have been completely replicated, resulting in two pairs of complementary strands (Figure 1.5 (bottom)), and the cell is ready to divide.

Yet while you were reading the description above, biology professors were writing a petition to have us fired and sent back to Biology 101. And they would be right, because our exposition suffers from a major flaw; we only described the replication process in this way so that you can better appreciate what we are about to reveal.

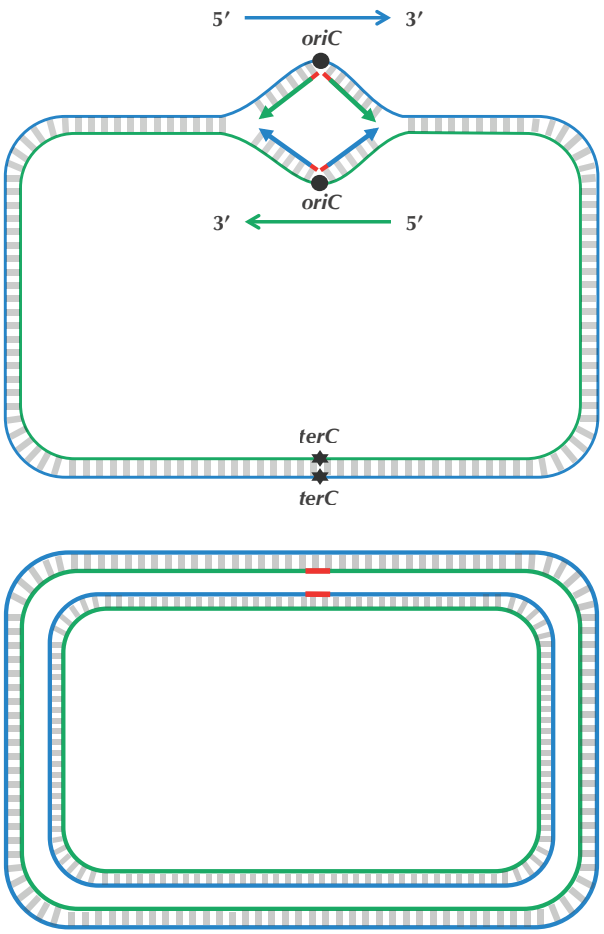


FIGURE 1.5 (Top) Four imaginary DNA polymerases at work replicating a chromosome as the replication forks extend from *oriC* to *terC*. The blue strand is directed clockwise, and the green strand is directed counterclockwise. (Bottom) Replication is complete.

The problem with our current description is that it assumes that DNA polymerases can copy DNA in *either* direction along a strand of DNA (i.e., both $5' \to 3'$ and $3' \to 5'$). However, nature has not yet equipped DNA polymerases with this ability, as they are **unidirectional**, meaning that they can only traverse a template strand of DNA in the $3' \to 5'$ direction. Notice that this is opposite from the $5' \to 3'$ direction of DNA.

STOP and Think: If you were a unidirectional DNA polymerase, how would you replicate DNA? How many DNA polymerases would be needed to complete this task?

The unidirectionality of DNA polymerase requires a major revision to our naive model of replication. Imagine that you decided to walk along DNA from *oriC* to *terC*. There are four different half-strands of parent DNA connecting *oriC* to *terC*, as highlighted in Figure 1.6. Two of these half-strands are traversed from *oriC* to *terC* in the $5' \rightarrow 3'$ direction and are thus called **forward half-strands** (represented by thin blue and green lines in Figure 1.6). The other two half-strands are traversed from *oriC* to *terC* in the $3' \rightarrow 5'$ direction and are thus called **reverse half-strands** (represented by thick blue and green lines in Figure 1.6).

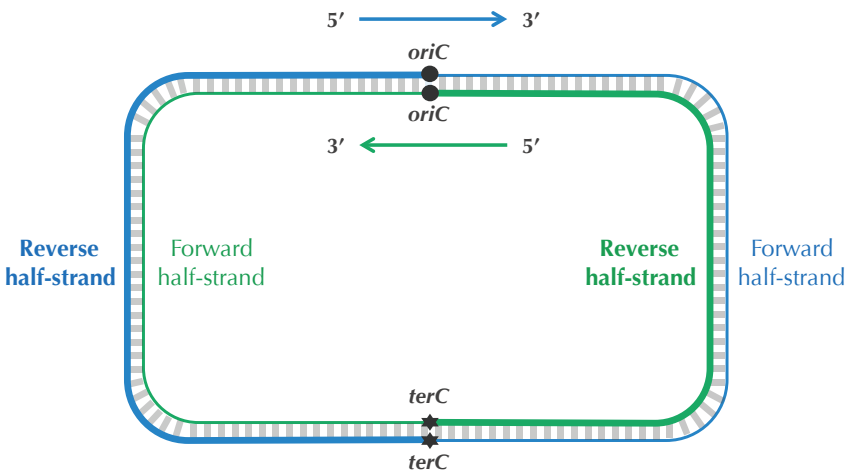


FIGURE 1.6 Complementary DNA strands with forward and reverse half-strands shown as thin and thick lines, respectively.

Asymmetry of Replication

While biologists will feel at home with the following description of DNA replication, computer scientists may find it overloaded with new terms. If it seems too biologically complex, then feel free to skim this section, as long as you believe us that the replication

process is **asymmetric**, i.e., that forward and reverse half-strands have very different fates with respect to replication.

Since a DNA polymerase can only move in the reverse $(3' \to 5')$ direction, it can copy nucleotides non-stop from *oriC* to *terC* along reverse half-strands. However, replication on forward half-strands is very different because a DNA polymerase cannot move in the forward $(5' \to 3')$ direction; on these half-strands, a DNA polymerase must replicate *backwards* toward *oriC*. Take a look at Figure 1.7 to see why this must be the case.

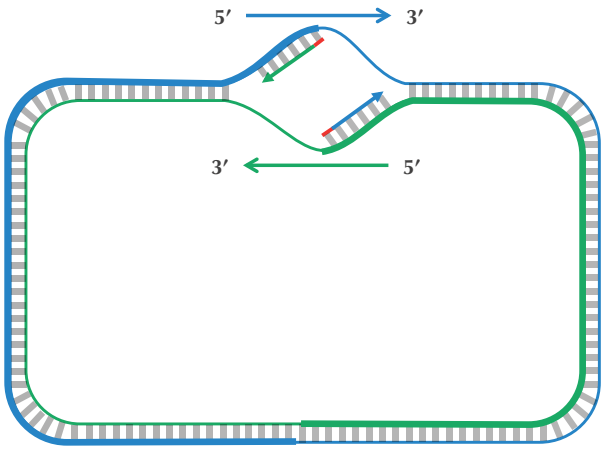


FIGURE 1.7 Replication begins at *oriC* (primers shown in red) with the synthesis of fragments on the reverse half-strands (shown by thick lines). A DNA polymerase must wait until the replication fork has opened some (small) distance before it starts copying the forward half-strands (shown by thin lines) back toward *oriC*.

On a forward half-strand, in order to replicate DNA, a DNA polymerase must wait for the replication fork to open a little (approximately 2,000 nucleotides) until a new primer is formed at the *end* of the replication fork; afterwards, the DNA polymerase starts replicating a small chunk of DNA starting from this primer and moving *backward* in the direction of *oriC*. When the two DNA polymerases on forward half-strands reach *oriC*, we have the situation shown in Figure 1.8. Note the contrast between this figure and Figure 1.5.

After this point, replication on each reverse half-strand progresses continuously; however, a DNA polymerase on a forward half-strand has no choice but to wait again until the replication fork has opened another 2,000 nucleotides or so. It then requires a new primer to begin synthesizing another fragment back toward *oriC*. On the whole,

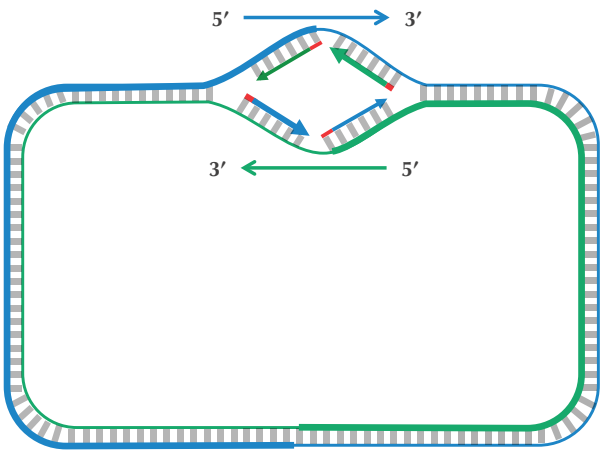


FIGURE 1.8 The daughter fragments are now synthesized (with some delay) on the forward half-strands (shown by thin lines).

replication on a forward half-strand requires occasional stopping and restarting, which results in the synthesis of short **Okazaki fragments** that are complementary to intervals on the forward half-strand. You can see these fragments forming in Figure 1.9 (top).

When the replication fork reaches *terC*, the replication process is almost complete, but gaps still remain between the disconnected Okazaki fragments (Figure 1.9 (middle)).

Finally, consecutive Okazaki fragments are sewn together by an enzyme called **DNA ligase**, resulting in two intact daughter chromosomes, each consisting of one parent strand and one newly synthesized daughter strand, as shown in Figure 1.9 (bottom). In reality, DNA ligase does not wait until after all the Okazaki fragments have been replicated to start sewing them together.

Biologists call a reverse half-strand a **leading half-strand** since a single DNA polymerase traverses this half-strand non-stop, and they call a forward half-strand a **lagging half-strand** since it is used as a template by many DNA polymerases stopping and starting replication. If you are confused about the differences between the leading and lagging half-strands, you are not alone — we and legions of biology students are also confused. The confusion is exacerbated by the fact that different textbooks use different terminology depending on whether the authors intend to refer to a leading *template* half-strand or a leading half-strand *that is being synthesized* from a (lagging) template half-strand. You hopefully see why we have chosen the terms "reverse" and "forward" half-strands in an attempt to mitigate some of this confusion.

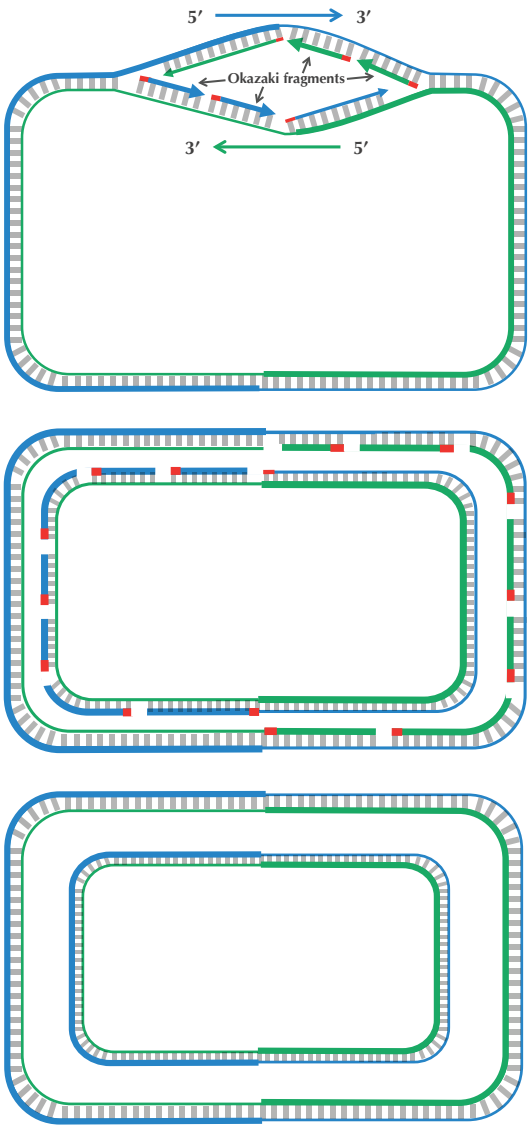


FIGURE 1.9 (Top) The replication fork continues growing. Only one primer is needed for each of the reverse half-strands (shown by thick lines), while the forward half-strands (shown by thin lines) require multiple primers in order to synthesize Okazaki fragments. Two of these primers are shown in red on each forward half-strand. (Middle) Replication is nearly complete, as all daughter DNA is synthesized. However, half of each daughter chromosome contains disconnected Okazaki fragments. (Bottom) Okazaki fragments have been sewn together, resulting in two intact daughter chromosomes.

Peculiar Statistics of the Forward and Reverse Half-Strands

Deamination

In the last section, we saw that as the replication fork expands, DNA polymerase synthesizes DNA quickly on the reverse half-strand but suffers delays on the forward half-strand. We will explore the asymmetry of DNA replication to design a new algorithm for finding *oriC*.

How in the world can the asymmetry of replication possibly help us locate *oriC*? Notice that since the replication of a reverse half-strand proceeds quickly, it lives double-stranded for most of its life. Conversely, a forward half-strand spends a much larger amount of its life single-stranded, waiting to be used as a template for replication. This discrepancy between the forward and reverse half-strands is important because single-stranded DNA has a much higher mutation rate than double-stranded DNA. In particular, if one of the four nucleotides in single-stranded DNA has a greater tendency than other nucleotides to mutate in single-stranded DNA, then we should observe a shortage of this nucleotide on the forward half-strand.

Following up on this thought, let's compare the nucleotide counts of the reverse and forward half-strands. If these counts differ substantially, then we will design an algorithm that attempts to track down these differences in genomes for which *oriC* is unknown. The nucleotide counts for *Thermotoga petrophila* are shown in Figure 1.10.

	#C	#G	#A	#T
Entire strand	427419	413241	491488	491363
Reverse half-strand	219518	201634	243963	246641
Forward half-strand	207901	211607	247525	244722
Difference	+11617	-9973	-3562	+1919

FIGURE 1.10 Counting nucleotides in the *Thermotoga petrophila* genome on the forward and reverse half-strands.

STOP and Think: Do you notice anything interesting about the nucleotide counts in Figure 1.10?

Although the frequencies of A and T are practically identical on the two half-strands, C is more frequent on the reverse half-strand than on the forward half-strand, resulting in a difference of $219518 - 207901 = +11617$. Its complementary nucleotide G is less

frequent on the reverse half-strand than on the forward half-strand, resulting in a difference of $201634 - 211607 = -9973$.

It turns out that we observe these discrepancies because cytosine (C) has a tendency to mutate into thymine (T) through a process called **deamination**. Deamination rates rise 100-fold when DNA is single-stranded, which leads to a decrease in cytosine on the forward half-strand, thus forming mismatched base pairs T–G. These mismatched pairs can further mutate into T–A pairs when the bond is repaired in the next round of replication, which accounts for the observed decrease in guanine (G) on the reverse half-strand (recall that a forward parent half-strand synthesizes a reverse daughter half-strand, and vice-versa).

STOP and Think: If deamination changes cytosine to thymine, why do you think that the forward half-strand still has some cytosine?

The skew diagram

Let's see if we can take advantage of these peculiar statistics caused by deamination to locate *oriC*. As Figure 1.10 illustrates, the difference between the total amount of guanine and the total amount of cytosine is negative on the reverse half-strand ($211607 - 207901 = 3706$) and positive on the forward half-strand ($201634 - 219518 = -17884$). Thus, our idea is to traverse the genome, keeping a running total of the difference between the counts of G and C. If this difference starts *increasing*, then we guess that we are on the forward half-strand; on the other hand, if this difference starts *decreasing*, then we guess that we are on the reverse half-strand. See Figure 1.11.

STOP and Think: Imagine that you are reading through the genome (in the $5' \rightarrow 3'$ direction) and notice that the difference between the guanine and cytosine counts just switched its behavior from decreasing to increasing. Where in the genome are you?

Since we don't know the location of *oriC* in a circular genome, let's linearize it (i.e., select an arbitrary position and pretend that the genome begins here), resulting in a linear string *Genome*. We define $\text{SKEW}_i(Genome)$ as the difference between the total number of occurrences of G and the total number of occurrences of C in the first i nucleotides of *Genome*. The **skew diagram** is defined by plotting $\text{SKEW}_i(Genome)$ as i ranges from 0 to $|Genome|$, where $\text{SKEW}_0(Genome)$ is set equal to zero. Figure 1.12 shows a skew diagram for a short DNA string.

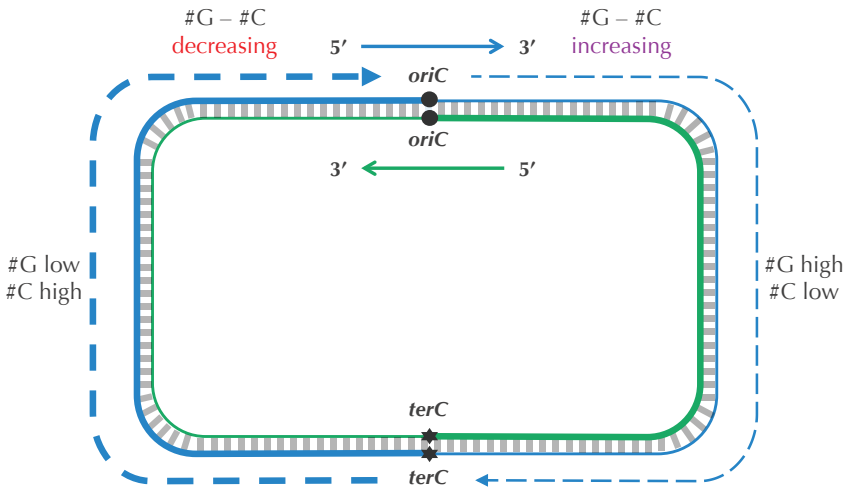


FIGURE 1.11 Because of deamination, each forward half-strand has a shortage of cytosine compared to guanine, and each reverse half-strand has a shortage of guanine compared to cytosine. The dashed blue line illustrates an imaginary walk along the outer strand of the genome counting the difference between the counts of G and C. We assume that the difference between these counts is positive on the forward half-strand and negative on the reverse half-strand.

Note that we can compute $\text{SKEW}_{i+1}(Genome)$ from $\text{SKEW}_i(Genome)$ according to the nucleotide in position i of *Genome*. If this nucleotide is **G**, then $\text{SKEW}_{i+1}(Genome) = \text{SKEW}_i(Genome) + 1$; if this nucleotide is **C**, then $\text{SKEW}_{i+1}(Genome) = \text{SKEW}_i(Genome) - 1$; otherwise, $\text{SKEW}_{i+1}(Genome) = \text{SKEW}_i(Genome)$.

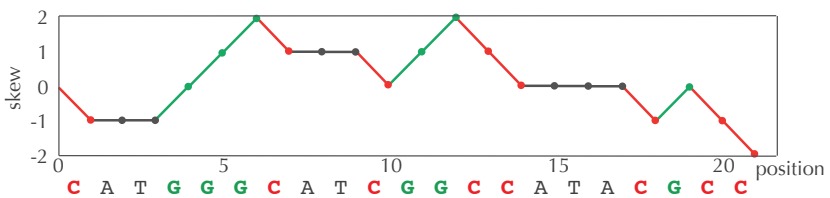


FIGURE 1.12 The skew diagram for *Genome* = CATGGGCATCGGCCATACGCC.

Figure 1.13 depicts the skew diagram for a linearized *E. coli* genome. Notice the very clear pattern! It turns out that the skew diagram for many bacterial genomes has a

similar characteristic shape.

STOP and Think: After looking at the skew diagram in Figure 1.13, where do you think that *oriC* is located in *E. coli*?

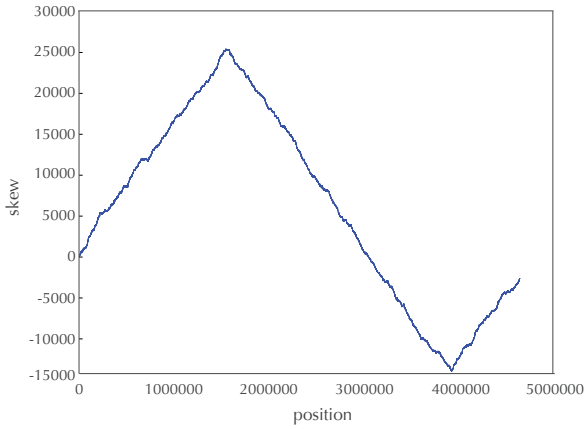


FIGURE 1.13 The skew diagram for *E. coli* achieves a maximum and minimum at positions 1550413 and 3923620, respectively.

Let's follow the $5' \rightarrow 3'$ direction of DNA and walk along the chromosome from *terC* to *oriC* (along a reverse half-strand), then continue on from *oriC* to *terC* (along a forward half-strand). In Figure 1.11, we saw that the skew is decreasing along the reverse half-strand and increasing along the forward half-strand. Thus, the skew should achieve a minimum at the position where the reverse half-strand ends and the forward half-strand begins, which is exactly the location of *oriC*! We have just developed an algorithm for locating *oriC*: it should be found where the skew attains a minimum.

Minimum Skew Problem:
Find a position in a genome where the skew diagram attains a minimum.

Input: A DNA string *Genome*.
Output: All integer(s) i minimizing $\text{SKEW}_i(Genome)$ among all values of i (from 0 to $|Genome|$).

STOP and Think: Note that the skew diagram changes depending on where we start our walk along the circular chromosome. Do you think that the minimum of the skew diagram points to the same position *in the genome* regardless of where we begin walking to generate the skew diagram?

Some Hidden Messages Are More Elusive Than Others

Solving the Minimum Skew Problem now provides us with an approximate location of *oriC* at position 3923620 in *E. coli*. In an attempt to confirm this hypothesis, let's look for a hidden message representing a potential *DnaA* box near this location. Solving the Frequent Words Problem in a window of length 500 starting at position 3923620 (shown below) reveals no 9-mers (along with their reverse complements) that appear three or more times! Even if we have located *oriC* in *E. coli*, it appears that we still have not found the *DnaA* boxes that jump-start replication in this bacterium ...

```
aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgcataacgcggt
atgaaaatggattgaagcccgggccgtggattctactcaactttgtcggcttgagaaaga
cctgggatcctgggtattaaaaagaagatctatttatttagagatctgttctattgtgat
ctcttattaggatcgcactgccctgtggataacaaggatccggcttttaagatcaacaac
ctggaaaggatcattaactgtgaatgatcggtgatcctggaccgtataagctgggatcag
aatgaggggttatacacaactcaaaaactgaacaacagttgttctttggataactaccgg
ttgatccaagcttcctgacagagttatccacagtagatcgcacgatctgtatacttattt
gagtaaattaacccacgatcccagccattcttctgccggatcttccggaatgtcgtgatc
aagaatgttgatcttcagtg
```

STOP and Think: What would you do next?

Before we give up, let's examine the *oriC* of *Vibrio cholerae* one more time to see if it provides us with any insights on how to alter our algorithm to find *DnaA* boxes in *E. coli*. You may have noticed that in addition to the three occurrences of **ATGATCAAG** and three occurrences of its reverse complement **CTTGATCAT**, the *Vibrio cholerae oriC* contains additional occurrences of **ATGATCAAC** and **CATGATCAT**, which differ from **ATGATCAAG** and **CTTGATCAT** in only a single nucleotide:

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

Finding eight *approximate* occurrences of our target 9-mer and its reverse complement in a short region is even more statistically surprising than finding the six *exact* occurrences of **ATGATCAAG** and its reverse complement **CTTGATCAT** that we stumbled upon in the beginning of our investigation. Furthermore, the discovery of these approximate 9-mers makes sense biologically, since *DnaA* can bind not only to "perfect" *DnaA* boxes but to their slight variations as well.

We say that position i in k-mers $p_1 \cdots p_k$ and $q_1 \cdots q_k$ is a **mismatch** if $p_i \neq q_i$. 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.

1G

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

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$(Pattern, Pattern') \leq d$. Our observation that a *DnaA* box may appear with slight variations leads to the following generalization of the Pattern Matching Problem.

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

1H

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.

Our goal now is to modify our previous algorithm for the Frequent Words Problem in order to find *DnaA* boxes by identifying frequent *k*-mers, possibly with mismatches. Given strings *Text* and *Pattern* as well as an integer d, we define $\text{COUNT}_d(\textit{Text}, \textit{Pattern})$ as the number of occurrences of *Pattern* in *Text* with at most d mismatches. For example,

$$\text{COUNT}_1(\texttt{AACAAGCATAAACATTAAAGAG}, \texttt{AAAAA}) = 4$$

because **AAAAA** appears four times in this string with at most one mismatch: **AACAA**, **ATAAA**, **AAACA**, and **AAAGA**. Notice that two of these occurrences overlap.

EXERCISE BREAK: Compute $\text{COUNT}_2(\texttt{AACAAGCATAAACATTAAAGAG}, \texttt{AAAAA})$.

Computing $\text{COUNT}_d(\textit{Text}, \textit{Pattern})$ simply requires us to compute the Hamming distance between *Pattern* and every *k*-mer substring of *Text*, as follows.

```
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
```

EXERCISE BREAK: Implement **APPROXIMATEPATTERNCOUNT**. What is its running time?

A **most frequent *k*-mer with up to *d* mismatches** in *Text* is simply a string *Pattern* maximizing $\text{COUNT}_d(\textit{Text}, \textit{Pattern})$ among all *k*-mers. Note that *Pattern* does not need to actually appear as a substring of *Text*; for example, as we saw above, **AAAAA** is the most frequent 5-mer with 1 mismatch in **AACAA**GC**ATAAA**C**A**TT**AAAGA**G, even though it does not appear exactly in this string. Keep this in mind while solving the following problem.

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*.
Output: All most frequent *k*-mers with up to *d* mismatches in *Text*.

CHARGING STATION (Solving the Frequent Words with Mismatches Problem): One way to solve the above problem is to generate all 4^k k-mers *Pattern*, compute **ApproximatePatternCount**(*Text*, *Pattern*, d) for each k-mer *Pattern*, and then output k-mers with the maximum number of approximate occurrences. This is an inefficient approach in practice, since many of the 4^k k-mers that this method analyzes should not be considered because neither they nor their mutated versions (with up to d mismatches) appear in *Text*. Check out this Charging Station to learn about a better approach that avoids analyzing such hopeless k-mers.

We now redefine the Frequent Words Problem to account for both mismatches and reverse complements. Recall that $\overline{Pattern}$ refers to the reverse complement of *Pattern*.

Frequent Words with Mismatches and Reverse Complements Problem:
Find the most frequent k-mers (with mismatches and reverse complements) in a string.

Input: A DNA string *Text* as well as integers k and d.
Output: All k-mers *Pattern* that maximize the sum $\text{Count}_d(Text, Pattern) + \text{Count}_d(Text, \overline{Pattern})$ over all possible k-mers.

A Final Attempt at Finding *DnaA* Boxes in *E. coli*

We now make a final attempt to find *DnaA* boxes in *E. coli* by finding the most frequent 9-mers with mismatches and reverse complements in the region suggested by the minimum skew as *oriC*. Although the minimum of the skew diagram for *E. coli* is found at position 3923620, we should not assume that its *oriC* is found exactly at this position due to random fluctuations in the skew. To remedy this issue, we could choose a larger window size (e.g., $L = 1000$), but expanding the window introduces the risk that we may bring in other clumped 9-mers that do not represent *DnaA* boxes but appear in this window more often than the true *DnaA* box. It makes more sense to try a small window either starting, ending, or centered at the position of minimum skew.

Let's cross our fingers and identify the most frequent 9-mers (with 1 mismatch and reverse complements) within a window of length 500 starting at position 3923620 of the *E. coli* genome. Bingo! The experimentally confirmed *DnaA* box in *E. coli* (**TTATCCACA**) is a most frequent 9-mer with 1 mismatch, along with its reverse complement **TGTGGATAA**:

aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgcataacgcggt
atgaaaatggattgaagcccgggccgtggattctactcaactttgtcggcttgagaaaga
cctgggatcctgggtattaaaaagaagatctatttatttagagatctgttctattgtgat
ctcttattaggatcgcactgccc**TGTGGATAA**caaggatccggcttttaagatcaacaac
ctggaaaggatcattaactgtgaatgatcggtgatcctggaccgtataagctgggatcag
aatgaggggg**TTATACACA**actcaaaaactgaacaacagttgttc**TTTGGATAA**ctaccgg
ttgatccaagcttcctgacagag**TTATCCACA**gtagatcgcacgatctgtatacttattt
gagtaaattaacccacgatcccagccattcttctgccggatcttccggaatgtcgtgatc
aagaatgttgatcttcagtg

You will notice that we highlighted an interior interval of this sequence with darker text. This region is the experimentally verified *oriC* of *E. coli*, which starts 37 nucleotides after position 3923620, where the skew reaches its minimum value.

We were very fortunate that the *DnaA* boxes of *E. coli* are captured in the window that we chose. Moreover, while **TTATCCACA** represents a most frequent 9-mer with 1 mismatch and reverse complements in this 500-nucleotide window, it is not the only one: GGATCCTGG, GATCCCAGC, GTTATCCAC, AGCTGGGAT, and CTGGGATCA also appear four times with 1 mismatch and reverse complements.

STOP and Think: In this chapter, every time we find *oriC*, we seem to find some other surprisingly frequent 9-mers. Why do you think this is?

We do not know what purpose — if any — these other 9-mers serve in the *E. coli* genome, but we do know that there are *many different types* of hidden messages in genomes; these hidden messages have a tendency to cluster within a genome, and most of them have nothing to do with replication. One example is the regulatory DNA motifs responsible for gene expression that we will study in Chapter 2. The important lesson is that existing approaches to *oriC* prediction remain imperfect and sometimes inconclusive. However, even providing biologists with a small collection of 9-mers as candidate *DnaA* boxes is a great aid as long as one of these 9-mers is correct.

Thus, the moral of this chapter is that even though computational predictions can be powerful, bioinformaticians should collaborate with biologists to verify their computational predictions. Or improve these predictions: the next question hints at how *oriC* predictions can be carried out using **comparative genomics**, a bioinformatics approach that uses evolutionary similarities to answer difficult questions about genomes.

STOP and Think: *Salmonella enterica* 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 enterica*?

You will have an opportunity to look for *DnaA* boxes in *Salmonella enterica* in the epilogue, which will feature a "Challenge Problem" asking you to apply what you have learned to a real dataset. Some chapters also have an "Open Problems" section outlining unanswered research questions.

Epilogue: Complications in *oriC* Predictions

In this chapter, we have considered three genomes and found three different hypothesized 9-mers encoding *DnaA* boxes: **ATGATCAAG** in *Vibrio cholerae*, **CCTACCACC** in *Thermotoga petrophila*, and **TTATCCACA** in *E. coli*. We must warn you that finding *oriC* is often more complex than in the three examples we considered. Some bacteria have even fewer *DnaA* boxes than *E. coli*, making it difficult to identify them. The *terC* region is often located not directly opposite to *oriC* but may be significantly shifted, resulting in reverse and forward half-strands having substantially different lengths. The position of the skew minimum is often only a rough indicator of *oriC* position, which forces researchers to expand their windows when searching for *DnaA* boxes, bringing in extraneous repeated substrings. Finally, skew diagrams do not always look as nice as that of *E. coli*; for example, the skew diagram for *Thermotoga petrophila* is shown in Figure 1.14.

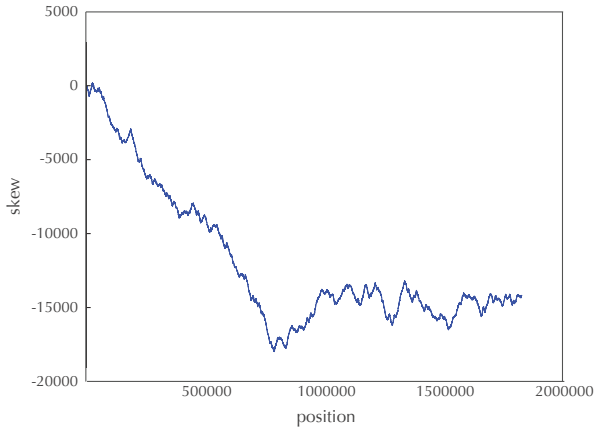


FIGURE 1.14 The skew diagram for *Thermotoga petrophila* achieves a minimum at position 787199 but does not have the same nice shape as the skew diagram for *E. coli*.

STOP and Think: What evolutionary process could possibly explain the shape of the skew diagram for *Thermotoga petrophila*?

Since the skew diagram for *Thermotoga petrophila* is complex and the *oriC* for this genome has not even been experimentally verified, there is a chance that the region predicted by Ori-Finder as the *oriC* region for *Thermotoga petrophila* (or even for *Vibrio cholerae*) is actually incorrect!

You now should have a good sense of how to locate *oriC* and *DnaA* boxes computationally. We will take the training wheels off and ask you to solve a challenge problem.

CHALLENGE PROBLEM: Find *DnaA* boxes in *Salmonella enterica*.

Open Problems

Multiple replication origins in a bacterial genome

Biologists long believed that each bacterial chromosome has only one *oriC*. Wang et al., 2011 genetically modified *E. coli* by inserting a synthetic *oriC* a million nucleotides away from the bacterium's known *oriC*. To their surprise, *E. coli* continued business as usual, starting replication at both locations!

Following the publication of this paper, the search for naturally occurring bacteria with multiple *oriC*s immediately started. In 2012, Xia raised doubts about the "single *oriC*" postulate and gave examples of bacteria with highly unusual skews. In fact, having more than one *oriC* makes sense in the light of evolution: if the genome is long and replication is slow, then multiple replication origins would decrease the amount of time that the bacterium must spend replicating its DNA.

For example, *Wigglesworthia glossinidia*, a symbiotic bacterium living in the intestines of tsetse flies, has the atypical skew diagram shown in Figure 1.15. Since this diagram has at least two pronounced local minima, Xia argued that this bacterium may have two or more *oriC* regions.

We should be careful with Xia's hypothesis that this bacterium has two *oriC*s, as there may be alternative explanations for multiple local minima in the skew. For example, **genome rearrangements** (which we will study in **??**) move genes within a genome and often reposition them from the forward to the reverse half-strand and vice-versa, thus resulting in irregularities in the skew diagram. One example of a genome rearrangement is a **reversal**, which flips around a segment of chromosome and switches it to the opposite strand; Figure 1.16 shows what happens to the skew diagram after a reversal.

Another difficulty is presented by the fact that different species of bacteria may exchange genetic material in **horizontal gene transfer**. If a gene from the forward half-strand of one bacterium is transferred to the reverse half-strand of another (or vice-versa), then we will observe an irregularity in the skew diagram. As a result, the question about the number of *oriC*s of *Wigglesworthia glossinidia* remains unresolved.

However, if you could demonstrate that there exist two sets of identical *DnaA* boxes in the vicinity of two local minima in the skew diagram of *Wigglesworthia glossinidia*, then you would have the first solid evidence in favor of multiple bacterial *oriC*s. Maybe simply applying your solution for the Frequent Words with Mismatches and Reverse Complements Problem will reveal these *DnaA* boxes. Can you find other bacterial genomes where a single *oriC* is in doubt and check whether they indeed have multiple *oriC*s?

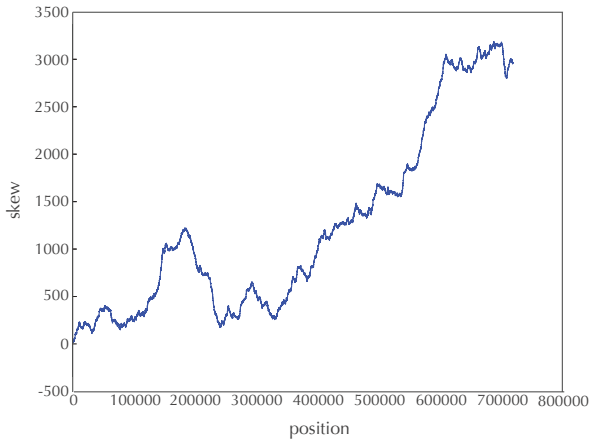


FIGURE 1.15 The skew diagram for *Wigglesworthia glossinidia*.

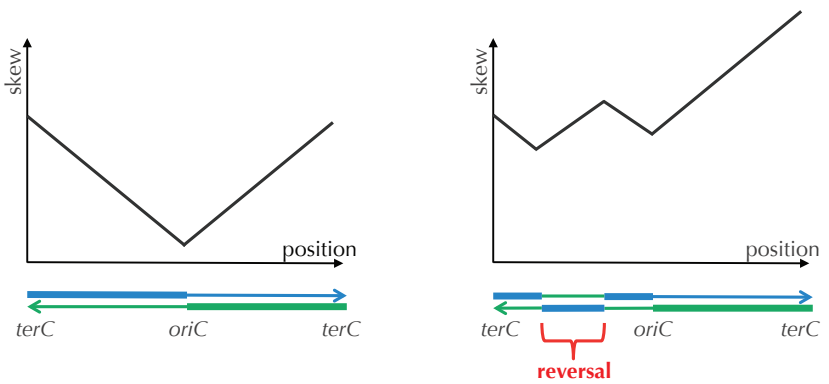


FIGURE 1.16 (Left) An "ideal" V-shaped skew diagram that achieves minimum skew at *oriC*. The skew diagram decreases along the reverse half-strand (shown by a thick line) and increases along the forward half-strand (shown by a thin line). We assume that a circular chromosome was cut at *terC*, resulting in a linear chromosome that starts and ends at *terC*. (Right) A skew diagram after a reversal that switches segments between the reverse and forward strands and alters the skew diagram. As before, the skew diagram still decreases along the segments of the genome shown by thick lines and increases along the segments shown by thin lines.

Finding replication origins in archaea

Archaea are unicellular organisms so distinct from other life forms that biologists have placed them into their own **domain of life** separate from bacteria and eukaryotes. Although archaea are visually similar to bacteria, they have some genomic features that are more closely related to eukaryotes. In particular, the replication machinery of archaea is more similar to eukaryotes than bacteria. Yet archaea use a much greater variety of energy sources than eukaryotes, feeding on ammonia, metals, or even hydrogen gas.

Figure 1.17 shows the skew diagram of *Sulfolobus solfataricus*, a species of archaea growing in acidic volcanic springs in temperatures over 80° C. In its skew diagram, you can see at least three local minima, represented by deep valleys, in addition to many more shallow valleys.

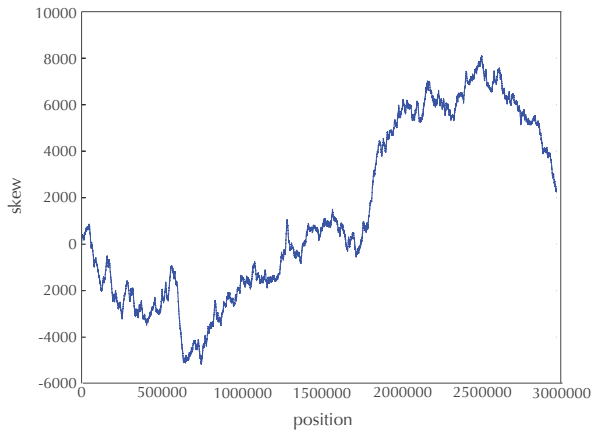


FIGURE 1.17 The skew diagram of *Sulfolobus solfataricus*.

Lundgren et al., 2004 demonstrated experimentally that *Sulfolobus solfataricus* indeed has three *ori*Cs. Since then, multiple *ori*Cs have been identified in many other archaea. However, no accurate computational approach has been developed to identify multiple *ori*Cs in a newly sequenced archaea genome. For example, the methane-producing archaea *Methanococcus jannaschii* is considered the workhorse of archaea genomics, but its *oriC*(s) still remain unidentified! Its skew diagram (shown in Figure 1.18) suggests that it may have multiple *ori*Cs: can you find them?

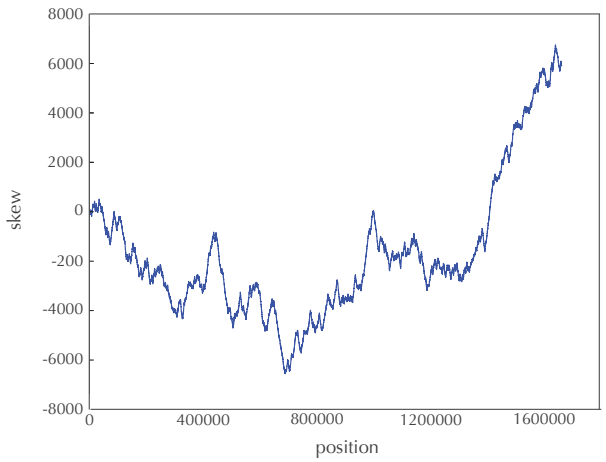


FIGURE 1.18 The skew diagram for *Methanococcus jannaschii*.

Finding replication origins in yeast

If you think that finding replication origins in bacteria is a complex problem, wait until you analyze replication origins in more complex organisms like yeast or humans, which have hundreds of replication origins. Among various yeast species, the budding yeast *Saccharomyces cerevisiae* has the best characterized replication origins. It has approximately 400 different *oriC*s, many of which may be used during the replication of any single yeast cell.

Having a large number of *oriC*s results in dozens of replication forks hurtling towards each other from different locations in the genome in ways that are not yet completely understood. However, researchers have discovered that the replication origins of *S. cerevisiae* share a (somewhat variable) pattern called the **ARS Consensus Sequence** (**ACS**). The ACS is the binding site for the so-called **Origin Recognition Complex**, which initiates the loading of additional proteins required for origin firing. Many ACSs correspond to the following canonical thymine-rich pattern of length 11.

TTTAT(G/A)TTT(T/A)(G/T)

Here, the notation (X/Y) indicates that either nucleotide X or nucleotide Y may appear in that position.

However, various ACSs may differ from this canonical pattern, with lengths varying from 11 to 17 nucleotides. For example, the 11-nucleotide long pattern shown above is often part of a 17-nucleotide pattern:

(T/A)(T/A)(T/A)(T/A)**TTTAT**(**G/A**)**TTT**(**T/A**)(**G/T**)(T/G)(T/C)

Recently, some progress has been made in characterizing the ACS in a few other yeast species. In some species like *S. bayanus*, the ACS is almost identical to that of *S. cerevisiae*, while in others such as *K. lactis*, it is very different. More alarmingly, at least for bioinformaticians, in some yeast species such as *S. pombe*, the Origin Recognition Complex binds to loosely defined AT-rich regions, which makes it next to impossible to find replication origins based on sequence analysis alone.

Despite recent efforts, finding *oriC*s in yeast remains an open problem, and no accurate software exists for predicting origins of replication from the sequence of yeast genomes. Can you explore this problem and devise an algorithm to predict replication origins in yeast?

Computing probabilities of patterns in a string

In the main text, we told you that the probability that a random DNA string of length 500 contains a 9-mer appearing three or more times is *approximately* 1/1300. In **DETOUR: Probabilities of Patterns in a String**, **PAGE 52** we describe a method to estimate this probability, but it is rather inaccurate. This open problem is aimed at finding better approximations or even deriving exact formulas for probabilities of patterns in strings.

We start by asking a seemingly simple question: what is the probability that a specific k-mer *Pattern* will appear (at least once) as a substring of a random string of length N? This simple question proved to be not so simple and was first addressed by Solov'ev, 1966 (see also Sedgewick and Flajolet, 2013).

The first surprise is that different k-mers may have different probabilities of appearing in a random string. For example, the probability that *Pattern* = "01" appears in a random binary string of length 4 is 11/16, while the probability that *Pattern* = "11" appears in a random binary string of length 4 is 8/16. This phenomenon is called the **overlapping words paradox** because different occurrences of *Pattern* can overlap each other for some patterns (e.g., "11") but not others (e.g., "01"). See **DETOUR: The Overlapping Words Paradox**. **PAGE 62**

We are interested in computing the following probabilities for a random N-letter string in an A-letter alphabet:

- $\Pr(N, A, Pattern, t)$, the probability that a string *Pattern* appears at least t times in a random string;
- $\Pr^*(N, A, Pattern, t)$, the probability that a string *Pattern* and its reverse complement $\overline{Pattern}$ appear at least t total times in a random string.

Note that the above two probabilities are relatively straightforward to compute. Several variants of these are open:

- $\Pr_d(N, A, Pattern, t)$, the probability that a string *Pattern* approximately appears at least t times in a random string (with at most d mismatches);
- $\Pr(N, A, k, t)$, the probability that there exists *any* k-mer appearing at least t times in a random string;
- $\Pr_d(N, A, k, t)$, the probability that there exists *any* k-mer with at least t approximate occurrences in a random string (with at most d mismatches).

Charging Stations

The frequency array

To make **FREQUENTWORDS** faster, we will think about why this algorithm is slow in the first place. It slides a window of length k down *Text*, identifying a k-mer *Pattern* of *Text* at each step. For each such k-mer, it must slide a window down the entire length of *Text* in order to compute PATTERNCOUNT(*Text*, *Pattern*). Instead of doing all this sliding, we aspire to slide a window down *Text* only once. As we slide this window, we will keep track of the number of times that each k-mer *Pattern* has already appeared in *Text*, updating these numbers as we proceed.

To achieve this goal, we will first order all 4^k k-mers lexicographically (i.e., according to how they would appear in the dictionary) and then convert them into the 4^k different integers between 0 and $4^k - 1$. Given an integer k, we define the **frequency array** of a string *Text* as an array of length 4^k, where the i-th element of the array holds the number of times that the i-th k-mer (in the lexicographic order) appears in *Text* (Figure 1.19).

k-mer	AA	AC	AG	AT	CA	CC	CG	CT	GA	GC	GG	GT	TA	TC	TG	TT
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
frequency	3	0	2	0	1	0	0	0	0	1	3	1	0	0	1	0

FIGURE 1.19 The lexicographic order of DNA 2-mers (top), along with the index of each k-mer in this order (middle), and the frequency array for AAGCAAAGGTGGG (bottom). For example, the frequency array at index 10 is equal to 3 because GG, the tenth DNA 2-mer according to lexicographic order, occurs three times in AAGCAAAGGTGGG.

To compute the frequency array, we need to determine how to transform a k-mer *Pattern* into an integer using a function PATTERNTONUMBER(*Pattern*). We also should know how to reverse this process, transforming an integer between 0 and $4^k - 1$ into a k-mer using a function NUMBERTOPATTERN(*index*, k). Figure 1.19 illustrates that PATTERNTONUMBER(GT) $= 11$ and NUMBERTOPATTERN$(11, 2) =$ GT.

EXERCISE BREAK: Compute the following:

1. PATTERNTONUMBER(ATGCAA)
2. NUMBERTOPATTERN$(5437, 7)$
3. NUMBERTOPATTERN$(5437, 8)$

CHARGING STATION (Converting Patterns Into Numbers and Vice-Versa): Check out this Charging Station to see how to implement PATTERNTONUMBER and NUMBERTOPATTERN.

The pseudocode below generates a frequency array by first initializing every element in the frequency array to zero (4^k operations) and then making a single pass down *Text* (approximately $|Text| \cdot k$ operations). For each k-mer *Pattern* that we encounter, we add 1 to the value of the frequency array corresponding to *Pattern*. As before, we refer to the k-mer beginning at position i of *Text* as $Text(i, k)$.

```
COMPUTINGFREQUENCIES(Text, k)
    for i ← 0 to 4^k − 1
        FREQUENCYARRAY(i) ← 0
    for i ← 0 to |Text| − k
        Pattern ← Text(i, k)
        j ← PATTERNTONUMBER(Pattern)
        FREQUENCYARRAY(j) ← FREQUENCYARRAY(j) + 1
    return FREQUENCYARRAY
```

We now have a faster algorithm for the Frequent Words Problem. After generating the frequency array, we can find all most frequent k-mers by simply finding all k-mers corresponding to the maximum element(s) in the frequency array.

```
FASTERFREQUENTWORDS(Text , k)
    FrequentPatterns ← an empty set
    FREQUENCYARRAY ← COMPUTINGFREQUENCIES(Text, k)
    maxCount ← maximal value in FREQUENCYARRAY
    for i ← 0 to 4^k − 1
        if FREQUENCYARRAY(i) = maxCount
            Pattern ← NUMBERTOPATTERN(i, k)
            add Pattern to the set FrequentPatterns
    return FrequentPatterns
```

CHARGING STATION (Finding Frequent Words by Sorting): Although **FASTERFREQUENTWORDS** is fast for small k (i.e., you can use it to find *DnaA* boxes in an *oriC* region), it becomes impractical when k is large. If you are familiar with sorting algorithms and are interested in seeing a faster algorithm, check out this Charging Station.

EXERCISE BREAK: Our claim that **FASTERFREQUENTWORDS** is faster than **FREQUENTWORDS** is only correct for certain values of $|Text|$ and k. Estimate the running time of **FASTERFREQUENTWORDS** and characterize the values of $|Text|$ and k when **FASTERFREQUENTWORDS** is indeed faster than **FREQUENTWORDS**.

Converting patterns to numbers and vice-versa

Our approach to computing PATTERNTONUMBER(*Pattern*) is based on a simple observation. If we remove the final symbol from all lexicographically ordered k-mers, the resulting list is still ordered lexicographically (think about removing the final letter from every word in a dictionary). In the case of DNA strings, every $(k-1)$-mer in the resulting list is repeated four times (Figure 1.20).

AAA	**AA**C	**AA**G	**AA**T	**AC**A	**AC**C	**AC**G	**AC**T
AGA	**AG**C	**AG**G	**AG**T	**AT**A	**AT**C	**AT**G	**AT**T
CAA	**CA**C	**CA**G	**CA**T	**CC**A	**CC**C	**CC**G	**CC**T
CGA	**CG**C	**CG**G	**CG**T	**CT**A	**CT**C	**CT**G	**CT**T
GAA	**GA**C	**GA**G	**GA**T	**GC**A	**GC**C	**GC**G	**GC**T
GGA	**GG**C	**GG**G	**GG**T	**GT**A	**GT**C	**GT**G	**GT**T
TAA	**TA**C	**TA**G	**TA**T	**TC**A	**TC**C	**TC**G	**TC**T
TGA	**TG**C	**TG**G	**TG**T	**TT**A	**TT**C	**TT**G	**TT**T

FIGURE 1.20 If we remove the final symbol from all lexicographically ordered DNA 3-mers, we obtain a lexicographic order of (red) 2-mers, where each 2-mer is repeated four times.

Thus, the number of 3-mers occurring before **AG**T is equal to four times the number of 2-mers occurring before **AG** plus the number of 1-mers occurring before T. Therefore,

$$\begin{aligned}\text{PATTERNTONUMBER}(\mathbf{AG}\text{T}) &= 4 \cdot \text{PATTERNTONUMBER}(\mathbf{AG}) + \text{SYMBOLTONUMBER}(\text{T}) \\ &= 8 + 3 = 11\,,\end{aligned}$$

where SYMBOLTONUMBER(*symbol*) is the function transforming symbols A, C, G, and T into the respective integers 0, 1, 2, and 3.

If we remove the final symbol of *Pattern*, denoted LASTSYMBOL(*Pattern*), then we will obtain a $(k-1)$-mer that we denote as PREFIX(*Pattern*). The preceding observation therefore generalizes to the formula

$$\text{PATTERNTONUMBER}(Pattern) = 4 \cdot \text{PATTERNTONUMBER}(\text{PREFIX}(Pattern)) + \text{SYMBOLTONUMBER}(\text{LASTSYMBOL}(Pattern)). \quad (*)$$

This equation leads to the following recursive algorithm, i.e., a program that calls itself. If you want to learn more about recursive algorithms, see **DETOUR: The Towers of Hanoi**. **PAGE 60**

```
PATTERNTONUMBER(Pattern)
    if Pattern contains no symbols
        return 0
    symbol ← LASTSYMBOL(Pattern)
    Prefix ← PREFIX(Pattern)
    return 4 · PATTERNTONUMBER(Prefix) + SYMBOLTONUMBER(symbol)
```

In order to compute the inverse function NUMBERTOPATTERN$(index, k)$, we return to (*) above, which implies that when we divide $index = \text{PATTERNTONUMBER}(Pattern)$ by 4, the remainder will be equal to SYMBOLTONUMBER(*symbol*), and the quotient will be equal to PATTERNTONUMBER(PREFIX(*Pattern*)). Thus, we can use this fact to peel away symbols at the end of *Pattern* one at a time, as shown in Figure 1.21.

STOP and Think: Once we have computed NUMBERTOPATTERN$(9904, 7)$ in Figure 1.21, how would you compute NUMBERTOPATTERN$(9904, 8)$?

In the pseudocode below, we denote the quotient and the remainder when dividing integer n by integer m as QUOTIENT(n, m) and REMAINDER(n, m), respectively. For example, QUOTIENT$(11, 4) = 2$ and REMAINDER$(11, 4) = 3$. This pseudocode uses the function NUMBERTOSYMBOL(*index*), which is the inverse of SYMBOLTONUMBER and transforms the integers 0, 1, 2, and 3 into the respective symbols A, C, G, and T.

n	Quotient$(n, 4)$	Remainder$(n, 4)$	NumberToSymbol
9904	2476	0	A
2476	619	0	A
619	154	3	T
154	38	2	G
38	9	2	G
9	2	1	C
2	0	2	G

Figure 1.21 When computing *Pattern* = NumberToPattern(9904, 7), we divide 9904 by 4 to obtain a quotient of 2476 and a remainder of 0. This remainder represents the final nucleotide of *Pattern*, or NumberToSymbol(0) = A. We then iterate this process, dividing each subsequent quotient by 4, until we obtain a quotient of 0. The symbols in the nucleotide column, read upward from the bottom, yield *Pattern* = GCGGTAA.

```
NumberToPattern(index , k)
    if k = 1
        return NumberToSymbol(index)
    prefixIndex ← Quotient(index, 4)
    r ← Remainder(index, 4)
    symbol ← NumberToSymbol(r)
    PrefixPattern ← NumberToPattern(prefixIndex, k − 1)
    return concatenation of PrefixPattern with symbol
```

Finding frequent words by sorting

To see how sorting can help us find frequent k-mers, we will consider a motivating example when $k = 2$. Given a string *Text* = AAGCAAAGGTGGG, list all its 2-mers in the order they appear in *Text*, and convert each 2-mer into an integer using PatternToNumber to produce an array Index, as shown below.

2-mer	AA	AG	GC	CA	AA	AA	AG	GG	GT	TG	GG	GG
Index	0	2	9	4	0	0	2	10	11	14	10	10

We will now sort Index to generate an array SortedIndex, as shown in Figure 1.22.

STOP and Think: How can the sorted array in Figure 1.22 help us find frequent words?

2-mer	AA	AA	AA	AG	AG	CA	GC	GG	GG	GG	GT	TG
SORTEDINDEX	0	0	0	2	2	4	9	10	10	10	11	14
COUNT	1	2	3	1	2	1	1	1	2	3	1	1

FIGURE 1.22 Lexicographically sorted 2-mers in AAGCAAAGGTGG (top), along with arrays SORTEDINDEX (middle) and COUNT (bottom).

Since identical k-mers clump together in the sorted array (like $(0, 0, 0)$ for AA or $(10, 10, 10)$ for GG in Figure 1.22), frequent k-mers are the longest runs of identical integers in SORTEDINDEX. This insight leads to **FINDINGFREQUENTWORDSBYSORTING**, whose pseudocode is shown below. This algorithm uses an array COUNT for which COUNT(i) computes the number of times that the integer at position i in the array SORTEDINDEX appears in the first i elements of this array (Figure 1.22 (bottom)). In the pseudocode for **FINDINGFREQUENTWORDSBYSORTING**, we assume that you already know how to sort an array using an algorithm **SORT**.

```
FINDINGFREQUENTWORDSBYSORTING(Text , k)
    FrequentPatterns ← an empty set
    for i ← 0 to |Text| − k
        Pattern ← Text(i, k)
        INDEX(i) ← PATTERNTONUMBER(Pattern)
        COUNT(i) ← 1
    SORTEDINDEX ← SORT(INDEX)
    for i ← 1 to |Text| − k
        if SORTEDINDEX(i) = SORTEDINDEX(i − 1)
            COUNT(i) = COUNT(i − 1) + 1
    maxCount ← maximum value in the array COUNT
    for i ← 0 to |Text| − k
        if COUNT(i) = maxCount
            Pattern ← NUMBERTOPATTERN(SORTEDINDEX(i), k)
            add Pattern to the set FrequentPatterns
    return FrequentPatterns
```

Solving the Clump Finding Problem

Note: This Charging Station assumes that you have read **CHARGING STATION: The Frequency Array**.

PAGE 39

The pseudocode below slides a window of length L down *Genome*. After computing the frequency array for the current window, it identifies (L, t)-clumps simply by finding which k-mers occur at least t times within the window. To keep track of these clumps, our algorithm uses an array CLUMP of length 4^k whose values are all initialized to zero. For each value of i between 0 and $4^k - 1$, we will set CLUMP(i) equal to 1 if NUMBERTOPATTERN(i, k) forms an (L, t)-clump in *Genome*.

```
ClumpFinding(Genome, k, t, L)
    FrequentPatterns ← an empty set
    for i ← 0 to 4^k − 1
        Clump(i) ← 0
    for i ← 0 to |Genome| − L
        Text ← the string of length L starting at position i in Genome
        FrequencyArray ← ComputingFrequencies(Text, k)
        for index ← 0 to 4^k − 1
            if FrequencyArray(index) ≥ t
                Clump(index) ← 1
    for i ← 0 to 4^k − 1
        if Clump(i) = 1
            Pattern ← NumberToPattern(i, k)
            add Pattern to the set FrequentPatterns
    return FrequentPatterns
```

EXERCISE BREAK: Estimate the running time of **CLUMPFINDING**.

CLUMPFINDING makes $|Genome| - L + 1$ iterations, generating a frequency array for a string of length L at each iteration. Since this task takes roughly $4^k + L \cdot k$ time, the overall running time of **CLUMPFINDING** is $\mathcal{O}\Big(|Genome| \cdot (4^k + L \cdot k)\Big)$. As a result, when searching for *DnaA* boxes ($k = 9$) in a typical bacterial genome ($|Genome| > 1000000$), **CLUMPFINDING** becomes too slow.

STOP and Think: Can you speed up **CLUMPFINDING** by eliminating the need to generate a new frequency array at every iteration?

To improve **CLUMPFINDING**, we observe that when we slide our window of length L one symbol to the right, the frequency array does not change much, and so regener-

ating the frequency array from scratch is inefficient. For example, Figure 1.23 shows the frequency arrays ($k = 2$) for the 13-mers $Text$ = **AA**GCAAAGGTGG**G** and $Text'$ = **A**GCAAAGGTGG**GC** starting at positions 0 and 1 of the 14-mer **AA**GCAAAGGTGG**GC**. These two frequency arrays differ in only two elements corresponding to the first k-mer in $Text$ (**AA**) and the last k-mer in $Text'$ (**GC**). Specifically, the frequency array value corresponding to the **first** k-mer of $Text$ is **reduced** by 1 in the frequency array of $Text'$, and the frequency array value corresponding to the **last** k-mer of $Text$ is **increased** by 1 in the frequency array of $Text'$.

This observation helps us modify **ClumpFinding** as shown below. Note that we now only call **ComputingFrequencies** once, updating the frequency array as we go along.

```
BetterClumpFinding(Genome, k, t, L)
    FrequentPatterns ← an empty set
    for i ← 0 to 4^k − 1
        Clump(i) ← 0
    Text ← Genome(0, L)
    FrequencyArray ← ComputingFrequencies(Text, k)
    for i ← 0 to 4^k − 1
        if FrequencyArray(i) ≥ t
            Clump(i) ← 1
    for i ← 1 to |Genome| − L
        FirstPattern ← Genome(i − 1, k)
        index ← PatternToNumber(FirstPattern)
        FrequencyArray(index) ← FrequencyArray(index) − 1
        LastPattern ← Genome(i + L − k, k)
        index ← PatternToNumber(LastPattern)
        FrequencyArray(index) ← FrequencyArray(index) + 1
        if FrequencyArray(index) ≥ t
            Clump(index) ← 1
    for i ← 0 to 4^k − 1
        if Clump(i) = 1
            Pattern ← NumberToPattern(i, k)
            add Pattern to the set FrequentPatterns
    return FrequentPatterns
```

k-mer	**AA**	AC	AG	AT	CA	CC	CG	CT	GA	**GC**	GG	GT	TA	TC	TG	TT
INDEX	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
frequency	**3**	0	2	0	1	0	0	0	0	**1**	3	1	0	0	1	0
frequency'	**2**	0	2	0	1	0	0	0	0	**2**	3	1	0	0	1	0

FIGURE 1.23 The frequency arrays for two consecutive substrings of length 13 starting at positions 0 and 1 of *Genome* = **AA**GCAAAGGTGG**GC** are very similar to each other.

Solving the Frequent Words with Mismatches Problem

Note: This Charging Station uses some notation from **CHARGING STATION: The Frequency Array**.

To prevent having to generate all 4^k *k*-mers in order to solve the Frequent Words with Mismatches Problem, our goal is to consider only those *k*-mers that are **close** to a *k*-mer in *Text*, i.e., those with Hamming distance at most *d* from this *k*-mer. Given a *k*-mer *Pattern*, we therefore define its ***d*-neighborhood** NEIGHBORS(*Pattern*, *d*) as the set of all *k*-mers that are close to *Pattern*. For example, NEIGHBORS(ACG, 1) consists of ten 3-mers:

ACG **C**CG **G**CG **T**CG A**A**G A**G**G A**T**G AC**A** AC**C** AC**T**

EXERCISE BREAK: Estimate the size of NEIGHBORS(*Pattern*, *d*).

We will also use an array CLOSE of size 4^k whose values we initialize to zero. In the **FREQUENTWORDSWITHMISMATCHES** pseudocode below, we set CLOSE(i) = 1 whenever *Pattern* = NUMBERTOPATTERN(i, k) is close to some *k*-mer in *Text*. This allows us to apply **APPROXIMATEPATTERNCOUNT** only to close *k*-mers, a smarter approach than applying it to all *k*-mers.

CHARGING STATION (Generating the Neighborhood of a String): **FREQUENTWORDSWITHMISMATCHES** also calls **NEIGHBORS**(*Pattern*, *d*), a function that generates the *d*-neighborhood of a *k*-mer *Pattern*. Check out this Charging Station to learn how to implement this function.

STOP and Think: Although **FREQUENTWORDSWITHMISMATCHES** is faster than the naive algorithm described in the main text for the typical parameters used in *oriC* searches, it is not necessarily faster for all parameter values. For which parameter values is **FREQUENTWORDSWITHMISMATCHES** slower than the naive algorithm?

```
FrequentWordsWithMismatches(Text, k, d)
    FrequentPatterns ← an empty set
    for i ← 0 to 4^k − 1
        Close(i) ← 0
        FrequencyArray ← 0
    for i ← 0 to |Text| − k
        Neighborhood ← Neighbors(Text(i, k), d)
        for each Pattern from Neighborhood
            index ← PatternToNumber(Pattern)
            Close(index) ← 1
    for i ← 0 to 4^k − 1
        if Close(i) = 1
            Pattern ← NumberToPattern(i, k)
            FrequencyArray(i) ← ApproximatePatternCount(Text, Pattern, d)
    maxCount ← maximal value in FrequencyArray
    for i ← 0 to 4^k − 1
        if FrequencyArray(i) = maxCount
            Pattern ← NumberToPattern(i, k)
            add Pattern to the set FrequentPatterns
    return FrequentPatterns
```

CHARGING STATION (Finding Frequent Words with Mismatches by Sorting): If you are familiar with sorting and are interested in seeing an even faster algorithm for the Frequent Words with Mismatches Problem, check out this Charging Station.

Generating the neighborhood of a string

Our goal is to generate the d-neighborhood NEIGHBORS($Pattern, d$), the set of all k-mers whose Hamming distance from *Pattern* does not exceed d. We will first generate the 1-neigborhood of *Pattern* using the following pseudocode.

```
IMMEDIATENEIGHBORS(Pattern)
    Neighborhood ← the set consisting of single string Pattern
    for i = 1 to |Pattern|
        symbol ← i-th nucleotide of Pattern
        for each nucleotide x different from symbol
            Neighbor ← Pattern with the i-th nucleotide substituted by x
            add Neighbor to Neighborhood
    return Neighborhood
```

Our idea for generating NEIGHBORS($Pattern, d$) is as follows. If we remove the first symbol of *Pattern* (denoted FIRSTSYMBOL(*Pattern*)), then we will obtain a $(k-1)$-mer that we denote by SUFFIX($Pattern$).

STOP and Think: If we know NEIGHBORS(SUFFIX($Pattern$), d), how does it help us construct NEIGHBORS($Pattern, d$)?

Now, consider a $(k-1)$-mer *Pattern*′ belonging to NEIGHBORS(SUFFIX($Pattern$), d). By the definition of the d-neighborhood NEIGHBORS(SUFFIX($Pattern$), d), we know that HAMMINGDISTANCE(*Pattern*′, SUFFIX(*Pattern*)) is either equal to d or less than d. In the first case, we can add FIRSTSYMBOL(*Pattern*) to the beginning of *Pattern*′ in order to obtain a k-mer belonging to NEIGHBORS($Pattern, d$). In the second case, we can add any symbol to the beginning of *Pattern*′ and obtain a k-mer belonging to NEIGHBORS($Pattern, d$).

In the following pseudocode, we use the notation *symbol* • *Text* to denote the concatenation of a character *symbol* and a string *Text*, e.g., A • GCATG = AGCATG.

```
NEIGHBORS(Pattern, d)
    if d = 0
        return {Pattern}
    if |Pattern| = 1
        return {A, C, G, T}
    Neighborhood ← an empty set
    SuffixNeighbors ← NEIGHBORS(SUFFIX(Pattern), d)
    for each string Text from SuffixNeighbors
        if HAMMINGDISTANCE(SUFFIX(Pattern), Text) < d
            for each nucleotide x
                add x • Text to Neighborhood
        else
            add FIRSTSYMBOL(Pattern) • Text to Neighborhood
    return Neighborhood
```

STOP and Think: Consider the following questions.

1. What is the running time of **NEIGHBORS**?
2. **NEIGHBORS** generates all k-mers of Hamming distance at most d from *Pattern*. Modify **NEIGHBORS** to generate all k-mers of Hamming distance exactly d from *Pattern*.

If you are still learning how recursive algorithms (like **NEIGHBORS**) work, you may want to implement an iterative version of **NEIGHBORS** instead, shown below.

```
ITERATIVENEIGHBORS(Pattern, d)
    Neighborhood ← set consisting of single string Pattern
    for j = 1 to d
        for each string Pattern' in Neighborhood
            add IMMEDIATENEIGHBORS(Pattern') to Neighborhood
            remove duplicates from Neighborhood
    return Neighborhood
```

Finding frequent words with mismatches by sorting

Note: This Charging Station uses some notation from **CHARGING STATION: Finding Frequent Words by Sorting**.

The following pseudocode reduces the Frequent Words with Mismatches Problem to sorting.

```
FindingFrequentWordsWithMismatchesBySorting(Text, k, d)
    FrequentPatterns ← an empty set
    Neighborhoods ← an empty list
    for i ← 0 to |Text| − k
        add Neighbors(Text(i, k), d) to Neighborhoods
    form an array NeighborhoodArray holding all strings in Neighborhoods
    for i ← 0 to |Neighborhoods| − 1
        Pattern ← NeighborhoodArray(i)
        Index(i) ← PatternToNumber(Pattern)
        Count(i) ← 1
    SortedIndex ← Sort(Index)
    for i ← 0 to |Neighborhoods| − 1
        if SortedIndex(i) = SortedIndex(i + 1)
            Count(i + 1) ← Count(i) + 1
    maxCount ← maximum value in array Count
    for i ← 0 to |Neighborhoods| − 1
        if Count(i) = maxCount
            Pattern ← NumberToPattern(SortedIndex(i), k)
            add Pattern to FrequentPatterns
    return FrequentPatterns
```

Detours

Big-O notation

Computer scientists typically measure an algorithm's efficiency in terms of its **worst-case running time**, which is the largest amount of time an algorithm can take for the most difficult input of a given size. The advantage to considering the worst-case running time is that we are guaranteed that our algorithm will never behave worse than our worst-case estimate.

Big-O notation compactly describes the running time of an algorithm. For example, if your algorithm for sorting an array of n numbers takes roughly n^2 operations for the most difficult dataset, then we say that the running time of your algorithm is $\mathcal{O}(n^2)$. In reality, depending on your implementation, it may use any number of operations, such as $1.5n^2$, $n^2 + n + 2$, or $0.5n^2 + 1$; all these algorithms are $\mathcal{O}(n^2)$ because big-O notation only cares about the term that grows the fastest with respect to the size of the input. This is because as n grows very large, the difference in behavior between two $\mathcal{O}(n^2)$ functions, like $999 \cdot n^2$ and $n^2 + 3n + 9999999$, is negligible when compared to the behavior of functions from different classes, say $\mathcal{O}(n^2)$ and $\mathcal{O}(n^6)$. Of course, we would prefer an algorithm requiring $1/2 \cdot n^2$ steps to an algorithm requiring $1000 \cdot n^2$ steps.

When we write that the running time of an algorithm is $\mathcal{O}(n^2)$, we technically mean that it does not grow faster than a function with a leading term of $c \cdot n^2$, for some constant c. Formally, a function $f(n)$ is Big-O of function $g(n)$, or $\mathcal{O}(g(n))$, when $f(n) \leq c \cdot g(n)$ for some constant c and sufficiently large n.

Probabilities of patterns in a string

We mentioned that the probability that some 9-mer appears 3 or more times in a random DNA string of length 500 is approximately $1/1300$. We assure you that this calculation does not appear out of thin air. Specifically, we can generate a **random string** modeling a DNA strand by choosing each nucleotide for any position with probability $1/4$. The construction of random strings can be generalized to an arbitrary alphabet with A symbols, where each symbol is chosen with probability $1/A$.

EXERCISE BREAK: What is the probability that two randomly generated strings of length n in an A-letter alphabet are identical?

We now ask a simple question: what is the probability that a specific k-mer *Pattern* will appear (at least once) as a substring of a random string of length N? For example, say

that we want to find the probability that "01" appears in a **binary string** ($A = 2$) of length 4. Here are all possible such strings.

0000 0001 0010 0011 0100 0101 0110 0111
1000 1001 1010 1011 1100 1101 1110 1111

Because "01" is a substring of 11 of these 4-mers, and because each 4-mer could be generated with probability 1/16, the probability that "01" appears in a random binary 4-mer is 11/16.

STOP and Think: What is the probability that *Pattern* = "11" appears as a substring of a random binary 4-mer?

Surprisingly, changing *Pattern* from "01" to "11" changes the probability that it appears as a substring of a random binary string. Indeed, "11" appears in only 8 binary 4-mers:

0000 0001 0010 0011 0100 0101 0110 0111
1000 1001 1010 1011 1100 1101 1110 1111

As a result, the probability of "11" appearing in a random binary string of length 4 is $8/16 = 1/2$.

STOP and Think: Why do you think that "11" is less likely than "01" to appear as a substring of a random binary 4-mer?

Let $\Pr(N, A, Pattern, t)$ denote the probability that a string *Pattern* appears t or more times in a random string of length N formed from an alphabet of A letters. We saw that $\Pr(4, 2, \text{"01"}, 1) = 11/16$ while $\Pr(4, 2, \text{"11"}, 1) = 1/2$. Interestingly, when we make t greater than 1, we see that "01" is *less* likely to appear multiple times than "11". For example, the probability of finding "01" twice or more in a random binary 4-mer is given by $\Pr(4, 2, \text{"01"}, 2) = 1/16$ because "0101" is the only binary 4-mer containing "01" twice, and yet $\Pr(4, 2, \text{"11"}, 2) = 3/16$ because binary 4-mers "0111", "1110" and "1111" all have at least two occurrences of "11".

EXERCISE BREAK: Compute $\Pr(100, 2, \text{"01"}, 1)$.

We have seen that different k-mers have different probabilities of occurring multiple times as a substring of a random string. In general, this phenomenon is called the **overlapping words paradox** because different substring occurrences of *Pattern* can

overlap each other for some choices of *Pattern* but not others (see **DETOUR: The Overlapping Words Paradox**). PAGE 62

For example, there are two overlapping occurrences of "11" in "1110", and three overlapping occurrences of "11" in "1111"; yet occurrences of "01" can never overlap with each other, and so "01" can never occur more than twice in a binary 4-mer. The overlapping words paradox makes computing $\Pr(N, A, Pattern, t)$ a rather complex problem because this probability depends heavily on the particular choice of *Pattern*. In light of the complications presented by the overlapping words paradox, we will try to *approximate* $\Pr(A, N, Pattern, t)$ rather than compute it exactly.

To approximate $\Pr(N, A, Pattern, t)$, we will assume that the k-mer *Pattern* is not overlapping. As a toy example, say we we wish to count the number of **ternary strings** ($A = 3$) of length 7 that contain "01" at least twice. Apart from the two occurrences of "01", we have three remaining symbols in the string. Let's assume that these symbols are all "2". The two occurrences of "**01**" can be inserted into "222" in ten different ways to form a 7-mer, as shown below.

0101222	**01**20122	**01**22012	**01**22201	20**01**22
20**12**012	20**12**201	22**01**012	22**01**201	222**01**01

We inserted these two occurrences of "01" into "222", but we could have inserted them into any other ternary 3-mer. Because there are $3^3 = 27$ ternary 3-mers, we obtain an approximation of $10 \cdot 27 = 270$ for the number of ternary 7-mers that contain two or more instances of "01". Because there are $3^7 = 2187$ ternary 7-mers, we estimate the probability $\Pr(7, 3, \text{"01"}, 2)$ as $270/2187$.

STOP and Think: Is $270/2187$ a good approximation for $\Pr(7, 3, \text{"01"}, 2)$? Is the true probability $\Pr(7, 3, \text{"01"}, 2)$ larger or smaller than $270/2187$?

To generalize the above method to approximate $\Pr(N, A, Pattern, t)$ for arbitrary parameter values, consider a string *Text* of length N having at least t occurrences of a k-mer *Pattern*. If we select exactly t of these occurrences, then we can think about *Text* as a sequence of $n = N - t \cdot k$ symbols interrupted by t insertions of the k-mer *Pattern*. If we fix these n symbols, then we wish to count the number of different strings *Text* that can be formed by inserting t occurrences of *Pattern* into a string formed by these n symbols.

For example, consider again the problem of embedding two occurrences of "01" into "222" ($n = 3$), and note that we have added five copies of a capital "X" below each 7-mer.

```
0101222   0120122   0122012   0122201   2010122
X X XXX   X XX XX   X XXX X   X XXXX    XX X XX

2012012   2012201   2201012   2201201   2220101
XX XX X   XX XXX    XXX X X   XXX XX    XXXX X
```

What do the "X" mean? Instead of counting the number of ways to insert two occurrences of "01" into "222", we can count the number of ways to select two of the five "X" to color blue.

```
XXXXX XXXXX XXXXX XXXXX XXXXX
XXXXX XXXXX XXXXX XXXXX XXXXX
```

In other words, we are counting the number of ways to choose 2 out of 5 objects, which can be counted by the **binomial coefficient** $\binom{5}{2} = 10$. More generally, the binomial coefficient $\binom{m}{k}$ represents the number of ways to choose k out of m objects and is equal to $m!/k!(m-k)!$

STOP and Think: How many ways are there to implant t instances of a (nonoverlapping) k-mer into a string of length n to produce a string of length $n + t \cdot k$?

To approximate $\Pr(N, A, Pattern, t)$, we want to count the number of ways to insert t instances of a k-mer $Pattern$ into a fixed string of length $n = N - t \cdot k$. We will therefore have $n + t$ occurrences of "X", from which we must select t for the placements of $Pattern$, giving a total of $\binom{n+t}{t}$. We then need to multiply $\binom{n+t}{t}$ by the number of strings of length n into which we can insert t instances of $Pattern$ to have an approximate total of $\binom{n+t}{t} \cdot A^n$ (the actual number will be smaller because of over-counting). Dividing by the number of strings of length N, we have our desired approximation,

$$\Pr(N, A, Pattern, t) \approx \frac{\binom{n+t}{t} \cdot A^n}{A^N} = \frac{\binom{N-t\cdot(k-1)}{t}}{A^{t\cdot k}}.$$

We will now compute the probability that the specific 5-mer ACTAT occurs at least $t = 3$ times in a random DNA string ($A = 4$) of length $N = 30$. Since $n = N - t \cdot k = 15$, our estimated probability is

$$\Pr(30, 4, \texttt{ACTAT}, 3) \approx \frac{\binom{30-3\cdot 4}{3}}{4^{15}} = \frac{816}{1073741824} \approx 7.599 \cdot 10^{-7}.$$

The exact probability is closer to $7.572 \cdot 10^{-7}$, illustrating that our approximation is relatively accurate for non-overlapping patterns. However, it becomes inaccurate for overlapping patterns, e.g., $\Pr(30, 4, \texttt{AAAAA}, 3) \approx 1.148 \cdot 10^{-3}$.

We should not be surprised that the probability of finding ACTAT in a random DNA string of length 30 is so low. However, remember that our original goal was to approximate the probability that there exists *some* 5-mer appearing three or more times. In general, the probability that some k-mer appears t or more times in a random string of length N formed over an A-letter alphabet is written $\Pr(N, A, k, t)$.

We approximated $\Pr(N, A, \mathit{Pattern}, t)$ as

$$p = \frac{\binom{N-t\cdot(k-1)}{t}}{A^{t\cdot k}}.$$

Notice that the approximate probability that *Pattern* does *not* appear t or more times is therefore $1 - p$. Thus, the probability that *all* A^k patterns appear fewer than t times in a random string of length N can be approximated as

$$(1-p)^{A^k}.$$

Moreover, the probability that there exists a k-mer appearing t or more times should be 1 minus this value, which gives us the following approximation:

$$\Pr(N, A, k, t) \approx 1 - (1-p)^{A^k}.$$

Your calculator may have difficulty with this formula, which requires raising a number close to 1 to a very large power and can cause round-off errors. To avoid this, if we assume that p is about the same for any *Pattern*, then we can approximate $\Pr(N, A, k, t)$ by multiplying p by the total number of k-mers A^k,

$$\Pr(N, A, k, t) \approx p \cdot A^k = \frac{\binom{N-t\cdot(k-1)}{t}}{A^{t\cdot k}} \cdot A^k = \frac{\binom{N-t\cdot(k-1)}{t}}{A^{(t-1)\cdot k}}.$$

We acknowledge again that this approximation is a gross over-simplification, since the probability $\Pr(N, A, \mathit{Pattern}, t)$ varies across different choices of k-mers and because it assumes that occurrences of different k-mers are independent events. For example, in the main text, we wish to approximate $\Pr(500, 4, 9, 3)$, and the above formula results in the approximation

$$\Pr(500, 4, 9, 3) \approx \frac{\binom{500-3\cdot 8}{3}}{4^{(3-1)\cdot 9}} = \frac{17861900}{68719476736} \approx \frac{1}{3847}.$$

Because of overlapping strings, this approximation deviates from the true value of which is closer to 1/1300. To see how to obtain more precise estimates, see Guibas and Odlyzko, 1981.

The most beautiful experiment in biology

The Meselson-Stahl experiment, conducted in 1958 by Matthew Meselson and Franklin Stahl, is sometimes called "the most beautiful experiment in biology". In the late 1950s, biologists debated three conflicting models of DNA replication, illustrated in Figure 1.24. The **semiconservative hypothesis** (recall Figure 1.1 from page 3), suggested that each parent strand acts as a template for the synthesis of a daughter strand. As a result, each of the two daughter molecules contains one parent strand and one newly synthesized strand. The **conservative hypothesis** proposed that the entire double-stranded parent DNA molecule serves as a template for the synthesis of a new daughter molecule, resulting in one molecule with two parent strands and another with two newly synthesized strands. The **dispersive hypothesis** proposed that some mechanism breaks the DNA backbone into pieces and splices intervals of synthesized DNA, so that each of the daughter molecules is a patchwork of old and new double-stranded DNA.

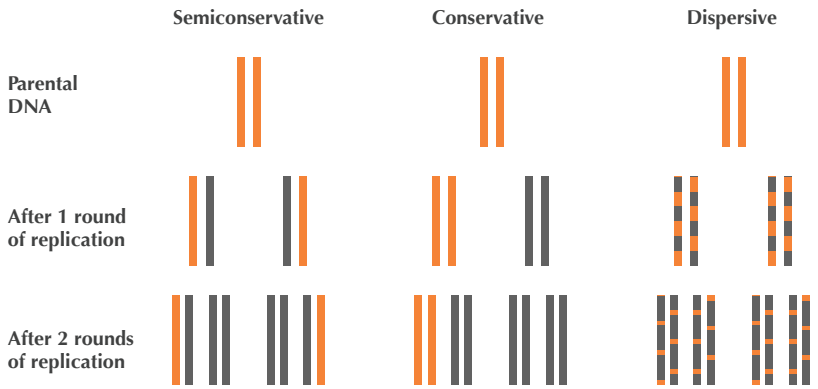


FIGURE 1.24 Semiconservative, conservative, and dispersive models of DNA replication make different predictions about the distribution of DNA strands after replication. Yellow strands indicate ^{15}N (heavy) segments of DNA, and black strands indicate ^{14}N (light) segments. The Meselson-Stahl experiment began with DNA consisting of 100% ^{15}N.

Meselson and Stahl's insight was that one isotope of nitrogen, **Nitrogen-14** (^{14}N), is lighter and more abundant than **Nitrogen-15** (^{15}N). Knowing that DNA naturally contains ^{14}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. When they were confident that the bacterial DNA was saturated with ^{15}N, they transferred the heavy *E. coli* cells to a less dense ^{14}N medium.

STOP and Think: What do you think happened when the "heavy" *E. coli* replicated in the "light" ^{14}N medium?

The brilliance of the Meselson-Stahl experiment is that all newly synthesized DNA would contain exclusively ^{14}N, and the three existing hypotheses for DNA replication predicted different outcomes for how this ^{14}N isotope would be incorporated into DNA. Specifically, after one round of replication, the conservative model predicted that half the *E. coli* DNA would still have only ^{15}N and therefore be heavier whereas the other half would have only ^{14}N and be lighter. Yet when they attempted to separate the *E. coli* DNA according to weight by using a centrifuge after one round of replication, all of the DNA had the same density! Just like that, they had refuted the conservative hypothesis once and for all.

Unfortunately, this experiment was not able to eliminate either of the other two models, as both the dispersive and semiconservative hypotheses predicted that all of the DNA after one round of replication would have the same density.

STOP and Think: What would the dispersive and semiconservative models predict about the density of *E. coli* DNA after two rounds of replication?

Let's first consider the dispersive model, which says that each daughter strand of DNA is formed by half mashed up pieces of the parent strand, and half new DNA. If this hypothesis were true, then after two replication cycles, any daughter strand of DNA should contain about 25% ^{15}N and about 75% ^{14}N. In other words, all the DNA should still have the same density. And yet when Meselson and Stahl spun the centrifuge after two rounds of *E. coli* replication, this is not what they observed!

Instead, they found that the DNA divided into two different densities. This is exactly what the semiconservative model predicted: after one cycle, every cell should possess one ^{14}N strand and one ^{15}N strand; after two cycles, half of the DNA molecules should have one ^{14}N strand and one ^{15}N strand, while the other half should have two ^{14}N strands, producing the two different densities they noticed.

STOP and Think: What does the semi-conservative model predict about the density of *E. coli* DNA after three rounds of replication?

Meselson and Stahl had rejected the conservative and dispersive hypotheses of replication, and yet they wanted to make sure that the semiconservative hypothesis was confirmed by further *E. coli* replication. This model predicted that after three rounds of

replication, one-quarter of the DNA molecules should still have a ^{15}N strand, causing 25% of the DNA to have an intermediate density, whereas the remaining 75% should be lighter, having only ^{14}N. This is indeed what Meselson and Stahl witnessed in the lab, and the semiconservative hypothesis has stood strong to this day.

Directionality of DNA strands

The sugar component of a nucleotide has a ring of five carbon atoms, which are labeled as 1′, 2′, 3′, 4′, and 5′ in Figure 1.25 (left). The 5′ atom is joined onto the phosphate group in the nucleotide and eventually to the 3′ end of the neighboring nucleotide. The 3′ atom is joined onto another neighboring nucleotide in the nucleic acid chain. As a result, we call the two ends of the nucleotide the **5′-end** and the **3′-end** (pronounced "five prime end" and "three prime end", respectively).

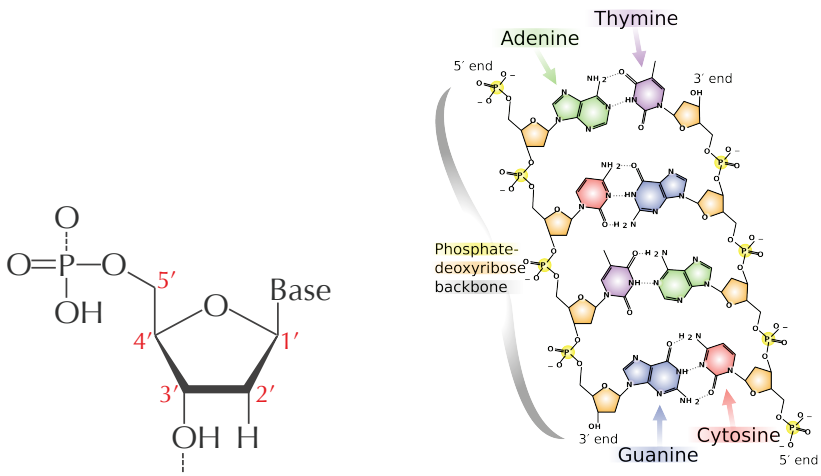


FIGURE 1.25 A nucleotide with sugar ring carbon atoms labeled 1′, 2′, 3′, 4′, and 5′.

When we zoom out to the level of the double helix, we can see in Figure 1.25 (right) that any DNA fragment is oriented with a 3′ atom on one end and a 5′ atom on the other end. As a standard, a DNA strand is always read in the 5′ $\rightarrow$ 3′ direction. Note that the orientations run opposite to each other in complementary strands.

The Towers of Hanoi

The **Towers of Hanoi** puzzle consists of three vertical pegs and a number of disks of different sizes, each with a hole in its center so that it fits on the pegs. The disks are initially stacked on the left peg (peg 1) so that disks increase in size from the top down (Figure 1.26). The puzzle is played by moving one disk at a time between pegs, with the goal of moving all disks from the left peg (peg 1) to the right peg (peg 3). However, you are not allowed to place a disk on top of a smaller disk.

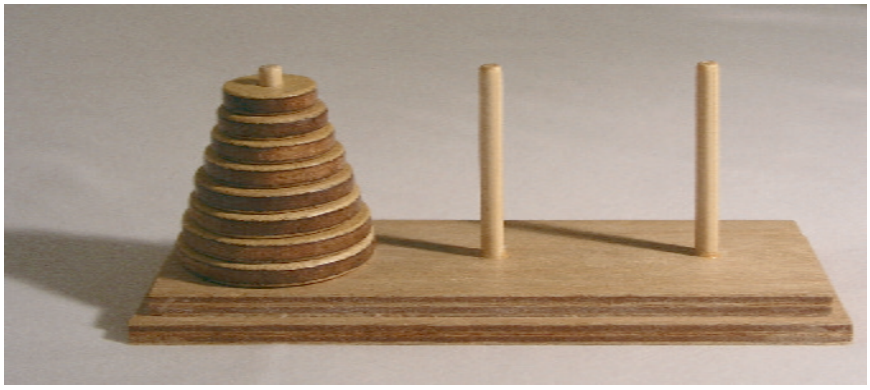


FIGURE 1.26 The Towers of Hanoi puzzle.

Towers of Hanoi Problem:
Solve the Towers of Hanoi puzzle.

Input: An integer n.
Output: A sequence of moves that will solve the Towers of Hanoi puzzle with n disks.

STOP and Think: What is the minimum number of steps needed to solve the Towers of Hanoi Problem for three disks?

Let's see how many steps are required to solve the Towers of Hanoi Problem for four disks. The first important observation is that sooner or later you will have to move the largest disk to the right peg. However, in order to move the largest disk, we first have to move all three smallest disks off the first peg. Furthermore, these three smallest disks

must all be on the same peg because the largest disk cannot be placed on top of another disk. Thus, we first have to move the top three disks to the middle peg (7 moves), then move the largest disk to the right peg (1 move), then again move the three smallest disks from the middle peg to the top of the largest disk on the right peg (another 7 moves), for a total of 15 moves.

More generally, let $T(n)$ denote the minimum number of steps required to solve the Towers of Hanoi puzzle with n disks. To move n disks from the left peg to the right peg, you first need to move the $n-1$ smallest disks from the left peg to the middle peg ($T(n-1)$ steps), then move the largest disk to the right peg (1 step), and finally move the $n-1$ smallest disks from the middle peg to the right peg ($T(n-1)$ steps). This yields the recurrence relation

$$T(n) = 2T(n-1) + 1\,.$$

STOP and Think: Using the above recurrence relation, can you find a formula for $T(n)$ that does not require recursion?

We now have a recursive algorithm to move n disks from the left peg to the right peg. We will use three variables (each taking a different value from 1, 2, and 3) to denote the three pegs: *startPeg*, *destinationPeg*, and *transitPeg*. These three variables always represent different pegs, and so *startPeg* + *destinationPeg* + *transitPeg* is always equal to $1+2+3=6$. **HANOITOWERS**$(n, \textit{startPeg}, \textit{destinationPeg})$ moves n disks from *startPeg* to *destinationPeg* (using *transitPeg* as a temporary destination).

```
HANOITOWERS(n, startPeg, destinationPeg)
    if n = 1
        Move top disk from startPeg to destinationPeg
        return
    transitPeg = 6 − startPeg − destinationPeg
    HANOITOWERS(n − 1, startPeg, transitPeg)
    Move top disk from startPeg to destinationPeg
    HANOITOWERS(n − 1, transitPeg, destinationPeg)
    return
```

Even though this algorithm may seem straightforward, moving a 100-disk tower would require more steps than the number of atoms in the universe! The fast growth of the number of moves required by **HANOITOWERS** is explained by the fact that every time

HANOITOWERS is called for n disks, it calls itself twice for $n-1$, which in turn triggers four calls for $n-2$, and so on. For example, a call to **HANOITOWERS**$(4,1,3)$ results in calls **HANOITOWERS**$(3,1,2)$ and **HANOITOWERS**$(3,2,3)$; these calls, in turn, call **HANOITOWERS**$(2,1,3)$, **HANOITOWERS**$(2,3,2)$, **HANOITOWERS**$(2,2,1)$, and **HANOITOWERS**$(2,1,3)$.

The overlapping words paradox

We illustrate the overlapping words paradox with a two-player game called "Best Bet for Simpletons". Player 1 selects a binary k-mer A, and Player 2, knowing what A is, selects a different binary k-mer B. The two players then flip a coin multiple times, with coin flips represented by strings of "1" ("heads") and "0" ("tails"); the game ends when A or B appears as a block of k consecutive coin flips.

STOP and Think: Do the two players always have the same chance of winning?

At first glance, you might guess that every k-mer has an equal chance of winning. Yet suppose that Player 1 chooses "00" and Player 2 chooses "10". After two flips, either Player 1 wins ("00"), Player 2 wins ("10"), or the game continues ("01" or "11"). If the game continues, then Player 1 should surrender, since Player 2 will win as soon as "tails" ("0") is next flipped. Player 2 is therefore three times more likely to win!

It may seem that Player 1 should have the advantage by simply selecting the "strongest" k-mer. However, an intriguing feature of Best Bet for Simpletons is that if $k > 2$, then Player 2 can always choose a k-mer B that beats A, *regardless* of Player 1's choice of A. Another surprise is that Best Bet for Simpletons is a **non-transitive game**: if A defeats B, and B defeats C, then we cannot automatically conclude that A defeats C (c.f. rock-paper-scissors).

The analysis of Best Bet for Simpletons is based on the notion of a **correlation polynomial**. We say that B ***i*-overlaps** with A if the last i digits of A coincide with the first i digits of B. For example, "110110" 1-overlaps, 2-overlaps, and 5-overlaps with "011011", as shown in Figure 1.27.

Given two k-mers A and B, the **correlation** of A and B, denoted $\text{CORR}(A,B) = (c_0,\ldots,c_{k-1})$, is a k-letter binary word such that $c_i = 1$ if B $(k-i)$-overlaps with A, and 0 otherwise. The correlation polynomial of A and B is defined as

$$K_{A,B}(t) = c_0 + c_1 \cdot t + c_2 \cdot t^2 + \cdots + c_{k-1} \cdot t^{k-1}.$$

	Corr(A, B)
B = 110110	**0**
B = **11011**0	**1**
B = 110110	**0**
B = 110110	**0**
B = **11**0110	**1**
B = **1**10110	**1**
A = **011011**	

FIGURE 1.27 The correlation of k-mers A = "011011" and B = "110110" is the string "**010011**".

For the strings A and B in Figure 1.27, their correlation is "010011" and their correlation polynomial is $K_{A,B}(t) = t + t^4 + t^5$.

Next, we write $K_{A,B}$ as shorthand for $K_{A,B}(1/2)$. For the example in Figure 1.27, $K_{A,B} = \frac{1}{2} + \frac{1}{16} + \frac{1}{32} = \frac{19}{32}$. John Conway suggested the following deceivingly simple formula to compute the odds that B will defeat A:

$$\frac{K_{A,A} - K_{A,B}}{K_{B,B} - K_{B,A}}$$

Conway never published a proof of this formula, and Martin Gardner, a leading popular mathematics writer, said the following about the formula:

> *I have no idea why it works. It just cranks out the answer as if by magic, like so many of Conway's other algorithms.*

Bibliography Notes

Using the skew to find replication origins was first proposed by Lobry, 1996 and also described in Grigoriev, 1998. Grigoriev, 2011 provides an excellent introduction to the skew approach, and Sernova and Gelfand, 2008 gave a review of algorithms and software tools for finding replication origins in bacteria. Lundgren et al., 2004 demonstrated that archaea may have multiple *oriC*s. Wang et al., 2011 inserted an artificial *oriC* into the *E. coli* genome and showed that it triggers replication. Xia, 2012 was the first to conjecture that bacteria may have multiple replication origins. Gao and Zhang, 2008 developed the Ori-Finder software program for finding bacterial replication origins. Liachko et al., 2013 provided the most comprehensive description of the replication origins of yeast. Solov'ev, 1966 was the first to derive accurate formulas for approximating the probabilities of patterns in a string. Gardner, 1974 wrote an excellent introductory article about the Best Bet for Simpletons paradox. Guibas and Odlyzko, 1981 provided an excellent coverage of the overlapping words paradox that illustrates the complexity of computing the probabilities of patterns in a random text. They also derived a rather complicated proof of Conway's formula for Best Bet for Simpletons. Sedgewick and Flajolet, 2013 gave an overview of various approaches for computing the probabilities of patterns in a string.

WHICH DNA PATTERNS PLAY THE ROLE OF MOLECULAR CLOCKS?

Randomized Algorithms

Do We Have a "Clock" Gene?

The daily schedules of animals, plants, and even bacteria are controlled by an internal timekeeper called the **circadian clock**. Anyone who has experienced the misery of jet lag knows that this clock never stops ticking. Rats and research volunteers alike, when placed in a bunker, naturally maintain a roughly 24-hour cycle of activity and rest in total darkness. And, like any timepiece, the circadian clock can malfunction, resulting in a genetic disease known as **delayed sleep-phase syndrome** (**DSPS**).

The circadian clock must have some basis on the molecular level, which presents many questions. How do *individual cells* in animals and plants (let alone bacteria) know when they should slow down or increase the production of certain proteins? Is there a "clock gene"? Can we explain why heart attacks occur more often in the morning, while asthma attacks are more common at night? And can we identify genes that are responsible for "breaking" the circadian clock to cause DSPS?

In the early 1970s, Ron Konopka and Seymour Benzer identified mutant flies with abnormal circadian patterns and traced the flies' mutations to a single gene. Biologists needed two more decades to discover a similar clock gene in mammals, which was just the first piece of the puzzle. Today, many more circadian genes have been discovered; these genes, having names like *timeless*, *clock*, and *cycle*, orchestrate the behavior of hundreds of other genes and display a high degree of evolutionary conservation across species.

We will first focus on plants, since maintaining the circadian clock in plants is a matter of life and death. Consider how many plant genes should pay attention to the time when the sun rises and sets; indeed, biologists estimate that over a thousand plant genes are circadian, including the genes related to photosynthesis, photo reception, and flowering. These genes must somehow know what time it is in order to change their gene transcript production, or **gene expression**, throughout the day (see **DETOUR: Gene Expression** **PAGE 107**).

It turns out that every plant cell keeps track of day and night independently of other cells, and that just three plant genes, called LCY, CCA1, and TOC1, are the clock's master timekeepers. Such regulatory genes, and the **regulatory proteins** that they encode, are often controlled by external factors (e.g., nutrient availability or sunlight) in order to allow organisms to adjust their gene expression.

For example, regulatory proteins controlling the circadian clock in plants coordinate circadian activity as follows. TOC1 promotes the expression of LCY and CCA1, whereas LCY and CCA1 repress the expression of TOC1, resulting in a **negative feedback loop**. In the morning, sunlight activates the transcription of LCY and CCA1, triggering the

repression of TOC1 transcription. As light diminishes, so does the production of LCY and CCA1, which in turn do not repress TOC1 any more. Transcription of TOC1 peaks at night and starts promoting the transcription of LCY and CCA1, which in turn repress the transcription of TOC1, and the cycle begins again.

LCY, CCA1, and TOC1 are able to control the transcription of other genes because the regulatory proteins that they encode are **transcription factors**, or master regulatory proteins that turn other genes on and off. A transcription factor regulates a gene by binding to a specific short DNA interval called a **regulatory motif**, or **transcription factor binding site**, in the gene's **upstream region**, a 600-1000 nucleotide-long region preceding the start of the gene. For example, CCA1 binds to AAAAAATCT in the upstream region of many genes regulated by CCA1.

The life of a bioinformatician would be easy if regulatory motifs were completely conserved, but the reality is more complex, as regulatory motifs may vary at some positions, e.g., CCA1 may instead bind to AA**G**AA**C**TCT. But how can we locate these regulatory motifs without knowing what they look like in advance? We need to develop algorithms for **motif finding**, the problem of discovering a "hidden message" shared by a collection of strings.

Motif Finding Is More Difficult Than You Think

Identifying the evening element

In 2000, Steve Kay used **DNA arrays** (see DETOUR: DNA Arrays) to determine which genes in the plant *Arabidopsis thaliana* are activated at different times of the day. He then extracted the upstream regions of nearly 500 genes that exhibited circadian behavior and looked for frequently appearing patterns in their upstream regions. If you concatenated these upstream regions into a single string, you would find that AAAATATCT is a surprisingly frequent word, appearing 46 times.

PAGE 107

EXERCISE BREAK: What is the expected number of occurrences of a 9-mer in 500 random DNA strings, each of length 1000?

Kay named AAAATATCT the **evening element** and performed a simple experiment to prove that it is indeed the regulatory motif responsible for circadian gene expression in *Arabidopsis thaliana*. After he mutated the evening element in the upstream region of one gene, the gene lost its circadian behavior.

Whereas the evening element in plants is very conserved, and thus easy to find, motifs having many mutations are more elusive. For example, if you infect a fly with a bacterium, the fly will switch on its **immunity genes** to fight the infection. Thus, some of the genes with elevated expression levels after the infection are likely to be immunity genes. Indeed, some of these genes have 12-mers similar to **TCGGGGATTTCC** in their upstream regions, the binding site of a transcription factor called **NF-κB** that activates various immunity genes in flies. However, NF-κB binding sites are nowhere near as conserved as the evening element. Figure 2.1 shows ten NF-κB binding sites from the *Drosophila melanogaster* genome; the most popular nucleotides in every column are shown by upper case colored letters.

```
 1   T C G G G G g T T T t t
 2   c C G G t G A c T T a C
 3   a C G G G G A T T T t C
 4   T t G G G G A c T T t t
 5   a a G G G G A c T T C C
 6   T t G G G G A c T T C C
 7   T C G G G G A T T c a t
 8   T C G G G G A T T c C t
 9   T a G G G G A a c T a C
10   T C G G G t A T a a C C
```

FIGURE 2.1 The ten candidate NF-κB binding sites appearing in the *Drosophila melanogaster* genome. The upper case colored letters indicate the most frequent nucleotide in each column.

Hide and seek with motifs

Our aim is to turn the biological challenge of finding regulatory motifs into a computational problem. Below, we have implanted a 15-mer hidden message at a randomly selected position in each of ten randomly generated DNA strings. This example mimics a transcription factor binding site hiding in the upstream regions of ten genes.

```
 1  atgaccgggatactgataaaaaaaaggggggggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
 2  acccctatttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaataaaaaaaaggggggga
 3  tgagtatccctgggatgacttaaaaaaaaggggggtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
 4  gctgagaattggatgaaaaaaaaggggggtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
 5  tccctttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaataaaaaaaaggggggcttatag
 6  gtcaatcatgttcttgtgaatggatttaaaaaaaaggggggggaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
 7  cggttttggcccttgttagaggcccccgtaaaaaaaaggggggcaattatgagagagctaatctatcgcgtgcgtgttcat
 8  aacttgagttaaaaaaaaggggggggctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
 9  ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcataaaaaaaaggggggggaccgaaaggaag
10  ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttaaaaaaaaggggggga
```

STOP and Think: Can you find the implanted hidden message?

This is a simple problem: applying an algorithm for the Frequent Words Problem to the concatenation of these strings will immediately reveal the most frequent 15-mer shown below as the implanted pattern. Since these short strings were randomly generated, it is unlikely that they contain other frequent 15-mers.

```
 1  atgaccgggatactgatAAAAAAAAGGGGGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
 2  acccctatttttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaactttccgaataAAAAAAAAGGGGGGGa
 3  tgagtatccctgggatgacttAAAAAAAAGGGGGGGtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
 4  gctgagaattggatgAAAAAAAAGGGGGGGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
 5  tccctttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAAAAAAAAGGGGGGGcttatag
 6  gtcaatcatgttcttgtgaatggatttAAAAAAAAGGGGGGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
 7  cggttttggccttgttagaggccccgtAAAAAAAAGGGGGGGcaattatgagagagctaatctatcgcgtgcgtgttcat
 8  aacttgagttAAAAAAAAGGGGGGGctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
 9  ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatAAAAAAAAGGGGGGGaccgaaagggaag
10  ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttAAAAAAAAGGGGGGGa
```

Now imagine that instead of implanting exactly the same pattern into all sequences, we mutate the pattern before inserting it into each sequence by randomly changing the nucleotides at four randomly selected positions within each implanted 15-mer, as shown below.

```
 1  atgaccgggatactgatAgAAgAAAGGttGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
 2  acccctatttttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaactttccgaatacAAtAAAAcGGcGGGa
 3  tgagtatccctgggatgacttAAAAtAAtGGaGtGGtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
 4  gctgagaattggatgcAAAAAAAGGGattGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
 5  tccctttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAtAAtAAAGGaaGGGcttatag
 6  gtcaatcatgttcttgtgaatggatttAAcAAtAAGGGctGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
 7  cggttttggccttgttagaggccccgtAtAAAcAAGGaGGGccaattatgagagagctaatctatcgcgtgcgtgttcat
 8  aacttgagttAAAAAAtAGGGaGccctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
 9  ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatActAAAAAGGaGcGGaccgaaagggaag
10  ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttActAAAAAGGaGcGGa
```

The Frequent Words Problem is not going to help us, since **AAAAAAAAGGGGGGG** does not even appear in the sequences above. Perhaps, then, we could apply our solution to the Frequent Words with Mismatches Problem. However, in Chapter 1, we implemented an algorithm for the Frequent Words with Mismatches Problem aimed at finding hidden messages with a small number of mismatches and a small k-mer size (e.g., one or two mismatches for *DnaA* boxes of length 9). This algorithm is likely to become too slow when searching for the implanted motif above, which is longer and has more mutations.

Furthermore, concatenating all the sequences into a single string is inadequate because it does not correctly model the biological problem of motif finding. A *DnaA* box is a pattern that clumps, or appears frequently, within a relatively short interval of the genome. In contrast, a regulatory motif is a pattern that appears at least once (perhaps with variation) in each of many different regions that are scattered throughout the genome.

A brute force algorithm for motif finding

Given a collection of strings *Dna* and an integer d, a k-mer is a **(k, d)-motif** if it appears in every string from *Dna* with at most d mismatches. For example, the implanted 15-mer in the strings above represents a $(15, 4)$-motif.

Implanted Motif Problem:
Find all (k, d)-motifs in a collection of strings.

Input: A collection of strings *Dna*, and integers k and d.
Output: All (k, d)-motifs in *Dna*.

Brute force (also known as **exhaustive search**) is a general problem-solving technique that explores all possible candidate solutions and checks whether each candidate solves the problem. Such algorithms require little effort to design and are guaranteed to produce a correct solution, but they may take an enormous amount of time, and the number of candidates may be too large to check.

A brute force approach for solving the Implanted Motif Problem is based on the observation that any (k, d)-motif must be at most d mismatches apart from some k-mer appearing in one of the strings of *Dna*. Therefore, we can generate all such k-mers and then check which of them are (k, d)-motifs. If you have forgotten how to generate these k-mers, recall **CHARGING STATION: Generating the Neighborhood of a String**.

PAGE 49


```
MotifEnumeration(Dna, k, d)
    Patterns ← an empty set
    for each k-mer Pattern in Dna
        for each k-mer Pattern' differing from Pattern by at most d mismatches
            if Pattern' appears in each string from Dna with at most d mismatches
                add Pattern' to Patterns
    remove duplicates from Patterns
    return Patterns
```

MotifEnumeration is unfortunately rather slow for large values of k and d, and so we will try a different approach instead. Maybe we can detect an implanted pattern simply by identifying the two most similar k-mers between each pair of strings in *Dna*? However, consider the implanted 15-mers **AgAAgAAAGGttGGG** and **cAAtAAAAcGGGGcG**, each of which differs from **AAAAAAAAGGGGGGG** by four mismatches. Although these

15-mers look similar to the correct motif **AAAAAAAAGGGGGGG**, they are not so similar when compared to each other, having eight mismatches:

```
AgAAgAAAGGttGGG
||  ||    |  ||  |
cAAtAAAAcGGGGcG
```

Since these two implanted patterns are so different, we should be concerned whether we will be able to find them by searching for the most similar k-mers among pairs of strings in *Dna*.

In the rest of the chapter, we will benchmark our motif finding algorithms by using a particularly challenging instance of the Implanted Motif Problem. The **Subtle Motif Problem** refers to implanting a 15-mer with four random mutations in ten randomly generated 600 nucleotide-long strings (the typical length of many upstream regulatory regions). The instance of the Subtle Motif Problem that we will use has the implanted 15-mer **AAAAAAAAGGGGGGG**.

It turns out that thousands of pairs of randomly occurring 15-mers in our dataset for the Subtle Motif Problem are fewer than 8 nucleotides apart from each other, preventing us from identifying the true implanted motifs by pairwise comparisons.

Scoring Motifs

From motifs to profile matrices and consensus strings

Although the Implanted Motif Problem offers a useful abstraction of the biological problem of motif finding, it has some limitations. For example, when Steve Kay used a DNA array to infer the set of circadian genes in plants, he did not expect that *all* genes in the resulting set would have the evening element (or its variants) in their upstream regions. Similarly, biologists do not expect that all genes with an elevated expression level in infected flies must be regulated by NF-$\varkappa$B. DNA array experiments are inherently noisy, and some genes identified by these experiments have nothing to do with the circadian clock in plants or immunity genes in flies. For such noisy datasets, any algorithm for the Implanted Motif Problem would fail, because as long as a single sequence does not contain the transcription factor binding site, a (k, d)-motif does not exist!

A more appropriate problem formulation would score individual instances of motifs depending on how similar they are to an "ideal" motif (i.e., a transcription factor binding site that binds the best to the transcription factor). However, since the ideal

motif is unknown, we attempt to select a k-mer from each string and score these k-mers depending on how similar they are *to each other*.

To define scoring, consider t DNA strings, each of length n, and select a k-mer from each string to form a collection *Motifs*, which we represent as a $t \times k$ **motif matrix**. In Figure 2.2, which shows the motif matrix for the NF-$\varkappa$B binding sites from Figure 2.1, we indicate the most frequent nucleotide in each column of the motif matrix by upper case letters. If there are multiple most popular nucleotides in a column, then we arbitrarily select one of them to break the tie. Note that positions 2 and 3 are the most conserved (nucleotide **G** is completely conserved in these positions), whereas position 10 is the least conserved.

By varying the choice of k-mers in each string, we can construct a large number of different motif matrices from a given sample of DNA strings. Our goal is to select k-mers resulting in the most "conserved" motif matrix, meaning the matrix with the most upper case letters (and thus the fewest number of lower case letters). Leaving aside the question of how we select such k-mers, we will first focus on how to score the resulting motif matrices, defining SCORE(*Motifs*) as the number of unpopular (lower case) letters in the motif matrix *Motifs*. Our goal is to find a collection of k-mers that *minimizes* this score.

EXERCISE BREAK: The minimum possible value of SCORE(*Motifs*) is 0 (if all rows in the $t \times k$ matrix *Motifs* are the same). What is the maximum possible value of SCORE(*Motifs*) in terms of t and k?

We can construct the $4 \times k$ **count matrix** COUNT(*Motifs*) counting the number of occurrences of each nucleotide in each column of the motif matrix; the (i, j)-th element of COUNT(*Motifs*) stores the number of times that nucleotide i appears in column j of *Motifs*. We will further divide all of the elements in the count matrix by t, the number of rows in *Motifs*. This results in a **profile matrix** $P = \text{PROFILE}(Motifs)$ for which $P_{i,j}$ is the *frequency* of the i-th nucleotide in the j-th column of the motif matrix. Note that the elements of any column of the profile matrix sum to 1.

Finally, we form a **consensus string**, denoted CONSENSUS(*Motifs*), from the most popular nucleotides in each column of the motif matrix (ties are broken arbitrarily). If we select *Motifs* correctly from the collection of upstream regions, then CONSENSUS(*Motifs*) provides an ideal candidate regulatory motif for these regions. For example, the consensus string for the NF-$\varkappa$B binding sites in Figure 2.2 is **TCGGGGATTTCC**.

Motifs

T	**C**	**G**	**G**	**G**	**G**	a	**T**	**T**	**T**	t	t
c	**C**	**G**	**G**	t	**G**	**A**	c	**T**	**T**	a	**C**
a	**C**	**G**	**G**	**G**	**G**	**A**	**T**	**T**	**T**	t	**C**
T	t	**G**	**G**	**G**	**G**	**A**	c	**T**	**T**	t	t
a	a	**G**	**G**	**G**	**G**	**A**	c	**T**	**T**	**C**	**C**
T	t	**G**	**G**	**G**	**G**	**A**	c	**T**	**T**	**C**	**C**
T	**C**	**G**	**G**	**G**	**G**	**A**	**T**	**T**	c	a	t
T	**C**	**G**	**G**	**G**	**G**	**A**	**T**	**T**	c	**C**	t
T	a	**G**	**G**	**G**	**G**	**A**	a	c	**T**	a	**C**
T	**C**	**G**	**G**	**G**	t	**A**	**T**	a	a	**C**	**C**

SCORE(*Motifs*) $3 + 4 + 0 + 0 + 1 + 1 + 1 + 5 + 2 + 3 + 6 + 4 = 30$

COUNT(*Motifs*)

A:	2	2	0	0	0	0	**9**	1	1	1	3	0
C:	1	**6**	0	0	0	0	0	4	1	2	4	**6**
G:	0	0	**10**	**10**	**9**	**9**	1	0	0	0	0	0
T:	**7**	2	0	0	1	1	0	**5**	**8**	**7**	3	4

PROFILE(*Motifs*)

A:	.2	.2	0	0	0	0	**.9**	.1	.1	.1	.3	0
C:	.1	**.6**	0	0	0	0	0	.4	.1	.2	.4	**.6**
G:	0	0	**1**	**1**	**.9**	**.9**	.1	0	0	0	0	0
T:	**.7**	.2	0	0	.1	.1	0	**.5**	**.8**	**.7**	.3	.4

CONSENSUS(*Motifs*) **T C G G G G A T T T C C**

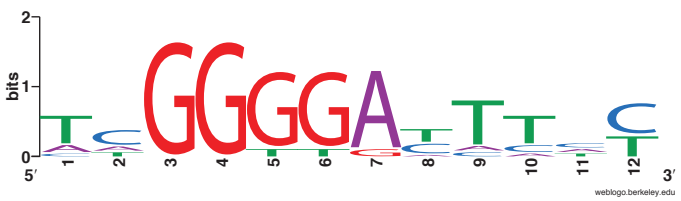


FIGURE 2.2 From motif matrix to count matrix to profile matrix to consensus string to motif logo. The NF-κB binding sites form a 10×12 motif matrix, with the most frequent nucleotide in each column shown in upper case letters and all other nucleotides shown in lower case letters. SCORE(*Motifs*) counts the total number of unpopular (lower case) symbols in the motif matrix. The motif matrix results in a 4×12 count matrix holding the nucleotide counts in every column of the motif matrix; a profile matrix holding the frequencies of nucleotides in every column of the motif matrix; and a consensus string formed by the most frequent nucleotide in each column of the count matrix. Finally, the motif logo is a common way to visualize the conservation of various positions within a motif. The total height of the letters depicts the information content of the position.

Towards a more adequate motif scoring function

Consider the second column (containing 6 C, 2 A, and 2 T) and the final column (containing 6 C and 4 T) in the motif matrix from Figure 2.2. Both of these columns contribute 4 to SCORE(*Motifs*).

STOP and Think: Does scoring these two columns equally make sense biologically?

For many biological motifs, certain positions feature two nucleotides with roughly the same ability to bind to a transcription factor. For example, the sixteen nucleotide-long CSRE transcription factor binding site in the yeast *S. cerevisiae* consists of five strongly conserved positions in addition to eleven weakly conserved positions, each of which features two nucleotides with similar frequencies (see Figure 2.3).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C	G/C	G/T	T/A	C/T	G/C	C/G	A	T	G/T	C/G	A	T	C/T	C/T	G/T

FIGURE 2.3 The CSRE transcription factor binding site in *S. cerevisiae* is 16 nucleotides long, but only five of these positions (1, 8, 9, 12, 13) are strongly conserved. The remaining 11 positions can take one of two different nucleotides.

Following this example, a more appropriate representation of the consensus string TCGGGGATTTCC for the NF-κB binding sites should include viable alternatives to the most popular nucleotides in each column (see Figure 2.4). In this sense, the last column (6 C, 4 T) in the motif matrix from Figure 2.2 is "more conserved" than the second column (6 C, 2 A, 2 T) and should receive a lower score.

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

FIGURE 2.4 Taking nucleotides in each column of the NF-κB binding site motif matrix from Figure 2.2 with frequency at least 0.4 yields a representation of the NF-κB binding sites with ten strongly conserved positions and two weakly conserved positions (8 and 12).

Entropy and the motif logo

Every column of PROFILE(*Motifs*) corresponds to a **probability distribution**, or a collection of nonnegative numbers that sum to 1. For example, the second column in Figure 2.2 corresponds to the probabilities 0.2, 0.6, 0.0, and 0.2 for A, C, G, and T, respectively.

Entropy is a measure of the uncertainty of a probability distribution $(p_1, \ldots, p_N)$, and is defined as

$$H(p_1, \ldots, p_N) = -\sum_{i=1}^{N} p_i \cdot \log_2 (p_i) .$$

For example, the entropy of the probability distribution (0.2, 0.6, 0.0, 0.2) corresponding to the second column of the profile matrix in Figure 2.2 is

$$-(0.2 \log_2 0.2 + 0.6 \log_2 0.6 + 0.0 \log_2 0.0 + 0.2 \log_2 0.2) \approx 1.371 ,$$

whereas the entropy of the more conserved final column $(0.0, 0.6, 0.0, 0.4)$ is

$$-(0.0 \log_2 0.0 + 0.6 \log_2 0.6 + 0.0 \log_2 0.0 + 0.4 \log_2 0.4) \approx 0.971 ,$$

and the entropy of the very conserved 5th column $(0.0, 0.0, 0.9, 0.1)$ is

$$-(0.0 \log_2 0.0 + 0.0 \log_2 0.0 + 0.9 \log_2 0.9 + 0.1 \log_2 0.1) \approx 0.467 .$$

Note that technically, $\log_2 0$ is not defined, but in the computation of entropy, we assume that $0 \cdot \log_2 0$ is equal to 0.

STOP and Think: What are the maximum and minimum possible values for the entropy of a probability distribution containing four values?

The entropy of the completely conserved third column of the profile matrix in Figure 2.2 is 0, which is the minimum possible entropy. On the other hand, a column with equally-likely nucleotides (all probabilities equal to $1/4$) has maximum possible entropy $-4 \cdot 1/4 \cdot \log_2 (1/4) = 2$. In general, the more conserved the column, the smaller its entropy. Thus, entropy offers an improved method of scoring motif matrices: the entropy of a motif matrix is defined as the sum of the entropies of its columns. In this book, we will continue to use SCORE(*Motifs*) for simplicity, but the entropy score is used more often in practice.

EXERCISE BREAK: Compute the entropy of the NF-κB motif matrix from Figure 2.2.

Another application of entropy is the **motif logo**, a diagram for visualizing motif conservation that consists of a stack of letters at each position (see the bottom of Figure 2.2). The relative sizes of letters indicate their frequency in the column. The total height of the letters in each column is based on the **information content** of the column, which is defined as $2 - H(p_1, \ldots, p_N)$. The lower the entropy, the higher the information content, meaning that tall columns in the motif logo are highly conserved.

From Motif Finding to Finding a Median String

The Motif Finding Problem

Now that we have a good grasp of scoring a collection of k-mers, we are ready to formulate the Motif Finding Problem.

Motif Finding Problem:
Given a collection of strings, find a set of k-mers, one from each string, that minimizes the score of the resulting motif.

Input: A collection of strings *Dna* and an integer k.
Output: A collection *Motifs* of k-mers, one from each string in *Dna*, minimizing SCORE(*Motifs*) among all possible choices of k-mers.

A brute force algorithm for the Motif Finding Problem, **BRUTEFORCEMOTIFSEARCH**, considers every possible choice of k-mers *Motifs* from *Dna* (one k-mer from each string of n nucleotides) and returns the collection *Motifs* having minimum score. Because there are $n - k + 1$ choices of k-mers in each of t sequences, there are $(n - k + 1)^t$ different ways to form *Motifs*. For each choice of *Motifs*, the algorithm calculates SCORE(*Motifs*), which requires $k \cdot t$ steps. Thus, assuming that k is much smaller than n, the overall running time of the algorithm is $\mathcal{O}(n^t \cdot k \cdot t)$. We need to come up with a faster algorithm!

Reformulating the Motif Finding Problem

Because **BRUTEFORCEMOTIFSEARCH** is inefficient, we will think about motif finding in a different way. Instead of exploring all *Motifs* in *Dna* and deriving the consensus

string from *Motifs* afterwards,

$$Motifs \rightarrow \text{CONSENSUS}(Motifs)\,,$$

we will explore all potential k-mer consensus strings first and then find the best possible collection *Motifs* for each consensus string,

$$\text{CONSENSUS}(Motifs) \rightarrow Motifs\,.$$

To reformulate the Motif Finding Problem, we need to devise an alternative way of computing $\text{SCORE}(Motifs)$. Until now, we have computed $\text{SCORE}(Motifs)$, the number of lower case letters in the motif matrix, column-by-column. For example, in Figure 2.2, we computed $\text{SCORE}(Motifs)$ for the NF-$\varkappa$B motif matrix as

$$3+4+0+0+1+1+1+5+2+3+6+4=30\,.$$

Figure 2.5 illustrates that $\text{SCORE}(Motifs)$ can just as easily be computed row-by-row as

$$3+4+2+4+3+2+3+2+4+3=30\,.$$

Note that each element in the latter sum represents the number of mismatches between the consensus string **TCGGGGATTTCC** and a motif in the corresponding row of the motif matrix, i.e., the Hamming distance between these strings. For the first row of the motif matrix in Figure 2.5, $d(\texttt{TCGGGGATTTCC}, \texttt{TCGGGGgTTTtt}) = 3$.

Given a collection of k-mers $Motifs = \{Motif_1, ..., Motif_t\}$ and a k-mer *Pattern*, we now define $d(Pattern, Motifs)$ as the sum of Hamming distances between *Pattern* and each $Motif_i$,

$$d(Pattern, Motifs) = \sum_{i=1}^{t} \text{HAMMINGDISTANCE}(Pattern, Motif_i)\,.$$

Because $\text{SCORE}(Motifs)$ corresponds to counting the lower case elements of *Motifs* column-by-column and $d(\text{CONSENSUS}(Motifs), Motifs)$ corresponds to counting these elements row-by-row, we obtain that

$$\text{SCORE}(Motifs) = d(\text{CONSENSUS}(Motifs), Motifs)\,.$$

	T	C	G	G	G	G	g	T	T	T	t	t	**3**
	c	C	G	G	t	G	A	c	T	T	a	C	**4**
	a	C	G	G	G	G	A	T	T	T	t	C	**2**
	T	t	G	G	G	G	A	c	T	T	t	t	**4**
Motifs	a	a	G	G	G	G	A	c	T	T	C	C	**3**
	T	t	G	G	G	G	A	c	T	T	C	C	**2**
	T	C	G	G	G	G	A	T	T	c	a	t	**3**
	T	C	G	G	G	G	A	T	T	c	C	t	**2**
	T	a	G	G	G	G	A	a	c	T	a	C	**4**
	T	C	G	G	G	t	A	T	a	a	C	C	**+ 3**
SCORE(*Motifs*)	3	+	4	+	0	+	0	+	1	+	1	+	1	+	5	+	2	+	3	+	6	+	4	=	**30**
CONSENSUS(*Motifs*)	T	C	G	G	G	G	A	T	T	T	C	C	

FIGURE 2.5 The motif and score matrices in addition to the consensus string for the NF-κB binding sites, reproduced from Figure 2.2. Rather than add the non-consensus elements (i.e., lower case nucleotides) column-by-column, we can add them row-by-row, as highlighted on the right of the motifs matrix. Each value at the end of a row corresponds to the Hamming distance between that row and the consensus string.

This equation gives us an idea. Instead of searching for a collection of k-mers *Motifs* minimizing SCORE(*Motifs*), let's instead search for a potential consensus string *Pattern* minimizing $d(Pattern, Motifs)$ among all possible k-mers *Pattern* and all possible choices of k-mers *Motifs* in *Dna*. This problem is equivalent to the Motif Finding Problem.

Equivalent Motif Finding Problem:
Given a collection of strings, find a pattern and a collection of k-mers (one from each string) that minimizes the distance between all possible patterns and all possible collections of k-mers.

Input: A collection of strings *Dna* and an integer k.
Output: A k-mer *Pattern* and a collection of k-mers *Motifs*, one from each string in *Dna*, minimizing $d(Pattern, Motifs)$ among all possible choices of *Pattern* and *Motifs*.

The Median String Problem

But wait a second — have we not just made our task more difficult? Instead of having to search for all *Motifs*, we now have to search all *Motifs* as well as all k-mers *Pattern*. The key observation for solving the Equivalent Motif Finding Problem is that, given *Pattern*, we don't need to explore all possible collections *Motifs* in order to minimize $d(Pattern, Motifs)$.

To explain how this can be done, we define MOTIFS$(Pattern, Dna)$ as a collection of k-mers that minimizes $d(Pattern, Motifs)$ for a given *Pattern* and all possible sets of k-mers *Motifs* in *Dna*. For example, for the strings *Dna* shown below, the five colored 3-mers represent MOTIFS(**AAA**, *Dna*).

```
        ttaccttAAC
        gATAtctgtc
Dna     ACGgcgttcg
        ccctAAAgag
        cgtcAGAggt
```

STOP and Think: Given a collection of strings *Dna* and a k-mer *Pattern*, design a fast algorithm for generating MOTIFS$(Pattern, Dna)$.

The reason why we don't need to consider all possible collections *Motifs* in $Dna = \{Dna_1, ..., Dna_t\}$ is that we can generate the k-mers in MOTIFS$(Pattern, Dna)$ one at a time; that is, we can select a k-mer in Dna_i independently of selecting k-mers in all other strings in *Dna*. Given a k-mer *Pattern* and a longer string *Text*, we use $d(Pattern, Text)$ to denote the minimum Hamming distance between *Pattern* and any k-mer in *Text*,

$$d(Pattern, Text) = \min_{\text{all } k\text{-mers } Pattern' \text{ in } Text} \text{HAMMINGDISTANCE}(Pattern, Pattern').$$

For example,

$$d(\text{GATTCTCA}, \text{gcaaaGACGCTGAccaa}) = 3.$$

A k-mer in *Text* that achieves the minimum Hamming distance with *Pattern* is denoted MOTIF$(Pattern, Text)$. For the above example,

$$\text{MOTIF}(\text{GATTCTCA}, \text{gcaaaGACGCTGAccaa}) = \text{GACGCTGA}.$$

We note that the notation MOTIF(*Pattern*, *Text*) is ambiguous because there may be multiple *k*-mers in *Text* that achieve the minimum Hamming distance with *Pattern*. For example, MOTIF(**AAG**, gc**AAT**cct**CAG**c) could be either **AAT** or **CAG**. However, this ambiguity does not affect the following analysis.

Given a *k*-mer *Pattern* and a set of strings $Dna = \{Dna_1, ..., Dna_t\}$, we define $d(Pattern, Dna)$ as the sum of distances between *Pattern* and all strings in *Dna*,

$$d(Pattern, Dna) = \sum_{i=1}^{t} d(Pattern, Dna_i).$$

For example, for the strings *Dna* shown below, $d(\mathbf{AAA}, Dna) = \mathbf{1} + \mathbf{1} + \mathbf{2} + \mathbf{0} + \mathbf{1} = \mathbf{5}$.

	ttaccttAAC	1
	gATAtctgtc	1
Dna	ACGgcgttcg	2
	ccctAAAgag	0
	cgtcAGAggt	1

Our goal is to find a *k*-mer *Pattern* that minimizes $d(Pattern, Dna)$ over all *k*-mers *Pattern*, the same task that the Equivalent Motif Finding Problem is trying to achieve. We call such a *k*-mer a **median string** for *Dna*.

Median String Problem:
Find a median string.

Input: A collection of strings *Dna* and an integer *k*.
Output: A *k*-mer *Pattern* minimizing $d(Pattern, Dna)$ among all *k*-mers *Pattern*.

2B

Notice that finding a median string requires solving a double minimization problem. We must find a *k*-mer *Pattern* that minimizes $d(Pattern, Dna)$, where this function is itself computed by taking a minimum over all choices of *k*-mers from each string in *Dna*. The pseudocode for a brute-force algorithm, MEDIANSTRING, is given below.

```
MEDIANSTRING(Dna, k)
    distance ← ∞
    for each k-mer Pattern from AA...AA to TT...TT
        if distance > d(Pattern, Dna)
            distance ← d(Pattern, Dna)
            Median ← Pattern
    return Median
```

CHARGING STATION (Solving the Median String Problem): Although this pseudocode is short, it is not without potential pitfalls. Check out this Charging Station if you fall into one of them.

STOP and Think: Instead of making a time-consuming search through all possible k-mers in **MEDIANSTRING**, can you only search through all k-mers that appear in *Dna*?

Why have we reformulated the Motif Finding Problem?

To see why we reformulated the Motif Finding Problem as the equivalent Median String Problem, consider the runtimes of **MEDIANSTRING** and **BRUTEFORCEMOTIFS**. The former algorithm computes $d(Pattern, Dna)$ for each of the 4^k k-mers *Pattern*. Each computation of $d(Pattern, Dna)$ requires a single pass over each string in *Dna*, which requires approximately $k \cdot n \cdot t$ operations for t strings of length n in *Dna*. Therefore, **MEDIANSTRING** has a running time of $\mathcal{O}\left(4^k \cdot n \cdot k \cdot t\right)$, which in practice compares favorably with the $\mathcal{O}\left(n^t \cdot k \cdot t\right)$ running time of **BRUTEFORCEMOTIFSEARCH** because the length of a motif (k) typically does not exceed 20 nucleotides, whereas t is measured in the thousands.

The Median String Problem teaches an important lesson, which is that sometimes rethinking how a problem is formulated can lead to dramatic improvements in the runtime required to solve it. In this case, our simple observation that SCORE(*Motifs*) could just as easily be computed row-by-row as column-by-column produced the faster **MEDIANSTRING** algorithm.

Of course, the ultimate test of a bioinformatics algorithm is how it performs in practice. Unfortunately, since **MEDIANSTRING** has to consider 4^k k-mers, it becomes too slow for the Subtle Motif Problem, for which $k = 15$. We will run **MEDIANSTRING**

with $k = 13$ in the hope that it will capture a substring of the correct 15-mer motif. The algorithm still requires half a day to run on our computer and returns the median string AAAAAtAGaGGGG (with distance 29). This 13-mer is not a substring of the implanted pattern AAAAAAAAGGGGGGG, but it does come close.

STOP and Think: How can a slightly incorrect median string of length 13 help us find the correct median string of length 15?

We have thus far assumed that the value of k is known in advance, which is not the case in practice. As a result, we are forced to run our motif finding algorithms for different values of k and then try to deduce the correct motif length. Since some regulatory motifs are rather long — later in the chapter, we will search for a biologically important motif of length 20 — MEDIANSTRING may be too slow to find them.

Greedy Motif Search

Using the profile matrix to roll dice

Many algorithms are iterative procedures that must choose among many alternatives at each iteration. Some of these alternatives may lead to correct solutions, whereas others may not. **Greedy algorithms** select the "most attractive" alternative at each iteration. For example, a greedy algorithm in chess might attempt to capture an opponent's most valuable piece at every move. Yet anyone who has played chess knows that this strategy of looking only one move ahead will likely produce disastrous results. In general, most greedy algorithms typically fail to find an exact solution of the problem; instead, they are often fast **heuristics** that trade accuracy for speed in order to find an *approximate* solution. Nevertheless, for many biological problems that we will study in this book, greedy algorithms will prove quite useful.

In this section, we will explore a greedy approach to motif finding. Again, let *Motifs* be a collection of k-mers taken from t strings *Dna*. Recall from our discussion of entropy that we can view each column of PROFILE(*Motifs*) as a four-sided biased die. Thus, a profile matrix with k columns can be viewed as a collection of k dice, which we will roll to randomly generate a k-mer. For example, if the first column of the profile matrix is (0.2, 0.1, 0.0, 0.7), then we generate A as the first nucleotide with probability 0.2, C with probability 0.1, G with probability 0.0, and T with probability 0.7.

In Figure 2.6, we reproduce the profile matrix for the NF-$\varkappa$B binding sites from Figure 2.2, where the lone colored entry in the i-th column corresponds to the i-th

nucleotide in ACGGGGATTACC. The probability Pr(ACGGGGATTACC|*Profile*) that *Profile* generates ACGGGGATTACC is computed by simply multiplying the highlighted entries in the profile matrix.

Profile	A:	**.2**	.2	.0	.0	.0	.0	**.9**	.1	.1	**.1**	.3	.0
	C:	.1	**.6**	.0	.0	.0	.0	.0	.4	.1	**.2**	**.4**	**.6**
	G:	.0	.0	**1**	**1**	**.9**	**.9**	.1	.0	.0	.0	.0	.0
	T:	.7	.2	.0	.0	.1	.1	.0	**.5**	**.8**	.7	.3	.4

Pr(ACGGGGATTACC|*Profile*) $= .2 \cdot .6 \cdot 1 \cdot 1 \cdot .9 \cdot .9 \cdot .9 \cdot .5 \cdot .8 \cdot .1 \cdot .4 \cdot .6 = 0.000839808$

FIGURE 2.6 We can generate a random string based on a profile matrix by selecting the *i*-th nucleotide in the string with the probability corresponding to that nucleotide in the *i*-th column of the profile matrix. The probability that a profile matrix will produce a given string is given by the product of individual nucleotide probabilities.

A *k*-mer tends to have a higher probability when it is more similar to the consensus string of a profile. For example, for the NF-κB profile matrix shown in Figure 2.6 and its consensus string TCGGGGATTTCC,

$$\begin{aligned}\text{Pr}(\text{TCGGGGATTTCC}|\textit{Profile}) &= 0.7 \cdot 0.6 \cdot 1.0 \cdot 1.0 \cdot 0.9 \cdot 0.9 \cdot 0.9 \cdot 0.5 \cdot 0.8 \cdot 0.7 \cdot 0.4 \cdot 0.6 \\ &= 0.0205753\,,\end{aligned}$$

which is larger than the value of Pr(ACGGGGATTACC|*Profile*) $= 0.000839808$ that we computed in Figure 2.6.

EXERCISE BREAK: Compute Pr(TCGTGGATTTCC|*Profile*), where *Profile* is the matrix shown in Figure 2.6.

Given a profile matrix *Profile*, we can evaluate the probability of every *k*-mer in a string *Text* and find a ***Profile*-most probable** *k*-mer in *Text*, i.e., a *k*-mer that was most likely to have been generated by *Profile* among all *k*-mers in *Text*. For the NF-κB profile matrix, ACGGGGATTACC is the *Profile*-most probable 12-mer in ggtACGGGGATTACCt. Indeed, every other 12-mer in this string has probability 0. In general, if there are multiple *Profile*-most probable *k*-mers in *Text*, then we select the first such *k*-mer occurring in *Text*.

***Profile*-most Probable *k*-mer Problem**:
Find a Profile-most probable k-mer in a string.

Input: A string *Text*, an integer k, and a $4 \times k$ matrix *Profile*.
Output: A *Profile*-most probable k-mer in *Text*.

Our proposed greedy motif search algorithm, **GREEDYMOTIFSEARCH**, tries each of the k-mers in Dna_1 as the first motif. For a given choice of k-mer $Motif_i$ in Dna_1, it then builds a profile matrix *Profile* for this lone k-mer, and sets $Motif_2$ equal to the *Profile*-most probable k-mer in Dna_2. It then iterates by updating *Profile* as the profile matrix formed from $Motif_1$ and $Motif_2$, and sets $Motif_2$ equal to the *Profile*-most probable k-mer in Dna_3. In general, after finding $i-1$ k-mers *Motifs* in the first $i-1$ strings of *Dna*, **GREEDYMOTIFSEARCH** constructs *Profile*(*Motifs*) and selects the *Profile*-most probable k-mer from Dna_i based on this profile matrix. After obtaining a k-mer from each string to obtain a collection *Motifs*, **GREEDYMOTIFSEARCH** tests to see whether *Motifs* outscores the current best scoring collection of motifs and then moves $Motif_1$ one symbol over in Dna_1, beginning the entire process of generating *Motifs* again.

```
GREEDYMOTIFSEARCH(Dna, k, t)
    BestMotifs ← motif matrix formed by first k-mers in each string from Dna
    for each k-mer Motif in the first string from Dna
        Motif_1 ← Motif
        for i = 2 to t
            form Profile from motifs Motif_1, ..., Motif_{i-1}
            Motif_i ← Profile-most probable k-mer in the i-th string in Dna
        Motifs ← (Motif_1, ..., Motif_t)
        if SCORE(Motifs) < SCORE(BestMotifs)
            BestMotifs ← Motifs
    return BestMotifs
```

If you are not satisfied with the performance of **GREEDYMOTIFSEARCH** — even if you implemented it correctly — then wait until we discuss this algorithm in the next section.

Analyzing greedy motif finding

In contrast to **MEDIANSTRING**, **GREEDYMOTIFSEARCH** is fast and can be run with $k = 15$ to solve the Subtle Motif Problem (recall that we settled for $k = 13$ in the

case of MEDIANSTRING). However, it trades speed for accuracy and returns the 15-mer **gtAAAtAgaGatGtG** (total distance: 58), which is very different from the true implanted motif **AAAAAAAAGGGGGGG**.

STOP and Think: Why does GREEDYMOTIFSEARCH perform so poorly?

At first glance, GREEDYMOTIFSEARCH may seem like a reasonable algorithm, but it is not! Let's see whether GREEDYMOTIFSEARCH will find the (4, 1)-motif **ACGT** implanted in the following strings *Dna*:

```
ttACCTtaac
gATGTctgtc
acgGCGTtag
ccctaACGAg
cgtcagAGGT
```

We will assume that the algorithm has already correctly chosen the implanted 4-mer **ACCT** from the first string in *Dna* and constructed the corresponding *Profile*:

A:	**1**	0	0	0
C:	0	**1**	**1**	0
G:	0	0	0	0
T:	0	0	0	**1**

The algorithm is now ready to search for a *Profile*-most probable 4-mer in the second sequence. The issue, however, is that there are so many zeros in the profile matrix that the probability of every 4-mer but **ACCT** is zero! Thus, unless **ACCT** is present in every string in *Dna*, there is little chance that GREEDYMOTIFSEARCH will find the implanted motif. Zeroes in the profile matrix are not just a minor annoyance but rather a persistent problem that we must address.

Motif Finding Meets Oliver Cromwell

What is the probability that the sun will not rise tomorrow?

In 1650, after the Scots proclaimed Charles II as king during the English Civil War, Oliver Cromwell made a famous appeal to the Church of Scotland. Urging them to see the error of their royal alliance, he pleaded,

I beseech you, in the bowels of Christ, think it possible that you may be mistaken.

The Scots rejected the appeal, and Cromwell invaded Scotland in response. His quotation later inspired the statistical maxim called **Cromwell's rule**, which states that we should not use probabilities of 0 or 1 unless we are talking about logical statements that can only be true or false. In other words, we should allow a small probability for extremely unlikely events, such as "this book was written by aliens" or "the sun will not rise tomorrow". We cannot speak to the likelihood of the former event, but in the 18th Century, the French mathematician Pierre-Simon Laplace actually estimated the probability that the sun will not rise tomorrow (1/1826251), given that it has risen every day for the past 5000 years. Although this estimate was ridiculed by his contemporaries, Laplace's approach to this question now plays an important role in statistics.

In any observed data set, there is the possibility, especially with low-probability events or small data sets, that an event with nonzero probability does not occur. Its observed frequency is therefore zero; however, setting the empirical probability of the event equal to zero represents an inaccurate oversimplification that may cause problems. By artificially adjusting the probability of rare events, these problems can be mitigated.

Laplace's Rule of Succession

Cromwell's rule is relevant to the calculation of the probability of a string based on a profile matrix. For example, consider the following *Profile*:

Profile	**A**:	.2	.2	.0	.0	.0	.0	**.9**	.1	.1	.1	.3	.0
	C:	.1	**.6**	.0	.0	.0	.0	.0	.4	.1	.2	**.4**	**.6**
	G:	.0	.0	**1**	1	**.9**	**.9**	.1	.0	.0	.0	.0	.0
	T:	**.7**	.2	.0	**.0**	.1	.1	.0	**.5**	**.8**	**.7**	.3	.4

$$\Pr(\texttt{TCGTGGATTTCC}|\mathit{Profile}) = .7 \cdot .6 \cdot 1 \cdot .0 \cdot .9 \cdot .9 \cdot .9 \cdot .5 \cdot .8 \cdot .7 \cdot .4 \cdot .6 = 0$$

The fourth symbol of **TCGTGGATTTCC** causes $\Pr(\texttt{TCGTGGATTTCC}|\mathit{Profile})$ to equal zero. As a result, the entire string is assigned a zero probability, even though **TCGTGGATTTCC** differs from the consensus string at only one position. For that matter, **TCGTGGATTTCC** has the same low probability as **AAATCTTGGAA**, which is very different from the consensus string.

In order to improve this unfair scoring, bioinformaticians often substitute zeroes with small numbers called **pseudocounts**. The simplest approach to introducing pseudocounts, called **Laplace's Rule of Succession**, is similar to the principle that Laplace

used to calculate the probability that the sun will not rise tomorrow. In the case of motifs, pseudocounts often amount to adding 1 (or some other small number) to each element of COUNT(*Motifs*). For example, say that we have the following motif, count, and profile matrices:

Motifs

```
T A A C
G T C T
A C T A
A G G T
```

COUNT(*Motifs*)

A:	2	1	1	1
C:	0	1	1	1
G:	1	1	1	0
T:	1	1	1	2

PROFILE(*Motifs*)

2/4	1/4	1/4	1/4
0	1/4	1/4	1/4
1/4	1/4	1/4	0
1/4	1/4	1/4	2/4

Laplace's Rule of Succession adds 1 to each element of COUNT(*Motifs*), updating the two matrices to the following:

COUNT(*Motifs*)

A:	2+1	1+1	1+1	1+1
C:	0+1	1+1	1+1	1+1
G:	1+1	1+1	1+1	0+1
T:	1+1	1+1	1+1	2+1

PROFILE(*Motifs*)

3/8	2/8	2/8	2/8
1/8	2/8	2/8	2/8
2/8	2/8	2/8	1/8
2/8	2/8	2/8	3/8

STOP and Think: How would you use Laplace's Rule of Succession to address the shortcomings of **GREEDYMOTIFSEARCH**?

An improved greedy motif search

The only change we need to introduce to **GREEDYMOTIFSEARCH** in order to eliminate zeroes from the profile matrices that it constructs is to replace line 6 of the pseudocode for **GREEDYMOTIFSEARCH**:

form *Profile* from motifs $Motif_1$, ... $Motif_{i-1}$

with the following line:

apply Laplace's Rule of Succession to form *Profile* from motifs $Motif_1$, ... $Motif_{i-1}$

We now will apply Laplace's Rule of Succession to search for the (4, 1)-motif **ACGT** implanted in the following strings *Dna*:

	tt**ACCT**taac
	g**ATGT**ctgtc
Dna	acg**GCGT**tag
	ccct**aACGA**g
	cgtcag**AGGT**

Again, let's assume that the algorithm has already chosen the implanted 4-mer **ACCT** from the first sequence. We can construct the corresponding count and profile matrices using Laplace's Rule of Succession:

Motifs **ACCT**

COUNT(*Motifs*)

A:	1+1	0+1	0+1	0+1
C:	0+1	1+1	1+1	0+1
G:	0+1	0+1	0+1	0+1
T:	0+1	0+1	0+1	1+1

PROFILE(*Motifs*)

2/5	1/5	1/5	1/5
1/5	2/5	2/5	1/5
1/5	1/5	1/5	1/5
1/5	1/5	1/5	2/5

We use this profile matrix to compute the probabilities of all 4-mers in the second string from *Dna*:

gATG	**ATGT**	**TGT**c	**GT**ct	**T**ctg	ctgt	tgtc
$1/5^4$	$4/5^4$	$1/5^4$	$4/5^4$	$2/5^4$	$2/5^4$	$1/5^4$

There are two *Profile*-most probable 4-mers in the second sequence (**ATGT** and **GT**ct); let's assume that we get lucky again and choose the implanted 4-mer **ATGT**. We now have the following motif, count, and profile matrices:

Motifs **ACCT**
ATGT

COUNT(*Motifs*)

A:	2+1	0+1	0+1	0+1
C:	0+1	1+1	1+1	0+1
G:	0+1	0+1	1+1	0+1
T:	0+1	1+1	0+1	2+1

PROFILE(*Motifs*)

3/6	1/6	1/6	1/6
1/6	2/6	2/6	1/6
1/6	1/6	2/6	1/6
1/6	2/6	1/6	3/6

We use this profile matrix to compute the probabilities of all 4-mers in the third string from *Dna*:

acg**G**	cg**GC**	g**GCG**	**GCGT**	**CGT**t	**GT**ta	**T**tag
$12/6^4$	$2/6^4$	$2/6^4$	$12/6^4$	$3/6^4$	$2/6^4$	$2/6^4$

Again, there are two *Profile*-most probable 4-mers in the second sequence (acg**G** and **GCGT**). This time, we will assume that acg**G** is selected instead of the implanted 4-mer **GCGT**. We now have the following motif, count, and profile matrices:

Motifs

ACCT
ATGT
acg**G**

COUNT(*Motifs*)

A:	3+1	0+1	0+1	1+1
C:	0+1	2+1	1+1	0+1
G:	0+1	0+1	2+1	1+1
T:	0+1	1+1	0+1	2+1

PROFILE(*Motifs*)

4/7	1/7	1/7	1/7
1/7	3/7	2/7	1/7
1/7	1/7	3/7	2/7
1/7	2/7	1/7	3/7

We use this profile matrix to compute probabilities of all 4-mers in the fourth string from *Dna*:

ccct	ccta	cta**A**	ta**AC**	a**ACG**	**ACGA**	**CGA**g
$18/7^4$	$3/7^4$	$2/7^4$	$1/7^4$	$16/7^4$	$36/7^4$	$2/7^4$

Despite the fact that we missed the implanted 4-mer in the third sequence, we have now found the implanted 4-mer in the fourth string in *Dna* as the *Profile*-most probable 4-mer **ACGA**. This provides us with the following motif, count, and profile matrices:

Motifs

ACCT
ATGT
acg**G**
ACGA

COUNT(*Motifs*)

A:	4+1	0+1	0+1	0+1
C:	0+1	3+1	1+1	0+1
G:	0+1	0+1	3+1	1+1
T:	0+1	1+1	0+1	2+1

PROFILE(*Motifs*)

5/8	1/8	1/8	2/8
1/8	4/8	2/8	1/8
1/8	1/8	4/8	2/8
1/8	2/8	1/8	3/8

We now use this profile to compute the probabilities of all 4-mers in the fifth string in *Dna*:

cgtc	gtca	tcag	cagA	agAG	gAGG	AGGT
$1/8^4$	$8/8^4$	$8/8^4$	$8/8^4$	$10/8^4$	$8/8^4$	$60/8^4$

The *Profile*-most probable 4-mer of the fifth string in *Dna* is **AGGT**, the implanted 4-mer. As a result, **GREEDYMOTIFSEARCH** has produced the following motif matrix, which implies the correct consensus string **ACGT**:

	ACCT
	ATGT
Motifs	acgG
	ACGA
	AGGT
CONSENSUS(*Motifs*)	ACGT

You have now seen the power of pseudocounts illustrated on a small example. Running **GREEDYMOTIFSEARCH** with pseudocounts to solve the Subtle Motif Problem returns a collection of 15-mers *Motifs* with SCORE(*Motifs*) = 41 and CONSENSUS(*Motifs*) = **AAAAAtAgaGGGGtt**. Thus, Laplace's Rule of Succession has provided a significant improvement over the original **GREEDYMOTIFSEARCH**, which returned the consensus string **gTtAAAtAgaGatGtG** with SCORE(*Motifs*) = 58.

You may be satisfied with the performance of **GREEDYMOTIFSEARCH**, but you should know by now that your authors are never satisfied. Can we design an even more accurate motif finding algorithm?

Randomized Motif Search

Rolling dice to find motifs

We will now turn to **randomized algorithms** that flip coins and roll dice in order to search for motifs. Making random algorithmic decisions may sound like a disastrous idea — just imagine a chess game in which every move would be decided by rolling a die. However, an 18th Century French mathematician and naturalist, Comte de Buffon, first proved that randomized algorithms are useful by randomly dropping needles onto parallel strips of wood and using the results of this experiment to accurately approximate the constant π (see **DETOUR: Buffon's Needle**).

Randomized algorithms may be nonintuitive because they lack the control of traditional algorithms. Some randomized algorithms are **Las Vegas algorithms**, which

deliver solutions that are guaranteed to be exact, despite the fact that they rely on making random decisions. Yet most randomized algorithms, including the motif finding algorithms that we will consider in this chapter, are **Monte Carlo algorithms**. These algorithms are not guaranteed to return exact solutions, but they do quickly find *approximate* solutions. Because of their speed, they can be run many times, allowing us to choose the best approximation from thousands of runs.

We previously defined PROFILE(*Motifs*) as the profile matrix constructed from a collection of *k*-mers *Motifs* in *Dna*. Now, given a collection of strings *Dna* and an arbitrary $4 \times k$ matrix *Profile*, we define MOTIFS(*Profile*, *Dna*) as the collection of *k*-mers formed by the *Profile*-most probable *k*-mers in each sequence from *Dna*. For example, consider the following *Profile* and *Dna*:

Profile

A:	4/5	0	0	1/5
C:	0	3/5	1/5	0
G:	1/5	1/5	4/5	0
T:	0	1/5	0	4/5

Dna

```
ttaccttaac
gatgtctgtc
acggcgttag
ccctaacgag
cgtcagaggt
```

Taking the *Profile*-most probable 4-mer from each row of *Dna* produces the following 4-mers (shown in red):

MOTIFS(*Profile*, *Dna*)

```
tt**acct**taac
g**atgt**ctgtc
acg**gcgt**tag
ccct**acga**g
cgtcag**aggt**
```

In general, we can begin from a collection of randomly chosen *k*-mers *Motifs* in *Dna*, construct PROFILE(*Motifs*), and use this profile to generate a new collection of *k*-mers:

$$\text{MOTIFS}(\text{PROFILE}(\textit{Motifs}), \textit{Dna})$$

Why would we do this? Because our hope is that MOTIFS(PROFILE(*Motifs*), *Dna*) has a better score than the original collection of *k*-mers *Motifs*. We can then form the profile matrix of these *k*-mers,

$$\text{PROFILE}(\text{MOTIFS}(\text{PROFILE}(\textit{Motifs}), \textit{Dna}))\,,$$

and use it to form the most probable *k*-mers,

$$\text{MOTIFS}(\text{PROFILE}(\text{MOTIFS}(\text{PROFILE}(\textit{Motifs}), \textit{Dna})), \textit{Dna}) .$$

We can continue to iterate...

$$\ldots \text{PROFILE}(\text{MOTIFS}(\text{PROFILE}(\text{MOTIFS}(\text{PROFILE}(\textit{Motifs}), \textit{Dna})), \textit{Dna}))\ldots$$

for as long as the score of the constructed motifs keeps improving, which is exactly what **RANDOMIZEDMOTIFSEARCH** does. To implement this algorithm, you will need to randomly select the initial collection of k-mers that form the motif matrix *Motifs*. To do so, you will need a **random number generator** (denoted RANDOM(N)) that is equally likely to return any integer from 1 to N. You might like to think about this random number generator as an unbiased N-sided die.

```
RANDOMIZEDMOTIFSEARCH(Dna, k, t)
    randomly select k-mers Motifs = (Motif_1, ..., Motif_t) in each string from Dna
    BestMotifs ← Motifs
    while forever
        Profile ← PROFILE(Motifs)
        Motifs ← MOTIFS(Profile, Dna)
        if SCORE(Motifs) < SCORE(BestMotifs)
            BestMotifs ← Motifs
        else
            return BestMotifs
```

EXERCISE BREAK: Prove that **RANDOMIZEDMOTIFSEARCH** will eventually terminate.

Since a single run of **RANDOMIZEDMOTIFSEARCH** may generate a rather poor set of motifs, bioinformaticians usually run this algorithm thousands of times. On each run, they begin from a new randomly selected set of k-mers, selecting the best set of k-mers found in all these runs.

Why randomized motif search works

At first glance, **RANDOMIZEDMOTIFSEARCH** appears to be doomed. How can this algorithm, which starts from a random guess, possibly find anything useful? To explore **RANDOMIZEDMOTIFSEARCH**, let's run it on five short strings with the implanted $(4, 1)$-motif ACGT (shown in upper case letters below) and imagine that it chooses the

following 4-mers *Motifs* (shown in red) at the first iteration. As expected, it misses the implanted motif in nearly every string.

Dna

ttACCT**taac**
gAT**GTct**gtc
ccgGCGTtag
c**acta**ACGAg
cgtcag**AGGT**

Below, we construct the profile matrix PROFILE(*Motifs*) of the chosen 4-mers.

Motifs

t	a	a	c
G	T	c	t
c	c	g	G
a	c	t	a
A	G	G	T

PROFILE(*Motifs*)

A:	0.4	0.2	0.2	0.2
C:	0.2	0.4	0.2	0.2
G:	0.2	0.2	0.4	0.2
T:	0.2	0.2	0.2	0.4

We can now compute the probabilities of every 4-mer in *Dna* based on this profile matrix. For example, the probability of the first 4-mer in the first string of *Dna* is $\text{PR}(\texttt{ttAC}|\textit{Profile}) = 0.2 \cdot 0.2 \cdot 0.2 \cdot 0.2 = 0.0016$. The maximum probabilities in every row are shown in red below.

ttAC	tACC	ACCT	CCTt	CTta	Ttaa	taac
.0016	.0016	**.0128**	.0064	.0016	.0016	.0016
gATG	ATGT	TGTc	GTct	Tctg	ctgt	tgtc
.0016	**.0128**	.0016	.0032	.0032	.0032	.0016
ccgG	cgGC	gGCG	GCGT	CGTt	GTta	Ttag
.0064	.0036	.0016	**.0128**	.0032	.0016	.0016
cact	acta	ctaA	taAC	aACG	ACGA	CGAg
.0032	.0064	.0016	.0016	.0032	**.0128**	.0016
cgtc	gtca	tcag	cagA	agAG	gAGG	AGGT
.0016	.0016	.0016	.0032	.0032	.0032	**.0128**

We select the most probable 4-mer in each row above as our new collection *Motifs* (shown below). Notice that this collection has captured all five implanted motifs in *Dna*!

	tt**ACCT**taac
	g**ATGT**ctgtc
Dna	ccg**GCGT**tag
	cacta**ACGA**g
	cgtcag**AGGT**

STOP and Think: How is it possible that randomly chosen *k*-mers have led us to the correct implanted *k*-mer? If you think we manufactured this example, select your own initial 4-mers and see what happens.

For the Subtle Motif Problem with implanted 15-mer **AAAAAAAAGGGGGGG**, when we run **RANDOMIZEDMOTIFSEARCH** 100,000 times (each time with new randomly selected *k*-mers), it returns the 15-mers shown in Figure 2.7 as the lowest scoring collection *Motifs* across all iterations, resulting in the consensus string **AAAAAAAAacaGGGG** with score 43. These strings are only slightly less conserved than the collection of implanted (15, 4)-motifs with score 40 (or the motif returned by **GREEDYMOTIFSEARCH** with score 41), and it largely captures the implanted motif. Furthermore, unlike **GREEDYMOTIFSEARCH**, **RANDOMIZEDMOTIFSEARCH** can be run for a larger number of iterations to discover better and better motifs.

		Score
	AAAt**A**c**A**g**ACAG**c**G**t	5
	AAAAAAt**A**g**C**A**GGG**t	3
	t**AAAA**t**AAAC**A**G**c**GG**	3
	Ac**A**g**AAAAA**a**AGGGG**	3
Motifs	**AAAA**t**AAAAC**t**G**c**G**a	4
	At**A**g**A**cg**AACA**c**GG**t	6
	c**AAAAA**g**A**ga**AGGGG**	4
	At**A**g**AAAA**gg**A**a**GGG**	5
	AAg**AAAAAA**g**AG**a**GG**	3
	c**A**t**AA**tg**AAC**t**G**t**G**a	7
CONSENSUS(*Motifs*)	**AAAAAAAAACAGGGG**	**43**

FIGURE 2.7 The lowest scoring collection of strings *Motifs* produced by 100,000 runs of **RANDOMIZEDMOTIFSEARCH**, along with their consensus string and score for the Subtle Motif Problem.

STOP and Think: Does your run of **RANDOMIZEDMOTIFSEARCH** return a similar consensus string? How many times do you need to run **RANDOMIZEDMOTIFSEARCH** to obtain the implanted $(15, 4)$-motif with distance 40?

Although the motifs returned by **RANDOMIZEDMOTIFSEARCH** are slightly less conserved than the motifs returned by **MEDIANSTRING**, **RANDOMIZEDMOTIFSEARCH** has the advantage of being able to find longer motifs (since **MEDIANSTRING** becomes too slow for longer motifs). In the epilogue, we will see that this feature is important in practice.

How Can a Randomized Algorithm Perform So Well?

In the previous section, we began with a collection of implanted motifs that resulted in the following profile matrix.

A:	**0.8**	0.0	0.0	0.2
C:	0.0	**0.6**	0.2	0.0
G:	0.2	0.2	**0.8**	0.0
T:	0.0	0.2	0.0	**0.8**

If the strings in *Dna* were truly random, then we would expect that all nucleotides in the selected k-mers would be equally likely, resulting in an expected *Profile* in which every entry is approximately 0.25:

A:	0.25	0.25	0.25	0.25
C:	0.25	0.25	0.25	0.25
G:	0.25	0.25	0.25	0.25
T:	0.25	0.25	0.25	0.25

Such a **uniform profile** is essentially useless for motif finding because no string is more probable than any other according to this profile and because it does not provide any clues on what an implanted motif looks like.

At the opposite end of the spectrum, if we were incredibly lucky, we would choose the implanted k-mers *Motifs* from the very beginning, resulting in the first of the two profile matrices above. In practice, we are likely to obtain a profile somewhere in between these two extremes, such as the following:

A:	**0.4**	0.2	0.2	0.2
C:	0.2	**0.4**	0.2	0.2
G:	0.2	0.2	**0.4**	0.2
T:	0.2	0.2	0.2	**0.4**

This profile matrix has already started to point us toward the implanted motif ACGT, i.e., ACGT is the most likely 4-mer that can be generated by this profile. Fortunately, **RANDOMIZEDMOTIFSEARCH** is designed so that subsequent steps have a good chance of leading us toward this implanted motif (although it is not certain).

If you still doubt the efficacy of randomized algorithms, consider the following argument. We have already noticed that if the strings in *Dna* were random, then **RANDOMIZEDMOTIFSEARCH** would start from a nearly uniform profile, and there would be nothing to work with. However, the key observation is that the strings in *Dna* are not random because they include the implanted motif! These multiple occurrences of the same motif may create a bias in the profile matrix, directing it away from the uniform profile and toward the implanted motif. For example, consider again the original randomly selected *k*-mers *Motifs* (shown in red):

	ttACCT**taac**
	gAT**GTct**gtc
Dna	**ccgG**CGTtag
	c**acta**ACGAg
	cgtcag**AGGT**

You will see that the 4-mer **AGGT** in the last string happened to capture the implanted motif simply by chance. In fact, the profile formed from the remaining 4-mers (**taac**, **GTct**, **ccgG**, and **acta**) is uniform.

EXERCISE BREAK: Compute the probability that ten randomly selected 15-mers from ten 600-nucleotide long strings (such as in the Subtle Motif Problem) capture at least one implanted 15-mer.

Although the probability that randomly selected *k*-mers match *all* implanted motifs is negligible, the probability that they capture *at least one* implanted motif is significant. Even in the case of difficult motif finding problems for which this probability is small, we can run **RANDOMIZEDMOTIFSEARCH** many times, so that it will almost certainly catch at least one implanted motif, thus creating a statistical bias pointing toward the correct motif.

Unfortunately, capturing a single implanted motif is often insufficient to steer **RANDOMIZEDMOTIFSEARCH** to an optimal solution. Therefore, since the number of starting positions of k-mers is huge, the strategy of randomly selecting motifs is often not as successful as in the simple example above. The chance that these randomly selected k-mers will be able to guide us to the optimal solution is relatively small.

EXERCISE BREAK: Compute the probability that ten randomly selected 15-mers from the ten 600-nucleotide long strings in the Subtle Motif Problem capture at least two implanted 15-mers.

Gibbs Sampling

Note that **RANDOMIZEDMOTIFSEARCH** may change all t strings in *Motifs* in a single iteration. This strategy may prove reckless, since some correct motifs (captured in *Motifs*) may potentially be discarded at the next iteration. **GIBBSSAMPLER** is a more cautious iterative algorithm that discards a single k-mer from the current set of motifs at each iteration and decides to either keep it or replace it with a new one. This algorithm thus moves with more caution in the space of all motifs, as illustrated below.

```
ttaccttaac      ttaccttaac          ttaccttaac      ttaccttaac
gatatctgtc      gatatctgtc          gatatctgtc      gatatctgtc
acggcgttcg  →   acggcgttcg          acggcgttcg  →   acggcgttcg
ccctaaagag      ccctaaagag          ccctaaagag      ccctaaagag
cgtcagaggt      cgtcagaggt          cgtcagaggt      cgtcagaggt
```

RANDOMIZEDMOTIFSEARCH (may change all k-mers in one step)

GIBBSSAMPLER (changes one k-mer in one step)

Like **RANDOMIZEDMOTIFSEARCH**, **GIBBSSAMPLER** starts with randomly chosen k-mers in each of t DNA sequences, but it makes a random rather than a deterministic choice at each iteration. It uses randomly selected k-mers $Motifs = (Motif_1, \ldots, Motif_t)$ to come up with another (hopefully better scoring) set of k-mers. In contrast with **RANDOMIZEDMOTIFSEARCH**, which deterministically defines new motifs as

$$\text{MOTIFS}(\text{PROFILE}(Motifs), Dna)\,,$$

GIBBSSAMPLER randomly selects an integer i between 1 and t and then randomly changes a single k-mer $Motif_i$.

To describe how **GIBBSSAMPLER** updates *Motifs*, we will need a slightly more advanced random number generator. Given a probability distribution $(p_1, \ldots, p_n)$, this random number generator, denoted $\text{RANDOM}(p_1, \ldots, p_n)$, models an n-sided biased die and returns integer i with probability p_i. For example, the standard six-sided fair die represents the random number generator

$$\text{RANDOM}(1/6, 1/6, 1/6, 1/6, 1/6, 1/6)\,,$$

whereas a biased die might represent the random number generator

$$\text{RANDOM}(0.1, 0.2, 0.3, 0.05, 0.1, 0.25)\,.$$

GIBBSSAMPLER further generalizes the random number generator by using the function $\text{RANDOM}(p_1, \ldots, p_n)$ defined for any set of non-negative numbers, i.e., not necessarily satisfying the condition $\sum_{i=1}^{n} p_i = 1$. Specifically, if $\sum_{i=1}^{n} p_i = C > 0$, then $\text{RANDOM}(p_1, \ldots, p_n)$ is defined as $\text{RANDOM}(p_1/C, \ldots, p_n/C)$, where $(p_1/C, \ldots, p_n/C)$ is a probability distribution. For example, given the values $(p_1, p_2, p_3) = (0.1, 0.2, 0.3)$ with $0.1 + 0.2 + 0.3 = 0.6$,

$$\begin{aligned}\text{RANDOM}(0.1, 0.2, 0.3) &= \text{RANDOM}(0.1/0.6, 0.2/0.6, 0.3/0.6)\\ &= \text{RANDOM}(1/6, 1/3, 1/2)\,.\end{aligned}$$

STOP and Think: Implement $\text{RANDOM}(p_1, \ldots, p_n)$ so that it uses $\text{RANDOM}(X)$ (for an appropriately chosen integer X) as a subroutine.

We have previously defined the notion of a *Profile*-most probable k-mer in a string. We now define a ***Profile*-randomly generated k-mer** in a string *Text*. For each k-mer *Pattern* in *Text*, compute the probability $\Pr(Pattern|Profile)$, resulting in $n = |Text| - k + 1$ probabilities $(p_1, \ldots, p_n)$. These probabilities do not necessarily sum to 1, but we can still form the random number generator $\text{RANDOM}(p_1, \ldots, p_n)$ based on them. **GIBBSSAMPLER** uses this random number generator to select a *Profile*-randomly generated k-mer at each step: if the die rolls the number i, then we define the *Profile*-randomly generated k-mer as the i-th k-mer in *Text*. While the pseudocode below repeats this procedure N times, in practice **GIBBSSAMPLER** depends on various stopping rules that are beyond the scope of this chapter.

```
GIBBSSAMPLER(Dna, k, t, N)
    randomly select k-mers Motifs = (Motif_1, ..., Motif_t) in each string from Dna
    BestMotifs ← Motifs
    for j ← 1 to N
        i ← RANDOM(t)
        Profile ← profile matrix formed from all strings in Motifs except for Motif_i
        Motif_i ← Profile-randomly generated k-mer in the i-th sequence
        if SCORE(Motifs) < SCORE(BestMotifs)
            BestMotifs ← Motifs
    return BestMotifs
```

STOP and Think: Note that in contrast to **RANDOMIZEDMOTIFSEARCH**, which always moves from higher to lower scoring *Motifs*, **GIBBSSAMPLER** may move from lower to higher scoring *Motifs*. Why is this reasonable?

Gibbs Sampling in Action

We illustrate how **GIBBSSAMPLER** works on the same strings *Dna* that we considered before. Imagine that, at the initial step, the algorithm has chosen the following 4-mers (shown in red) and has randomly selected the third string for removal.

	ttACCT**taac**		ttACCT**taac**
	gAT**GTct**gtc		gAT**GTct**gtc
Dna	**ccgG**CGTtag	⟶	----------
	c**acta**ACGAg		c**acta**ACGAg
	cgtcag**AGGT**		cgtcag**AGGT**

This results in the following motif, count, and profile matrices.

Motifs

	t	a	a	c
	G	T	c	t
	a	c	t	a
	A	G	G	T

COUNT(*Motifs*)

A:	2	1	1	1
C:	0	1	1	1
G:	1	1	1	0
T:	1	1	1	2

PROFILE(*Motifs*)

A:	2/4	1/4	1/4	1/4
C:	0	1/4	1/4	1/4
G:	1/4	1/4	1/4	0
T:	1/4	1/4	1/4	2/4

Note that the profile matrix is only slightly more conserved than the uniform profile, making us wonder whether we have any chance to be steered toward the implanted motif. We now use this profile matrix to compute the probabilities of all 4-mers in the deleted string `ccgGCGTtag`:

`ccgG`	`cgGC`	`gGCG`	`GCGT`	`CGTt`	`GTta`	`Ttag`
0	0	0	1/128	0	1/256	0

Note that all but two of these probabilities are zero. This situation is similar to the one we encountered with **GREEDYMOTIFSEARCH**, and as before, we need to augment zero probabilities with small pseudocounts to avoid disastrous results.

Application of Laplace's Rule of Succession to the count matrix above yields the following updated count and profile matrices:

$$\text{COUNT}(Motifs) \begin{matrix} \texttt{A}: & 3 & 2 & 2 & 2 \\ \texttt{C}: & 1 & 2 & 2 & 2 \\ \texttt{G}: & 2 & 2 & 2 & 1 \\ \texttt{T}: & 2 & 2 & 2 & 3 \end{matrix} \qquad \text{PROFILE}(Motifs) \begin{matrix} \texttt{A}: & 3/8 & 2/8 & 2/8 & 2/8 \\ \texttt{C}: & 1/8 & 2/8 & 2/8 & 2/8 \\ \texttt{G}: & 2/8 & 2/8 & 2/8 & 1/8 \\ \texttt{T}: & 2/8 & 2/8 & 2/8 & 3/8 \end{matrix}$$

After adding pseudocounts, the 4-mer probabilities in the deleted string `ccgGCGTtag` are recomputed as follows:

`ccgG`	`cgGC`	`gGCG`	`GCGT`	`CGTt`	`GTta`	`Ttag`
$4/8^4$	$8/8^4$	$8/8^4$	$24/8^4$	$12/8^4$	$16/8^4$	$8/8^4$

Since these probabilities sum to $C = 80/8^4$, our hypothetical seven-sided die is represented by the random number generator

$$\begin{aligned} &\text{RANDOM}\left(\frac{4/8^4}{80/8^4}, \frac{8/8^4}{80/8^4}, \frac{8/8^4}{80/8^4}, \frac{24/8^4}{80/8^4}, \frac{12/8^4}{80/8^4}, \frac{16/8^4}{80/8^4}, \frac{8/8^4}{80/8^4}\right) \\ &= \text{RANDOM}\left(\frac{4}{80}, \frac{8}{80}, \frac{8}{80}, \frac{24}{80}, \frac{12}{80}, \frac{16}{80}, \frac{8}{80}\right). \end{aligned}$$

Let's assume that after rolling this seven-sided die, we arrive at the *Profile*-randomly generated 4-mer `GCGT` (the fourth 4-mer in the deleted sequence). The deleted string `ccgGCGTtag` is now added back to the collection of motifs, and **`GCGT`** substitutes the previously chosen `ccgG` in the third string in *Dna*, as shown below. We then roll a fair five-sided die and randomly select the first string from *Dna* for removal.

	ttACCT**taac**		----------
	gAT**GTct**gtc		gAT**GTct**gtc
Dna	ccg**GCGT**tag	⟶	ccg**GCGT**tag
	c**acta**ACGAg		c**acta**ACGAg
	cgtcag**AGGT**		cgtcag**AGGT**

After constructing the motif and profile matrices, we obtain the following:

$$\textit{Motifs} = \begin{matrix} G & T & c & t \\ G & C & G & T \\ a & c & t & a \\ A & G & G & T \end{matrix} \qquad \text{PROFILE}(\textit{Motifs}) = \begin{matrix} A: & 2/4 & 0 & 0 & 1/4 \\ C: & 0 & 2/4 & 1/4 & 0 \\ G: & 2/4 & 1/4 & 2/4 & 0 \\ T: & 0 & 1/4 & 1/4 & 3/4 \end{matrix}$$

Note that the profile matrix looks more biased toward the implanted motif than the previous profile matrix did. We update the count and profile matrices with pseudocounts:

$$\text{COUNT}(\textit{Motifs}) = \begin{matrix} A: & 3 & 1 & 1 & 2 \\ C: & 1 & 3 & 2 & 1 \\ G: & 3 & 2 & 3 & 1 \\ T: & 1 & 2 & 2 & 4 \end{matrix} \qquad \text{PROFILE}(\textit{Motifs}) = \begin{matrix} A: & 3/8 & 1/8 & 1/8 & 2/8 \\ C: & 1/8 & 3/8 & 2/8 & 1/8 \\ G: & 3/8 & 2/8 & 3/8 & 1/8 \\ T: & 1/8 & 2/8 & 2/8 & 4/8 \end{matrix}$$

Then, we compute the probabilities of all 4-mers in the deleted string `ttACCTtaac`:

ttAC	tACC	ACCT	CCTt	CTta	Ttaa	taac
$2/8^4$	$2/8^4$	$72/8^4$	$24/8^4$	$8/8^4$	$4/8^4$	$1/8^4$

When we roll a seven-sided die, we arrive at the *Profile*-randomly generated *k*-mer **ACCT**, which we add to the collection *Motifs*. After rolling the five-sided die once again, we randomly select the fourth string for removal.

	ttACCTtaac		ttACCTtaac
	gATGTctgtc		gATGTctgtc
Dna	ccgGCGTtag	⟶	ccgGCGTtag
	cactaACGAg		----------
	cgtcagAGGT		cgtcagAGGT

We further add pseudocounts and construct the resulting count and profile matrices:

Motifs

```
A C C T
G T c t
G C G T
A G G T
```

COUNT(*Motifs*)

```
A: 3 1 1 1
C: 1 3 3 1
G: 3 2 3 1
T: 1 2 1 5
```

PROFILE(*Motifs*)

```
A: 3/8 1/8 1/8 1/8
C: 1/8 3/8 3/8 1/8
G: 3/8 2/8 3/8 1/8
T: 1/8 2/8 1/8 5/8
```

We now compute the probabilities of all 4-mers in the deleted string `cactaACGAg`:

`cact`	`acta`	`ctaA`	`taAC`	`aACG`	`ACGA`	`CGAg`
$15/8^4$	$9/8^4$	$2/8^4$	$1/8^4$	$9/8^4$	$27/8^4$	$2/8^4$

We need to roll a seven-sided die to produce a *Profile*-randomly generated 4-mer. Assuming the most probable scenario in which we select **ACGA**, we update the selected 4-mers as follows:

Dna

```
ttACCTtaac
gATGTctgtc
ccgGCGTtag
cactaACGAg
cgtcagAGGT
```

You can see that the algorithm is beginning to converge. Rest assured that a subsequent iteration will produce all implanted motifs after we select the second string in *Dna* (when the incorrect 4-mer `GTct` will likely change into the implanted (4, 1)-motif `ATGT`).

STOP and Think: Run **GIBBSSAMPLER** on the Subtle Motif Problem. What do you find?

Although **GIBBSSAMPLER** performs well in many cases, it may converge to a suboptimal solution, particularly for difficult search problems with elusive motifs. A **local optimum** is a solution that is optimal within a small neighboring set of solutions, which is in contrast to a **global optimum**, or the optimal solution among all possible solutions. Since **GIBBSSAMPLER** explores just a small subset of solutions, it may "get stuck" in a local optimum. For this reason, similarly to **RANDOMIZEDMOTIFSEARCH**, it should be run many times with the hope that one of these runs will produce the best-scoring

motifs. Yet convergence to a local optimum is just one of many issues we must consider in motif finding; see **DETOUR: Complications in Motif Finding** for some other challenges. PAGE 111

When we run GIBBSSAMPLER 2,000 times on the Subtle Motif Problem with implanted 15-mer **AAAAAAAAGGGGGGG** (each time with new randomly selected k-mers for $N = 200$ iterations), it returns a collection *Motifs* with consensus **AAAAAAgAGGGGGGt** and SCORE(*Motifs*) equal to 38. This score is even lower than the score of 40 expected from the implanted motifs!

Epilogue: How Does Tuberculosis Hibernate to Hide from Antibiotics?

Tuberculosis (TB) is an infectious disease that is caused by the *Mycobacterium tuberculosis* bacterium (MTB) and is responsible for over a million deaths each year. Although the spread of TB has been greatly reduced due to antibiotics, strains that resist all available treatments are now emerging. MTB is successful as a pathogen because it can persist in humans for decades without causing disease; in fact, one-third of the world population has **latent MTB infections**, in which MTB lies dormant within the host's body and may or may not reactivate at a later time. The widespread prevalence of latent infections makes it difficult to control TB epidemics. Biologists are therefore interested in finding out what makes the disease latent and how MTB activates itself within a host.

It remains unclear why MTB can stay latent for so long and how it survives during latency. The resistance of latent TB to antibiotics implies that MTB may have an ability to shut down expression of most genes and stay dormant, not unlike bears hibernating in the winter. Hibernation in bacteria is called **sporulation** because many bacteria form protective and metabolically dormant **spores** that can survive in tough conditions, allowing the bacteria to persist in the environment until conditions improve.

Hypoxia, or oxygen shortage, is often associated with latent forms of TB. Biologists have found that MTB becomes dormant in low-oxygen environments, presumably with the idea that the host's lungs will recover enough to potentially spread the disease in the future. Since MTB shows a remarkable ability to survive for years without oxygen, it is important to identify MTB genes responsible for the development of the latent state under hypoxic conditions. Biologists are interested in finding a **master regulator** (transcription factor) that "senses" the shortage of oxygen and starts a genetic program that affects the expression of many genes, allowing MTB to adapt to hypoxia.

In 2003, biologists found the **dormancy survival regulator** (**DosR**), a transcription factor that regulates many genes whose expression dramatically changes under hypoxic

conditions. However, it remained unclear how DosR regulates these genes, and its transcription factor binding site remained unknown. In an attempt to resolve this puzzle, biologists performed a DNA array experiment and found 25 genes whose expression levels significantly changed in hypoxic conditions. Given the upstream regions of these genes, each of which is 250 nucleotides long, we would like to discover the "hidden message" that DosR uses to control the expression of these genes.

To simplify the problem a bit, we have selected just 10 of the 25 genes, resulting in the **DosR dataset**. We will try to identify motifs in this dataset using the arsenal of motif finding tools that we have developed. However, we will not give you a hint about the DosR motif.

What k-mer size should we choose in order to analyze the DosR dataset using MEDIANSTRING and RANDOMIZEDMOTIFSEARCH? Taking a wild guess and running these algorithms for k from 8 to 12 returns the consensus strings shown below.

	MEDIANSTRING			RANDOMIZEDMOTIFSEARCH	
k	Consensus	Score	k	Consensus	Score
8	CATCGGCC	11	8	CCGACGGG	13
9	GGCGGGGAC	16	9	CCATCGGCC	16
10	GGTGGCCACC	19	10	CCATCGGCCC	21
11	GGACTTCCGGC	20	11	ACCTTCGGCCC	25
12	GGACTTCCGGCC	23	12	GGACCAACGGCC	28

STOP and Think: Can you infer the DosR binding site from these median strings? What do you think is the length of the binding site?

Note that although the consensus strings returned by RANDOMIZEDMOTIFSEARCH generally deviate from the median strings, the consensus 12-mer (GGACCAACGGCC, with score 28) is very similar to the median string (GGACTTCCGGCC, with score 23).

While the motifs returned by RANDOMIZEDMOTIFSEARCH are slightly less conserved than the motifs returned by MEDIANSTRING, the former algorithm has the advantage of being able to find longer motifs (since MEDIANSTRING becomes too slow for longer motifs). The motif of length 20 returned by RANDOMIZEDMOTIFSEARCH is CGGGACCTACGTCCCTAGCC (with score 57). As shown below, the consensus strings of length 12 found by RANDOMIZEDMOTIFSEARCH and MEDIANSTRING are "embedded" with small variations in the longer motif of length 20:

GGACCAACGGCC
CGGGACCTACGTCCCTAGCC
GGACTTCCGGCC

Finally, in 2,000 runs with $N = 200$, **GIBBSSAMPLER** returned the same consensus string of length 20 for the DosR dataset as **RANDOMIZEDMOTIFSEARCH** but generated a different collection of motifs with a smaller score of 55.

As you have seen in this chapter, different motif finding algorithms generate somewhat different results, and it remains unclear how to identify the DosR motif in MTB. Try to answer this question and find all putative DosR motifs in MTB as well as all genes that they regulate. We will provide you with the upstream regions of all 25 genes identified in the DosR study to help you address the following problem.

CHALLENGE PROBLEM: Infer the profile of the DosR motif and find all its putative occurrences in *Mycobacterium tuberculosis.*

Charging Stations

Solving the Median String Problem

The first potential issue with implementing **MEDIANSTRING** is writing a function to compute $d(Pattern, Dna) = \sum_{i=1}^{t} d(Pattern, Dna_i)$, the sum of distances between *Pattern* and each string in $Dna = \{Dna_1, \ldots, Dna_t\}$. This task is achieved by the following pseudocode.

```
DISTANCEBETWEENPATTERNANDSTRINGS(Pattern, Dna)
    k ← |Pattern|
    distance ← 0
    for each string Text in Dna
        HammingDistance ← ∞
        for each k-mer Pattern′ in Text
            if HammingDistance > HAMMINGDISTANCE(Pattern, Pattern′)
                HammingDistance ← HAMMINGDISTANCE(Pattern, Pattern′)
        distance ← distance + HammingDistance
    return distance
```

To solve the Median String Problem, we need to iterate through all possible 4^k k-mers *Pattern* before computing $d(Pattern, Dna)$. The pseudocode below is a modification of **MEDIANSTRING** using the function NUMBERTOPATTERN (implemented in **CHARGING STATION: Converting Patterns Into Numbers and Vice-Versa**), which is applied to convert all integers from 0 to $4^k - 1$ into all possible k-mers.


```
MEDIANSTRING(Dna, k)
    distance ← ∞
    for i ← 0 to 4^k − 1
        Pattern ← NUMBERTOPATTERN(i, k)
        if distance > DISTANCEBETWEENPATTERNANDSTRINGS(Pattern, Dna)
            distance ← DISTANCEBETWEENPATTERNANDSTRINGS(Pattern, Dna)
            Median ← Pattern
    return Median
```

Detours

Gene expression

Genes encode proteins, and proteins dictate cell function. To respond to changes in their environment, cells must therefore control their protein levels. The flow of information from DNA to RNA to protein means that the cell can adjust the amount of proteins that it produces during both transcription (DNA to RNA) and translation (RNA to protein).

Transcription begins when an RNA polymerase binds to a **promoter sequence** on the DNA molecule, which is often located just upstream from the starting point for transcription. The initiation of transcription is a convenient control point for the cell to regulate gene expression since it is at the very beginning of the protein production process. The genes transcribed in a cell are controlled by various transcription regulators that may increase or suppress transcription.

DNA arrays

A **DNA array** is a collection of DNA molecules attached to a solid surface. Each spot on the array is assigned a unique DNA sequence called a **probe** that measures the expression level of a specific gene, known as a **target**. In most arrays, probes are synthesized and then attached to a glass or silicon chip (Figure 2.8).

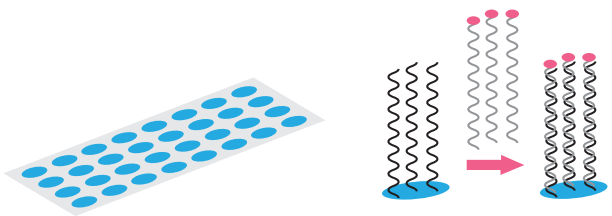


FIGURE 2.8 Fluorescently labeled DNA binds to a complementary probe on a DNA array.

Fluorescently labeled targets then bind to their corresponding probe (e.g., when their sequences are complementary), generating a fluorescent signal. The strength of this signal depends upon the amount of target sample that binds to the probe at a given spot. Thus, the higher the expression level of a gene, the higher the intensity of its fluorescent signal on the array. Since an array may contain millions of probes, biologists can measure the expression of many genes in a single array experiment. The DNA array

experiment that identified the evening element in *Arabidopsis thaliana* measured the expression of 8,000 genes.

Buffon's needle

Comte de Buffon was a prolific 18th Century naturalist whose writings on natural history were popular at the time. However, his first paper was in mathematics; in 1733, he wrote an essay on a Medieval French game called "Le jeu de franc carreau". In this game, a single player flips a coin into the air, and the coin lands on a checkerboard. The player wins if the coin lands completely within one of the squares on the board, and loses otherwise (Figure 2.9 (left)). Buffon asked a natural question: what is the probability that the player wins?

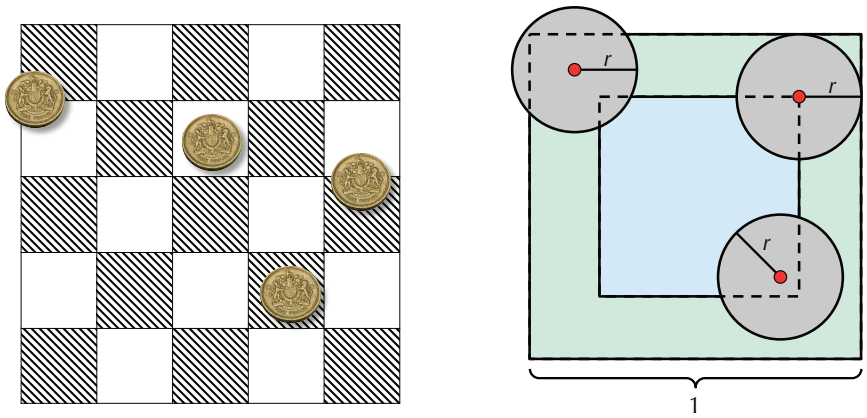


FIGURE 2.9 (Left) A game of "franc carreau" with four coins. Two of the coins have landed within one of the squares of the checkerboard and are considered winners, whereas the other two have landed on a boundary and are considered losers. (Right) Three coins shown on a single square of the checkerboard (the green outside square); one coin is a loser, another is a winner, and the third corresponds to a boundary case. You can see that if the coin has radius r, then the probability of winning the game corresponds to the probability that the center of the coin (shown as a red dot) lands within the blue square, which has side length $1 - 2r$. This probability is the ratio of the squares' areas, which is $(1 - 2r)^2$.

Let's assume that the checkerboard consists of just a single square with side length 1, that the coin has radius $r < 1/2$, and that the center of the coin always lands within the square. Then the player can only win if the center of the circle falls within an imaginary

central square of side length $1 - 2r$ (Figure 2.9 (right)). Assuming that the coin lands anywhere on the larger square with uniform probability, then the probability that the coin falls completely within the smaller square is given by the ratio of the areas of the two squares, or $(1 - 2r)^2$.

Four decades later, Buffon published a paper describing a similar game in which the player uniformly drops a needle onto a floor covered by long wooden panels of equal width. In this game, which has become known as **Buffon's needle**, the player wins if the needle falls entirely within one of the panels. Note that computing the probability of a win is now complicated by the fact that the needle is described by an orientation in addition to its position. Nevertheless, the first game gives us an idea for how to solve this problem: once we fix a position for the center of the needle, its collection of different possible orientations sweep out a circle (Figure 2.10 (left)).

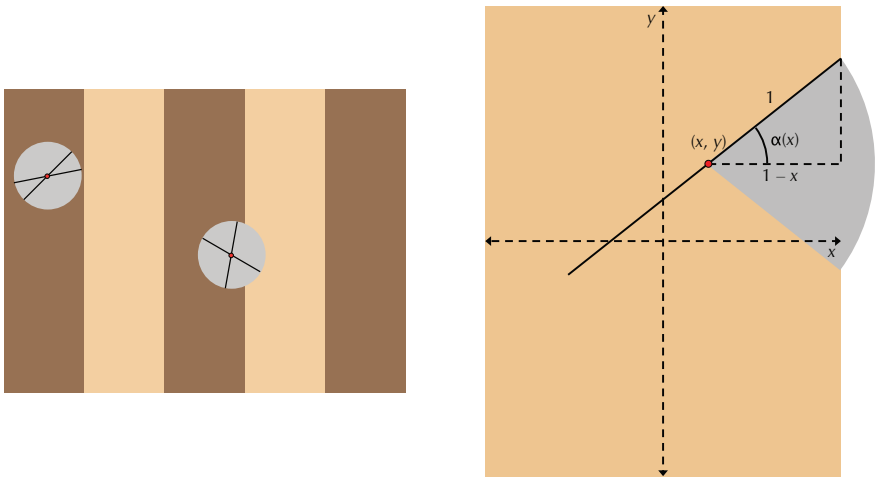


FIGURE 2.10 (Left) Once we fix a point for the center of the needle (shown as a red dot), its collection of possible orientations sweep out a circle. In the circle on the left, the needle will always lie within the dark brown panel, regardless of its orientation. In the circle on the right, one of the two needles lies within the dark brown panel, whereas another is shown crossing over into the adjacent panel. (Right) Once we fix a point (x, y) for the center of the needle, there is a critical angle $\alpha(x)$ such that all angles between $-\alpha(x)$ and $\alpha(x)$ will cause the needle to cross over into the next panel. In this figure, the length of the needle is equal to the width of the panel.

The probability that the player wins depends on the length of the needle with respect to the distance between wooden panels. We will assume that both of these lengths are

equal to 2, and we will find the probability of a *loss* instead of a win. To this end, we first ask a simpler question: if the center of the needle were to land in the same place every time, then what is the likelihood that the needle crosses a panel?

To address this question, let's map the panel into which the needle falls onto a coordinate plane, with the y-axis dividing the panel into two smaller panels of width 1 (Figure 2.10 (right)). If the center of the needle lands at position (x, y) with $x > 0$, then its orientation can be described by an angle θ, where θ ranges from $-\pi/2$ to $\pi/2$ radians. If $\theta = 0$, then the needle will cross the line $y = 1$; if $\theta = \pi/2$, then the needle will not cross the line $y = 1$. Yet more importantly, since the needle's center position is fixed, there must be some critical angle $\alpha(x)$ such that the needle always touches this line if $-\alpha(x) \leq \theta \leq \alpha(x)$. If the needle is dropped randomly, then any value of θ is equally likely, and so we obtain that the probability of a loss given this position of the needle is equal to $2 \cdot \alpha(x)/\pi$.

Following the same reasoning, the needle can take any position x with equal probability. To find the probability of a loss, $\Pr(\text{loss})$, we must therefore compute an "average" of the values $2 \cdot \alpha(x)/\pi$ as x continuously ranges from -1 to 1. This average can be represented using an integral,

$$\Pr(\text{loss}) = \frac{\int_{-1}^{1} \frac{2 \cdot \alpha(x)}{\pi} \, dx}{1 - (-1)} = \int_{-1}^{1} \frac{\alpha(x)}{\pi} \, dx = 2 \int_{0}^{1} \frac{\alpha(x)}{\pi} \, dx \, .$$

Revisiting Figure 2.10 (right), applying some basic trigonometry tells us that $\cos \alpha(x)$ is equal to $1 - x$, so that $\alpha(x) = \arccos(1 - x)$. After making this substitution into the above equation — and consulting our dusty calculus textbook — $\Pr(\text{loss})$ must be equal to $2/\pi$. It is not difficult to see that this probability will be the same when the needle is dropped onto any number of wooden panels.

But what does Buffon's needle have to do with randomized algorithms? In 1812, none other than Laplace pointed out that Buffon's needle could be used to approximate π, and the world's first Monte Carlo algorithm was born. Specifically, we can approximate the probability P_e of a loss *empirically* by dully flipping a needle into the air thousands of times (or asking a computer to do it for us). Once we have computed this empirical probability, we can conclude that P_e is approximately equal to $2/\pi$, and thus

$$\pi \approx \frac{2}{P_e} \, .$$

STOP and Think: How does this approximation change in the following cases?

1. The needle is shorter than the width between panels.
2. The needle is longer than the width between panels.

Complications in motif finding

Motif finding becomes difficult if the **background nucleotide distribution** in the sample is skewed. In this case, searching for k-mers with the minimum score or entropy may lead to a biologically irrelevant motif composed from the most frequent nucleotides in the sample. For example, if `A` has frequency 85% and `T`, `G`, and `C` have frequencies of 5%, then k-mer `AA...AA` may represent a motif with minimum score, thus disguising biologically relevant motifs. For example, the relevant motif `GCCG` with score 5 in the example below loses out to the motif **`aaaa`** with score 1.

```
        taaaaGTCGa
        acGCTGaaaa
Dna     aaaaGCCTat
        aCCCGaataa
        agaaaaGGCG
```

To find biologically relevant motifs in samples with biased nucleotide frequencies, you may therefore want to use a generalization of entropy called "relative entropy" (see **DETOUR: Relative Entropy**).

PAGE 111

Another complication in motif finding is that many motifs are best represented in a different alphabet than the alphabet of 4 nucleotides. Let `W` denote either `A` or `T`, `S` denote either `G` or `C`, `K` denote either `G` or `T`, and `Y` denote either `C` or `T`. Now, consider the motif `CSKWYWWATKWATYYK`, which represents the CSRE motif in yeast (recall Figure 2.3 from page 74). This strong motif in a hybrid alphabet corresponds to 2^{11} different motifs in the standard 4-letter alphabet of nucleotides. However, each of these 2^{11} motifs is too weak to be found by the algorithms we have considered in this chapter.

Relative entropy

Given a collection of strings *Dna*, the **relative entropy** of a $4 \times k$ profile matrix $P = (p_{r,j})$ is defined as

$$\sum_{j=1}^{k} \sum_{r \in \{A,C,G,T\}} p_{r,j} \cdot \log_2(p_{r,j}/b_r) =$$
$$\sum_{j=1}^{k} \sum_{r \in \{A,C,G,T\}} p_{r,j} \cdot \log_2(p_{r,j}) - \sum_{j=1}^{k} \sum_{r \in \{A,C,G,T\}} p_{r,j} \cdot \log_2(b_r),$$

where b_r is the frequency of nucleotide r in *Dna*. Note that the sum in the expression for entropy is preceded by a negative sign ($-\sum_{j=1}^{k} \sum_{r \in \{A,C,G,T\}} p_{r,j} \cdot \log_2(p_{r,j})$), whereas the sum on the left side of the *relative* entropy equation does not have this negative sign. Therefore, although we minimized the entropy of a motif matrix, we will now attempt to maximize the relative entropy.

The term $-\sum_{j=1}^{k} \sum_{r \in \{A,C,G,T\}} p_{r,j} \cdot \log_2(b_r)$ is called the **cross-entropy** of the profile matrix P; note that the relative entropy of a profile matrix is simply the difference between the profile's cross-entropy and its entropy. For example, the relative entropy for the motif GCCG in the example from **DETOUR: Complications in Motif Finding** **PAGE 111** is equal to $9.85 - 3.53 = 6.32$, as shown below. In this example, $b_A = 0.5$, $b_C = 0.18$, $b_G = 0.2$, and $b_T = 0.12$.

Motifs		G	T	C	G
		G	C	T	G
		G	C	C	T
		C	C	C	G
		G	G	C	G
PROFILE(*Motifs*)	A:	0.0	0.0	0.0	0.0
	C:	0.2	0.6	0.8	0.0
	G:	0.8	0.2	0.0	0.8
	T:	0.0	0.2	0.2	0.2
Entropy		$0.72 + 1.37 + 0.72 + 0.72 =$			3.53
Cross-entropy		$2.35 + 2.56 + 2.47 + 2.47 =$			9.85

For the more conserved but irrelevant motif AAAA, the relative entropy is equal to $4.18 - 0.72 = 3.46$, as shown below. Thus, GCCG loses to AAAA with respect to entropy but wins with respect to relative entropy.

Motifs		a	a	a	a	
		a	a	a	a	
		a	a	a	a	
		a	t	a	a	
		a	a	a	a	
PROFILE(*Motifs*)	A:	1.0	0.8	1.0	1.0	
	C:	0.0	0.0	0.0	0.0	
	G:	0.0	0.0	0.0	0.0	
	T:	0.0	0.2	0.0	0.0	
Entropy		0.0	+ 0.72 +	0.0	+ 0.0	= 0.72
Cross-entropy		0.94	+ 1.36 +	0.94	+ 0.94	= 4.18

Bibliography Notes

Konopka and Benzer, 1971 bred flies with abnormally short (19 hours) and long (28 hours) circadian rhythms and then traced these abnormalities to a single gene. Harmer et al., 2000 discovered the evening transcription factor binding site that orchestrates the circadian clock in plants. Excellent coverage of this discovery is given by Cristianini and Hahn, 2006. Park et al., 2003 found a transcription factor that mediates the hypoxic response of *Mycobacterium tuberculosis*.

Hertz and Stormo, 1999 described the first greedy algorithm for motif finding. The general framework for Gibbs sampling was described by Geman and Geman, 1984 and was named Gibbs sampling in reference to its similarities with some approaches in statistical mechanics (Josiah Willard Gibbs was one of the founders of statistical mechanics). Lawrence et al., 1993 adapted Gibbs sampling for motif finding.

A

Introduction to Pseudocode

What is Pseudocode?

An **algorithm** is a sequence of instructions to solve a well-formulated computational problem specified in terms of its **input** and **output**. An algorithm uses the input to generate the output. For example, the algorithm **PATTERNCOUNT** uses strings *Text* and *Pattern* as input to generate the number COUNT(*Text*, *Pattern*) as its output.

In order to solve a computational problem, you need to carry out the instructions specified by the algorithm. For example, if we want you to count how many times *Pattern* appears in *Text*, we could tell you to do the following:

1. Start from the first position of *Text* and check whether *Pattern* appears in *Text* starting at its first position.
2. If yes, draw a dot on a piece of paper.
3. Move to the second position of *Text* and check whether *Pattern* appears in *Text* starting at its second position.
4. If yes, draw another dot on the same piece of paper.
5. Continue until you reach the end of *Text*.
6. Count the number of dots on the paper.

Since humans are slow, make mistakes, and hate repetitive work, we invented computers, which are fast, love repetitive work, and never make mistakes. However, while you may easily understand the above instructions for counting the number of occurrences of *Pattern* in *Text*, no computer in the world can execute them. The only

reason you can understand them is because you have been trained for many years to understand human language. For example, we didn't specify that you should start from a *blank* piece of paper without any dots, but you assumed it. We didn't explain what it means to "reach the end of *Text*"; at what position of *Text* should we stop?

Because computers do not understand human language, algorithms must be rephrased in a **programming language** (such as Python, Java, C++, Perl, Ruby, Go, or dozens of others) in order to give the computer specific instructions. However, we don't want to describe algorithms in a specific language because it may not be your favorite

Our focus is on algorithmic ideas rather than on implementation details, which is why we will meet you halfway between human languages and programming languages by using **pseudocode**. By emphasizing ideas rather than implementation details, pseudocode is able to describe algorithms without being too formal, ignoring many of the tedious details that are required in a specific programming language. At the same time, pseudocode is more precise and less ambiguous than the instructions we gave above for counting a pattern in a text.

For example, consider the following pseudocode for an algorithm called **DISTANCE**, whose input is four numbers (x_1, y_1, x_2, y_2) and whose output is one number d. Can you guess what it does?

```
DISTANCE(x_1, y_1, x_2, y_2)
    d ← (x_2 − x_1)^2 + (y_2 − y_1)^2
    d ← √d
    return d
```

The first line of pseudocode specifies the name of an algorithm (**DISTANCE**), followed by a list of **arguments** that it requires as input (x_1, y_1, x_2, y_2). Subsequent lines contain the statements that describe the algorithm's actions, and the operation **return** reports the result of the algorithm.

We can invoke an algorithm by passing it the appropriate values for its arguments. For example, **DISTANCE**$(1, 3, 7, 5)$ would return the distance between points $(1, 3)$ and $(7, 5)$ in two-dimensional space, by first computing

$$d \leftarrow 7 - 1)^2 + (5 - 3)^2 = 40$$

and then computing

$$d \leftarrow \sqrt{40}\ .$$

The pseudocode for **DISTANCE** uses the concept of a **variable**, which contains some value and can be assigned a new value at different points throughout the course of an algorithm. To assign a new value to a variable, we use the notation

$$a \leftarrow b\,,$$

which sets the variable a equal to the value stored in variable b. For example, in the pseudocode above, when we compute **DISTANCE**$(1, 3, 7, 5)$, d is first assigned the value $(7-1)^2 + (5-3)^2 = 40$ and then is assigned the value $\sqrt{40}$.

Furthermore, we can use any name we like for variable names. For example, the following pseudocode is equivalent to the previous pseudocode for **DISTANCE**.

```
DISTANCE(x, y, z, w)
    abracadabra ← (z − x)^2 + (w − y)^2
    abracadabra ← √abracadabra
    return abracadabra
```

Whereas computer scientists are accustomed to pseudocode, we fear that some biologists reading this book might decide that pseudocode is too cryptic and therefore useless. Although modern biologists deal with algorithms on a daily basis, the language they use to describe an algorithm may be closer to a series of steps described in plain English.

Accordingly, some bioinformatics books are written without pseudocode. Unfortunately, this language is insufficient to describe the complex algorithmic ideas behind various bioinformatics tools that biologists use every day.

To be able to explain complex algorithmic ideas, we will need to delve deeper into the details of pseudocode. As a result, you will be able not only to understand the algorithms in this book, but use pseudocode to design your own algorithms!

Nuts and Bolts of Pseudocode

We have thus far described pseudocode only superficially. We will now discuss some of the details of pseudocode that we use throughout this book. We will often avoid tedious details by specifying parts of an algorithm in English, using operations that are not listed below, or by omitting noncritical details.

if conditions

The algorithm **MINIMUM2**(a, b) shown below has two numbers (a and b) as its input and a single number as its output. What do you think that it does?

```
MINIMUM2(a, b)
    if a > b
        return b
    else
        return a
```

This algorithm, which computes the minimum of two numbers, uses the following construction:

```
if statement X is true
    execute instructions Y
else
    execute instructions Z
```

If statement X is true, then the algorithm executes instructions Y; otherwise, it executes instructions Z. For example, **MINIMUM2**$(1, 9)$ returns 1 because the condition "$1 > 9$" is false.

The following pseudocode computes the minimum of three numbers.

```
MINIMUM3(a, b, c)
    if a > b
        if b > c
            return c
        else
            return b
    else
        if a < c
            return c
        else
            return b
```

We may also use **else if**, which allows us to consider more than two different possibilities in the same **if** statement. For example, we can compute the minimum of three numbers as follows.

```
MINIMUM3(a, b, c)
    if a > c and b > c
        return c
    else if a > b and c > b
        return b
    else
        return a
```

Both of these algorithms are correct, but below is a more compact version that uses the **MINIMUM2** function that we already wrote as a **subroutine**, or a function that is called within another function. Programmers break their programs into subroutines in order to keep the length of functions short and to improve readability.

```
MINIMUM3(a, b, c)
    if a > b
        return MINIMUM2(b, c)
    else
        return MINIMUM2(a, c)
```

STOP and Think: Can you rewrite **MINIMUM3** using just a single line of pseudocode?

EXERCISE BREAK: Write pseudocode for an algorithm **MINIMUM4**(a, b, c, d) that computes the minimum of four numbers.

Sometimes, we may omit the "**else**" statement.

for *loops*

Consider the following problem.

Summing Integers Problem:
Compute the sum of the first n positive integers.

Input: A positive integer n.
Output: The sum of the first n positive integers.

If n were a fixed number, then we could solve this problem using our existing framework. For example, the following program **SUM5** returns the sum of the first five integers (i.e., $1 + 2 + 3 + 4 + 5 = 15$).

```
SUM5()
    sum ← 0
    i ← 1
    sum ← sum + i
    i ← i + 1
    sum ← sum + i
    i ← i + 1
    sum ← sum + i
    i ← i + 1
    sum ← sum + i
    i ← i + 1
    sum ← sum + i
    return sum
```

We could then write **SUM6**, **SUM7**, and so on. However, we cannot endorse this programming style! After all, to solve the Summing Integers Problem for an *arbitrary* integer n, we will need an algorithm that takes n as its input. This is achieved by the following algorithm, which we call **GAUSS**.

```
GAUSS(n)
    sum ← 0
    for i ← 1 to n
        sum ← sum + i
    return sum
```

GAUSS employs a **for** loop that has the following format:

```
for i ← a to b
    execute X
```

This **for** loop first sets i equal to a and executes instructions X. Afterwards, it increases i by 1 and executes X again. It repeats this process by increasing i by 1 until it becomes

equal to b, when it makes a final execution of X. That is, i varies through the values $a, a+1, a+2, \ldots, b-1, b$ during execution of the loop.

while loops

A different way of summing the first n integers, called **AnotherGauss**, is shown below.

```
AnotherGauss(n)
    sum ← 0
    i ← 1
    while i ≤ n
        sum ← sum + i
    return sum
```

This algorithm uses a **while** loop having the following format:

```
while statement X is true
    execute Y
```

The **while** loop checks condition X; if X is true, then it executes instructions Y. This process is repeated until X is false. (Note: if X winds up always being true, then the **while** loop enters an **infinite loop**, which you should avoid at all costs, because your algorithm will never end.) In the case of **AnotherGauss**, the loop stops executing after n trips through the loop, when i eventually becomes equal to $n+1$ and the numbers 1 through n have been added to *sum*.

Recursive algorithms

Below is yet another algorithm solving the Summing Integers Problem.

```
RecursiveGauss(n)
    if n > 0
        sum ← RecursiveGauss(n − 1) + n
    else
        sum ← 0
    return sum
```

You may be confused by the fact that **RECURSIVEGAUSS**(n) calls **RECURSIVEGAUSS**$(n-1)$ as a subroutine. So imagine that you are computing the sum of the first 100 positive integers, but you are lazy and ask your friend to compute the sum of the first 99 positive integers for you. As soon as your friend has computed it, you will simply add 100 to the result, and you are done! Yet little do you know that your friend is just as lazy, and she asks her friend to compute the sum of the first 98 integers, to which she adds 99 and then passes to you. The story continues until a friend is assigned 1. Although every individual in this chain of friends is lazy, the group is nevertheless able to compute the sum.

RECURSIVEGAUSS presents an example of a **recursive algorithm**, which subcontracts a job by calling itself (on a smaller input).

EXERCISE BREAK: Can you write one line of pseudocode solving the Summing Integers Problem?

You have undoubtedly been wondering why we have named all these summing algorithms "Gauss". In 1785, a primary school teacher asked his class to sum the integers from 1 to 100, assuming that this task would occupy them for the rest of the day. He was shocked when an 8 year old boy thought for a few seconds and wrote down the answer, 5,050. This boy was Karl Gauss, and he would go on to become one of the greatest mathematicians of all time. The following one-line algorithm implements his idea for solving the Summing Integers Problem. (Why does it work?)

```
GAUSS(n)
    return (n + 1) * n/2
```

Arrays

Finally, when writing pseudocode, we may also use an **array**, or an ordered sequence of variables. We often use a single letter to denote an array, e.g., $a = (a_1, a_2, \ldots, a_n)$.

EXERCISE BREAK: Consider the algorithm following this exercise. What does it do?

```
RABBITS(n)
    a_1 ← 1
    a_2 ← 1
    for i ← 3 to n
        a_i ← a_{i−1} + a_{i−2}
    return a
```

RABBITS(n) computes the first n Fibonacci numbers and places them in an array. Why do you think that we called it **RABBITS**?

Bibliography

Cristianini, N. and M. W. Hahn (2006). *Introduction to Computational Genomics: A Case Studies Approach*. Cambridge University Press.

Gao, F. and C.-T. Zhang (2008). "Ori-Finder: A web-based system for finding *oriCs* in unannotated bacterial genomes". *BMC Bioinformatics* Vol. 9: 79.

Gardner, M. (1974). "Mathematical Games". *Scientific American* Vol. 230: 120–125.

Geman, S. and D. Geman (1984). "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images". *IEEE Transactions on Pattern Analysis and Machine Intelligence* Vol. PAMI-6: 721–741.

Grigoriev, A. (1998). "Analyzing genomes with cumulative skew diagrams". *Nucleic Acids Research* Vol. 26: 2286–2290.

Grigoriev, A. (2011). "How do replication and transcription change genomes?" In: *Bioinformatics for Biologists*. Ed. by P. A. Pevzner and R. Shamir. Cambridge University Press, 111–125.

Guibas, L. and A. Odlyzko (1981). "String overlaps, pattern matching, and nontransitive games". *Journal of Combinatorial Theory, Series A* Vol. 30: 183 –208.

Harmer, S. L., J. B. Hogenesch, M. Straume, H. S. Chang, B. Han, T. Zhu, X. Wang, J. A. Kreps, and S. A. Kay (2000). "Orchestrated transcription of key pathways in *Arabidopsis* by the circadian clock". *Science* Vol. 290: 2110–2113.

Hertz, G. Z. and G. D. Stormo (1999). "Identifying DNA and protein patterns with statistically significant alignments of multiple sequences." *Bioinformatics* Vol. 15: 563–577.

Konopka, R. J. and S. Benzer (1971). "Clock mutants of Drosophila melanogaster". *Proceedings of the National Academy of Sciences of the United States of America* Vol. 68: 2112–2116.

Lawrence, C. E., S. F. Altschul, M. S. Boguski, J. S. Liu, A. F. Neuwald, and J. C. Wootton (1993). "Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment". *Science* Vol. 262: 208–214.

Liachko, I., R. A. Youngblood, U. Keich, and M. J. Dunham (2013). "High-resolution mapping, characterization, and optimization of autonomously replicating sequences in yeast". *Genome Research* Vol. 23: 698–704.

Lobry, J. R. (1996). "Asymmetric substitution patterns in the two DNA strands of bacteria". *Molecular Biology and Evolution* Vol. 13: 660–665.

Lundgren, M., A. Andersson, L. Chen, P. Nilsson, and R. Bernander (2004). "Three replication origins in *Sulfolobus* species: Synchronous initiation of chromosome replication and asynchronous termination". *Proceedings of the National Academy of Sciences of the United States of America* Vol. 101: 7046–7051.

Park, H. D., K. M. Guinn, M. I. Harrell, R. Liao, M. I. Voskuil, M. Tompa, G. K. Schoolnik, and D. R. Sherman (2003). "Rv3133c/dosR is a transcription factor that mediates the hypoxic response of Mycobacterium tuberculosis". *Molecular Microbiology* Vol. 48: 833–843.

Sedgewick, R. and P. Flajolet (2013). *An Introduction to the Analysis of Algorithms*. Addison-Wesley.

Sernova, N. V. and M. S. Gelfand (2008). "Identification of replication origins in prokaryotic genomes". *Briefings in Bioinformatics* Vol. 9: 376–391.

Solov'ev, A. (1966). "A combinatorial identity and its application to the problem about the first occurence of a rare event". *Theory of Probability and its Applications* Vol. 11: 276–282.

Wang, X., C. Lesterlin, R. Reyes-Lamothe, G. Ball, and D. J. Sherratt (2011). "Replication and segregation of an *Escherichia coli* chromosome with two replication origins". *Proceedings of the National Academy of Sciences* Vol. 108: E243–E250.

Xia, X. (2012). "DNA replication and strand asymmetry in prokaryotic and mitochondrial genomes". *Current Genomics* Vol. 13: 16–27.