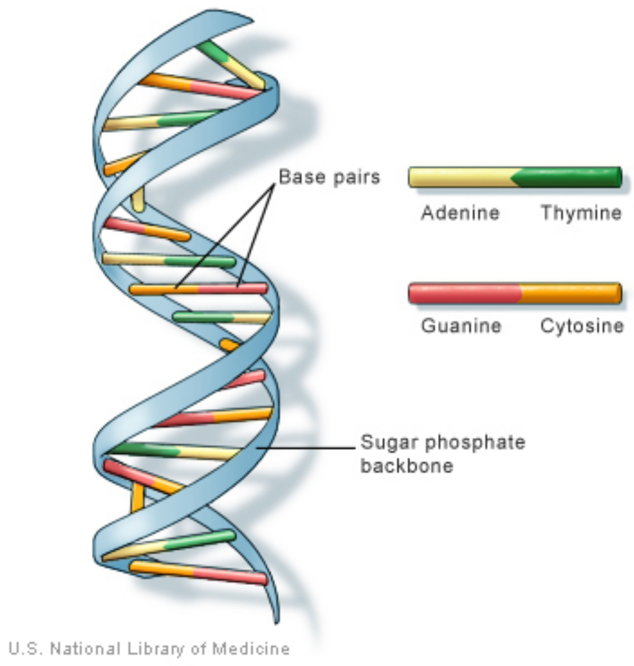
*6.8 COMPUTATIONAL GENOMICS

Every living cell contains molecules of DNA (deoxyribonucleic acid) that encode the genetic blueprint of the organism. Decoding the information contained in DNA, and understanding how it is used in all the processes of life, is an ongoing grand challenge at the center of modern biology and medicine. Comparing the DNA of different organisms also provides insight into evolution and the tree of life. The lengths of DNA molecules and the sheer quantity of DNA data that have been read from living cells and recorded in text files require that biologists use computational methods to answer this challenge.

A genomics primer

As illustrated in Figure 1, DNA is a long double-stranded molecule in the shape of a double helix. Each strand is a chain of smaller units called *nucleotides*. A nucleotide consists of a sugar (deoxyribose), a phosphate group, and one of four nitrogenous bases: adenine (A), thymine (T), cytosine (C), or guanine (G). Each nucleotide in a molecule can be represented by the letter corresponding to the base it contains, and an entire strand can be represented by a string of characters corresponding to its sequence of nucleotides. For example, the following string represents a small DNA molecule consisting of an adenine nucleotide followed by a guanine nucleotide followed by a cytosine nucleotide, etc.:



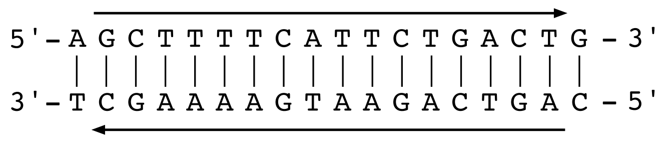
http://ghr.nlm.nih.gov/handbook/basics/dna

Figure 1 An illustration of a DNA double helix.

```
>>> dna = 'agcttttcattctgactg'
```

(The case of the characters is irrelevant; some sequence repositories use lowercase and some use uppercase.) Real DNA sequences are stored in large text files; we will look more closely at these later.

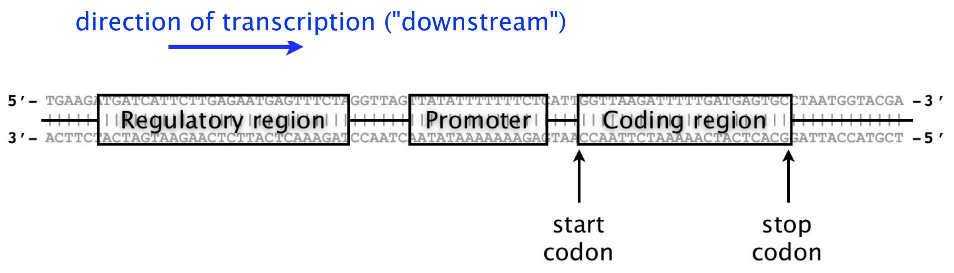
A base on one strand is connected via a hydrogen bond to a complementary base on the other strand. C and G are complements, as are A and T. A base and its connected complement are called a *base pair*. The two strands are "antiparallel" in the sense that they are traversed in opposite directions by the cellular machinery that copies and reads DNA. On each strand, DNA is read in an "upstream-to-downstream" direction, but the "upstream" and "downstream" ends are reversed on the two strands. For reasons that are not relevant here, the "upstream" end is called 5' (read "five prime") and the downstream end is called 3' (read "three prime"). For example, the sequence in the top strand below, read from 5' to 3', is `AGCTT...CTG`, while the bottom strand, called its *reverse complement*, also read 5' to 3', is `CAGTC...GCT`.


5'- A G C T T T T C A T T C T G A C T G -3'
3'- T C G A A A A G T A A G A C T G A C -5'

Reflection 1 *When DNA sequences are stored in text files, only the sequence on one strand is stored. Why?*

RNA (ribonucleic acid) is a similar molecule, but each nucleotide contains a different sugar (ribose instead of deoxyribose), and a base named uracil (U) takes the place of thymine (T). RNA molecules are also single-stranded. As a result, RNA tends to "fold" when complementary bases on the same strand pair with each other. This folded structure forms a unique three-dimensional shape that plays a significant role in the molecule's function.

Some regions of a DNA molecule are called *genes* because they encode genetic information. A gene is *transcribed* by an enzyme called *RNA polymerase* to produce a complementary molecule of RNA. For example, the DNA sequence `5'-GACTGAT-3'` would be transcribed into the RNA sequence `3'-CUGACUA-5'`. If the RNA is a messenger RNA (mRNA), then it contains a *coding region* that will ultimately be used to build a protein. Other RNA products of transcription, called *RNA genes*, are not translated into proteins, and are often instead involved in regulating whether genes are transcribed or translated into proteins. Upstream from the transcribed sequence is a *promoter* region that binds to RNA polymerase to initiate transcription. Upstream from there is often a *regulatory region* that influences whether or not the gene is transcribed.



The coding region of a mRNA is *translated* into a protein by a molecular machine called a *ribosome*. A ribosome reads the coding region in groups of three nucleotides called *codons*.

	U			**C**			**A**			**G**			
U	UUU	Phe	F	UCU	Ser	S	UAU	Tyr	Y	UGU	Cys	C	**U**
	UUC	Phe	F	UCC	Ser	S	UAC	Tyr	Y	UGC	Cys	C	**C**
	UUA	Leu	L	UCA	Ser	S	UAA	Stop	*	UGA	Stop	*	**A**
	UUG	Leu	L	UCG	Ser	S	UAG	Stop	*	UGG	Trp	W	**G**
C	CUU	Leu	L	CCU	Pro	P	CAU	His	H	CGU	Arg	R	**U**
	CUC	Leu	L	CCC	Pro	P	CAC	His	H	CGC	Arg	R	**C**
	CUA	Leu	L	CCA	Pro	P	CAA	Gln	Q	CGA	Arg	R	**A**
	CUG	Leu	L	CCG	Pro	P	CAG	Gln	Q	CGG	Arg	R	**G**
A	AUU	Ile	I	ACU	Thr	T	AAU	Asn	N	AGU	Ser	S	**U**
	AUC	Ile	I	ACC	Thr	T	AAC	Asn	N	AGC	Ser	S	**C**
	AUA	Ile	I	ACA	Thr	T	AAA	Lys	K	AGA	Arg	R	**A**
	AUG	Met	M	ACG	Thr	T	AAG	Lys	K	AGG	Arg	R	**G**
G	GUU	Val	V	GCU	Ala	A	GAU	Asp	D	GGU	Gly	G	**U**
	GUC	Val	V	GCC	Ala	A	GAC	Asp	D	GGC	Gly	G	**C**
	GUA	Val	V	GCA	Ala	A	GAA	Glu	E	GGA	Gly	G	**A**
	GUG	Val	V	GCG	Ala	A	GAG	Glu	E	GGG	Gly	G	**G**

Table 1 The standard genetic code that translates between codons and amino acids. For each codon, both the three letter code and the single letter code are shown for the corresponding amino acid.

In prokaryotes (e.g., bacteria), the coding region begins with a particular *start codon* and ends with one of three *stop codons*. The ribosome translates each mRNA codon between the start and stop codons into an amino acid, according to the *genetic code*, shown in Table 1. It is this sequence of amino acids that comprises the final protein. The genes of eukaryotes (organisms whose cells contain a nucleus) are more complicated, partially because they are interspersed with untranslated regions called *introns* that must be spliced out before translation can occur.

Reflection 2 *What amino acid sequence is represented by the mRNA sequence* `CAU UUU GAG`*?*

Reflection 3 *Notice that, in the genetic code (Table 1), most amino acids are represented by several similar codons. Keeping in mind that nucleotides can mutate over time, what evolutionary advantage might this hold?*

The complete sequence of an organism's DNA is called its *genome*. The size of a genome can range from 10^5 base pairs (bp) in the simplest bacteria to 1.5×10^{11} bp in some plants. The human genome contains about 3.2×10^9 (3.2 billion) bp. Interestingly, the size of an organism's genome does not necessarily correspond to its complexity; plants have some of the largest genomes, far exceeding the size of the human genome.

The subfield of biology that studies the structure and function of genomes is called *genomics*. To better understand a genome, genomicists ask questions such as:

- What is the frequency of each base in the genome? What is the frequency of each codon? For each amino acid, is there a bias toward particular codons?

- Where are the genes and what are their functions?
- How similar are two sequences? Sequence comparison can be used to determine whether two genes have a shared ancestry, called *homology*. Sequence comparison between homologous sequences can also give clues to an unidentified gene's function.
- How are a set of organisms related evolutionarily? We can use sequence comparison to build a *phylogenetic tree* that specifies genetic relations between species.
- Scientists have discovered that mammalian genomes are largely reshuffled collections of similar blocks, called *synteny blocks*. What is the most likely sequence of rearrangement events to have changed one genome into the other?
- What genes are regulated together? Identifying groups of genes that are regulated in the same way can lead to insights into genes' functions, especially those related to disease.

Because genomes are so large, questions like these can only be answered computationally. In the next few pages, we will look at how the methods and techniques from previous sections can be used to answer some of the questions above. We leave many additional examples as exercises. We will begin by working with small sequences; later, we will discuss how to read some longer sequences from files and the web.

Basic DNA analysis

We can use some of the techniques from the previous section to answer questions in the first bullet above. For example, bacterial genomes are sometimes categorized according to their ratio of G and C bases to A and T bases. The fraction of bases that are G or C is called the genome's *GC content*. There are a few ways that we can compute the GC content. First, we could simply use the `count` method:

```
>>> gc = (dna.count('c') + dna.count('g')) / len(dna)
>>> gc
0.3888888888888889
```

Or, we could use a `for` loop like this:

```
def gcContent(dna):
    """Compute the GC content of a DNA sequence.

    Parameter:
        dna: a string representing a DNA sequence

    Return value: the GC content of the DNA sequence
    """

    dna = dna.lower()
    count = 0
    for nt in dna:              # nt is short for "nucleotide"
        if nt in 'cg':
            count = count + 1
    return count / len(dna)
```

Reflection 4 *Why do we convert* `dna` *to lowercase at the beginning of the function?*

Because DNA sequences can be in either upper or lowercase, we need to account for both possibilities when we write a function. But rather than check for both possibilities in our functions, it is easier to just convert the parameter to either upper or lowercase at the beginning.

To gather statistics about the codons in genes, we need to count the number of non-overlapping occurrences of any particular codon. This is very similar to iterating over slices of a text, except that we need to increment the index variable by three in each step:

```python
def countCodon(dna, codon):
    """Find the number of occurrences of a codon in a DNA sequence.

    Parameters:
        dna:   a string representing a DNA sequence
        codon: a string representing a codon

    Return value: the number of occurrences of the codon in dna
    """

    dna = dna.lower()
    codon = codon.lower()
    count = 0
    for index in range(0, len(dna) - 2, 3):
        if dna[index:index + 3] == codon:
            count = count + 1
    return count
```

Reflection 5 *Why do we subtract* 2 *from* `len(dna)` *above?*

Transforming sequences

When DNA sequences are stored in databases, only the sequence of one strand is stored. But genes and other features may exist on either strand, so we need to be able to derive the reverse complement sequence from the original. To first compute the complement of a sequence, we can use a string accumulator, but append the complement of each base in each iteration.

```python
def complement(dna):
    """Return the complement of a DNA sequence.

    Parameter:
        dna: a string representing a DNA sequence

    Return value: the complement of the DNA sequence
    """

    dna = dna.lower()
    compdna = ''
    for nt in dna:
        if nt == 'a':
            compdna = compdna + 't'
        elif nt == 'c':
            compdna = compdna + 'g'
        elif nt == 'g':
            compdna = compdna + 'c'
        else:
            compdna = compdna + 'a'
    return compdna
```

To turn our complement into a reverse complement, we need to write an algorithm to reverse a string of DNA. Let's look at two different ways we might do this. First, we could iterate over the original string in reverse order and append the characters to the new string in that order.

Reflection 6 *How can we iterate over indices in reverse order?*

We can iterate over the indices in reverse order by using a `range` that starts at `len(dna) - 1` and goes down to, but not including, −1 using a step of −1. A function that uses this technique follows.

```
def reverse(dna):
    """Return the reverse of a DNA sequence.

    Parameter:
        dna: a string representing a DNA sequence

    Return value: the reverse of the DNA sequence
    """

    revdna = ''
    for index in range(len(dna) - 1, -1, -1):
        revdna = revdna + dna[index]
    return revdna
```

A more elegant solution simply iterates over the characters in `dna` in the normal order, but *prepends* each character to the `revdna` string.

```
def reverse(dna):
    """ (docstring omitted) """

    revdna = ''
    for nt in dna:
        revdna = nt + revdna
    return revdna
```

Finally, we can combine the `complement` and `reverse` functions to create a function for reverse complement:

```
def reverseComplement(dna):
    """Return the reverse complement of a DNA sequence.

    Parameter:
        dna: a string representing a DNA sequence

    Return value: the reverse complement of the DNA sequence
    """

    return reverse(complement(dna))
```

This function first computes the `complement` of `dna`, then calls `reverse` on the result. We can now, for example, use the `reverseComplement` function to count the frequency of a particular codon on both strands of a DNA sequence:

```
countForward = countCodon(dna, 'atg')
countBackward = countCodon(reverseComplement(dna), 'atg')
```

Comparing sequences

Measuring the similarity between DNA sequences, called *comparative genomics*, has become an important area in modern biology. Comparing the genomes of different species can provide fundamental insights into evolutionary relationships. Biologists can also discover the function of an unknown gene by comparing it to genes with known functions in evolutionarily related species.

Dot plots are used heavily in computational genomics to provide visual representations of how similar large DNA and amino sequences are. Consider the following two sequences. The `dotplot1` function in Section 6.6 would show pairwise matches between the nucleotides in black only.

```
Sequence 1:  agctttgcattctgacag
Sequence 2:  acctttttaattctgtacag
```

But notice that the last four bases in the two sequences are actually the same; if you insert a gap in the first sequence, above the last T in the second sequence, then the number of differing bases drops from eight to four, as illustrated below.

```
Sequence 1:  agctttgcattctg-acag
Sequence 2:  accttttaattctgtacag
```

Evolutionarily, if the DNA of the closest common ancestor of these two species contained a T in the location of the gap, then we interpret the gap as a deletion in the first sequence. Or, if the closest common ancestor did not have this T, then we interpret the gap as an insertion in the second sequence. These insertions and deletions, collectively called *indels*, are common; therefore, sequence comparison algorithms must take them into account. So, just as the first `dotplot1` function did not adequately represent the similarity between texts, neither does it for DNA sequences. Figure 2 shows a complete dot plot, using the `dotplot` function from Section 6.6, for the sequences above.

As we saw in the previous section, dot plots tend to be more useful when we reduce the "noise" by instead comparing subsequences of a given length, say ℓ, within a sliding window. Because there are 4^ℓ different possible subsequences with length ℓ, fewer matches are likely.

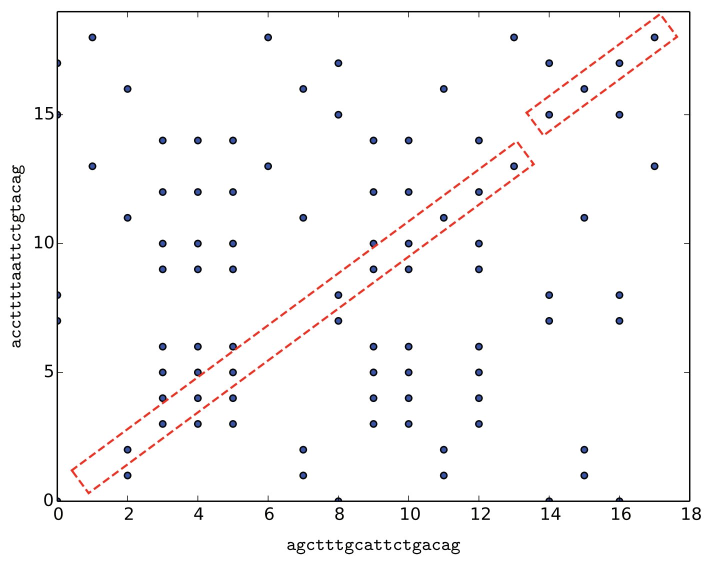


Figure 2 A dot plot comparing individual bases in `agctttgcattctgacag` and `accttttaattctgtacag`. The dots representing the main alignment are highlighted.

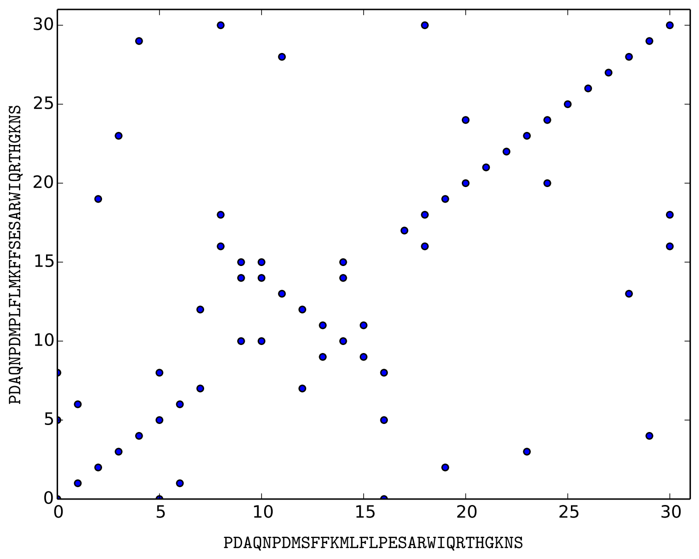


Figure 3 A dot plot comparing two hypothetical short proteins `PDAQNPDMSFFKMLFLPESARWIQRTHGKNS` and `PDAQNPDMPLFLMKFFSESARWIQRTHGKNS`.

Dot plots are also more useful when comparing sequences of amino acids. Since there are 20 different amino acids, we tend to see less noise in the plots. For example, Figure 3 shows a dot plot for the following hypothetical small proteins. Each letter corresponds to a different amino acid. (See Table 1 for the meanings of the letters if you are interested.)

```
seq1 = 'PDAQNPDMSFFKMLFLPESARWIQRTHGKNS'
seq2 = 'PDAQNPDMPLFLMKFFSESARWIQRTHGKNS'
```

Notice how the plot shows an *inversion* in one sequence compared the other, highlighted in red above.

Reading sequence files

DNA sequences are archived and made publicly available in a database managed by the National Center for Biotechnology Information (NCBI), a division of the National Institutes of Health (NIH). Each sequence in the database has an associated identifier known as an *accession number*. For example, the accession number for the first segment of the Burmese python (*Python molurus bivittatus*) genome is `AEQU02000001`. You can find this data by visiting the NCBI website at `http://www.ncbi.nlm.nih.gov`, entering the accession number in the search box, and selecting "Nucleotide" from the drop-down menu.

Genomic data is commonly stored in a format called *FASTA* [47]. A FASTA file begins with a header preceded by a greater-than sign (>), followed by several lines of sequence data. Each line of sequence is terminated with a newline character. For example, the FASTA file for the first segment of the Burmese python genome looks like this:

```
>gi|541020484|gb|AEQU02000001.1| Python bivittatus ...
AGGCCTGGCGCAATATGGTTCATGGGGTCACGAAGAGTCGGACACGACTTAACGACTAAACAACAATTAA
TTCTAAACCCTAGCTATCTGGGTTGCTTCCCTACATTATATTTCCGTGAATAAATCTCACTAAACTCAGA
AGGATTTACTTTTGAGTGAACACTCATGATCAAGAACTCAGAAAACTGAAAGGATTCGGGAATAGAAGTG
TTATTTCAGCCCTTTCCAATCTTCTAAAAAGCTGACAGTAATACCTTTTTTTGTTTTAAAAATTTAAAAA
GAGCACTCACATTTATGCTACAGAATGAGTCTGTAACAGGGAAGCCAGAAGGGAAAAAGACTGAAACGTA
AACCAAGGACATAATGCAGGGATCATAATTTCTAGTTGGCAAGACTAAAGTCTATACATGTTTTATTGAA
AAACCACTGCTATTTGCTCTTACAAGAACTTATCCCTCCAGAAAAAGTAGCACTGGCTTGATACTGCCTG
CTCTAACTCAACAGTTTCAGGAAGTGCTGGCAGCCCCTCCTAACACCATCTCAGCTAGCAAGACCAAGTT
GTGCAAGTCCACTCTATCTCATAACAAAACCTCTTTAATTAAACTTGAAAGCCTAGAGCTTTTTTTACCT
TGGTTCTTCAAGTCTTCGTTTGTACAATATGGGCCAATGTCATAGCTTGGTGTATTAGTATTATTATTGT
TACTGTTGGTTACACAGTCAGTCAGATGTTTAGACTGGATTTGACATTTTTACCCATGTATCGAGTCCTT
CCCAAGGACCTGGGATAGGCAGATGTTGGTGTTTGATGGTGTTAAAGGT
```

To use this DNA sequence in a program, we need to extract the unadorned sequence into a string, removing the header and newline characters. A function to do this is very similar to those in Section 6.2.

```
def readFASTA(fileName):
    """Read a FASTA file containing a single sequence and return
       the sequence as a string.

    Parameter:
        fileName: a string representing the name of a FASTA file

    Return value: a string containing the sequence in the FASTA file
    """
    fastaFile = open(fileName, 'r', encoding = 'utf-8')
    header = fastaFile.readline()
    dna = ''
    for line in fastaFile:
        dna = dna + line[:-1]
    fastaFile.close()
    return dna
```

After opening the FASTA file, we read the header line with the `readline` function. We will not need this header, so this `readline` just serves to move the file pointer past the first line, to the start of the actual sequence. To read the rest of the file, we iterate over it, line by line. In each iteration, we append the line, without the newline character at the end, to a growing string named `dna`. There is a link on the book website to the FASTA file above so that you can try out this function.

We can also directly access FASTA files by using a URL that sends a query to NCBI. The URL below submits a request to the NCBI website to download the FASTA file for the first segment of the Burmese python genome (with accession number `AEQU02000001`).

```
http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nuccore
   &id=AEQU02000001&rettype=fasta&retmode=text
```

To retrieve a different sequence, we would just need to insert a different accession number (in red). The following function puts all of the pieces together.

```
import urllib.request as web

def getFASTA(id):
    """Fetch the DNA sequence with the given id from NCBI and return
       it as a string.

    Parameter:
        id: the identifier of a DNA sequence

    Return value: a string representing the sequence with the given id
    """

    prefix1 = 'http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi'
    prefix2 = '?db=nuccore&id='
    suffix = '&rettype=fasta&retmode=text'
    url = prefix1 + prefix2 + id + suffix
    readFile = web.urlopen(url)
    header = readFile.readline()
    dna = ''
    for line in readFile:
        line = line[:-1]
        dna = dna + line.decode('utf-8')
    readFile.close()
    return dna
```

The function first creates a string containing the URL for the accession number passed in parameter `id`. The first common parts of the URL are assigned to the `prefix` variables, and the last part of the URL is assigned to `suffix`. We construct the custom URL by concatenating these pieces with the accession number. We then open the URL and use the code from the previous `readFASTA` function to extract the DNA sequence and return it as a string.

Reflection 7 *Use the `getFASTA` function to read the first segment of the Burmese python genome (accession number `AEQU02000001`).*

We have just barely touched on the growing field of computational genomics. The following exercises explore some additional problems and provide many opportunities to practice your algorithm design and programming skills.

Exercises

6.8.1* Write a function

```
countACG(dna)
```

that returns the fraction of nucleotides in the given DNA sequence that are not T. Use a `for` loop that iterates over the characters in the string.

6.8.2. Repeat the previous exercise, but use a `for` loop that iterates over the indices of the string instead.

6.8.3* Write a function

```
printCodons(dna)
```

that prints the codons, starting at the left end, in the string `dna`. For example, `printCodons('ggtacactgt')` would print

```
ggt
aca
ctg
```

6.8.4. Write a function

```
findCodon(dna, codon)
```

that returns the index of the first occurrence of the given `codon` in the string `dna`. Unlike in the `countCodon` function, increment the loop index by 1 instead of 3, so you can find the codon at any position.

6.8.5. Write a function

```
findATG(dna)
```

that builds up a list of indices of *all* of the positions of the codon `ATG` in the given DNA string. Do not use the built-in `index` method or `find` method. Constructing the list of indices is very similar to what we have done several times in building lists to plot.

6.8.6. Since codons consist of three bases each, transcription can occur in one of three independent *reading frames* in any sequence of DNA by starting at offsets 0, 1, and 2 from the left end. For example, if our DNA sequence was `ggtacactgtcat`, the codons in the three reading frames are:

```
RF 1:  ggt aca ctg tca t
RF 2:  g gta cac tgt cat
RF 3:  gg tac act gtc at
```

Write a function

```
printCodonsAll(dna)
```

that uses your function from Exercise 6.8.3 to print the codons in all three reading frames.

6.8.7. Most vertebrates have much lower density of CG pairs (called *CG dinucleotides*) than would be expected by chance. However, they often have relatively high concentrations upstream from genes (coding regions). For this reason, finding these so-called “CpG islands” is often a good way to find putative sites of genes. (The “p” between C and G denotes the phosphodiester bond between the C and G, versus a hydrogen bond across two complementary strands.) Without using the built-in `count` method, write a function

```
CpG(dna)
```

that returns the fraction of dinucleotides that are `cg`. For example, if `dna` were `atcgttcg`, then the function should return 0.5 because half of the sequence is composed of CG dinucleotides.

6.8.8* A ***microsatellite*** or *simple sequence repeat (SSR)* is a contiguous repeat of a short sequence of DNA. The repeated sequence is typically 2–4 bases in length and can be repeated 10–100 times or more. For example, `cacacacaca` is a short SSR of `ca`, a very common repeat in humans. Microsatellites are very common in the DNA of most organisms, but their lengths are highly variable within populations because of replication errors resulting from “slippage” during copying. Comparing the distribution of length variants within and between populations can be used to determine genetic relationships and learn about evolutionary processes.

Write a function

```
ssr(dna, repeat)
```

that returns the length (number of repeats) of the first SSR in `dna` that repeats the parameter `repeat`. If `repeat` is not found, return 0. Use the `find` method of the string class to find the first instance of `repeat` in `dna`.

6.8.9. Write another version of the `ssr` function that finds the length of the *longest* SSR in `dna` that repeats the parameter `repeat`. Your function should repeatedly call the `ssr` function in the previous problem. You will probably want to use a `while` loop.

6.8.10. Write a third version of the `ssr` function that uses the function in the previous problem to find the longest SSR of *any dinucleotide* (sequence with length 2) in `dna`. Your function should return the longest repeated dinucleotide.

6.8.11. Write a function

```
palindrome(dna)
```

that returns `True` if `dna` is the same as its reverse complement, and `False` otherwise. (Note that this is different from the standard definition of palindrome.) For example, `gaattc` is a palindrome because it and its reverse complement are the same. These sequences turn out to be very important because certain restriction enzymes target specific palindromic sequences and cut the DNA molecule at that location. For example, the EcoR1 restriction enzyme cuts DNA of the bacteria *Escherichia coli* at the sequence `gaattc` in the following way:

```
5'--gaattc--3'             5'--g    \ aattc--3'
    ||||||          ⇒          |      \    |
3'--cttaag--5'             3'--cttaa  \   g--5'
```

6.8.12* Write a function

```
dna2rna(dna)
```

that returns returns a copy of `dna` with every T replaced by a U.

6.8.13. Write a function

```
transcribe(dna)
```

that returns the RNA equivalent of the reverse complement of `dna`. (Utilize previously written functions to accomplish this in one line.)

6.8.14. Write a function

```
clean(dna)
```

that returns a new DNA string in which every character that is not an A, C, G, or T is replaced with an N. For example, `clean('goat')` should return the string `'gnat'`.

6.8.15. When DNA molecules are sequenced, there are often ambiguities that arise in the process. To handle these situations, there are also "ambiguous symbols" defined that code for one of a set of bases.

Symbol	Possible Bases	Symbol	Possible Bases
R	A G	B	C G T
Y	C T	D	A G T
K	G T	H	A C T
M	A C	V	A C G
S	C G	N	A C G T
W	A T		

Write a function

```
fix(dna)
```

that returns a DNA string in which each ambiguous symbol is replaced with one of the possible bases it represents, each with equal probability. For example, if an `R` exists in `dna`, it should be replaced with either an `A` with probability 1/2 or a `G` with probability 1/2.

6.8.16. Write a function

```
mark(dna)
```

that returns a new DNA string in which every start codon ATG is replaced with >>>, and every stop codon (TAA, TAG, or TGA) is replaced with <<<. Your loop should increment by 3 in each iteration so that you are only considering non-overlapping codons. For example, `mark('ttgatggagcattagaag')` should return the string `'ttg>>>gagcat<<<aag'`.

6.8.17. The accession number for the hepatitis A virus is `NC_001489`. Write a program that uses the `getFASTA` function to get the DNA sequence for hepatitis A, and then the `gcContent` function from this section to find the GC content of hepatitis A.

6.8.18. The DNA of the hepatitis C virus encodes a single long protein that is eventually processed by enzymes called proteases to produce ten smaller proteins. There are seven different types of the virus. The accession numbers for the proteins produced by type 1 and type 2 are `NP_671491` and `YP_001469630`. Write a program that uses the `getFASTA` function and your dot plot function from

Exercise 6.6.14 to produce a dot plot comparing these two proteins using a window of size 4.

6.8.19. *Hox* genes control key aspects of embryonic development. Early in development, the embryo consists of several segments that will eventually become the main axis of the head-to-tail body plan. The *Hox* genes dictate what body part each segment will become. One particular *Hox* gene, called *Hox A1*, seems to control development of the hindbrain. Write a program that uses the `getFASTA` function and your dot plot function from Exercise 6.6.14 to produce a dot plot comparing the human and mouse *Hox A1* genes, using a window of size 8. The human *Hox A1* gene has accession number `U10421.1` and the mouse *Hox A1* gene has accession number `NM_010449.4`.

Selected Exercise Solutions

6.8.1
```
def countACG(dna):
    dna = dna.lower()
    count = 0
    for nt in dna:
        if nt != 't':
            count = count + 1
    return count / len(dna)
```

6.8.3
```
def printCodons(dna):
    for index in range(0, len(dna) - 2, 3):
        print(dna[index:index+3])
```

6.8.8
```
def ssr(dna, repeat):
    start = dna.find(repeat)
    if start < 0:
        return 0
    count = 0
    for index in range(start, len(dna), len(repeat)):
        if dna[index : index + len(repeat)] == repeat:
            count = count + 1
        else:
            return count

# OR

def ssr(dna, repeat):
    start = dna.find(repeat)
    if start < 0:
        return 0
    count = 0
    index = start
    while index < len(dna) and dna[index:index + len(repeat)] == repeat:
        count = count + 1
        index = index + len(repeat)
    return count
```

6.8.12
```
def dna2rna(dna):
    dna = dna.lower()
    rna = ''
    for nt in dna:
        if nt == 't':
            rna = rna + 'u'
        else:
            rna = rna + nt
    return rna
```