Interestingly, among the four most frequent 9-mers in the *ori* 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 tccttaaccctctattttttacggaaga**ATGATCAAG**ctgctgct**CTTGATCAT**cgtttc

Finding a 9-mer that appears six or more 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 or more times alone. This statistical evidence leads us to the working hypothesis that "**ATGATCAAG**" and its reverse complement "**CTTGATCAT**" indeed represent *DnaA* boxes in*Vibrio cholerae*. Our computational conclusion makes sense biologically because the *DnaA* protein that binds to *Dna*A boxes and initiates replication does not care which of the two strands it binds to. For our purposes, both "**ATGATCAAG**" and "**CTTGATCAT**" represent *DnaA* boxes.

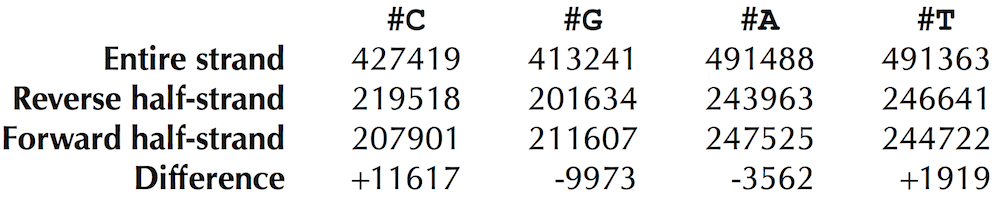
After solving the Pattern Matching Problem, we discover that "**ATGATCAAG**" appears 17 times in the following positions of the *Vibrio cholerae* genome:

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 *ori* 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. The preceding exercise verifies 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.

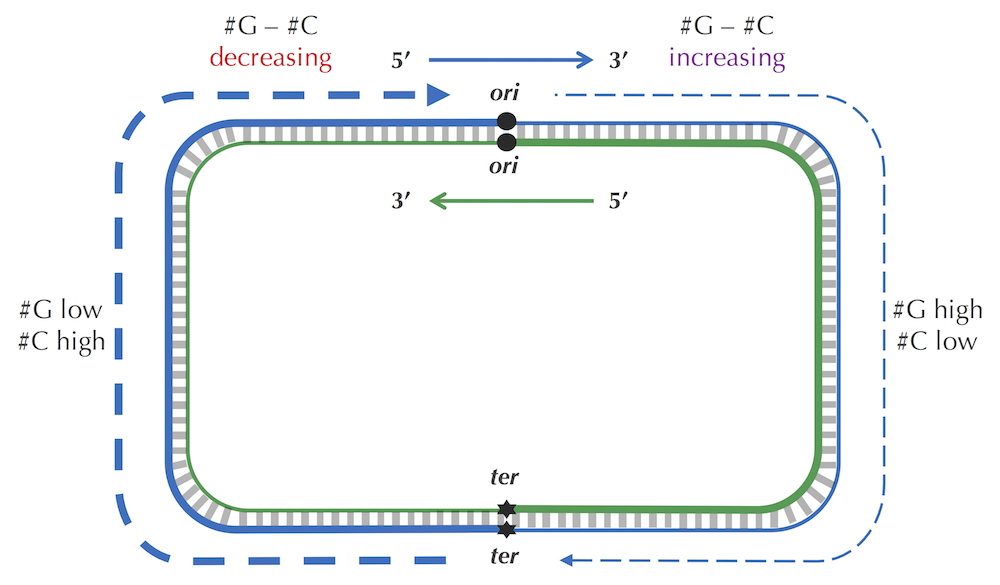
**STOP and Think:** Is it safe to conclude that "**ATGATCAAG**"/"**CTTGATCAT**" also represents a *DnaA* box in other bacterial genomes?

In the table containing nucleotide counts for *T. petrophila* (reproduced below), we noted that not just C but also G has peculiar statistics on the forward and reverse half-strands.



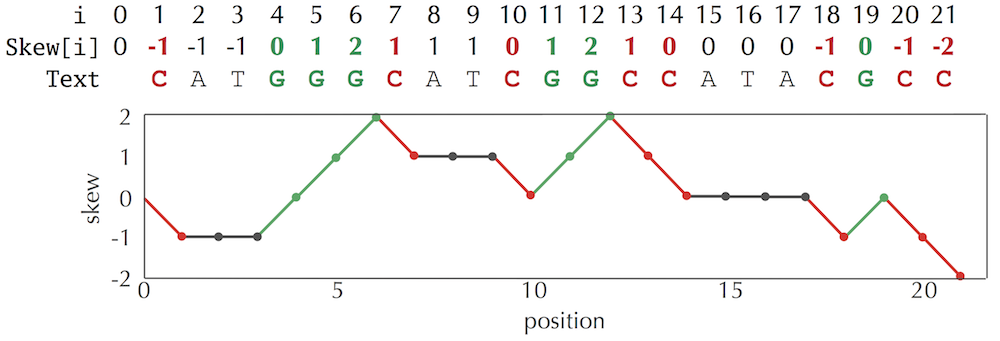
In practice, scientists use a more accurate approach that accounts for both G and C when searching for *ori*. As the above figure illustrates, the difference between the total amount of guanine and the total amount of cytosine is negative on the reverse half-strand and positive on the forward half-strand.

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 below).



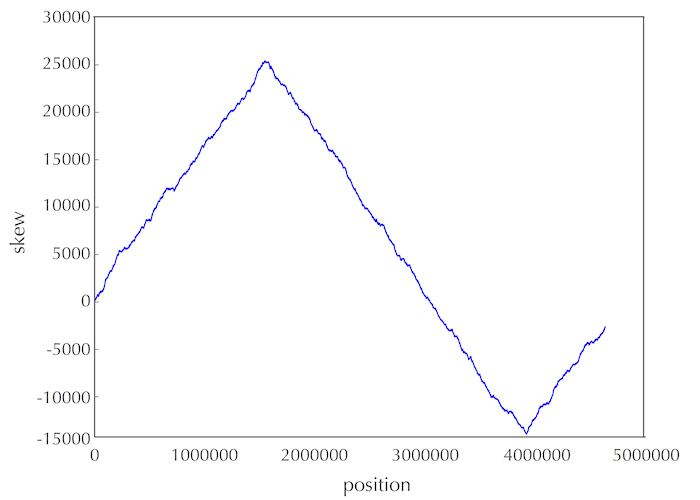
**Figure:** Because of deamination, each forward half-strand has more guanine than cytosine, and each reverse half-strand has more cytosine than guanine. The difference between the counts of G and C is therefore positive on the forward half-strand and negative on the reverse half-strand.

The **skew diagram** of Genome is defined by plotting i against Skew[i] as i ranges from 0 to len(Genome). The figure below shows the skew diagram for the genome from the previous step.



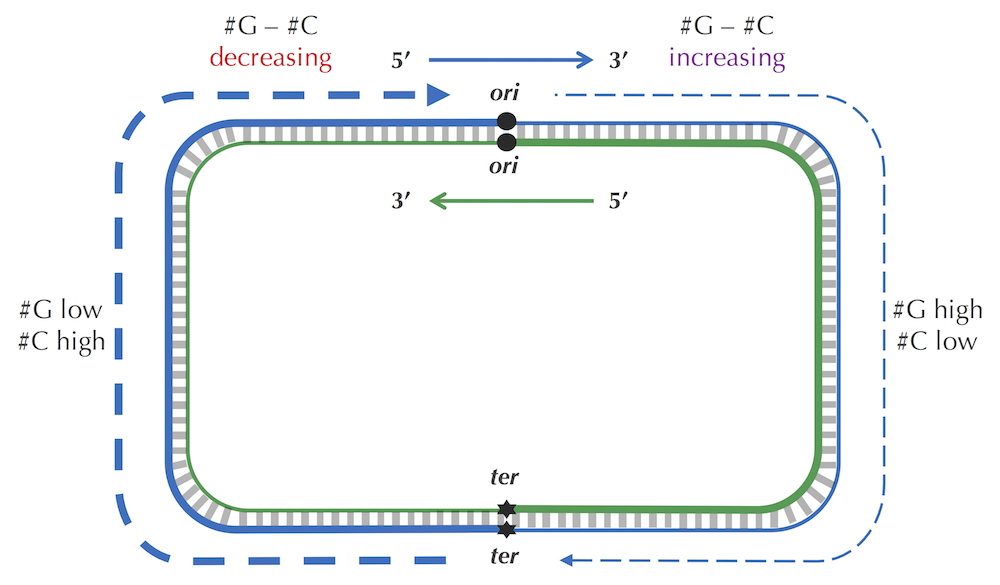
**Figure:**(Top) The skew array for Genome = CATGGGCATCGGCCATACGCC, reproduced from the previous step. (Bottom) The skew diagram corresponding to Genome. The skew increases when we encounter **G** and decreases when we encounter **C**.

The figure below depicts the skew diagram for a linearized *E. coli*genome. The pattern is even stronger than the pattern observed when we visualized the symbol array! It turns out that the skew diagram for many bacterial genomes has a similar characteristic shape.



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

Let’s follow the 5' → 3' direction of DNA and walk along the chromosome from *ter* to *ori* (along a reverse half-strand), then continue on from *ori* to *ter* (along a forward half-strand). In the figure below, we see 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 *ori*!



Solving the Minimum Skew Problem now provides us with an approximate location of *ori* 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 the position of *ori* 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

Before we give up, let’s examine the *ori* 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* *ori* 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 tccttaaccctctattttttacggaaga**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 modifications as well.

We now make a final attempt to find *DnaA* boxes in the region of the *E. coli* genome hypothesized by the minimum skew as *ori*. Although the minimum of the skew diagram for *E. coli* is found at position 3923620, we should not assume that its *ori* 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., 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) 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 indeed a most frequent 9-mer, along with its reverse complement **TGTGGATAA** (with 1 mismatch):

aatgatgatgacgtcaaaaggatccggataaaacat**ggtgattgcctcgcataacgcggt atgaaaatggattgaagcccgggccgtggattctactcaactttgtcggcttgagaaaga cctgggatcctgggtattaaaaagaagatctatttatttagagatctgttctattgtgat ctcttattaggatcgcactgcccTGTGGATAAcaaggatccggcttttaagatcaacaac ctggaaaggatcattaactgtgaatgatcggtgatcctggaccgtataagctgggatcag aatgaggggTTATACACAactcaaaaactgaacaacagttgttcTTTGGATAActaccgg ttgatccaagcttcctgacagagTTATCCACAgtagatcg**cacgatctgtatacttattt gagtaaattaacccacgatcccagccattcttctgccggatcttccggaatgtcgtgatc  
aagaatgttgatcttcagtg

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

We now make a final attempt to find DnaA boxes in the region of the E. coli genome hypothesized by the minimum skew as ori. Although the minimum of the skew diagram for E. coli is found at position 3923620, we should not assume that its ori 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., 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) 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 indeed a most frequent 9-mer, along with its reverse complement TGTGGATAA (with 1 mismatch):

aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgcataacgcggt atgaaaatggattgaagcccgggccgtggattctactcaactttgtcggcttgagaaaga cctgggatcctgggtattaaaaagaagatctatttatttagagatctgttctattgtgat ctcttattaggatcgcactgcccTGTGGATAAcaaggatccggcttttaagatcaacaac ctggaaaggatcattaactgtgaatgatcggtgatcctggaccgtataagctgggatcag aatgaggggTTATACACAactcaaaaactgaacaacagttgttcTTTGGATAActaccgg ttgatccaagcttcctgacagagTTATCCACAgtagatcgcacgatctgtatacttattt gagtaaattaacccacgatcccagccattcttctgccggatcttccggaatgtcgtgatc

aagaatgttgatcttcagtg

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

Thus, the moral of this chapter is that even though computational analysis can be powerful, bioinformaticians should collaborate with biologists to verify their predictions.

**WEEK 3**

**The circadian clock**

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 what time it is? 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.

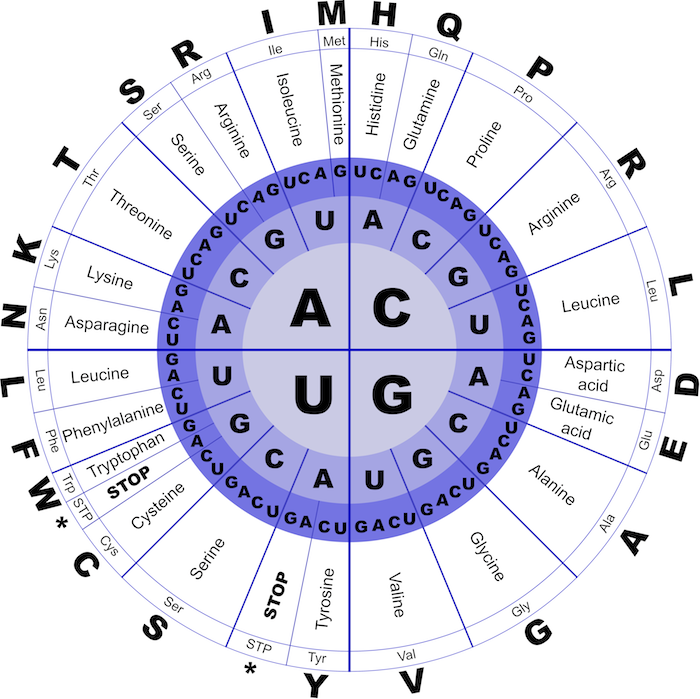
We will first focus on plants, since maintaining the circadian clock in plants is a matter of life and death. Consider how many plant functions depend on when the sun rises and sets. Indeed, biologists estimate that over a thousand plant genes are circadian, including the genes related to photosynthesis, photo reception, and flowering. But what does it mean for a gene be circadian?

## Gene expression

The **Central Dogma of Molecular Biology** states that “DNA makes RNA makes protein.” The DNA corresponding to a gene is first **transcribed** into a strand of RNA composed of four **ribonucleotides**: adenine, cytosine, guanine, and uracil (which replaces thymine in DNA). Then, the RNA transcript is **translated** into the amino acid sequence of a protein, which performs some function in the cell.

Much like DNA replication, the chemical machinery underlying transcription and translation is fascinating, but from a computational perspective, both processes are straightforward. Transcription simply replaces all occurrences of T in a DNA string with U. The resulting strand of RNA is translated into an amino acid sequence as follows. During translation, the RNA strand is partitioned into non-overlapping 3-mers called **codons**. Then, each codon is converted into one of 20 amino acids via the **genetic code**; the resulting sequence can be represented as an **amino acid string** over a 20-letter alphabet. As illustrated in the figure below, each of the 64 RNA codons encodes its own amino acid (some codons encode the same amino acid), with the exception of three **stop codons** that do not translate into amino acids and serve to halt translation (see [DETOUR: Discovery of Codons and Split Genes](https://stepik.org/lesson/Detour-Discovery-of-Codons-and-Split-Genes-24205)). For example, the DNA string "**TATACGAAA**" transcribes into the RNA string "**UAUACGAAA**", which in turn translates into the amino acid string "**YTK**".

The key point is that cells are able to transcribe different genes into RNA at different rates. This variance in the production of a gene’s transcripts, or **gene expression**, explains how a brain cell and a skin cell can have the same DNA but perform vastly different functions. Variation in gene expression through the day also accounts for how the cell can keep track of time. .



**Figure:** The genetic code describes the translation of an RNA 3-mer (codon) into one of 20 amino acids. The first three circles, moving from the inside out, represent the first, second, and third nucleotides of a codon. The fourth, fifth, and sixth circles define the translated amino acid in three ways: the amino acid’s full name, its 3-letter abbreviation, and its single-letter abbreviation. Three of the 64 total RNA codons are stop codons, which halt translation.

## Regulatory proteins

It turns out that every plant cell keeps track of day and night independently of other cells, and that just three plant genes, called LHY, CCA1, and TOC1, are the clock’s master timekeepers. Such 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 LHY and CCA1, whereas LHY and CCA1 repress the expression of TOC1, resulting in a **negative feedback loop**. In the morning, sunlight activates the transcription of LHY and CCA1, triggering the repression of TOC1 transcription. As light diminishes, so does the production of LHY and CCA1, which in turn do not repress TOC1 any more. Transcription of TOC1 peaks at night and starts promoting the transcription of LHY and CCA1, which in turn repress the transcription of TOC1, and the cycle begins again.

LHY, 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.

n 2000, Steve Kay used **DNA arrays** (see [DETOUR: DNA Arrays](https://stepik.org/lesson/Detour-DNA-Arrays-24206/step/1?course=Which-DNA-Patterns-Play-The-Role-of-Molecular-Clocks-(Part-1)&unit=6801)) 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.

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.

**STOP and Think:** What is the possible downside of concatenating all the upstream regions into a single string and looking for frequent words in order to find a motif?

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-kB** that activates various immunity genes in flies. However, NF-κB binding sites are nowhere near as conserved as the evening element. The figure below 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:** The ten candidate NF-κB binding sites appearing in the *Drosophila melanogaster* genome. The colored upper case 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 "atgaccgggatactgataaaaaaaagggggggggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg"  
 2 "acccctattttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaataaaaaaaaaggggggga"  
 3 "tgagtatccctgggatgacttaaaaaaaagggggggtgctctcccgatttttgaatatgtaggatcattcgccagggtccga"  
 4 "gctgagaattggatgaaaaaaaagggggggtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga"  
 5 "tcccttttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaataaaaaaaagggggggcttatag"  
 6 "gtcaatcatgttcttgtgaatggatttaaaaaaaaggggggggaccgcttggcgcacccaaattcagtgtgggcgagcgcaa"  
 7 "cggttttggcccttgttagaggcccccgtaaaaaaaagggggggcaattatgagagagctaatctatcgcgtgcgtgttcat"  
 8 "aacttgagttaaaaaaaagggggggctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta"  
  9 "ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcataaaaaaaagggggggaccgaaagggaag"   
10 "ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttaaaaaaaaggggggga"

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

This is a simple problem: applying the FrequentWords algorithm that we developed in the previous chapter 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 "atgaccgggatactgat**AAAAAAAAGGGGGGG**ggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg"  
 2 "acccctattttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaata**AAAAAAAAGGGGGGG**a"  
 3 "tgagtatccctgggatgactt**AAAAAAAAGGGGGGG**tgctctcccgatttttgaatatgtaggatcattcgccagggtccga"  
 4 "gctgagaattggatg**AAAAAAAAGGGGGGG**tccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga"  
 5 "tcccttttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaat**AAAAAAAAGGGGGGG**cttatag"  
 6 "gtcaatcatgttcttgtgaatggattt**AAAAAAAAGGGGGGG**gaccgcttggcgcacccaaattcagtgtgggcgagcgcaa"  
 7 "cggttttggcccttgttagaggcccccgt**AAAAAAAAGGGGGGG**caattatgagagagctaatctatcgcgtgcgtgttcat"  
 8 "aacttgagtt**AAAAAAAAGGGGGGG**ctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta"  
  9 "ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcat**AAAAAAAAGGGGGGG**accgaaagggaag"   
10 "ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagctt**AAAAAAAAGGGGGGG**a"

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

 1 "atgaccgggatactgat**AgAAgAAAGGttGGG**ggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg"  
 2 "acccctattttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaata**cAAtAAAAcGGcGGG**a"  
 3 "tgagtatccctgggatgactt**AAAAtAAtGGaGtGG**tgctctcccgatttttgaatatgtaggatcattcgccagggtccga"  
 4 "gctgagaattggatg**cAAAAAAAGGGattG**tccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga"  
 5 "tcccttttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaat**AtAAtAAAGGaaGGG**cttatag"  
 6 "gtcaatcatgttcttgtgaatggattt**AAcAAtAAGGGctGG**gaccgcttggcgcacccaaattcagtgtgggcgagcgcaa"  
 7 "cggttttggcccttgttagaggcccccgt**AtAAAcAAGGaGGG**ccaattatgagagagctaatctatcgcgtgcgtgttcat"  
 8 "aacttgagtt**AAAAAAtAGGGaGcc**ctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta"  
  9 "ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcat**ActAAAAAGGaGcGG**accgaaagggaag"   
10 "ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagctt**ActAAAAAGGaGcGG**a"

FrequentWords is no longer going to help us, since **AAAAAAAAGGGGGGG** does not even appear in the strings above. We could adapt the Frequent Words Problem into a “Frequent Words with Mismatches Problem”. However, concatenating all the strings 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 DNA string. In contrast, a regulatory motif is a pattern that appears at least once in each one of several different regions that are scattered throughout the genome.

Furthermore, 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-κB. 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.

## From motifs to profile matrices and consensus strings

A computational problem formulation for motif finding 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 a list of t DNA strings Dna, where each string has length n, and select a k-mer from each string to form a collection Motifs, which we represent as a t x k motif matrix. In the figure below, which shows the motif matrix for the NF-κB binding sites from the figure below, we indicate the most frequent nucleotide in each column of the motif matrix by upper case letters. If there are multiple most frequent 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.

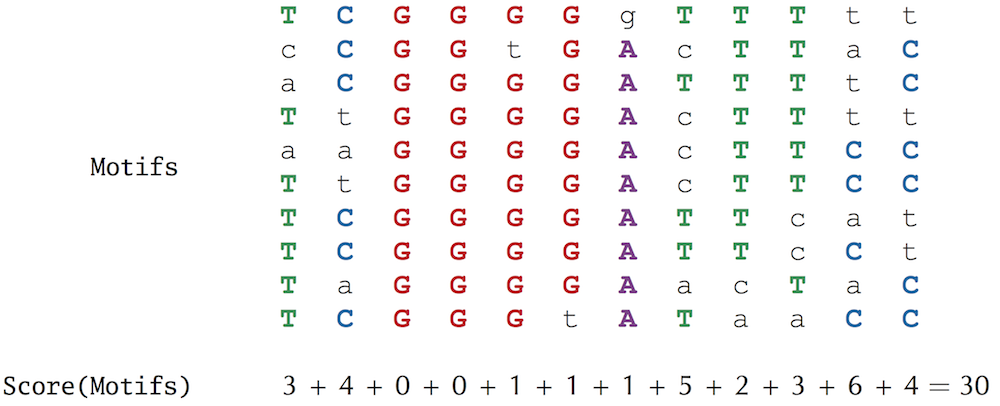


**Figure:** The NF-κB binding sites form a 10 x 12 motif matrix, with the most frequent nucleotide in each column shown in upper case letters and all other nucleotides shown in lower case letters.

In Python, we will represent a motif matrix as a list of strings Motifs. We can access the i-th string in the motif matrix by calling Motifs[i]; we can access the j-th symbol in this string by calling Motifs[i][j].

**Python Practice:** Make sure that you have completed Units 1-8 of the Python track at Codecademy if you have not done so already.

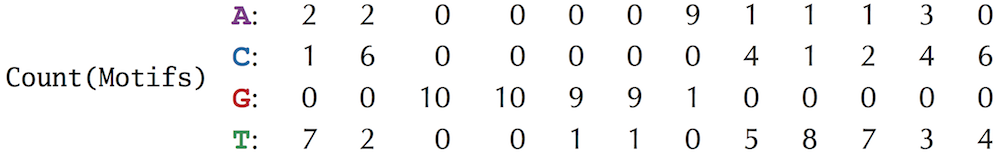
By varying the choice of k-mers in each string, we can construct a large number of different motif matrices from a given sample of DNA strings. Our goal is to select k-mers resulting in the most “conserved” motif matrix, meaning the matrix with the most upper case letters (and thus the fewest number 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 (see updated figure below). Our goal is to find a collection of k-mers that minimizes this score (for more on motif scoring functions, see [DETOUR: Motif Scoring Functions](https://stepik.org/lesson/Detour-Motif-Scoring-Functions-24208)).



For a given choice of Motifs, we can construct a 4 x k **count matrix**, denoted Count(Motifs), counting the number of occurrences of each nucleotide in each column of the motif matrix; element (i,j) of Count(Motifs) stores the number of times that nucleotide i appears in column j of Motifs. (See updated figure below).



One way of representing a count matrix in Python is to create a list for each row of the matrix and then organize these lists into a larger dictionary (thus creating a dictionary whose keys are nucleotides and whose values are lists). For example, given the count matrix from the previous step:



We can represent this count matrix in Python as follows:

count = {"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]

       }

To generate a count matrix from an arbitrary list of strings Motifs, we need to first initialize the count matrix, represented as a dictionary:

count = {}

We then range over all nucleotides symbol and create a list of zeroes corresponding to count[symbol].

k = len(Motifs[0])

for symbol in "ACGT":

count[symbol] = []

for j in range(k):

count[symbol].append(0)

Note that the first line above sets k equal to the length of Motifs[0], the first string in Motifs, which is the length of every string in Motifs. Also, note the difference between the line count = {} (which forms an empty dictionary) and the line count[symbol] = [] (which forms an empty list). Finally, we need to range over all elements symbol = Motifs[i][j] of the count matrix and add 1 to count[symbol][j].

    t = len(Motifs)

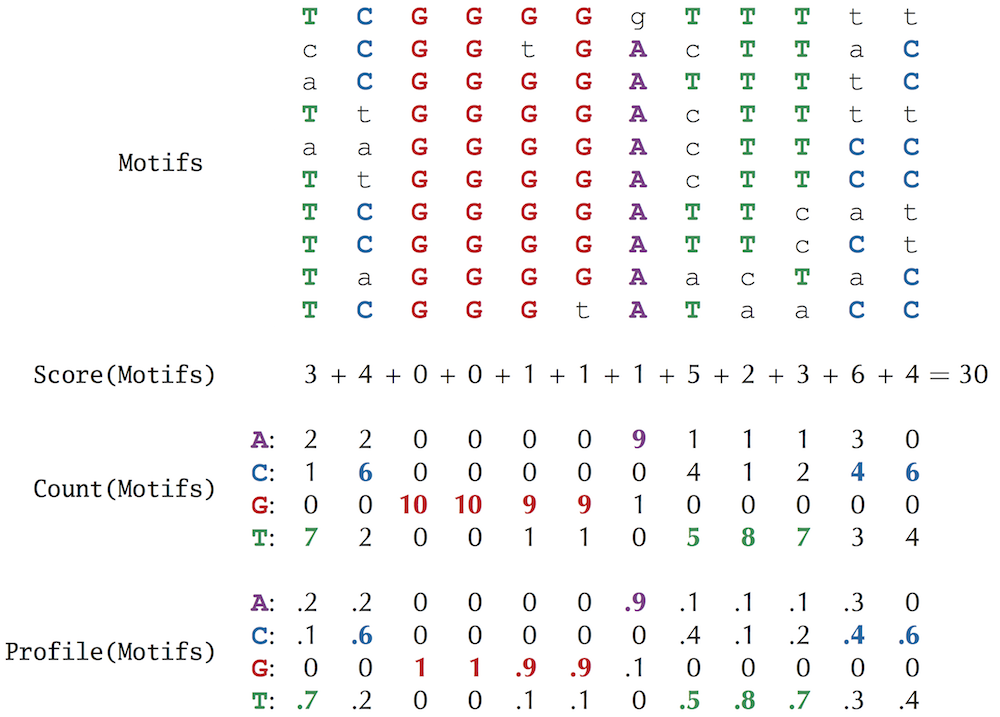
for i in range(t):

for j in range(k):

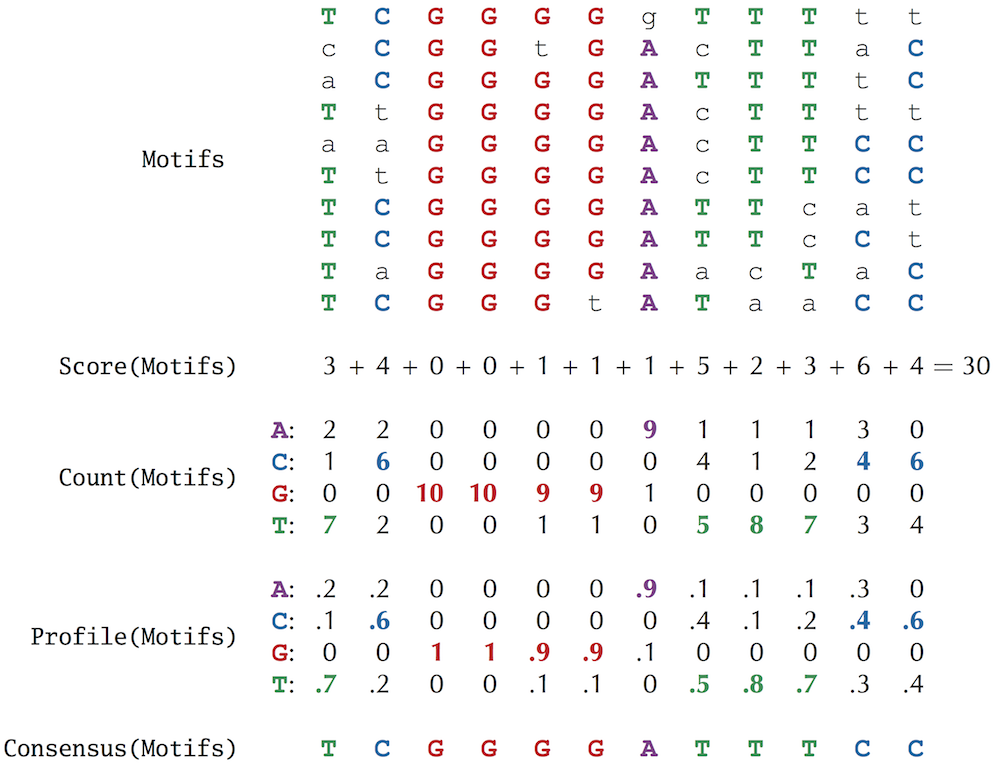
symbol = Motifs[i][j]

count[symbol][j] += 1

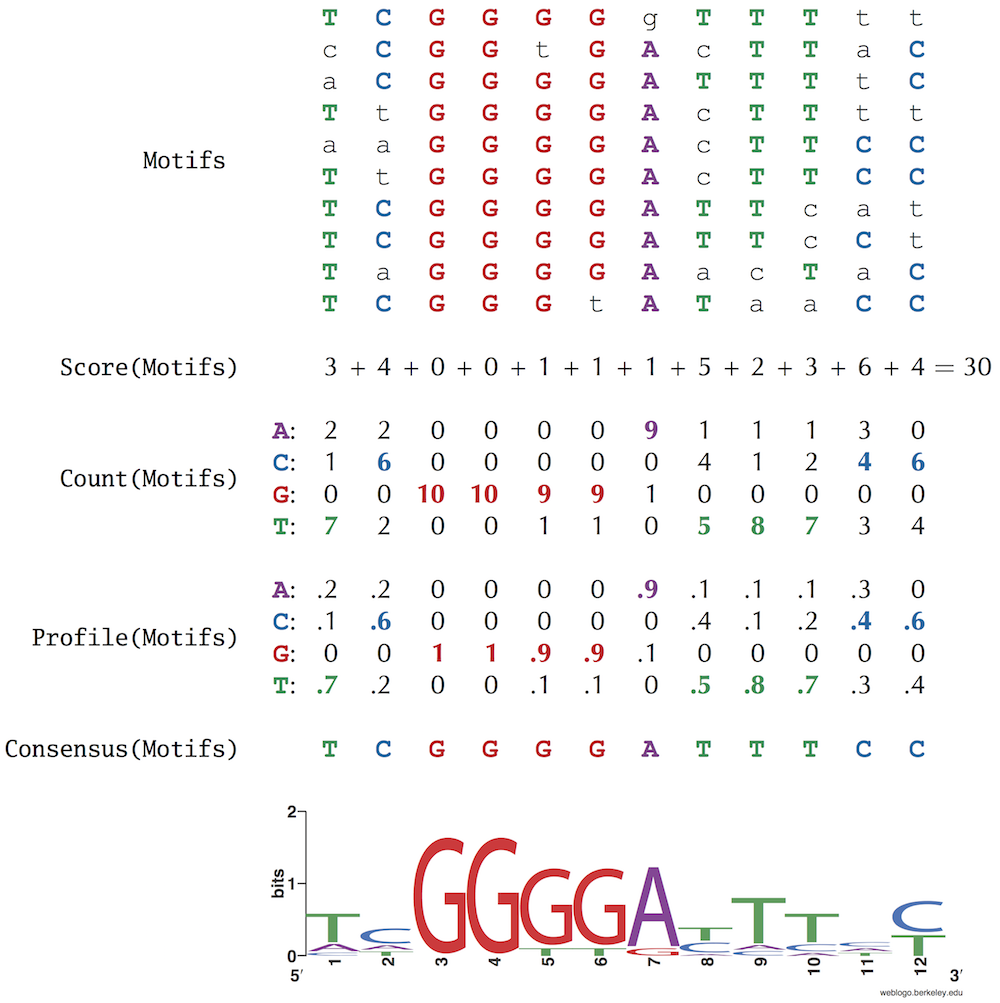
As shown below, we will further divide all of the elements in the count matrix by t, the number of rows in Motifs. This results in a **profile matrix** Profile(Motifs) for which element (i,j) is the *frequency* of the i-th nucleotide in the j-th column of the motif matrix (i.e., the number of occurrences of the i-th nucleotide divided by t, the number of nucleotides in the column). Note that the elements of any column of the profile matrix sum to 1.



Finally, we can 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 a candidate regulatory motif for these regions. For example, as shown below, the consensus string for the NF-κB binding sites is "**TCGGGGATTTCC**".



Biologists also commonly use a **motif logo**, a diagram for visualizing motif conservation that consists of a stack of letters at each position (see the figure below). The relative sizes of letters indicate their frequency in the column, i.e., highly conserved columns in the motif matrix correspond to tall symbols in the motif logo. (For more on motif logos, see [DETOUR: Motif Scoring Functions](https://stepik.org/lesson/Detour-Motif-Scoring-Functions-24208)).



## The Motif Finding Problem

Now that we have a good grasp of scoring a collection of k-mers, we are ready to formulate a computational problem for motif finding.

**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.

**Brute force search** (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 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.

## Analyzing the runtime of brute force motif finding

Throughout this chapter, we will benchmark our motif finding algorithms by using a **Subtle Motif Problem** that 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**".

To benchmark BruteForceMotifSearch, note that there are n-k+1 choices of k-mers in each of t strings, so that there are (n-k+1)t different ways to form Motifs. For each choice of Motifs, the algorithm calculates Score(Motifs), which requires k⋅t steps. Thus, assuming that k is much smaller than n (as is the case for biological datasets), the overall running time of the brute force motif finding algorithm is on the order of ((n-k+1)t)⋅k⋅t steps. For the Subtle Motif Problem, this is on the order of 1029 steps. You may recall that the naive algorithm we developed to generate a symbol array in Chapter 1 took several days to carry out an algorithm with just 1013 steps. In this case, the earth will have been destroyed by the sun long before BruteForceMotifSearch will terminate. It goes without saying that we need to devise a faster algorithm!

We have also 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, BruteForceMotifSearch will be too slow to find them.

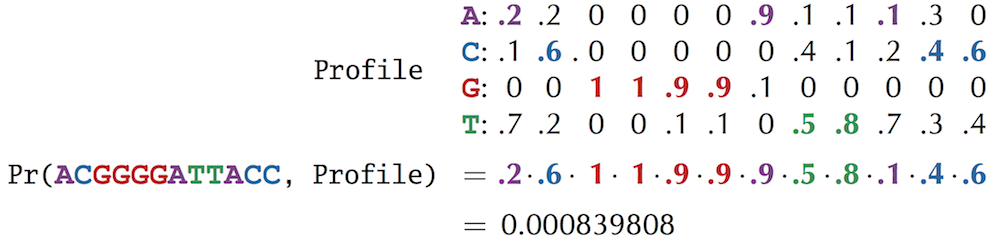
## Using the profile matrix to roll dice

Many algorithms are iterative procedures that must choose among various 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 a strategy 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, 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. We can view each column of Profile(Motifs) as a four-sided biased die with one nucleotide on each side. Thus, a profile matrix with k columns can be viewed as a collection of k dice that 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 the figure below, we reproduce the profile matrix for the NF-kB binding sites, 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 multiplying the highlighted entries in the profile matrix.



**Figure:** Generating 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.

## Motifs in tuberculosis

**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**([click here](http://bioinformaticsalgorithms.com/data/challengedatasets/DosR.txt) to download). In this chapter, we will try to identify motifs in this dataset using the motif finding algorithms that we will develop. However, we will not give you a hint about the DosR motif.

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:

tt**ACCT**taac  
g**ATGT**ctgtc  
acg**GCG**Ttag  
cccta**ACGA**g  
cgtcag**AGGT**

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 string. The issue, however, is that there are so many zeroes 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.

n 1961, Sydney Brenner and Francis Crick established the rule of “one codon, one amino acid” during protein translation. They observed that deleting a single nucleotide or two consecutive nucleotides in a gene dramatically altered the protein product. Paradoxically, deleting three consecutive nucleotides resulted in only minor changes in the protein. For example, the phrase

**THE** · **SLY** · **FOX** · **AND** · **THE** · **SHY** · **DOG**

turns into gibberish after deleting one letter:

**THE** · **SYF** · **OXA** · **NDT** · **HES** · **HYD** · **OG**

or after deleting two letters:

**THE** · **SFO** · **XAN** · **DTH** · **ESH** · **YDO** · **G**

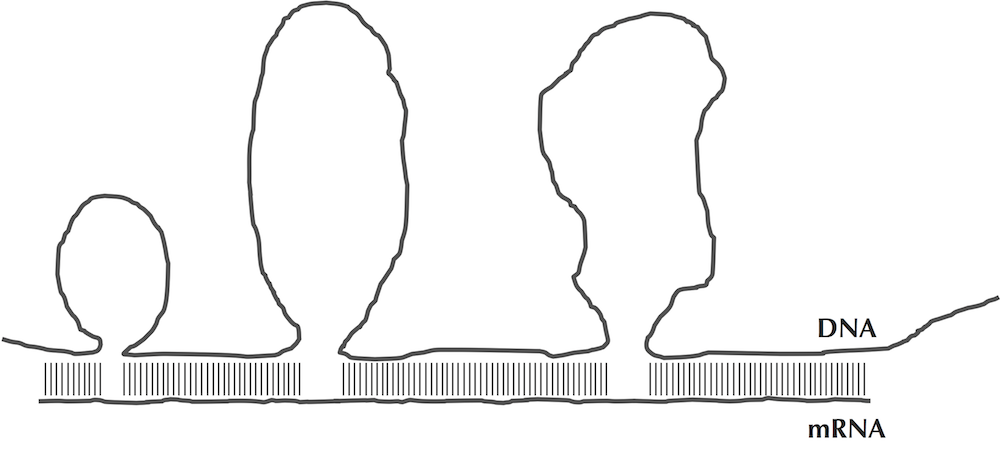
but it makes sense after deleting three letters:

**THE** · **FOX** · **AND** · **THE** · **SHY** · **DOG**

In 1964, Charles Yanofsky demonstrated that a gene and the protein that it produces are **collinear**, meaning that the first codon codes for the first amino acid in the protein, the second codon codes for the second amino acid, etc. For the next thirteen years, biologists believed that a protein was encoded by a long string of *contiguous* nucleotide triplets. However, the independent discovery of **split genes** in 1977 by Phillip Sharp and Richard Roberts proved otherwise and necessitated the computational problem of predicting the locations of genes using only the genomic sequence.

Sharp hybridized RNA encoding an adenovirus protein called **hexon** against a single-strand of adenovirus DNA. If the hexon gene were contiguous, then he expected to see a one-to-one hybridization of RNA bases with DNA bases.

Yet to Sharp’s surprise, when he viewed the RNA-DNA hybridization under an electron microscope, he saw three loop structures, rather than the continuous duplex segment suggested by the contiguous gene model (figure below). This observation implied that the hexon mRNA must be built from four non-contiguous fragments of the adenovirus genome. These four segments, called**exons**, are separated by three fragments (the loops in the figure below) called **introns**, to form a split gene. Split genes are analogous to a magazine article printed on pages 12, 17, 40, and 95, with many pages of advertising appearing in-between.



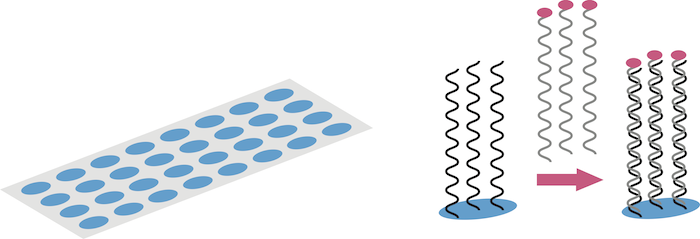
**Figure:**A rendering of Sharp’s electron microscopy experiment that led to the discovery of split genes. When hexon RNA is hybridized against the DNA that generated it, three distinct loops are formed. Because the loops are present in the DNA and are not present in RNA, these loops (called introns) must be removed during the process of RNA formation.

The discovery of split genes caused an interesting quandary: *What happens to the introns*? In other words, the RNA that is transcribed from a split gene (called **precursor mRNA** or **pre-mRNA**) should be longer than the RNA that is used as a template for protein synthesis (called **messenger RNA** or **mRNA**). Some biological process must remove the introns in pre-mRNA and concatenate the exons into a single mRNA string. This process of converting pre-mRNA into mRNA is known as **splicing**, and it is carried out by a molecular machine called the **spliceosome**.

The discovery of split genes led to many new avenues of research. Biologists still debate what purpose introns serve; some introns are viewed as “junk DNA”, while others contain important regulatory elements. Furthermore, the partition of a gene into exons often varies from species to species. For example, a gene in the chicken genome may have a different number of exons than the related gene in the human genome

A **DNA array** is a collection of DNA molecules attached to a solid surface. Each spot on the array is assigned a unique DNA string 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 below).

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.



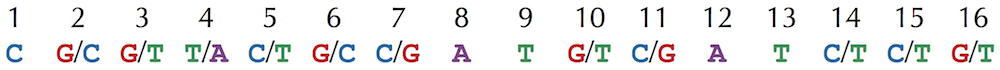
**Figure:**Fluorescently labeled DNA binds to a complementary probe on a DNA array.

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 the figure below. 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 (the figure below).



**Figure:** 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 (figure below). In this sense, the last column (6 C, 4 T) in the motif matrix from the figure in the first step is “more conserved” than the second column (6 C, 2 A, 2 T) and should receive a lower score. (Note that the corresponding position in the motif logo in the figure in the first step is taller as well.)



**Figure:**Taking nucleotides in each column of the NF-κB binding site motif matrix from the figure on the first step 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).

In order to define a motif scoring function taking this into account, first note that 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 the figure in the first step 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, . . . ,*pN*), and is defined as

H(p\_1, \ldots, p\_N) = -\sum\_{i=1}^{N}{p\_i · \log\_{2}{p\_i}}*H*(*p*1​,…,*pN*​)=−*i*=1∑*N*​*pi*​⋅log2​*pi*​

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 the figure in the first step 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−(0.2log2​0.2+0.6log2​0.6+0.0log2​0.0+0.2log2​0.2)≈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−(0.0log2​0.0+0.6log2​0.6+0.0log2​0.0+0.4log2​0.4)≈0.971

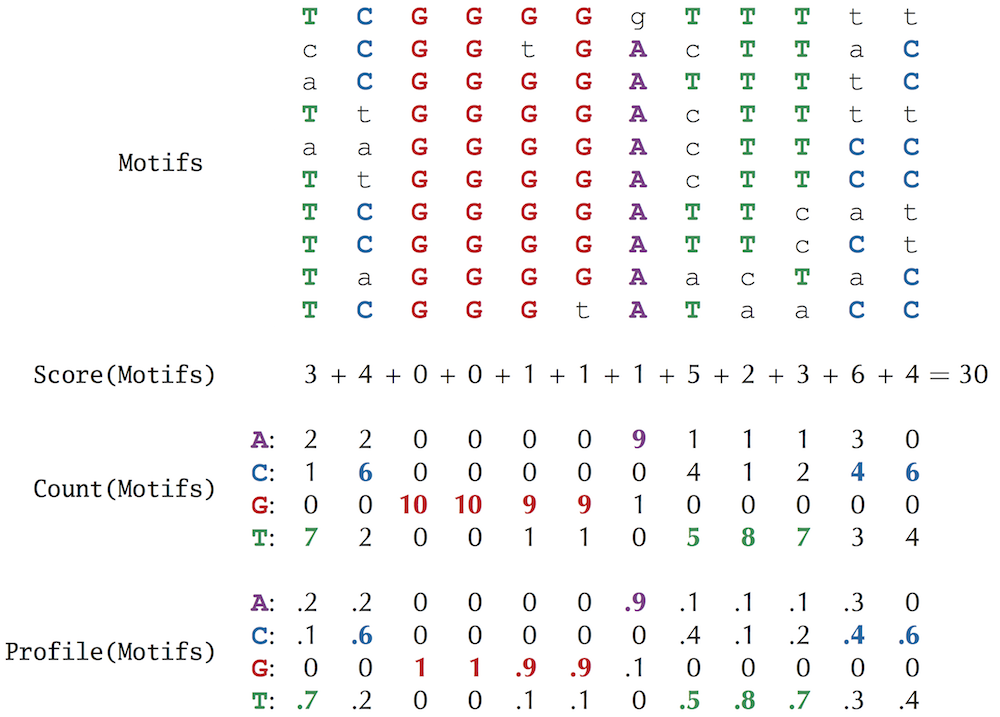
and the entropy of the very conserved 5th column (0.0, 0.0, 0.9, 0.1) is

-(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−(0.0log2​0.0+0.0log2​0.0+0.9log2​0.9+0.1log2​0.1)≈0.467

Note that technically, log2(0) is not defined, but in the computation of entropy, we assume that 0 · log2(0) is equal to 0.

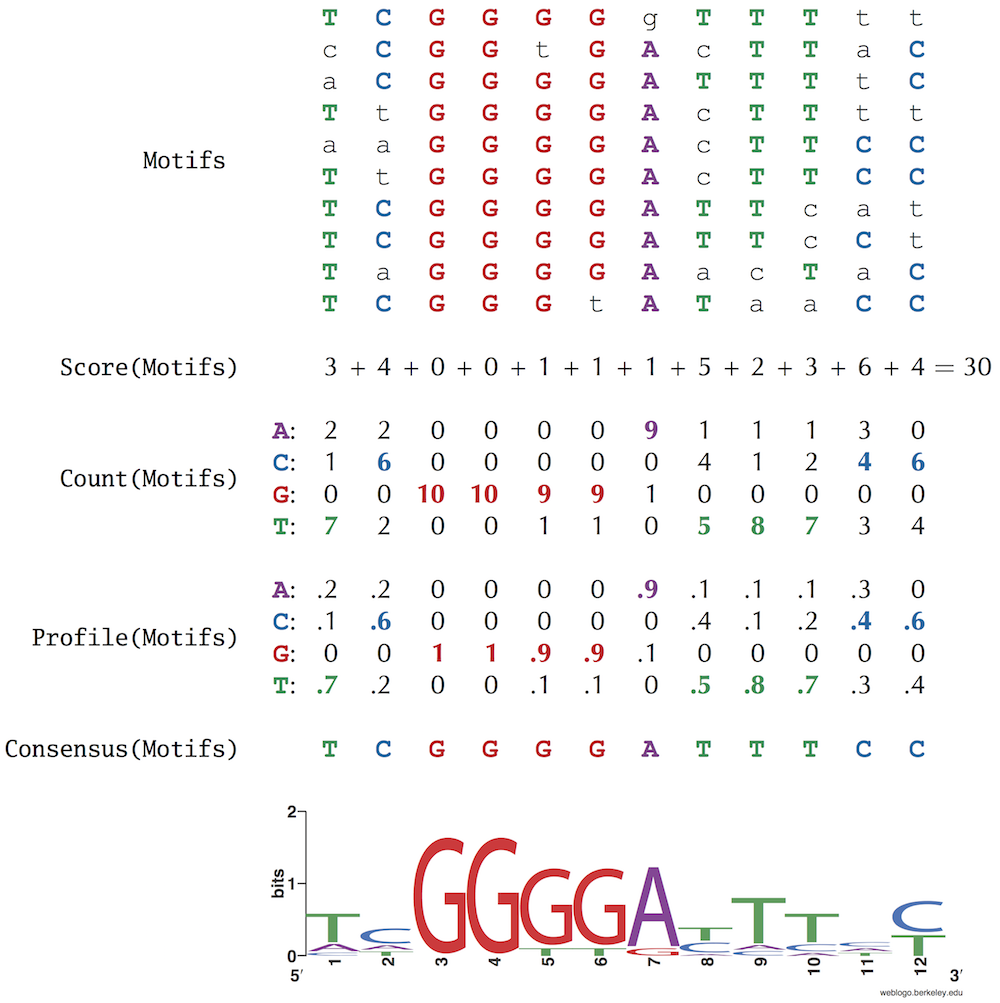
**STOP and Think:**What are the maximum and minimum possible values for the entropy of a probability distribution containing four values?

The entropy of the completely conserved third column of the profile matrix in the figure in the first step 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 · 1/4 · log2 (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 (0 points):**Compute the entropy of the NF-κB motif matrix above.

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 the NF-κB figure, reproduced below). 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,...,*pN*). The lower the entropy, the higher the information content, meaning that tall columns in the motif logo are highly conserved.



## 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,

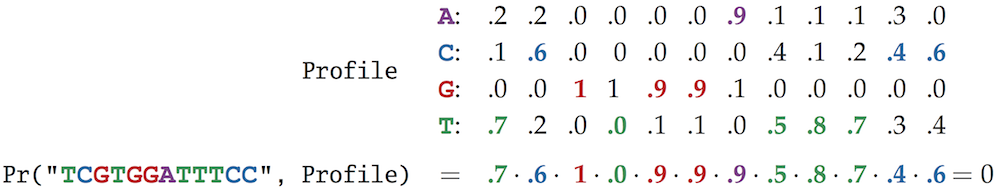
*I beseech you, in the bowels of Christ, think it possible that you may be mistaken.*

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.

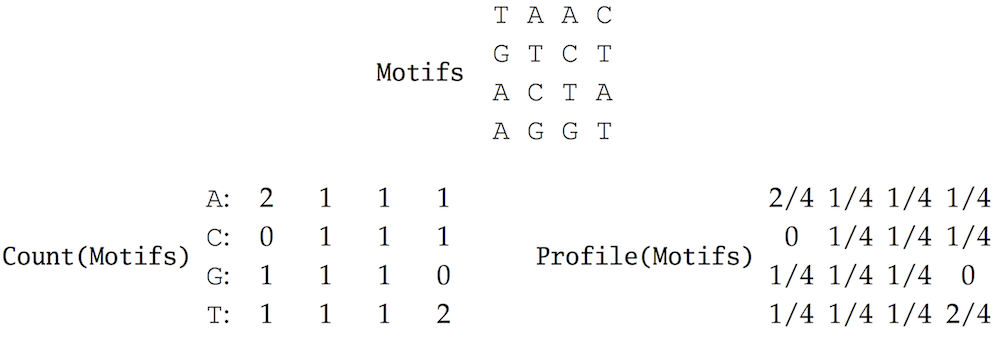
## Laplace’s Rule of Succession

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:



The fourth symbol of "**TCGTGGATTTCC**" causes Pr("**TCGTGGATTTCC**", Profile) to be equal to 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 differs from the consensus string at every position.

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:



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



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) = "**AAAAA**t**A**ga**GGGG**tt". Thus, Laplace’s Rule of Succession has provided a significant improvement over the original GreedyMotifSearch, which returned the consensus string "gtt**AAA**t**A**ga**G**at**G**t**G**" 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?

## 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](https://stepik.org/lesson/Detour-Buffons-Needle-24209)).

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.

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:

  Motifs(Profile(Motifs), 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,

Profile(Motifs(Profile(Motifs), Dna))

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

Motifs(Profile(Motifs(Profile(Motifs), Dna)), Dna).

We can continue to iterate. . .

...Profile(Motifs(Profile(Motifs(Profile(Motifs), Dna)), Dna))...

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

**STOP and Think:** How would you implement an algorithm for generating a random integer?

Simulating the process of generating a random integer is more difficult than you might think and requires more mathematics than we would like to describe here. Fortunately, since the task of generating random numbers arises in so many applications, Python provides a **module** called random for generating them. You can think of a module as a “bundle” of related functions. To use the random module, we place the following statement at the top of our file.

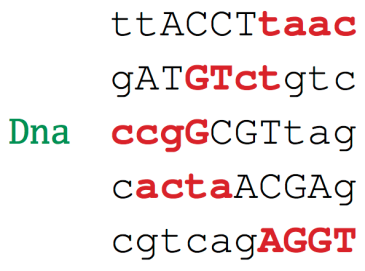
import random

Inside of the random module is a built-in function called randint(1, M) that generates a random integer between 1 and M, inclusively. To call this function, we use

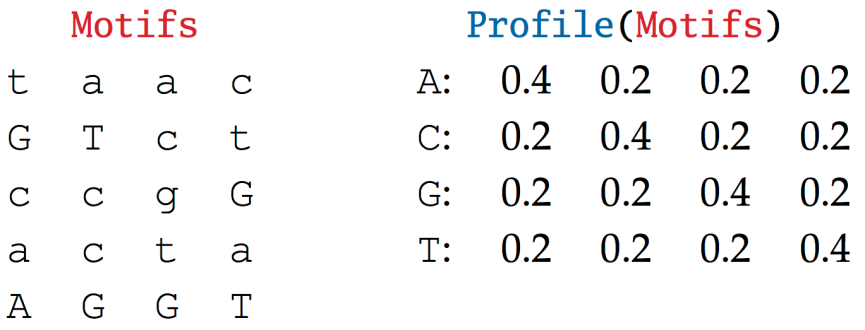
random.randint(1, M)

## Why randomized motif search works

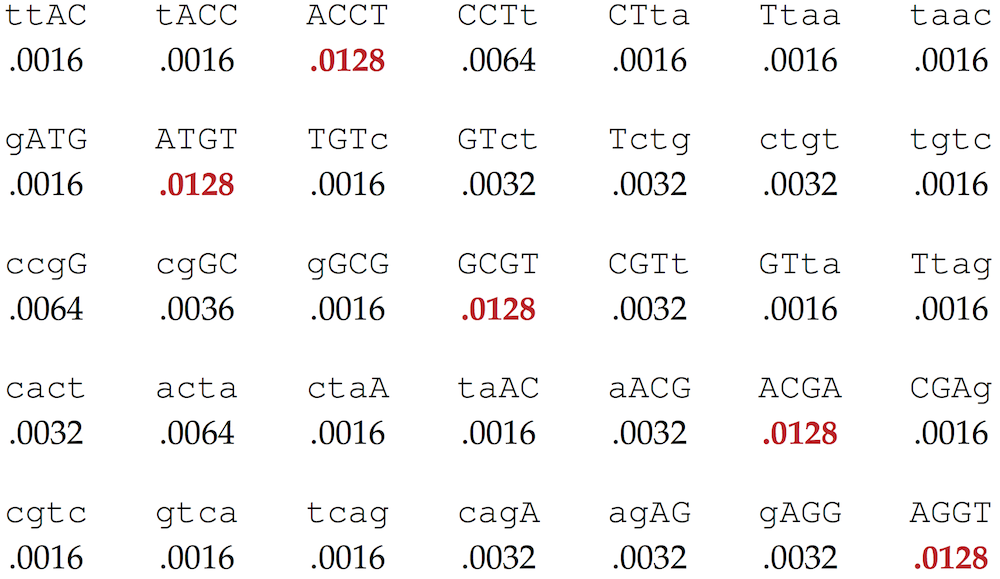
At first glance, RandomizedMotifSearch appears to be doomed. How can this algorithm, which starts from a random guess, possibly find anything useful? To explore RandomizedMotifSearch, let’s run it on five short strings with the implanted (4, 1)-motif ACGT (shown in upper case letters below) and imagine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



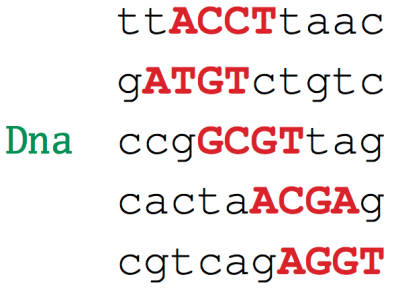
Below, we construct the profile matrix Profile(Motifs) of the chosen 4-mers.



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 Pr("ttAC", Profile) = 0.2 · 0.2 · 0.2 · 0.2 = 0.0016. The maximum probabilities in every row are shown in red below.



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!



**STOP and Think:** How is it possible that randomly chosen k-mers have led us to the correct implanted k-mer?

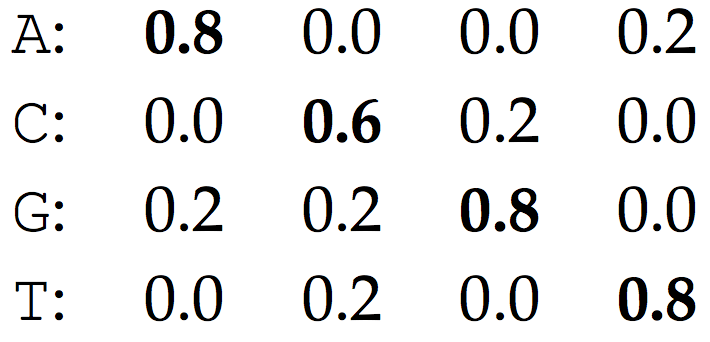
For the Subtle Motif Problem with implanted 15-mer "**AAAAAAAAGGGGGGG**", when we run RandomizedMotifSearch for N = 100,000 times (each time with new randomly selected k-mers), it returns the 15-mers shown in the figure below as the lowest scoring collection Motifs across all iterations, resulting in the consensus string "**AAAAAAAA**aca**GGGG**" 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 pseudocounts having 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.

RandomizedMotifSearch has the advantage of being able to find longer motifs. In the epilogue, we will see that this feature is important in practice.

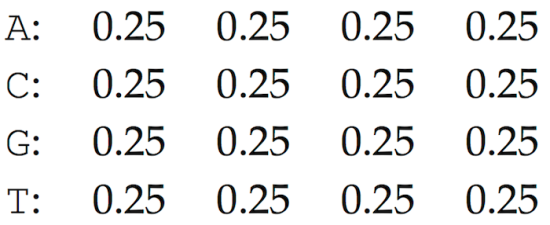


**Figure:** The lowest scoring collection of strings Motifs produced by 100,000 runs of RandomizedMotifSearch, along with their consensus string and score for the Subtle Motif Problem.

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

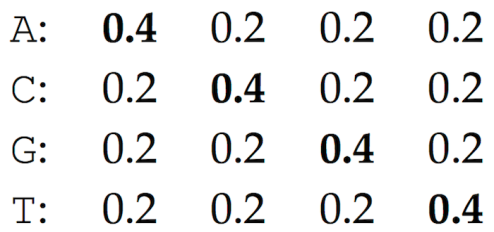


If the strings in Dna were truly random, then we would expect that all nucleotides in the selected k-mers would be equally likely, resulting in an expected Profile in which every entry is approximately 0.25:



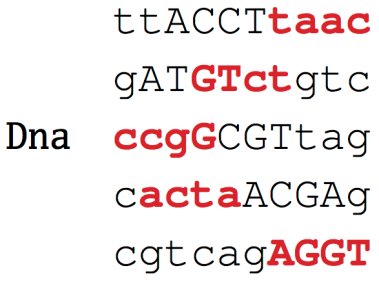
Such a **uniform profile** is essentially useless for motif finding because no string is more probable than any other according to this profile and because it does not provide any clues on what an implanted motif looks like.

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 matrix somewhere in between these two extremes, such as the following:



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).

f 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 direct the profile matrix away from the uniform profile and toward the implanted motif. For example, consider again the original randomly selected k-mers Motifs (shown in red):



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. Note that only completely captured motifs (like "**AGGT**") rather than partially captured motifs (like "**GTct**" or "**ccgG**") contribute to the statistical bias in the profile matrix.

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.

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.

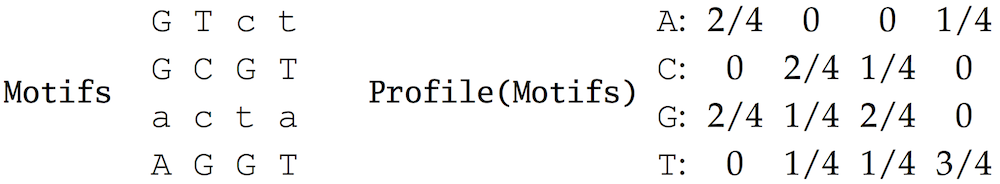
Note that RandomizedMotifSearch may change all t strings in Motifs in a single iteration. This strategy may prove reckless, since some correct motifs (captured in Motifs) may potentially be discarded at the next iteration. GibbsSampler is a more cautious iterative algorithm that discards a single k-mer 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.



Let’s assume that after “rolling the seven-sided die” represented by the function Die(Probabilities), we arrive at the Profile-randomly generated 4-mer GCGT (the fourth 4-mer in the deleted string). 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.

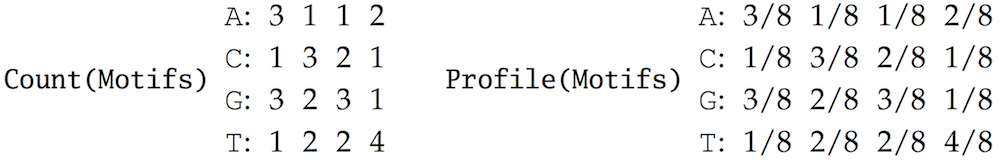


After constructing the motif and profile matrices, we obtain the following:

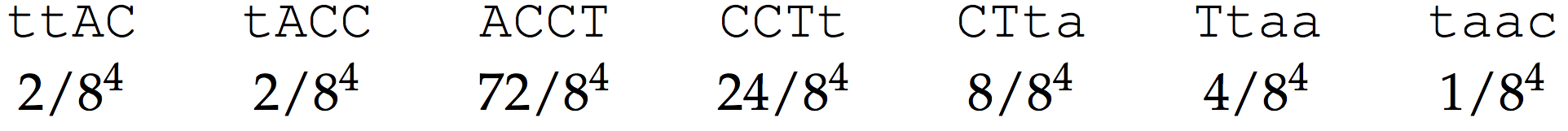


Note that the profile matrix looks more biased toward the implanted motif than the previous profile matrix did.

We update the count and profile matrices with pseudocounts:

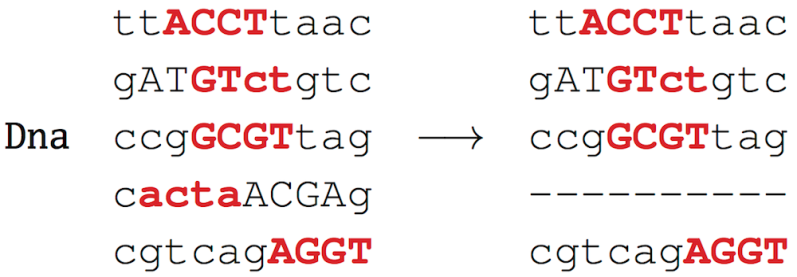


Then, we compute the probabilities of all 4-mers in the deleted string "ttACCTtaac":

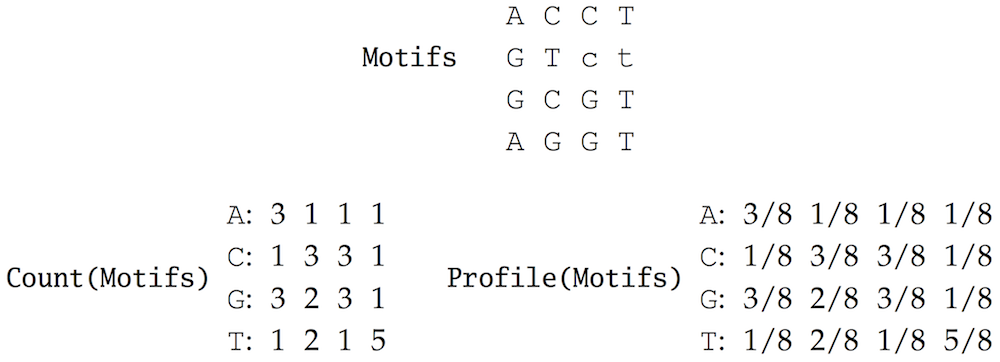


When we rescale these numbers so that they sum to 1 and roll a seven-sided die based on the results, we arrive at the Profile-randomly generated k-mer "**ACCT**", which we add to the collection Motifs.

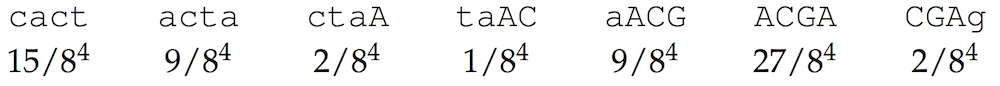
After rolling the five-sided die once again, we randomly select the fourth string for removal.



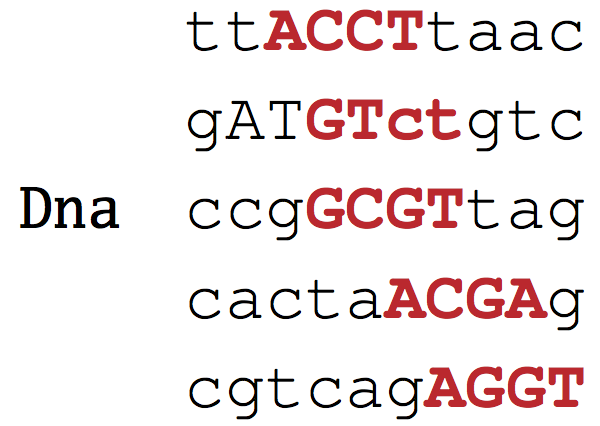
We further add pseudocounts and construct the resulting count and profile matrices:



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



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:



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 "ACGT").

**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. Does this make sense?

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](https://stepik.org/lesson/Detour-Complications-in-Motif-Finding-24210) for some other challenges.