2016 CS696 Programming Problems In Bioinformatics

Problem 1

Installing Python

Why Python?

Rosalind problems can be solved using any programming language. Our language of choice is **Python**. Why? Because it's simple, powerful, and even funny. You'll see what we mean.

If you don't already have Python software, please download and install the appropriate version for your platform (Windows, Linux or Mac OS X). Please install Python of version 2.x (not 3.x) — it has more libraries support and many well-written guides.

After completing installation, launch **IDLE** (default Python development environment; it's usually installed with Python, however you may need to install it separately on Linux).

You'll see a window containing three arrows, like so:

```
>>>
```

The three arrows are Python's way of saying that it is ready to serve your every need. You are in interactive mode, meaning that any command you type will run immediately. Try typing `1+1` and see what happens.

Of course, to become a Rosalind pro, you will need to write programs having more than one line. So select **File → New Window** from the IDLE menu. You can now type code as you would into a text editor. For example, type the following:

```
print "Hello, World!"
```

Select **File → Save** to save your creation with an appropriate name (e.g., `hello.py`).

To run your program, select **Run → Run Module**. You'll see the result in the interactive mode window (Python Shell).

Congratulations! You just ran your first program in Python!

Problem

After downloading and installing Python, type `import this` into the Python command line and see what happens. Then, click the "Download dataset" button below and copy the Zen of Python into the space provided.

Problem 2

Introduction to the Bioinformatics Armory

Let's Be Practical

If you are an accomplished coder, then you can write a separate program for every new task you encounter. In practice, these programs only need to be written once and posted to the web, where those of us who are not great coders can use them quickly and efficiently. In the Armory, we will familiarize ourselves with a sampling of some of the more popular bioinformatics tools taken from "out of the box" software.

To be equitable, we will focus mainly on free, internet-based software and on programs that are compatible with multiple operating systems. The "Problem" section will contain links to this software, with short descriptions about how to use it.

Problem

This initial problem is aimed at familiarizing you with Rosalind's task-solving pipeline. To solve it, you merely have to take a given DNA sequence and find its nucleotide counts; this problem is equivalent to "Counting DNA Nucleotides" in the Stronghold.

Of the many tools for DNA sequence analysis, one of the most popular is the Sequence Manipulation Suite. Commonly known as SMS 2, it comprises a collection of programs for generating, formatting, and analyzing short strands of DNA and polypeptides.

One of the simplest SMS 2 programs, called `DNA stats`, counts the number of occurrences of each nucleotide in a given strand of DNA. An online interface for `DNA stats` can be found here.

Given: A DNA string s of length at most 1000 bp.

Return: Four integers (separated by spaces) representing the respective number of times that the symbols 'A', 'C', 'G', and 'T' occur in s. **Note**: You must provide your answer in the format shown in the sample output below.

Sample Dataset

```
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
```

Sample Output

```
20 12 17 21
```

Programming Shortcut

Our default choice for existing functions and modules to analyze biological data is **BioPython**, a set of freely available tools for computational biology that are written in Python. We will give you tips on how to solve certain problems (like this one) using BioPython functions and methods.

Detailed installation instructions for BioPython are available in PDF and HTML formats.

BioPython offers a specific data structure called `Seq` for representing sequences. `Seq` represents an extension of the "str" (string) object type that is built into Python by supporting additional biologically relevant

methods like `translate()` and `reverse_complement()`.

In this problem, you can easily use the built-in Python method `.count()` for strings. Here's how you could count the occurrences of 'A' found in a `Seq` object.

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq.count("A")
3
```

Problem 3

Counting DNA Nucleotides

A Rapid Introduction to Molecular Biology

Making up all living material, the **cell** is considered to be the building block of life. The **nucleus**, a component of most **eukaryotic** cells, was identified as the hub of cellular activity 150 years ago. Viewed under a light microscope, the nucleus appears only as a darker region of the cell, but as we increase magnification, we find that the nucleus is densely filled with a stew of macromolecules called **chromatin**. During **mitosis** (eukaryotic cell division), most of the chromatin condenses into long, thin strings called **chromosomes**. See **Figure 1** for a figure of cells in different stages of mitosis.

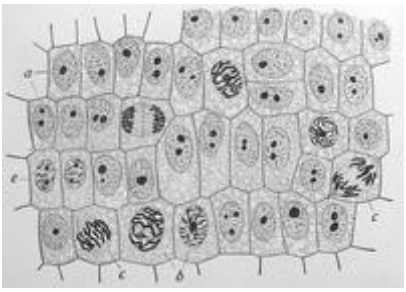

Figure 1. A 1900 drawing by Edmund Wilson of onion cells at different stages of mitosis. The sample has been dyed, causing chromatin in the cells (which soaks up the dye) to appear in greater contrast to the rest of the cell.

One class of the macromolecules contained in chromatin are called **nucleic acids**. Early 20th century research into the chemical identity of nucleic acids culminated with the conclusion that nucleic acids are **polymers**, or repeating chains of smaller, similarly structured molecules known as **monomers**. Because of their tendency to be long and thin, nucleic acid polymers are commonly called **strands**.

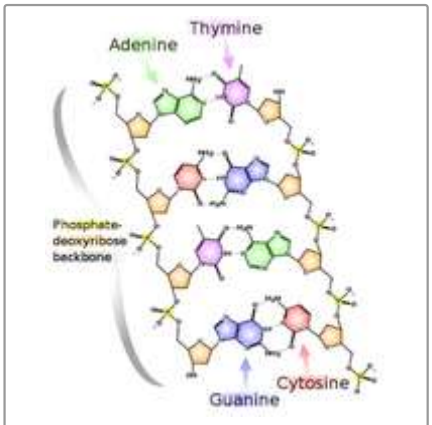

Figure 2. A sketch of DNA's primary structure.

The nucleic acid monomer is called a **nucleotide** and is used as a unit of strand length (abbreviated to nt). Each nucleotide is formed of three parts: a **sugar** molecule, a negatively charged **ion** called a **phosphate**, and a compound called a **nucleobase** ("base" for short). Polymerization is achieved as the sugar of one nucleotide bonds to the phosphate of the next nucleotide in the chain, which forms a **sugar-phosphate backbone** for the nucleic acid strand. A key point is that the nucleotides of a specific type of nucleic acid always contain the same sugar and phosphate molecules, and they differ only in their choice of base. Thus, one strand of a nucleic acid can be differentiated from another based solely on the *order* of its bases; this ordering of bases defines a nucleic acid's **primary structure**.

For example, **Figure 2** shows a strand of **deoxyribose nucleic acid** (DNA), in which the sugar is called **deoxyribose**, and the only four choices for nucleobases are molecules called **adenine** (A), **cytosine** (C),

> **guanine** (G), and **thymine** (T).
>
> For reasons we will soon see, DNA is found in all living organisms on Earth, including bacteria; it is even found in many viruses (which are often considered to be nonliving). Because of its importance, we reserve the term **genome** to refer to the sum total of the DNA contained in an organism's chromosomes.

Problem

A **string** is simply an ordered collection of symbols selected from some **alphabet** and formed into a word; the **length** of a string is the number of symbols that it contains.

An example of a length 21 **DNA string** (whose alphabet contains the symbols 'A', 'C', 'G', and 'T') is "ATGCTTCAGAAAGGTCTTACG."

Given: A DNA string s of length at most 1000 nt.

Return: Four integers (separated by spaces) counting the respective number of times that the symbols 'A', 'C', 'G', and 'T' occur in s.

Sample Dataset

```
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
```

Sample Output

```
20 12 17 21
```

Problem 4

Find Patterns Forming Clumps in a String

Given integers *L* and *t*, a string *Pattern* forms an **(*L*, *t*)-clump** inside a (larger) string *Genome* if there is an interval of *Genome* of length *L* in which *Pattern* appears at least *t* times. For example, TGCA forms a (25,3)-clump in the following *Genome*: gatcagcataagggtccc**TGCA**A**TGCA**TGACAAGCC**TGCA**gttgttttac.

Clump Finding Problem

Find patterns forming clumps in a string.

Given: A string *Genome*, and integers *k*, *L*, and *t*.

Return: All distinct *k*-mers forming (*L*, *t*)-clumps in *Genome*.

Sample Dataset

```
CGGACTCGACAGATGTGAAGAAATGTGAAGACTGAGTGAAGAGAAGAGGAAACACGACACGACATTGCGACATAATGTACGAA
TGTAATGTGCCTATGGC
```

```
5 75 4
```

Sample Output

```
CGACA GAAGA AATGT
```

Extra Datasets

Click for an extra dataset

Click for debug datasets

Problem 5

Complementing a Strand of DNA

The Secondary and Tertiary Structures of DNA

In "Counting DNA Nucleotides", we introduced nucleic acids, and we saw that the primary structure of a nucleic acid is determined by the ordering of its nucleobases along the sugar-phosphate backbone that constitutes the bonds of the nucleic acid polymer. Yet primary structure tells us nothing about the larger, 3-dimensional shape of the molecule, which is vital for a complete understanding of nucleic acids.

The search for a complete chemical structure of nucleic acids was central to molecular biology research in the mid-20th Century, culminating in 1953 with a publication in *Nature* of fewer than 800 words by James Watson and Francis Crick. Consolidating a high resolution X-ray image created by Rosalind Franklin and Raymond Gosling with a number of established chemical results, Watson and Crick proposed the following structure for DNA:

1. The DNA molecule is made up of two strands, running in opposite directions.
2. Each base bonds to a base in the opposite strand. Adenine always bonds with thymine, and cytosine always bonds with guanine; the **complement** of a base is the base to which it always bonds; see **Figure 1**.
3. The two strands are twisted together into a long spiral staircase structure called a **double helix**; see **Figure 2**.

Because they dictate how bases from different strands interact with each other, (1) and (2) above compose the **secondary structure** of DNA. (3) describes the 3-dimensional shape of the DNA molecule, or its **tertiary structure**.

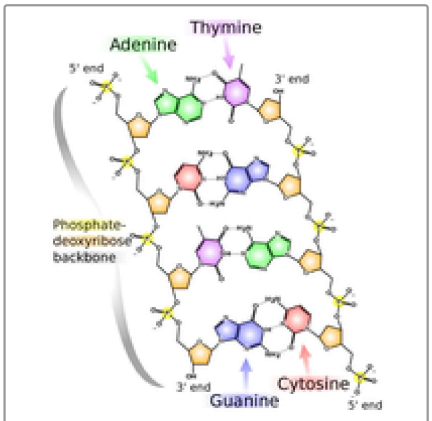

Figure 1. Base pairing across the two strands of DNA.

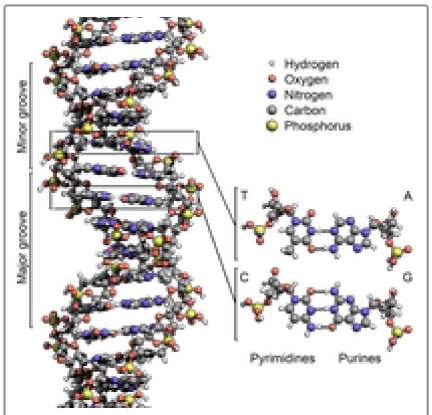

Figure 2. The double helix of DNA on the molecular scale.

In light of Watson and Crick's model, the bonding of two complementary bases is called a **base pair** (bp). Therefore, the length of a DNA molecule will commonly be given in bp instead of nt. By complementarity, once we know the order of bases on one strand, we can immediately deduce the sequence of bases in the complementary strand. These bases will run in the opposite order to match the fact that the two strands of DNA run in opposite directions.

Problem

In DNA strings, symbols 'A' and 'T' are complements of each other, as are 'C' and 'G'.

The **reverse complement** of a DNA string s is the string s^c formed by reversing the symbols of s, then taking the complement of each symbol (e.g., the reverse complement of "GTCA" is "TGAC").

Given: A DNA string s of length at most 1000 bp.

Return: The reverse complement s^c of s.

Sample Dataset

```
AAAACCCGGT
```

Sample Output

```
ACCGGGTTTT
```

Problem 6

Computing GC Content

Identifying Unknown DNA Quickly

A quick method used by early computer software to determine the language of a given piece of text was to analyze the frequency with which each letter appeared in the text. This strategy was used because each language tends to exhibit its own letter frequencies, and as long as the text under consideration is long enough, software will correctly recognize the language quickly and with a very low error rate. See **Figure 1** for a table compiling English letter frequencies.

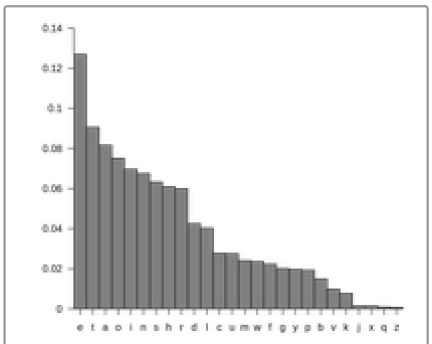

Figure 1. The table above was computed from a large number of English words and shows for any letter the frequency with which it appears in those words. These frequencies can be used to reliably identify a piece of English text and differentiate it

You may ask: what in the world does this linguistic problem have to do with biology? Although two members of the same species will have different genomes, they still share the vast percentage of their DNA; notably, 99.9% of the 3.2 billion base pairs in a human genome are common to almost all humans (i.e., excluding people having major genetic defects). For this reason, biologists will speak of *the* human

genome, meaning an average-case genome derived from a collection of individuals. Such an average case genome can be assembled for any species, a challenge that we will soon discuss.

from that of another language. Taken from http://en.wikipedia.org/wiki/File:English_lette

The biological analog of identifying unknown text arises when researchers encounter a molecule of DNA deriving from an unknown species. Because of the base pairing relations of the two DNA strands, cytosine and guanine will always appear in equal amounts in a double-stranded DNA molecule. Thus, to analyze the symbol frequencies of DNA for comparison against a database, we compute the molecule's **GC-content**, or the percentage of its bases that are *either* cytosine or guanine.

In practice, the GC-content of most eukaryotic genomes hovers around 50%. However, because genomes are so long, we may be able to distinguish species based on very small discrepancies in GC-content; furthermore, most prokaryotes have a GC-content significantly higher than 50%, so that GC-content can be used to quickly differentiate many prokaryotes and eukaryotes by using relatively small DNA samples.

Problem

The GC-content of a DNA string is given by the percentage of symbols in the string that are 'C' or 'G'. For example, the GC-content of "AGCTATAG" is 37.5%. Note that the reverse complement of any DNA string has the same GC-content.

DNA strings must be labeled when they are consolidated into a database. A commonly used method of string labeling is called **FASTA format**. In this format, the string is introduced by a line that begins with '>', followed by some labeling information. Subsequent lines contain the string itself; the first line to begin with '>' indicates the label of the next string.

In Rosalind's implementation, a string in FASTA format will be labeled by the ID "Rosalind_xxxx", where "xxxx" denotes a four-digit code between 0000 and 9999.

Given: At most 10 DNA strings in FASTA format (of length at most 1 kbp each).

Return: The ID of the string having the highest GC-content, followed by the GC-content of that string. Rosalind allows for a default error of 0.001 in all decimal answers unless otherwise stated; please see the note on **absolute error** below.

Sample Dataset

```
>Rosalind_6404
CCTGCGGAAGATCGGCACTAGAATAGCCAGAACCGTTTCTCTGAGGCTTCCGGCCTTCCC
TCCCACTAATAATTCTGAGG
>Rosalind_5959
CCATCGGTAGCGCATCCTTAGTCCAATTAAGTCCCTATCCAGGCGCTCCGCCGAAGGTCT
ATATCCATTTGTCAGCAGACACGC
>Rosalind_0808
CCACCCTCGTGGTATGGCTAGGCATTCAGGAACCGGAGAACGCTTCAGACCAGCCCGGAC
TGGGAACCTGCGGGCAGTAGGTGGAAT
```

Sample Output

```
Rosalind_0808
60.919540
```

Note on Absolute Error

We say that a number x is within an absolute error of y to a correct solution if x is within y of the correct solution. For example, if an exact solution is 6.157892, then for x to be within an absolute error of 0.001, we must have that $|x - 6.157892| < 0.001$, or $6.156892 < x < 6.158892$

Error bounding is a vital practical tool because of the inherent round-off error in representing decimals in a computer, where only a finite number of decimal places are allotted to any number. After being compounded over a number of operations, this round-off error can become evident. As a result, rather than testing whether two numbers are equal with $x = z$, you may wish to simply verify that $|x - z|$ is very small.

The mathematical field of **numerical analysis** is devoted to rigorously studying the nature of computational approximation.

Problem 7

Protein Translation

The Genetic Code

Given a nucleotide sequence obtained from sequencing or a database, we want to know whether this sequence corresponds to a coding region of the genome. If so, you need only apply the genetic code to translate your sequence into an amino acid chain.

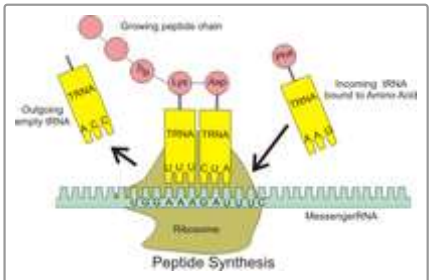

Figure 1. Schematic image of the translation process.

The apparent difficulty of translation is that somehow 4 RNA bases must correspond to a protein language of 20 amino acids; in order for every possible amino acid to be used, we must translate 3-nucleotide codons into amino acids (see **Figure 1**). Note that there are $4^3 = 64$ possible codons, so that multiple codons may encode the same amino acid. Two special types of codons are the **start codon** (AUG), which codes for the amino acid methionine and always indicates the start of translation, and the three **stop codons** (UAA, UAG, UGA), which do not code for an amino acid and terminate the translation process.

It is important to note that some organisms and DNA-containing organelles use an alternative form of the genetic code. This phenomenon is called **genetic code variation**. For example, vertebrate mitochondria treat AGA and AGG as stop codons instead of having these two codons code for arginine.

Thus, it is important to check the source of your genome data prior to translation.

Problem

The 20 commonly occurring amino acids are abbreviated by using 20 letters from the English alphabet (all letters except for B, J, O, U, X, and Z). **Protein strings** are constructed from these 20 symbols. The **RNA codon table** shows the encoding from each RNA codon to the amino acid alphabet.

The `Translate` tool from the SMS 2 package can be found here in the SMS 2 package

A detailed list of genetic code variants (codon tables) along with indexes representing these codes (1 = standard genetic code, etc.) can be obtained here.

For now, when translating DNA and RNA strings, we will start with the first letter of the string and ignore stop codons.

Given: A DNA string s of length at most 10 kbp, and a protein string translated by s.

Return: The index of the genetic code variant that was used for translation. (If multiple solutions exist, you may return any one.)

Sample Dataset

```
ATGGCCATGGCGCCCAGAACTGAGATCAATAGTACCCGTATTAACGGGTGA
MAMAPRTEINSTRING
```

Sample Output

```
1
```

Programming Shortcut

BioPython possesses a `translate()` method for converting RNA strings to protein strings:

`translate(sequence, table='Standard', stop_symbol='*', to_stop=False)`

The `translate()` method has the following parameters:

`sequence` : the DNA or RNA string to translate. `table` : the codon table to use. This can be either a name (string) or NCBI identifier (integer). Defaults to the "Standard" table, which has a value of 1. `stop_symbol` : a single symbol used to mark any terminators, which defaults to the asterisk "*". `to_stop` : a Boolean value. If `True`, translation is terminated at the first stop codon appearing in the frame; defaults to `False`.

Here are some examples of `translate()` in action:

```
>>> from Bio.Seq import translate
>>> coding_dna = "GTGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG"
>>> translate(coding_dna)
'VAIVMGR*KGAR*'
>>> translate(coding_dna, stop_symbol="@")
'VAIVMGR@KGAR@'
>>> translate(coding_dna, to_stop=True)
'VAIVMGR'
>>> translate(coding_dna, table=2)
'VAIVMGRWKGAR*'
>>> translate(coding_dna, table=2, to_stop=True)
'VAIVMGRWKGAR'
```

Problem 8

Transcribing DNA into RNA

The Second Nucleic Acid

In "Counting DNA Nucleotides", we described the primary structure of a nucleic acid as a polymer of nucleotide units, and we mentioned that the omnipresent nucleic acid DNA is composed of a varied sequence of four bases.

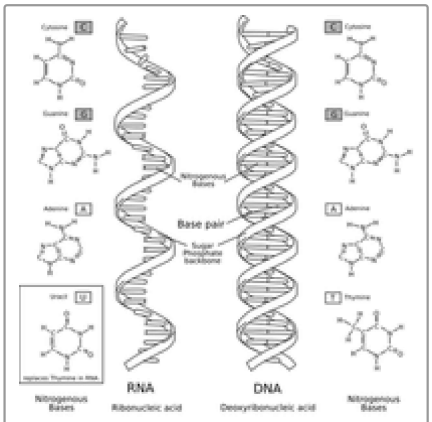

Figure 1. Structural differences between RNA and DNA

Yet a second nucleic acid exists alongside DNA in the chromatin; this molecule, which possesses a different sugar called **ribose**, came to be known as **ribose nucleic acid**, or RNA. RNA differs further from DNA in that it contains a base called **uracil** in place of thymine; structural differences between DNA and RNA are shown in **Figure 1**. Biologists initially believed that RNA was only contained in plant cells, whereas DNA was restricted to animal cells. However, this hypothesis dissipated as improved chemical methods discovered both nucleic acids in the cells of all life forms on Earth.

The primary structure of DNA and RNA is so similar because the former serves as a blueprint for the creation of a special kind of RNA molecule called **messenger RNA**, or mRNA. mRNA is created during **RNA transcription**, during which a strand of DNA is used as a template for constructing a strand of RNA by copying nucleotides one at a time, where uracil is used in place of thymine.

In eukaryotes, DNA remains in the nucleus, while RNA can enter the far reaches of the cell to carry out DNA's instructions. In future problems, we will examine the process and ramifications of RNA transcription in more detail.

Problem

An **RNA string** is a string formed from the alphabet containing 'A', 'C', 'G', and 'U'.

Given a DNA string t corresponding to a coding strand, its transcribed RNA string u is formed by replacing all occurrences of 'T' in t with 'U' in u.

Given: A DNA string t having length at most 1000 nt.

Return: The transcribed RNA string of t.

Sample Dataset

```
GATGGAACTTGACTACGTAAATT
```

Sample Output

```
GAUGGAACUUGACUACGUAAAUU
```

Problem 9

Translating RNA into Protein

The Genetic Code

Just as nucleic acids are polymers of nucleotides, **proteins** are chains of smaller molecules called **amino acids**; 20 amino acids commonly appear in every species. Just as the primary structure of a nucleic acid is given by the order of its nucleotides, the **primary structure** of a protein is the order of its amino acids. Some proteins are composed of several subchains called **polypeptides**, while others are formed of a single polypeptide; see **Figure 1**.

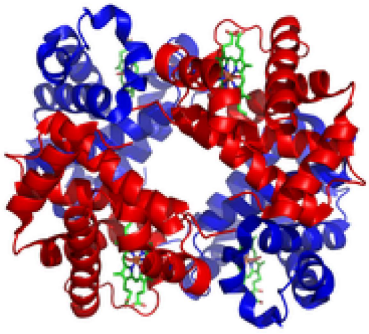

Figure 1. The human hemoglobin molecule consists of 4 polypeptide chains; α subunits are shown in red and β subunits are shown in blue

Proteins power every practical function carried out by the cell, and so presumably, the key to understanding life lies in interpreting the relationship between a chain of amino acids and the function of the protein that this chain of amino acids eventually constructs. **Proteomics** is the field devoted to the study of proteins.

How are proteins created? The **genetic code**, discovered throughout the course of a number of ingenious experiments in the late 1950s, details the **translation** of an RNA molecule called **messenger RNA** (mRNA) into amino acids for protein creation. The apparent difficulty in translation is that somehow 4 RNA bases must be translated into a language of 20 amino acids; in order for every possible amino acid to be created, we must translate 3-nucleobase strings (called **codons**) into amino acids. Note that there are $4^3 = 64$ possible codons, so that multiple codons may encode the same amino acid. Two special types of codons are the **start codon** (AUG), which codes for the amino acid methionine always indicates the start of translation, and the three **stop codons** (UAA, UAG, UGA), which do not code for an amino acid and cause translation to end.

The notion that protein is always created from RNA, which in turn is always created from DNA, forms the **central dogma of molecular biology**. Like all dogmas, it does not always hold; however, it offers an excellent approximation of the truth.

An organelle called a **ribosome** creates peptides by using a helper molecule called **transfer RNA** (tRNA). A single tRNA molecule possesses a string of three RNA nucleotides on one end (called an **anticodon**) and an amino acid at the other end. The ribosome takes an RNA molecule transcribed from DNA (see "Transcribing DNA into RNA"), called **messenger RNA** (mRNA), and examines it one codon at a time. At each step, the tRNA possessing the complementary anticodon bonds to the mRNA at this location, and the amino acid found on the opposite end of the tRNA is added to the growing peptide chain before the remaining part of the tRNA is ejected into the cell, and the ribosome looks for the next tRNA molecule.

Not every RNA base eventually becomes translated into a protein, and so an interval of RNA (or an interval of DNA translated into RNA) that does code for a protein is of great biological interest; such an interval of DNA or RNA is called a **gene**. Because protein creation drives cellular processes, genes differentiate organisms and serve as a basis for **heredity**, or the process by which traits are inherited.

Problem

The 20 commonly occurring amino acids are abbreviated by using 20 letters from the English alphabet (all letters except for B, J, O, U, X, and Z). **Protein strings** are constructed from these 20 symbols. Henceforth, the term

genetic string will incorporate protein strings along with DNA strings and RNA strings.

The **RNA codon table** dictates the details regarding the encoding of specific codons into the amino acid alphabet.

Given: An RNA string s corresponding to a strand of mRNA (of length at most 10 kbp).

Return: The protein string encoded by s.

Sample Dataset

```
AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA
```

Sample Output

```
MAMAPRTEINSTRING
```

Problem 10

Inferring mRNA from Protein

Pitfalls of Reversing Translation

When researchers discover a new protein, they would like to infer the strand of mRNA from which this protein could have been translated, thus allowing them to locate genes associated with this protein on the genome.

> Unfortunately, although any RNA string can be translated into a unique protein string, reversing the process yields a huge number of possible RNA strings from a single protein string because most amino acids correspond to multiple RNA codons (see the RNA Codon Table).
>
> Because of memory considerations, most data formats that are built into languages have upper bounds on how large an integer can be: in some versions of Python, an "int" variable may be required to be no larger than $2^{31}-1$, or 2,147,483,647. As a result, to deal with very large numbers in Rosalind, we need to devise a system that allows us to manipulate large numbers without actually having to store large numbers.

Problem

For positive integers a and n, a **modulo** n (written $a \mod n$ in shorthand) is the remainder when a is divided by n. For example, $29 \mod 11 = 7$ because $29 = 11 \times 2 + 7$.

Modular arithmetic is the study of addition, subtraction, multiplication, and division with respect to the modulo operation. We say that a and b are **congruent** modulo n if $a \mod n = b \mod n$; in this case, we use the notation $a \equiv b \mod n$.

Two useful facts in modular arithmetic are that if $a \equiv b \mod n$ and $c \equiv d \mod n$, then $a + c \equiv b + d \mod n$ and $a \times c \equiv b \times d \mod n$. To check your understanding of these rules, you may wish to verify these relationships for $a = 29$, $b = 73$, $c = 10$, $d = 32$, and $n = 11$.

As you will see in this exercise, some Rosalind problems will ask for a (very large) integer solution modulo a smaller number to avoid the computational pitfalls that arise with storing such large numbers.

Given: A protein string of length at most 1000 aa.

Return: The total number of different RNA strings from which the protein could have been translated, modulo 1,000,000. (Don't neglect the importance of the stop codon in protein translation.)

Sample Dataset

```
MA
```

Sample Output

```
12
```

> **Hint**
>
> What does it mean intuitively to take a number modulo 1,000,000?

Problem 11

Calculating Protein Mass

Chaining the Amino Acids

In "Translating RNA into Protein", we examined the translation of RNA into an amino acid chain for the construction of a protein. When two amino acids link together, they form a **peptide bond**, which releases a molecule of water; see **Figure 1**. Thus, after a series of amino acids have been linked together into a polypeptide, every pair of adjacent amino acids has lost one molecule of water, meaning that a polypeptide containing n amino acids has had $n-1$ water molecules removed.

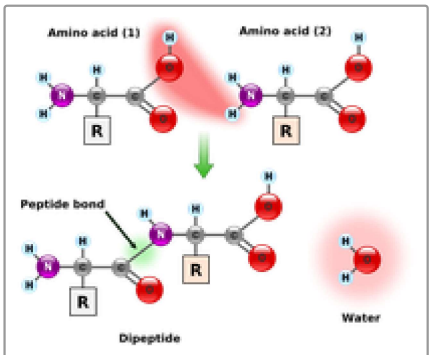

Figure 1. Formation of a peptide bond

More generally, a **residue** is a molecule from which a water molecule has been removed; every amino acid in a protein are residues except the leftmost and the rightmost ones. These outermost amino acids are special in that one has an "unstarted" peptide bond, and the other has an "unfinished" peptide bond. Between them, the two molecules have a single "extra" molecule of water (see the atoms marked in blue in **Figure 2**). Thus, the mass of a protein is the sum of masses of all its residues plus the mass of a single water molecule.

There are two standard ways of computing the mass of a residue by summing the masses of its individual atoms. Its **monoisotopic mass** is computed by using the principal (most abundant) isotope of each atom in the amino acid, whereas its **average mass** is taken by taking the average mass of each atom in the molecule (over all naturally appearing isotopes).

Many applications in proteomics rely on **mass spectrometry**, an analytical chemical technique used to determine the mass, elemental composition, and structure of molecules. In mass spectrometry, monoisotopic mass is used more often than average mass, and so all amino acid masses are assumed to be monoisotopic unless otherwise stated.

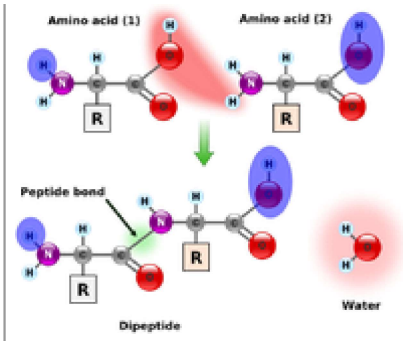

Figure 2. Outermost acids

The standard unit used in mass spectrometry for measuring mass is the **atomic mass unit**, which is also called the **dalton** (Da) and is defined as one twelfth of the mass of a neutral atom of carbon-12. The mass of a protein is the sum of the monoisotopic masses of its amino acid residues plus the mass of a single water molecule (whose monoisotopic mass is 18.01056 **Da**).

In the following several problems on applications of mass spectrometry, we avoid the complication of having to distinguish between residues and non-residues by only considering peptides excised from the middle of the protein. This is a relatively safe assumption because in practice, peptide analysis is often performed in **tandem mass spectrometry**. In this special class of mass spectrometry, a protein is first divided into peptides, which are then broken into ions for mass analysis.

Problem

In a **weighted alphabet**, every symbol is assigned a positive real number called a **weight**. A string formed from a weighted alphabet is called a **weighted string**, and its **weight** is equal to the sum of the weights of its symbols.

The standard weight assigned to each member of the 20-symbol amino acid alphabet is the monoisotopic mass of the corresponding amino acid.

Given: A protein string P of length at most 1000 aa.

Return: The total weight of P. Consult the **monoisotopic mass table**.

Sample Dataset

```
SKADYEK
```

Sample Output

```
821.392
```

Problem 12

Open Reading Frames

Transcription May Begin Anywhere

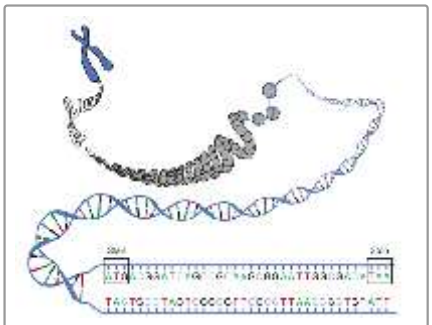

Figure 1. Schematic image of the particular ORF with start and stop codons shown.

In "Transcribing DNA into RNA", we discussed the transcription of DNA into RNA, and in "Translating RNA into Protein", we examined the translation of RNA into a chain of amino acids for the construction of proteins. We can view these two processes as a single step in which we directly translate a DNA string into a protein string, thus calling for a **DNA codon table**.

However, three immediate wrinkles of complexity arise when we try to pass directly from DNA to proteins. First, not all DNA will be transcribed into RNA: so-called **junk DNA** appears to have no practical purpose for cellular function. Second, we can begin translation at any position along a strand of RNA, meaning that any substring of a DNA string can serve as a template for translation, as long as it begins with a start codon, ends with a stop codon, and has no other stop codons in the middle. See **Figure 1**. As a result, the same RNA string can actually be translated in three different ways, depending on how we group triplets of symbols into codons. For example, ...AUGCUGAC... can be translated as ...AUGCUG..., ...UGCUGA..., and ...GCUGAC..., which will typically produce wildly different protein strings.

Problem

Either strand of a DNA double helix can serve as the coding strand for RNA transcription. Hence, a given DNA string implies six total **reading frames**, or ways in which the same region of DNA can be translated into amino acids: three reading frames result from reading the string itself, whereas three more result from reading its reverse complement.

An **open reading frame** (ORF) is one which starts from the start codon and ends by stop codon, without any other stop codons in between. Thus, a candidate protein string is derived by translating an open reading frame into amino acids until a stop codon is reached.

Given: A DNA string s of length at most 1 kbp in FASTA format.

Return: Every distinct candidate protein string that can be translated from ORFs of s. Strings can be returned in any order.

Sample Dataset

```
>Rosalind_99
AGCCATGTAGCTAACTCAGGTTACATGGGGATGACCCCGCGACTTGGATTAGAGTCTCTTTTGGAATAAGCCTGAATGATCCG
AGTAGCATCTCAG
```

Sample Output

```
MLLGSFRLIPKETLIQVAGSSPCNLS
M
MGMTPRLGLESLLE
MTPRLGLESLLE
```

Problem 13

Enumerating k-mers Lexicographically

Organizing Strings

When cataloguing a collection of genetic strings, we should have an established system by which to organize them. The standard method is to organize strings as they would appear in a dictionary, so that "APPLE" precedes "APRON", which in turn comes before "ARMOR".

Problem

Assume that an alphabet $\mathscr{A}$ has a predetermined order; that is, we write the alphabet as a permutation $\mathscr{A} = (a_1, a_2, \ldots, a_k)$, where $a_1 < a_2 < \cdots < a_k$. For instance, the English alphabet is organized as $(\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z})$.

Given two strings s and t having the same length n, we say that s precedes t in the **lexicographic order** (and write $s <_{\mathrm{Lex}} t$) if the first symbol $s[j]$ that doesn't match $t[j]$ satisfies $s_j < t_j$ in $\mathscr{A}$.

Given: A collection of at most 10 symbols defining an ordered alphabet, and a positive integer n ($n \leq 10$).

Return: All strings of length n that can be formed from the alphabet, ordered lexicographically.

Sample Dataset

```
T A G C
2
```

Sample Output

```
AA
AC
AG
AT
CA
CC
CG
CT
GA
GC
GG
GT
TA
TC
TG
TT
```

Note

As illustrated in the sample, the alphabet order in this problem is defined by the order in which symbols are provided in the dataset, which is not necessarily the traditional order of the English alphabet.

Problem 14

k-Mer Composition

Generalizing GC-Content

A length k substring of a genetic string is commonly called a **k-mer**. A genetic string of length n can be seen as composed of $n-k+1$ overlapping k-mers. The **k-mer composition** of a genetic string encodes the number of times that each possible k-mer occurs in the string. See **Figure 1**. The 1-mer composition is a generalization of the GC-content of a strand of DNA, and the 2-mer, 3-mer, and 4-mer compositions of a DNA string are also commonly known as its di-nucleotide, tri-nucleotide, and tetra-nucleotide compositions.

AA	AC	AG	AT	CA	CC	CG	CT	GA	GC	GG	GT	TA	TC	TG	TT
3	5	0	8	6	2	1	3	2	2	0	4	5	3	7	8

Figure 1. The 2-mer composition of TTGATTACCTTATTTGATCATTACACATTGT

The biological significance of k-mer composition is manyfold. GC-content is helpful not only in helping to identify a piece of unknown DNA (see "Computing GC Content"), but also because a genomic region having high GC-content compared to the rest of the genome signals that it may belong to an exon. Analyzing k-mer composition is vital to fragment assembly as well.

In "Computing GC Content", we also drew an analogy between analyzing the frequency of characters and identifying the underlying language. For larger values of k, the k-mer composition offers a more robust fingerprint of a string's identity because it offers an analysis on the scale of substrings (i.e., words) instead of that of single symbols. As a basis of comparison, in language analysis, the k-mer composition of a text can be used not only to pin down the language, but also often the *author*.

Problem

For a fixed positive integer k, order all possible k-mers taken from an underlying alphabet lexicographically.

Then the k-mer composition of a string s can be represented by an array A for which $A[m]$ denotes the number of times that the mth k-mer (with respect to the lexicographic order) appears in s.

Given: A DNA string s in FASTA format (having length at most 100 kbp).

Return: The 4-mer composition of s.

Sample Dataset

```
>Rosalind_6431
CTTCGAAAGTTTGGGCCGAGTCTTACAGTCGGTCTTGAAGCAAAGTAACGAACTCCACGG
CCCTGACTACCGAACCAGTTGTGAGTACTCAACTGGGTGAGAGTGCAGTCCCTATTGAGT
TTCCGAGACTCACCGGGATTTTCGATCCAGCCTCAGTCCAGTCTTGTGGCCAACTCACCA
AATGACGTTGGAATATCCCTGTCTAGCTCACGCAGTACTTAGTAAGAGGTCGCTGCAGCG
```

```
GGGCAAGGAGATCGGAAAATGTGCTCTATATGCGACTAAAGCTCCTAACTTACACGTAGA
CTTGCCCGTGTTAAAAACTCGGCTCACATGCTGTCTGCGGCTGGCTGTATACAGTATCTA
CCTAATACCCTTCAGTTCGCCGCACAAAAGCTGGGAGTTACCGCGGAAATCACAG
```

Sample Output

```
4 1 4 3 0 1 1 5 1 3 1 2 2 1 2 0 1 1 3 1 2 1 3 1 1 1 1 2 2 5 1 3 0 2 2 1 1 1 1 3 1 0
0 1 5 5 1 5 0 2 0 2 1 2 1 1 1 2 0 1 0 0 1 1 3 2 1 0 3 2 3 0 0 2 0 8 0 0 1 0 2 1 3 0
0 0 1 4 3 2 1 1 3 1 2 1 3 1 2 1 2 1 1 1 2 3 2 1 1 0 1 1 3 2 1 2 6 2 1 1 1 2 3 3 3 2
3 0 3 2 1 1 0 0 1 4 3 0 1 5 0 2 0 1 2 1 3 0 1 2 2 1 1 0 3 0 0 4 5 0 3 0 2 1 1 3 0 3
2 2 1 1 0 2 1 0 2 2 1 2 0 2 2 5 2 2 1 1 2 1 2 2 2 2 1 1 3 4 0 2 1 1 0 1 2 2 1 1 1 5
2 0 3 2 1 1 2 2 3 0 3 0 1 3 1 2 3 0 2 1 2 2 1 2 3 0 1 2 3 1 1 3 1 0 1 1 3 0 2 1 2 2
0 2 1 1
```

Problem 15

Rabbits and Recurrence Relations

Wascally Wabbits

In 1202, Leonardo of Pisa (commonly known as Fibonacci) considered a mathematical exercise regarding the reproduction of a population of rabbits. He made the following simplifying assumptions about the population:

1. The population begins in the first month with a pair of newborn rabbits.
2. Rabbits reach reproductive age after one month.
3. In any given month, every rabbit of reproductive age mates with another rabbit of reproductive age.
4. Exactly one month after two rabbits mate, they produce one male and one female rabbit.
5. Rabbits never die or stop reproducing.

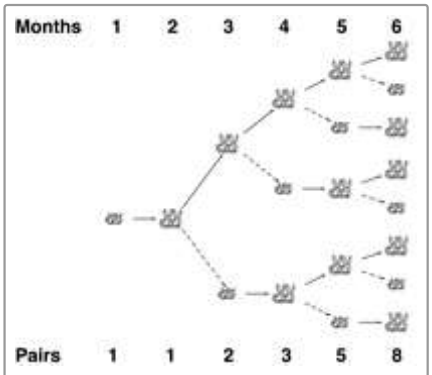

Figure 1. The growth of Fibonacci's rabbit population for the first six months.

Fibonacci's exercise was to calculate how many pairs of rabbits would remain in one year. We can see that in the second month, the first pair of rabbits reach reproductive age and mate. In the third month, another pair of rabbits is born, and we have two rabbit pairs; our first pair of rabbits mates again. In the fourth month, another pair of rabbits is born to the original pair, while the second pair reach maturity and mate (with three total pairs). The dynamics of the rabbit population are illustrated in **Figure 1**. After a year, the rabbit population boasts 144 pairs.

Figure 2. Erosion at Lake Mungo in New South Wales, which was initiated by

Although Fibonacci's assumption of the rabbits' immortality may seem a bit farfetched, his model was not unrealistic for reproduction in a predator-free environment: European rabbits were introduced to

European rabbits in the 19th Century. Courtesy Pierre Pouliquin.

> Australia in the mid 19th Century, a place with no real indigenous predators for them. Within 50 years, the rabbits had already eradicated many plant species across the continent, leading to irreversible changes in the Australian ecosystem and turning much of its grasslands into eroded, practically uninhabitable parts of the modern Outback (see **Figure 2**). In this problem, we will use the simple idea of counting rabbits to introduce a new computational topic, which involves building up large solutions from smaller ones.

Problem

A **sequence** is an ordered collection of objects (usually numbers), which are allowed to repeat. Sequences can be finite or infinite. Two examples are the finite sequence $(\pi, -\sqrt{2}, 0, \pi)$ and the infinite sequence of odd numbers $(1, 3, 5, 7, 9, \ldots)$. We use the notation a_n to represent the n-th term of a sequence.

A **recurrence relation** is a way of defining the terms of a sequence with respect to the values of previous terms. In the case of Fibonacci's rabbits from the introduction, any given month will contain the rabbits that were alive the previous month, plus any new offspring. A key observation is that the number of offspring in any month is equal to the number of rabbits that were alive two months prior. As a result, if F_n represents the number of rabbit pairs alive after the n-th month, then we obtain the **Fibonacci sequence** having terms F_n that are defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ (with $F_1 = F_2 = 1$ to initiate the sequence). Although the sequence bears Fibonacci's name, it was known to Indian mathematicians over two millennia ago.

When finding the n-th term of a sequence defined by a recurrence relation, we can simply use the recurrence relation to generate terms for progressively larger values of n. This problem introduces us to the computational technique of **dynamic programming**, which successively builds up solutions by using the answers to smaller cases.

Given: Positive integers $n \leq 40$ and $k \leq 5$.

Return: The total number of rabbit pairs that will be present after n months, if we begin with 1 pair and in each generation, every pair of reproduction-age rabbits produces a litter of k rabbit pairs (instead of only 1 pair).

Sample Dataset

```
5 3
```

Sample Output

```
19
```

Problem 16

Mortal Fibonacci Rabbits

> **Wabbit Season**

In "Rabbits and Recurrence Relations", we mentioned the disaster caused by introducing European rabbits into Australia. By the turn of the 20th Century, the situation was so out of control that the creatures could not be killed fast enough to slow their spread (see **Figure 1**).

Figure 1. A c.1905 photo from Australia of a cart loaded to the hilt with rabbit skins.

> The conclusion? Build a fence! The fence, intended to preserve the sanctity of Western Australia, was completed in 1907 after undergoing revisions to push it back as the bunnies pushed their frontier ever westward (see **Figure 2**). If it sounds like a crazy plan, the Australians at the time seem to have concurred, as shown by the cartoon in **Figure 3**.
>
> By 1950, Australian rabbits numbered 600 million, causing the government to decide to release a virus (called myxoma) into the wild, which cut down the rabbits until they acquired resistance. In a final Hollywood twist, another experimental rabbit virus escaped in 1991, and some resistance has already been observed.
>
> The bunnies will not be stopped, but they don't live forever, and so in this problem, our aim is to expand Fibonacci's rabbit population model to allow for mortal rabbits.

Figure 2. Western Australia's rabbit fence is actually not the longest fence in the world as the sign claims. That honor goes to a 3,500 mile fence in southeastern Australia built to keep out dingoes. Courtesy Matt Pounsett.

Figure 3. An 1884 cartoon from the Queensland Figaro proposing how the rabbits viewed their fence.

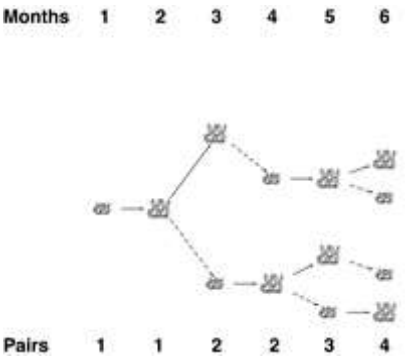

Figure 4. A figure illustrating the propagation of Fibonacci's rabbits if they die after three months.

Problem

Recall the definition of the Fibonacci numbers from "Rabbits and Recurrence Relations", which followed the recurrence relation $F_n = F_{n-1} + F_{n-2}$ and assumed that each pair of rabbits reaches maturity in one month and produces a single pair of offspring (one male, one female) each subsequent month.

Our aim is to somehow modify this recurrence relation to achieve a dynamic programming solution in the case that all rabbits die out after a fixed number of months. See **Figure 4** for a depiction of a rabbit tree in which rabbits live for three months (meaning that they reproduce only twice before dying).

Given: Positive integers $n \leq 100$ and $m \leq 20$.

Return: The total number of pairs of rabbits that will remain after the n-th month if all rabbits live for m months.

Sample Dataset

```
6 3
```

Sample Output

```
4
```

Problem 17

Introduction to Random Strings

Modeling Random Genomes

We already know that the genome is not just a random strand of nucleotides; recall from "Finding a Motif in DNA" that motifs recur commonly across individuals and species. If a DNA motif occurs in many different organisms, then chances are good that it serves an important function.

> At the same time, if you form a long enough DNA string, then you should theoretically be able to locate every possible short substring in the string. And genomes are very long; the human genome contains about 3.2 billion base pairs. As a result, when analyzing an unknown piece of DNA, we should try to ensure that a motif does not occur out of random chance.
>
> To conclude whether motifs are random or not, we need to quantify the likelihood of finding a given motif randomly. If a motif occurs randomly with high probability, then how can we really compare two organisms to begin with? In other words, all very short DNA strings will appear randomly in a genome, and very few long strings will appear; what is the critical motif length at which we can throw out random chance and conclude that a motif appears in a genome for a reason?
>
> In this problem, our first step toward understanding random occurrences of strings is to form a simple model for constructing genomes randomly. We will then apply this model to a somewhat simplified exercise: calculating the probability of a given motif occurring randomly at a *fixed* location in the genome.

Problem

An **array** is a structure containing an ordered collection of objects (numbers, strings, other arrays, etc.). We let $A[k]$ denote the k-th value in array A. You may like to think of an array as simply a matrix having only one row.

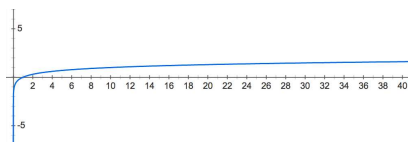

Figure 1. The graph of the common logarithm function of x. For a given x-value, the corresponding y-value is the exponent to which we must raise 10 to obtain x. Note that x-values between 0 and 1 get mapped to y-values between -infinity and 0.

A **random string** is constructed so that the probability of choosing each subsequent symbol is based on a fixed underlying symbol frequency.

GC-content offers us natural symbol frequencies for constructing random DNA strings. If the GC-content is x, then we set the symbol frequencies of C and G equal to $\frac{x}{2}$ and the symbol frequencies of A and T equal to $\frac{1-x}{2}$. For example, if the GC-content is 40%, then as we construct the string, the next symbol is 'G'/'C' with probability 0.2, and the next symbol is 'A'/'T' with probability 0.3.

In practice, many probabilities wind up being very small. In order to work with small probabilities, we may plug them into a function that "blows them up" for the sake of comparison. Specifically, the **common logarithm** of x (defined for $x > 0$ and denoted $\log_{10}(x)$) is the exponent to which we must raise 10 to obtain x.

See **Figure 1** for a graph of the common logarithm function $y = \log_{10}(x)$. In this graph, we can see that the logarithm of x-values between 0 and 1 always winds up mapping to y-values between $-\infty$ and 0: x-values near 0 have logarithms close to $-\infty$, and x-values close to 1 have logarithms close to 0. Thus, we will select the common logarithm as our function to "blow up" small probability values for comparison.

Given: A DNA string s of length at most 100 bp and an array A containing at most 20 numbers between 0 and 1.

Return: An array B having the same length as A in which $B[k]$ represents the common logarithm of the probability that a random string constructed with the GC-content found in $A[k]$ will match s exactly.

Sample Dataset

```
ACGATACAA
0.129 0.287 0.423 0.476 0.641 0.742 0.783
```

Sample Output

```
-5.737 -5.217 -5.263 -5.360 -5.958 -6.628 -7.009
```

> **Hint**
>
> One property of the logarithm function is that for any positive numbers x and y,
> $\log_{10}(x \cdot y) = \log_{10}(x) + \log_{10}(y)$

Problem 18

Introduction to Pattern Matching

If At First You Don't Succeed...

We introduced the problem of finding a motif in a genetic string in "Finding a Motif in DNA". More commonly, we will have a collection of motifs that we may wish to find in a larger string, for example when searching a genome for a collection of known genes.

> This application sets up the algorithmic problem of **pattern matching**, in which we are searching a large string (called a **text**) for instances of a collection of smaller strings, called **patterns**. For the moment, we will focus on requiring that all matches should be exact.
>
> The most obvious method for finding exact patterns in a text is to simply apply a simple "sliding window" algorithm for each pattern. However, this method is time-consuming if we have a large number of patterns to consider (which will often be the case when dealing with a database of genes). It would be better if instead of traversing the genome for every pattern, we could somehow only traverse it once. To this end, we will need a data structure that can efficiently organize a collection of patterns.

Problem

Given a collection of strings, their **trie** (often pronounced "try" to avoid ambiguity with the general term tree) is a rooted tree formed as follows. For every unique first symbol in the strings, an edge is formed connecting the root to a new vertex. This symbol is then used to label the edge.

We may then iterate the process by moving down one level as follows. Say that an edge connecting the root to a node v is labeled with 'A'; then we delete the first symbol from every string in the collection beginning with 'A' and then treat v as our root. We apply this process to all nodes that are adjacent to the root, and then we move down another level and continue. See **Figure 1** for an example of a trie.

As a result of this method of construction, the symbols along the edges of any path in the trie from the root to a leaf will spell out a unique string from the collection, as long as no string is a prefix of another in the collection (this would cause the first string to be encoded as a path terminating at an internal node).

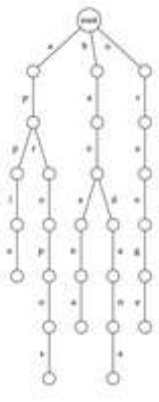

Figure 1. The trie corresponding to the strings 'apple', 'apropos', 'banana', 'bandana', and 'orange'. Each path from root to leaf encodes one of these strings.

Given: A list of at most 100 DNA strings of length at most 100 bp, none of which is a prefix of another.

Return: The adjacency list corresponding to the trie T for these patterns, in the following format. If T has n nodes, first label the root with 1 and then label the remaining nodes with the integers 2 through n in any order you like. Each edge of the adjacency list of T will be encoded by a triple containing the integer representing the edge's parent node, followed by the integer representing the edge's child node, and finally the symbol labeling the edge.

Sample Dataset

```
ATAGA
ATC
GAT
```

Sample Output

```
1 2 A
2 3 T
3 4 A
4 5 G
5 6 A
3 7 C
1 8 G
8 9 A
9 10 T
```

Problem 19

Finding a Motif in DNA

> ### Combing Through the Haystack

Finding the same interval of DNA in the genomes of two different organisms (often taken from different species) is highly suggestive that the interval has the same function in both organisms.

We define a **motif** as such a commonly shared interval of DNA. A common task in molecular biology is to search an organism's genome for a known motif.

Figure 1. The human chromosomes stained with a probe for Alu elements, shown in green.

The situation is complicated by the fact that genomes are riddled with intervals of DNA that occur multiple times (possibly with slight modifications), called **repeats**. These repeats occur far more often than would be dictated by random chance, indicating that genomes are anything but random and in fact illustrate that the language of DNA must be very powerful (compare with the frequent reuse of common words in any human language).

The most common repeat in humans is the **Alu repeat**, which is approximately 300 bp long and recurs around a million times throughout every human genome (see **Figure 1**). However, Alu has not been found to serve a positive purpose, and appears in fact to be parasitic: when a new Alu repeat is inserted into a genome, it frequently causes genetic disorders.

Problem

Given two strings s and t, t is a **substring** of s if t is contained as a contiguous collection of symbols in s (as a result, t must be no longer than s).

The **position** of a symbol in a string is the total number of symbols found to its left, including itself (e.g., the positions of all occurrences of 'U' in "AUGCUUCAGAAAGGUCUUACG" are 2, 5, 6, 15, 17, and 18). The symbol at position i of s is denoted by $s[i]$.

A substring of s can be represented as $s[j:k]$, where j and k represent the starting and ending positions of the substring in s; for example, if s = "AUGCUUCAGAAAGGUCUUACG", then $s[2:5]$ = "UGCU".

The **location** of a substring $s[j:k]$ is its beginning position j; note that t will have multiple locations in s if it occurs more than once as a substring of s (see the Sample below).

Given: Two DNA strings s and t (each of length at most 1 kbp).

Return: All locations of t as a substring of s.

Sample Dataset

```
GATATATGCATATACTT
ATAT
```

Sample Output

```
2 4 10
```

Note

Different programming languages use different notations for positions of symbols in strings. Above, we use **1-based numbering**, as opposed to **0-based numbering**, which is used in Python. For s =

"AUGCUUCAGAAAGGUCUUACG", 1-based numbering would state that $s[1]$ = 'A' is the first symbol of the string, whereas this symbol is represented by $s[0]$ in 0-based numbering. The idea of 0-based numbering propagates to substring indexing, so that $s[2:5]$ becomes "GCUU" instead of "UGCU".

Note that in some programming languages, such as Python, s[j:k] returns only fragment from index j up to but *not* including index k, so that s[2:5] actually becomes "UGC", not "UGCU".

Problem 20

Finding a Protein Motif

Motif Implies Function

As mentioned in "Translating RNA into Protein", proteins perform every practical function in the cell. A structural and functional unit of the protein is a **domain**: in terms of the protein's primary structure, the domain is an interval of amino acids that can evolve and function independently.

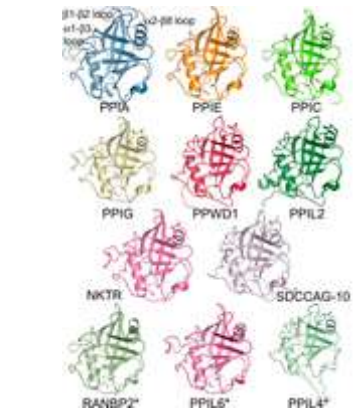

Figure 1. The human cyclophilin family, as represented by the structures of the isomerase domains of some of its members.

Each domain usually corresponds to a single function of the protein (e.g., binding the protein to DNA, creating or breaking specific chemical bonds, etc.). Some proteins, such as myoglobin and the Cytochrome complex, have only one domain, but many proteins are multifunctional and therefore possess several domains. It is even possible to artificially fuse different domains into a protein molecule with definite properties, creating a **chimeric protein**.

Just like species, proteins can evolve, forming homologous groups called **protein families**. Proteins from one family usually have the same set of domains, performing similar functions; see **Figure 1**.

A component of a domain essential for its function is called a motif, a term that in general has the same meaning as it does in nucleic acids, although many other terms are also used (blocks, signatures, fingerprints, etc.) Usually protein motifs are evolutionarily conservative, meaning that they appear without much change in different species.

Proteins are identified in different labs around the world and gathered into freely accessible databases. A central repository for protein data is UniProt, which provides detailed protein annotation, including function description, domain structure, and post-translational modifications. UniProt also supports protein similarity search, taxonomy analysis, and literature citations.

Problem

To allow for the presence of its varying forms, a protein motif is represented by a shorthand as follows: [XY] means "either X or Y" and {X} means "any amino acid except X." For example, the N-glycosylation motif is written as N{P}[ST]{P}.

You can see the complete description and features of a particular protein by its access ID "uniprot_id" in the UniProt database, by inserting the ID number into

```
http://www.uniprot.org/uniprot/uniprot_id
```

Alternatively, you can obtain a protein sequence in FASTA format by following

```
http://www.uniprot.org/uniprot/uniprot_id.fasta
```

For example, the data for protein B5ZC00 can be found at http://www.uniprot.org/uniprot/B5ZC00.

Given: At most 15 UniProt Protein Database access IDs.

Return: For each protein possessing the N-glycosylation motif, output its given access ID followed by a list of locations in the protein string where the motif can be found.

Sample Dataset

```
A2Z669
B5ZC00
P07204_TRBM_HUMAN
P20840_SAG1_YEAST
```

Sample Output

```
B5ZC00
85 118 142 306 395
P07204_TRBM_HUMAN
47 115 116 382 409
P20840_SAG1_YEAST
79 109 135 248 306 348 364 402 485 501 614
```

> ### Note
>
> Some entries in UniProt have one primary (citable) accession number and some secondary numbers, appearing due to merging or demerging entries. In this problem, you may be given any type of ID. If you type the secondary ID into the UniProt query, then you will be automatically redirected to the page containing the primary ID. You can find more information about UniProt IDs here.

Problem 21

Finding a Shared Motif

> ### Searching Through the Haystack
>
> In "Finding a Motif in DNA", we searched a given genetic string for a motif; however, this problem assumed that we know the motif in advance. In practice, biologists often do not know exactly what they are looking for.

Rather, they must hunt through several different genomes at the same time to identify regions of similarity that may indicate genes shared by different organisms or species.

The simplest such region of similarity is a motif occurring without mutation in every one of a collection of genetic strings taken from a database; such a motif corresponds to a substring shared by all the strings. We want to search for long shared substrings, as a longer motif will likely indicate a greater shared function.

Problem

A **common substring** of a collection of strings is a substring of every member of the collection. We say that a common substring is a **longest common substring** if there does not exist a longer common substring. For example, "CG" is a common substring of "A**CG**TACGT" and "AAC**CG**TATA", but it is not as long as possible; in this case, "CGTA" is a longest common substring of "A**CGTA**CGT" and "AAC**CGTA**TA".

Note that the longest common substring is not necessarily unique; for a simple example, "AA" and "CC" are both longest common substrings of "AACC" and "CCAA".

Given: A collection of k ($k \leq 100$) DNA strings of length at most 1 kbp each in FASTA format.

Return: A longest common substring of the collection. (If multiple solutions exist, you may return any single solution.)

Sample Dataset

```
>Rosalind_1
GATTACA
>Rosalind_2
TAGACCA
>Rosalind_3
ATACA
```

Sample Output

```
AC
```

Problem 22

Speeding Up Motif Finding

Shortening the Motif Search

In "Finding a Motif in DNA", we discussed the problem of searching a genome for a known motif. Because of the large scale of eukaryotic genomes, we need to accomplish this computational task as efficiently as possible.

> The standard method for locating one string t as a substring of another string s (and perhaps one you implemented in "Finding a Motif in DNA") is to move a sliding window across the larger string, at each step starting at $s[k]$ and matching subsequent symbols of t to symbols of s. After we have located a match or mismatch, we then shift the window backwards to begin searching at $s[k+1]$.
>
> The potential weakness of this method is as follows: say we have matched 100 symbols of t to s before reaching a mismatch. The window-sliding method would then move back 99 symbols of s and start comparing t to s; can we avoid some of this sliding?
>
> For example, say that we are looking for $t = \mathrm{ACGTACGT}$ in $s = \mathrm{TAGGTACGTACGGCATCACG}$. From $s[6]$ to $s[12]$, we have matched seven symbols of t, and yet $s[13]$ = G produces a mismatch with $t[8]$ = T. We don't need to go all the way back to $s[7]$ and start matching with t because $s[7] = \mathrm{C}$, $s[8] = \mathrm{G}$, and $s[9] = \mathrm{T}$ are all different from $t[1] = \mathrm{A}$. What about $s[10]$? Because $t[1:4] = t[5:8] = \mathrm{ACGT}$, the previous mismatch of $s[13] = \mathrm{G}$ and $t[8] = \mathrm{T}$ guarantees the *same* mismatch with $s[13]$ and $t[4]$. Following this analysis, we may advance directly to $s[14]$ and continue sliding our window, without ever having to move it backward.
>
> This method can be generalized to form the framework behind the **Knuth-Morris-Pratt algorithm** (KMP), which was published in 1977 and offers an efficiency boost for determining whether a given motif can be located within a larger string.

Problem

A **prefix** of a length n string s is a substring $s[1:j]$; a **suffix** of s is a substring $s[k:n]$.

The **failure array** of s is an array P of length n for which $P[k]$ is the length of the longest substring $s[j:k]$ that is equal to some prefix $s[1:k-j+1]$ where j cannot equal 1 (otherwise, $P[k]$ would always equal k). By convention, $P[1] = 0$.

Given: A DNA string s (of length at most 100 kbp) in FASTA format.

Return: The failure array of s.

Sample Dataset

```
>Rosalind_87
CAGCATGGTATCACAGCAGAG
```

Sample Output

```
0 0 0 1 2 0 0 0 0 0 0 1 2 1 2 3 4 5 3 0 0
```

> ### Extra Information
>
> If you would like a more precise technical explanation of the Knuth-Morris-Pratt algorithm, please take a look at this site

Problem 23

Longest Increasing Subsequence

A Simple Measure of Gene Order Similarity

In "Enumerating Gene Orders", we started talking about comparing the order of genes on a chromosome taken from two different species and moved around by rearrangements throughout the course of evolution.

> One very simple way of comparing genes from two chromosomes is to search for the largest collection of genes that are found in the same order in both chromosomes. To do so, we will need to apply our idea of permutations. Say that two chromosomes share n genes; if we label the genes of one chromosome by the numbers 1 through n in the order that they appear, then the second chromosome will be given by a permutation of these numbered genes. To find the largest number of genes appearing in the same order, we need only to find the largest collection of increasing elements in the permutation.

Problem

A **subsequence** of a permutation is a collection of elements of the permutation in the order that they appear. For example, (5, 3, 4) is a subsequence of (5, 1, 3, 4, 2).

A subsequence is **increasing** if the elements of the subsequence increase, and **decreasing** if the elements decrease. For example, given the permutation (8, 2, 1, 6, 5, 7, 4, 3, 9), an increasing subsequence is (2, 6, 7, 9), and a decreasing subsequence is (8, 6, 5, 4, 3). You may verify that these two subsequences are as long as possible.

Given: A positive integer $n \leq 10000$ followed by a permutation π of length n.

Return: A longest increasing subsequence of π, followed by a longest decreasing subsequence of π.

Sample Dataset

```
5
5 1 4 2 3
```

Sample Output

```
1 2 3
5 4 2
```

> ### Citation
>
> Adapted from Jones & Pevzner, *An Introduction to Bioinformatics Algorithms, Problem 6.48.

Problem 24

Consensus and Profile

Finding a Most Likely Common Ancestor

In "Counting Point Mutations", we calculated the minimum number of symbol mismatches between two strings of equal length to model the problem of finding the minimum number of point mutations occurring on the evolutionary path between two homologous strands of DNA. If we instead have several homologous strands that we wish to analyze simultaneously, then the natural problem is to find an average-case strand to represent the most likely common ancestor of the given strands.

Problem

A **matrix** is a rectangular table of values divided into rows and columns. An $m \times n$ matrix has m rows and n columns. Given a matrix A, we write $A_{i,j}$ to indicate the value found at the intersection of row i and column j.

Say that we have a collection of DNA strings, all having the same length n. Their **profile matrix** is a $4 \times n$ matrix P in which $P_{1,j}$ represents the number of times that 'A' occurs in the jth position of one of the strings, $P_{2,j}$ represents the number of times that C occurs in the jth position, and so on (see below).

A **consensus string** c is a string of length n formed from our collection by taking the most common symbol at each position; the jth symbol of c therefore corresponds to the symbol having the maximum value in the j-th column of the profile matrix. Of course, there may be more than one most common symbol, leading to multiple possible consensus strings.

```
                A T C C A G C T
                G G G C A A C T
                A T G G A T C T
DNA Strings     A A G C A A C C
                T T G G A A C T
                A T G C C A T T
                A T G G C A C T

          A     5 1 0 0 5 5 0 0
Profile   C     0 0 1 4 2 0 6 1
          G     1 1 6 3 0 1 0 0
          T     1 5 0 0 0 1 1 6

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

Given: A collection of at most 10 DNA strings of equal length (at most 1 kbp) in FASTA format.

Return: A consensus string and profile matrix for the collection. (If several possible consensus strings exist, then you may return any one of them.)

Sample Dataset

```
>Rosalind_1
ATCCAGCT
>Rosalind_2
GGGCAACT
>Rosalind_3
ATGGATCT
>Rosalind_4
AAGCAACC
>Rosalind_5
TTGGAACT
>Rosalind_6
ATGCCATT
>Rosalind_7
ATGGCACT
```

Sample Output

```
ATGCAACT
A: 5 1 0 0 5 5 0 0
C: 0 0 1 4 2 0 6 1
G: 1 1 6 3 0 1 0 0
T: 1 5 0 0 0 1 1 6
```

Problem 25

Finding the Longest Multiple Repeat

Long Repeats

We saw in "Introduction to Pattern Matching" that a data structure commonly used to encode the relationships among a collection of strings was the trie, which is particularly useful when the strings represent a collection of patterns that we wish to match to a larger text.

> The trie is helpful when processing multiple strings at once, but when we want to analyze a single string, we need something different.
>
> In this problem, we will use a new data structure to handle the problem of finding long repeats in the genome. Recall from "Finding a Motif in DNA" that cataloguing these repeats is a problem of the utmost interest to molecular biologists, as a natural correlation exists between the frequency of a repeat and its influence on RNA transcription. Our aim is therefore to identify long repeats that occur more than some predetermined number of times.

Problem

A **repeated substring** of a string s of length n is simply a substring that appears in more than one location of s; more specifically, a **k-fold**

substring appears in at least k distinct locations.

The **suffix tree** of s, denoted $T(s)$, is defined as follows:

- $T(s)$ is a rooted tree having exactly n leaves.
- Every edge of $T(s)$ is labeled with a substring of s^*, where s^* is the string formed by adding a placeholder symbol `$` to the end of s.
- Every internal node of $T(s)$ other than the root has at least two children; i.e., it has degree at least 3.
- The substring labels for the edges leading from a node to its children must begin with different symbols.
- By concatenating the substrings along edges, each path from the root to a leaf corresponds to a unique suffix of s^*.

Figure 1. The suffix tree for s = GTCCGAAGCTCCGG. Note that the dollar sign has been appended to a substring of the tree to mark the end of s. Every path from the root to a leaf corresponds to a unique suffix of GTCCGAAGCTCCGG, and each leaf is labeled with the location in s of the suffix ending at that leaf.

See **Figure 1** for an example of a suffix tree.

Given: A DNA string s (of length at most 20 kbp) with `$` appended, a positive integer k, and a list of edges defining the suffix tree of s. Each edge is represented by four components:

1. the label of its parent node in $T(s)$;
2. the label of its child node in $T(s)$;
3. the location of the substring t of s^* assigned to the edge; and
4. the length of t.

Return: The longest substring of s that occurs at least k times in s. (If multiple solutions exist, you may return any single solution.)

Sample Dataset

```
CATACATAC$
2
node1 node2 1 1
node1 node7 2 1
node1 node14 3 3
node1 node17 10 1
node2 node3 2 4
node2 node6 10 1
node3 node4 6 5
node3 node5 10 1
node7 node8 3 3
node7 node11 5 1
node8 node9 6 5
node8 node10 10 1
node11 node12 6 5
node11 node13 10 1
node14 node15 6 5
node14 node16 10 1
```

Sample Output

```
CATAC
```

Hint

How can repeated substrings of s be located in $T(s)$?

Problem 26

Introduction to Protein Databases

Four Commonly Used Protein Databases

Proteins are identified in different labs around the world, and data about them is gathered into freely accessible databases. A central repository for protein data is **UniProt**, a comprehensive high-quality database established by an international consortium. UniProt provides detailed protein annotation, including function description, domain structure, and post-translational modifications. It also supports protein similarity search, taxonomy analysis, and literature citations. See **Figure 1** for a screenshot of the UniProt homepage.

Figure 1. Screenshot of the UniProt homepage.

UniProt is part of ExPASy, the resource portal of Swiss Institute of Bioinformatics. ExPASy provides access to numerous databases and software tools in different areas of the life sciences.

The most important component of UniProt is the UniProt Knowledgebase, or UniProtKB (see the help page here), a protein database partially curated by experts. UniProtKB comprises two major parts:

- Swiss-Prot: a manually annotated and reviewed, non-redundant protein sequence database. Each Swiss-Prot entry holds all relevant information about a particular protein, including data taken from scientific literature data and the results of computational analysis of the protein.
- TrEMBL (stands for "Translated EMBL"): an automatically annotated database that is not reviewed. TrEMBL is not a true protein database; instead, it holds translated versions of nucleic acid sequences taken from multiple sources, including the European Molecular Biology Laboratory (EMBL) database, worldwide genome sequencing projects, and the coding regions of genes accessed from GenBank.

The National Center for Biotechnology Information (NCBI) also maintains protein data. Data equivalent to that of Swiss-Prot is part of the NCBI's **RefSeq** database (http://www.ncbi.nlm.nih.gov/RefSeq/), a curated collection of genetic sequences. RefSeq records are annotated by NCBI personnel, and they provide reliable information for genomic DNA along with RNA transcribed from DNA and the corresponding translated proteins. NCBI's equivalent project to TrEmbl is the NCBI Protein database, which is part of the larger GenBank database and holds unannotated protein sequences. If a nucleotide sequence contained in GenBank codes for protein, then the corresponding amino acid sequence is automatically annotated and included in the NCBI database with its own protein ID. The NCBI databases also recognize UniProt IDs as search terms.

Because Swiss-Prot annotation provides so much information, NCBI protein records usually provide links to corresponding Swiss-Prot entries whenever possible.

The NCBI Protein Database can be found here.

Problem

The UniProt Knowledgebase can be found here.

You can see a complete description of a protein by entering its UniProt access ID into the site's query field. Equivalently, you may simply insert its ID (`uniprot_id`) directly into a UniProt hyperlink as follows:

```
http://www.uniprot.org/uniprot/uniprot_id
```

For example, the data for protein B5ZC00 can be found at http://www.uniprot.org/uniprot/B5ZC00.

Swiss-Prot holds protein data as a structured .txt file. You can obtain it by simply adding `.txt` to the link:

```
http://www.uniprot.org/uniprot/uniprot_id.txt
```

Given: The UniProt ID of a protein.

Return: A list of biological processes in which the protein is involved (biological processes are found in a subsection of the protein's "Gene Ontology" (GO) section).

Sample Dataset

```
Q5SLP9
```

Sample Output

```
DNA recombination
DNA repair
DNA replication
```

Programming Shortcut

ExPASy databases can be accessed automatically via Biopython's `Bio.ExPASy` module. The function `.get_sprot_raw` will find a target protein by its ID.

We can obtain data from an entry by using the `SwissProt` module. The `read()` function will handle one SwissProt record and `parse` will allow you to read multiple records at a time.

Let's get the data for the B5ZC00 protein:

```
>>>from Bio import ExPASy
>>>from Bio import SwissProt
>>>handle = ExPASy.get_sprot_raw('B5ZC00') #you can give several IDs separated b
y commas
>>>record = SwissProt.read(handle) # use SwissProt.parse for multiple proteins
```

We now can check the list of attributes for the obtained Swiss-Prot record:

```
 >>> dir(record)
 [..., 'accessions', 'annotation_update', 'comments', 'created', 'cross_referenc
es',
 'data_class', 'description', 'entry_name', 'features', 'gene_name', 'host_organ
```

```
ism', 'host_taxonomy_id', 'keywords',
 'molecule_type', 'organelle', 'organism', 'organism_classification', 'reference
s', 'seqinfo', 'sequence',
 'sequence_length', 'sequence_update', 'taxonomy_id']
```

To see the list of references to other databases, we can check the `.cross_references` attribute of our record:

```
>>>record.cross_references[0]
('EMBL', 'CP001184', 'ACI60310.1', '-', 'Genomic_DNA')
```

Problem 27

Data Formats

Same Data, Different Formats

A number of different data presentation formats have been used to represent genetic strings. The history of file formats presents its own kind of evolution: some formats have died out, being replaced by more successful ones. Three file formats are currently the most popular:

- FASTA (.fas, .fasta): used by practically all modern software and databases, including ClustalX, Paup, HyPhy, Rdp, and Dambe.
- **NEXUS** (.nex, .nexus, .nxs): used by Paup, MrBayes, FigTree, and SplitsTree.
- **PHYLIP**, or "**Phyl**ogeny **I**nference **P**ackage" (.phy): used by Phylip, Tree-Puzzle, PhyML, and a number of databases.

Figure 1. An example of a GenBank record header.

A simple reference on file formats can be found here.

In this problem, we will familarize ourselves with FASTA. We will save the other two formats for later problems.

In FASTA format, a string is introduced by a line that begins with '>', followed by some information labeling the string. Subsequent lines contain the string itself; the next line beginning with '>' indicates that the current string is complete and begins the label of the next string in the file.

GenBank hosts its own file format for storing genome data, containing a large amount of information about each interval of DNA. The GenBank file describes the interval's source, taxonomic position, authors, and features (see **Figure 1**).

A sample GenBank entry can be found here. You may export an entry to a variety of file formats by selecting the appropriate file format under the `Send To:` dropdown menu at the top of the page.

Problem

GenBank can be accessed here. A detailed description of the GenBank format can be found here. A tool, from the SMS 2 package, for converting GenBank to FASTA can be found here.

Given: A collection of n ($n \leq 10$) GenBank entry IDs.

Return: The shortest of the strings associated with the IDs in FASTA format.

Sample Dataset

```
FJ817486 JX069768 JX469983
```

Sample Output

```
>gi|408690371|gb|JX469983.1| Zea mays subsp. mays clone UT3343 G2-like transcriptio
n factor mRNA, partial cds
ATGATGTATCATGCGAAGAATTTTTCTGTGCCCTTTGCTCCGCAGAGGGCACAGGATAATGAGCATGCAA
GTAATATTGGAGGTATTGGTGGACCCAACATAAGCAACCCTGCTAATCCTGTAGGAAGTGGGAAACAACG
GCTACGGTGGACATCGGATCTTCATAATCGCTTTGTGGATGCCATCGCCCAGCTTGGTGGACCAGACAGA
GCTACACCTAAAGGGGTTCTCACTGTGATGGGTGTACCAGGGATCACAATTTATCATGTGAAGAGCCATC
TGCAGAAGTATCGCCTTGCAAAGTATATACCCGACTCTCCTGCTGAAGGTTCCAAGGACGAAAAGAAAGA
TTCGAGTGATTCCCTCTCGAACACGGATTCGGCACCAGGATTGCAAATCAATGAGGCACTAAAGATGCAA
ATGGAGGTTCAGAAGCGACTACATGAGCAACTCGAGGTTCAAAGACAACTGCAACTAAGAATTGAAGCAC
AAGGAAGATACTTGCAGATGATCATTGAGGAGCAACAAAAGCTTGGTGGATCAATTAAGGCTTCTGAGGA
TCAGAAGCTTTCTGATTCACCTCCAAGCTTAGATGACTACCCAGAGAGCATGCAACCTTCTCCCAAGAAA
CCAAGGATAGACGCATTATCACCAGATTCAGAGCGCGATACAACACAACCTGAATTCGAATCCCATTTGA
TCGGTCCGTGGGATCACGGCATTGCATTCCCAGTGGAGGAGTTCAAAGCAGGCCCTGCTATGAGCAAGTC
A
```

Programming Shortcut

Here we can again use the `Bio.Entrez` module introduced in "GenBank Introduction". To search for particular access IDs, you can use the function `Bio.Entrez.efetch(db, rettype)`, which takes two parameters: the `db` parameter takes the database to search, and the `rettype` parameter takes the data format to be returned. For example, we use "nucleotide" (or "nuccore") as the `db` parameter for Genbank and "fasta" as the `rettype` parameter for FASTA format.

The following code illustrates `efetch()` in action. It obtains plain text records in FASTA format from NCBI's [Nucleotide] database.

```
>>>from Bio import Entrez
>>>Entrez.email = "your_name@your_mail_server.com"
>>>handle = Entrez.efetch(db="nucleotide", id=["FJ817486, JX069768, JX469983"],
rettype="fasta")
>>>records = handle.read()
>>>print records
```

To work with FASTA format, we can use the `Bio.SeqIO` module, which provides an interface to input and output methods for different file formats. One of its main functions is `Bio.SeqIO.parse()`, which takes a handle and format name as parameters and returns entries as SeqRecords.

```
>>>from Bio import Entrez
>>>from Bio import SeqIO
>>>Entrez.email = "your_name@your_mail_server.com"
>>>handle = Entrez.efetch(db="nucleotide", id=["FJ817486, JX069768, JX469983"],
```

```
rettype="fasta")
>>>records = list (SeqIO.parse(handle, "fasta")) #we get the list of SeqIO objec
ts in FASTA format
>>>print records[0].id  #first record id
gi|227437129|gb|FJ817486.1|
>>>print len(records[-1].seq)  #Length of the last record
771
```

Problem 28

FASTQ format introduction

How to Handle Quality

Modern sequencing instruments generally produce a unbelievable number of reads, which are generally short sequences of <~200 bases. All sequencing machines are prone to some degree of error, so it is important to quantify the certainty with which each read is identified, or "called." This is often referred to as the quality of the base call.

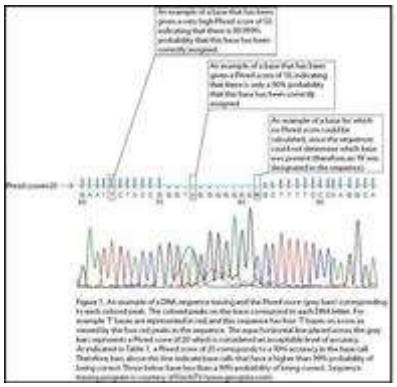

Figure 1. Phred quality scores

This quality is customarily measured on the Phred quality scale. Lower scores indicate less confidence in the accuracy of the base call. The Phred quality score Q is calculated using the following equation:

$$Q = -10\ log_{10} P$$

where P is the probability that the corresponding base call is incorrect. Thus, for Q=10, the probability that the base is called incorrectly is 1 in 10, and the accuracy is 90%. For Q=20, the probability of error is 1 in 100 and the accuracy is 99%, and so on. See **Figure 1** for an illustration.

The standard format for storing the output of high throughput sequencing instruments is FASTQ format. FASTQ is an extension of FASTA format; it provides both the sequence and the per-base quality scores for each read. Nucleotides are represented by a single symbol, while Phred scores are usually 2-digit numbers. To be able to conveniently list the two together in FASTQ format, an offset of 33 is added to the score, which is then represented by the corresponding symbol from the ASCII table. Here are two examples:

- The quality of a base with a Phred score of 10 (90% accuracy) is denoted by "+" (ASCII symbol #43)
- The quality of a base with a Phred score of 40 (99.99% accuracy) is denoted by "I" (ASCII symbol #73).

The offset of 33 was chosen because the symbols 0-33 in the ASCII table are reserved for a non-printable characters, such as backspace or vertical tab.

Each record in a FASTQ file typically consists of four lines:

- A line starting with @ that contains the sequence identifier
- The actual sequence
- A line starting with + containing an optional sequence identifier or comment
- A line with quality scores encoded as ASCII symbols

Notice that the 2nd and 4th line should always have the same length.

The following is an example FASTQ record:

```
@HWI-ST999:102:D1N6AACXX:1:1101:1235:1936 1:N:0:
ATGTCTCCTGGACCCCTCTGTGCCCAAGCTCCTCATGCATCCTCCTCAGCAACTTGTCCTGTAGCTGAGGCTCACTGACT
ACCAGCTGCAG
+
1:DAADDDF\<B\<AGF=FGIEHCCD9DG=1E9?D>CF@HHG??B\<GEBGHCG;;CDB8==C@@>>GII@@5?A?@B>C
EDCFCC:;?CCCAC
```

Generally, a FASTQ file has the suffix *.fq* or *.fastq*.

Problem

Sometimes it's necessary to convert data from FASTQ format to FASTA format. For example, you may want to perform a BLAST search using reads in FASTQ format obtained from your brand new Illumina Genome Analyzer.

Links:

- A FASTQ to FASTA converter can be accessed from the Sequence conversion website
- A free GUI converter developed by BlastStation is available here for download or as an add-on to Google Chrome.
- There is a FASTQ to FASTA converter in the Galaxy web platform. Note that you should register in the Galaxy and upload your file prior to using this tool.

Given: FASTQ file

Return: Corresponding FASTA records

Sample Dataset

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!*((((***+))%%%++)(%%%%).1***-+*****))**55CCF>>>>>>CCCCCCC65
```

Sample Output

```
>SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
```

Programming Shortcut

BioPython's Bio.SeqIO module will save the day.

Problem 29

Counting Point Mutations

Evolution as a Sequence of Mistakes

A **mutation** is simply a mistake that occurs during the creation or copying of a nucleic acid, in particular DNA. Because nucleic acids are vital to cellular functions, mutations tend to cause a ripple effect throughout the cell. Although mutations are technically mistakes, a very rare mutation may equip the cell with a beneficial attribute. In fact, the macro effects of evolution are attributable by the accumulated result of beneficial microscopic mutations over many generations.

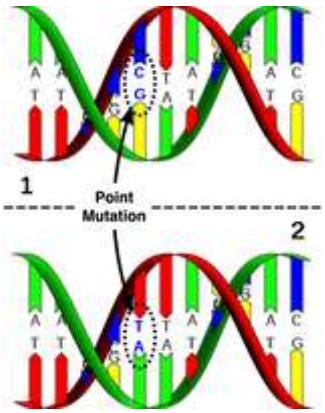

Figure 1. A point mutation in DNA changing a C-G pair to an A-T pair.

The simplest and most common type of nucleic acid mutation is a **point mutation**, which replaces one base with another at a single nucleotide. In the case of DNA, a point mutation must change the complementary base accordingly; see **Figure 1**.

Two DNA strands taken from different organism or species genomes are **homologous** if they share a recent ancestor; thus, counting the number of bases at which homologous strands differ provides us with the minimum number of point mutations that could have occurred on the evolutionary path between the two strands.

We are interested in minimizing the number of (point) mutations separating two species because of the biological principle of **parsimony**, which demands that evolutionary histories should be as simply explained as possible.

Problem

Given two strings s and t of equal length, the **Hamming distance** between s and t, denoted $d_{\mathrm{H}}(s, t)$, is the number of corresponding symbols that differ in s and t. See **Figure 2**.

GAGCCTACTAACGGGAT
CATCGTAATGACGGCCT

Figure 2. The Hamming distance between these two strings is 7. Mismatched symbols are colored red.

Given: Two DNA strings s and t of equal length (not exceeding 1 kbp).

Return: The Hamming distance $d_{\mathrm{H}}(s, t)$.

Sample Dataset

```
GAGCCTACTAACGGGAT
CATCGTAATGACGGCCT
```

Sample Output

```
7
```

Problem 30

Edit Distance

Point Mutations Include Insertions and Deletions

In "Counting Point Mutations", we saw that Hamming distance gave us a preliminary notion of the evolutionary distance between two DNA strings by counting the minimum number of single nucleotide substitutions that could have occurred on the evolutionary path between the two strands.

> However, in practice, homologous strands of DNA or protein are rarely the same length because point mutations also include the insertion or deletion of a single nucleotide (and single amino acids can be inserted or deleted from peptides). Thus, we need to incorporate these insertions and deletions into the calculation of the minimum number of point mutations between two strings. One of the simplest models charges a unit "cost" to any single-symbol insertion/deletion, then (in keeping with parsimony) requests the minimum cost over all transformations of one genetic string into another by point substitutions, insertions, and deletions.

Problem

Given two strings s and t (of possibly different lengths), the **edit distance** $d_{\mathrm{E}}(s, t)$ is the minimum number of **edit operations** needed to transform s into t, where an edit operation is defined as the substitution, insertion, or deletion of a single symbol.

The latter two operations incorporate the case in which a contiguous interval is inserted into or deleted from a string; such an interval is called a **gap**. For the purposes of this problem, the insertion or deletion of a gap of length k still counts as k distinct edit operations.

Given: Two protein strings s and t in FASTA format (each of length at most 1000 aa).

Return: The edit distance $d_{\mathrm{E}}(s, t)$.

Sample Dataset

```
>Rosalind_39
PLEASANTLY
>Rosalind_11
MEANLY
```

Sample Output

```
5
```

Problem 31

Edit Distance Alignment

Reconstructing Edit Distance

In "Counting Point Mutations", the calculation of Hamming distance gave us a clear way to model the sequence of point mutations transforming one genetic string into another. By simply writing one string directly over the other, we could count each mismatched symbol as a substitution.

> However, in the calculation of edit distance (see "Edit Distance"), the two strings can have different lengths; thus, simply superimposing one string over the other does us no good when it comes to visualizing a sequence of edit operations transforming one string into the other. To remedy this, we will introduce a new symbol to serve as a placeholder representing an inserted or deleted symbol; furthermore, this placeholder will allow us to align two strings of differing lengths.

Problem

An **alignment** of two strings s and t is defined by two strings s' and t' satisfying the following three conditions: 1. s' and t' must be formed from adding **gap symbols** "-" to each of s and t, respectively; as a result, s and t will form subsequences of s' and t'. 2. s' and t' must have the same length. 3. Two gap symbols may not be aligned; that is, if $s'[j]$ is a gap symbol, then $t'[j]$ cannot be a gap symbol, and vice-versa.

We say that s' and t' **augment** s and t. Writing s' directly over t' so that symbols are *aligned* provides us with a scenario for transforming s into t. Mismatched symbols from s and t correspond to symbol substitutions; a gap symbol $s'[j]$ aligned with a non-gap symbol $t'[j]$ implies the insertion of this symbol into t; a gap symbol $t'[j]$ aligned with a non-gap symbol $s'[j]$ implies the deletion of this symbol from s.

Thus, an alignment represents a transformation of s into t via edit operations. We define the corresponding **edit alignment score** of s' and t' as $d_{\mathrm{H}}(s', t')$ (Hamming distance is used because the gap symbol has been introduced for insertions and deletions). It follows that $d_{\mathrm{E}}(s, t) = \min_{s', t'} d_{\mathrm{H}}(s', t')$ where the minimum is taken over all alignments of s and t. We call such a minimum score alignment an **optimal alignment** (with respect to edit distance).

Given: Two protein strings s and t in FASTA format (with each string having length at most 1000 aa).

Return: The edit distance $d_{\mathrm{E}}(s, t)$ followed by two augmented strings s' and t' representing an optimal alignment of s and t.

Sample Dataset

```
>Rosalind_43
PRETTY
>Rosalind_97
PRTTEIN
```

Sample Output

```
4
PRETTY--
PR-TTEIN
```

Problem 32

Pairwise Global Alignment

Comparing Strings Online

Regions of similarity in two genetic strings that are presumed to be homologous can be found by alignment.

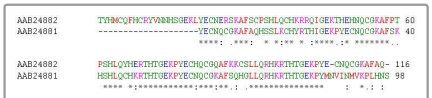

Figure 1. An alignment of two human zinc finger proteins, identified by their GenBank accession numbers on the left. A '*' below aligned symbols denotes identical amino acids, a ':' denotes similar amino acids, a '.' denotes less similar amino acids, and a blank space denotes dissimilar amino acids. Color-coding represents the amino acid's chemical properties (e.g., blue denotes acidic and green denotes basic).

As shown in **Figure 1**, a global alignment of two strings is formed by adding gap symbols ('-') to each string to make them the same length, so that every symbol in one string corresponds to a symbol in the other. A gap symbol in an aligned string represents a deletion at that position or an insertion in the string it is aligned to. Mismatches can be interpreted as point mutations that change symbols at single positions in the strings.

There are many possible alignments for a pair of strings. In practice, we cannot know which is the absolute "best" alignment of two genetic strings. This is because discriminating between alignments requires assumptions about how the strings have evolved, and this information about their evolution is precisely what we hope to learn from the alignment. The best we can do is to compare alignments quantitatively by using a scoring approach. Many such alignment scores assign 0 to a match, a negative number to a mismatch, and an even larger negative number to a gap. Summing all these scores over an alignment gives the alignment's score; in keeping with the assumption of parsimony, the optimal alignment will have the highest score, implying the fewest possible changes between the strings.

A scoring matrix contains a score for matches and mismatches between each pair of symbols. These scores are derived empirically based on the frequency with which these matches and mismatches occur in highly similar biological sequences for which the true alignment is practically unequivocal.

A gap penalty is the score deducted for the presence of a gap. One of the most commonly used gap penalties is an affine gap penalty, in which one penalty is assigned for adding the first symbol to a gap and different penalty is charged for every additional symbol in the gap. The gap opening penalty is usually larger than the gap extension penalty because a single deletion of k nucleotides is more likely than k independent, contiguous, single-nucleotide deletions.

For example, say that we use a simple scoring matrix that rewards +1 for all matching symbols and charges -1 for all mismatches, along with an affine gap penalty that has a gap opening penalty of -5 and a gap extension penalty of -1. The following alignment has 14 matched symbols, 3 mismatches, and two gaps (one of which is extended by three symbols), which gives a total score for the alignment of
$14 - 3 - (2 \cdot 5 + 3) = -2$

```
GARFIELDTHEVERYFA-TCAT
:::|||||||||    || ||||
WINFIELDTHE----FASTCAT
```

If you would like to learn about the algorithmic ideas behind the global alignment algorithms, you may be interested in solving "Global Alignment with Scoring Matrix" and "Global Alignment with Scoring Matrix and Affine Gap Penalty" in the Bioinformatics Stronghold.

> EBI hosts many different online user interfaces for bioinformatics tools, including `Needle` and `Stretcher`. `Needle` and `Stretcher` are tools from the EMBOSS package that are for aligning genetic strings. Both can be accessed from here
>
> The programs are very similar. One difference is that `Stretcher` always uses an end gap penalty, while `Needle` includes the option but does not use it by default. In either of these interfaces, you can enter strings directly or in any supported format: GCG, FASTA, EMBL, GenBank, PIR/NBRF, Phylip, or the UniProtKB/Swiss-Prot format. Feel free to play around with these interfaces by entering pairs of strings. Notice that clicking on `More options...` under "Step 2" will allow you to change the alignment parameters as well as the output format.

Problem

An online interface to EMBOSS's `Needle` tool for aligning DNA and RNA strings can be found here.

Use:

- The DNAfull scoring matrix; note that DNAfull uses IUPAC notation for ambiguous nucleotides.
- Gap opening penalty of 10.
- Gap extension penalty of 1.

For our purposes, the "pair" output format will work fine; this format shows the two strings aligned at the bottom of the output file beneath some statistics about the alignment.

Given: Two GenBank IDs.

Return: The maximum global alignment score between the DNA strings associated with these IDs.

Sample Dataset

```
JX205496.1 JX469991.1
```

Sample Output

```
257
```

> ### Programming Shortcut
>
> You can download the EMBOSS suite from the EMBOSS homepage on Sourceforge and run `Needle` and `Stretcher` locally via the command line (or by using a graphical user interface, such as Jemboss for UNIX systems).
>
> The Needle help page contains detailed descriptions of command line arguments. Do not forget to count gaps on the ends of an alignment using the arguments `-endweight`, `-endopen` and `-endextend`.

Problem 33

Global Alignment with Scoring Matrix

Generalizing the Alignment Score

The edit alignment score in "Edit Distance Alignment" counted the total number of edit operations implied by an alignment; we could equivalently think of this scoring function as assigning a cost of 1 to each such operation. Another common scoring function awards matched symbols with 1 and penalizes substituted/inserted/deleted symbols equally by assigning each one a score of 0, so that the maximum score of an alignment becomes the length of a longest common subsequence of s and t (see "Finding a Shared Spliced Motif"). In general, the **alignment score** is simply a scoring function that assigns costs to edit operations encoded by the alignment.

One natural way of adding complexity to alignment scoring functions is by changing the alignment score based on which symbols are substituted; many methods have been proposed for doing this. Another way to do so is to vary the penalty assigned to the insertion or deletion of symbols.

In general, alignment scores can be either maximized or minimized depending on how scores are established. The general problem of optimizing a particular alignment score is called **global alignment**.

Problem

To penalize symbol substitutions differently depending on which two symbols are involved in the substitution, we obtain a **scoring matrix** S in which $S_{i,j}$ represents the (negative) score assigned to a substitution of the ith symbol of our alphabet $\mathscr{A}$ with the jth symbol of $\mathscr{A}$.

A **gap penalty** is the component deducted from alignment score due to the presence of a gap. A gap penalty may be a function of the length of the gap; for example, a **linear gap penalty** is a constant g such that each inserted or deleted symbol is charged g; as a result, the cost of a gap of length L is equal to gL.

Given: Two protein strings s and t in FASTA format (each of length at most 1000 aa).

Return: The maximum alignment score between s and t. Use:

- The BLOSUM62 scoring matrix.
- Linear gap penalty equal to 5 (i.e., a cost of -5 is assessed for each gap symbol).

Sample Dataset

```
>Rosalind_67
PLEASANTLY
>Rosalind_17
MEANLY
```

Sample Output

```
8
```

Problem 34

Global Alignment with Constant Gap Penalty

Penalizing Large Insertions and Deletions

In dealing with global alignment in "Global Alignment with Scoring Matrix", we encountered a linear gap penalty, in which the insertion or deletion of a gap is penalized by some constant times the length of the gap. However, this model is not necessarily the most practical model, as one large rearrangement could have inserted or deleted a long gap in a single step to transform one genetic string into another.

Problem

In a **constant gap penalty**, every gap receives some predetermined constant penalty, regardless of its length. Thus, the insertion or deletion of 1000 contiguous symbols is penalized equally to that of a single symbol.

Given: Two protein strings s and t in FASTA format (each of length at most 1000 aa).

Return: The maximum alignment score between s and t. Use:

- The BLOSUM62 scoring matrix.
- Constant gap penalty equal to 5.

Sample Dataset

```
>Rosalind_79
PLEASANTLY
>Rosalind_41
MEANLY
```

Sample Output

```
13
```

Problem 35

Global Alignment with Scoring Matrix and Affine Gap Penalty

Mind the Gap

In "Global Alignment with Scoring Matrix", we considered a linear gap penalty, in which each inserted/deleted symbol contributes the exact same amount to the calculation of alignment score. However, as we mentioned in "Global Alignment with Constant Gap Penalty", a single large insertion/deletion (due to a rearrangement is then punished very strictly, and so we

> proposed a constant gap penalty.
>
> Yet large insertions occur far more rarely than small insertions and deletions. As a result, a more practical method of penalizing gaps is to use a hybrid of these two types of penalties in which we charge one constant penalty for *beginning* a gap and another constant penalty for every *additional* symbol added or deleted.

Problem

An **affine gap penalty** is written as $a + b \cdot (L - 1)$ where L is the length of the gap, a is a positive constant called the **gap opening penalty**, and b is a positive constant called the **gap extension penalty**.

We can view the gap opening penalty as charging for the first gap symbol, and the gap extension penalty as charging for each subsequent symbol added to the gap.

For example, if $a = 11$ and $b = 1$, then a gap of length 1 would be penalized by 11 (for an average cost of 11 per gap symbol), whereas a gap of length 100 would have a score of 110 (for an average cost of 1.10 per gap symbol).

Consider the strings "PRTEINS" and "PRTWPSEIN". If we use the BLOSUM62 scoring matrix and an affine gap penalty with $a = 11$ and $b = 1$, then we obtain the following optimal alignment.

```
PRT---EINS
|||   |||
PRTWPSEIN-
```

Matched symbols contribute a total of 32 to the calculation of the alignment's score, and the gaps cost 13 and 11 respectively, yielding a total score of 8.

Given: Two protein strings s and t in FASTA format (each of length at most 100 aa).

Return: The maximum alignment score between s and t, followed by two augmented strings s' and t' representing an optimal alignment of s and t. Use:

- The BLOSUM62 scoring matrix.
- Gap opening penalty equal to 11.
- Gap extension penalty equal to 1.

Sample Dataset

```
>Rosalind_49
PRTEINS
>Rosalind_47
PRTWPSEIN
```

Sample Output

```
8
PRT---EINS
PRTWPSEIN-
```

Problem 36

Character-Based Phylogeny

Introduction to Character-Based Phylogeny

In "Creating a Character Table", we discussed the construction of a character table from a collection of characters represented by subsets of our taxa. However, the ultimate goal is to be able to construct a phylogeny from this character table.

> The issues at hand are that we want to ensure that we have enough characters to actually construct a phylogeny, and that our characters do not conflict with each other.

Problem

Because a tree having n nodes has $n-1$ edges (see "Completing a Tree"), removing a single edge from a tree will produce two smaller, disjoint trees. Recall from "Creating a Character Table" that for this reason, each edge of an unrooted binary tree corresponds to a split $S \mid S^c$, where S is a subset of the taxa.

A **consistent character table** is one whose characters' splits do not conflict with the edge splits of some unrooted binary tree T on the n taxa. More precisely, $S_1 \mid S_1^c$ conflicts with $S_2 \mid S_2^c$ if all four intersections $S_1 \cap S_2$, $S_1 \cap S_2^c$, $S_1^c \cap S_2$, and $S_1^c \cap S_2^c$ are nonempty. As a simple example, consider the conflicting splits $\{a, b\} \mid \{c, d\}$ and $\{a, c\} \mid \{b, d\}$.

More generally, given a consistent character table C, an unrooted binary tree T "models" C if the edge splits of T agree with the splits induced from the characters of C.

Given: A list of n species ($n \leq 80$) and an n-column character table C in which the jth column denotes the jth species.

Return: An unrooted binary tree in Newick format that models C.

Sample Dataset

```
cat dog elephant mouse rabbit rat
011101
001101
001100
```

Sample Output

```
(dog,(cat,rabbit),(rat,(elephant,mouse)));
```

Problem 37

Alignment-Based Phylogeny

From Characters Toward Alignments

In "Creating a Character Table from Genetic Strings", we used strings to create a collection of characters from which we could create a phylogeny. However, the strings all had to share the same length, which was a problem. In practice, we would like to create a phylogeny from genetic strings having differing lengths; specifically, our aim is to construct a phylogeny from a multiple alignment.

> Unfortunately, constructing a phylogeny from the ground up based only on an alignment can be difficult. In order to produce an efficient solution, we will need to assume that the structure of the phylogeny has already been provided (perhaps from character-based methods), and our aim instead is to reconstruct the genetic strings corresponding to the internal nodes (i.e., ancestors) in the tree.
>
> The ancestor strings should have the property that the total number of point mutations separating adjacent nodes in the tree is minimized (in keeping with parsimony).

Problem

Say that we have n taxa represented by strings $s_1, s_2, \ldots, s_n$ with a multiple alignment inducing corresponding augmented strings $\overline{s}_1, \overline{s}_2, \ldots, \overline{s}_n$.

Recall that the number of single-symbol substitutions required to transform one string into another is the Hamming distance between the strings (see "Counting Point Mutations"). Say that we have a rooted binary tree T containing $\overline{s}_1, \overline{s}_2, \ldots, \overline{s}_n$ at its leaves and additional strings $\overline{s}_{n+1}, \overline{s}_{n+2}, \ldots, \overline{s}_{2n-1}$ at its internal nodes, including the root (the number of internal nodes is $n-1$ by extension of "Counting Phylogenetic Ancestors"). Define $d_{\mathrm{H}}(T)$ as the sum of $d_{\mathrm{H}}(\overline{s}_i, \overline{s}_j)$ over all edges $\{\overline{s}_i, \overline{s}_j\}$ in T:

$$d_{\mathrm{H}}(T) = \sum_{\{\overline{s}_i, \overline{s}_j\} \in E(T)} d_{\mathrm{H}}(\overline{s}_i, \overline{s}_j)$$

Thus, our aim is to minimize $d_{\mathrm{H}}(T)$.

Given: A rooted binary tree T on n ($n \leq 500$) species, given in Newick format, followed by a multiple alignment of m ($m \leq n$) augmented DNA strings having the same length (at most 300 bp) corresponding to the species and given in FASTA format.

Return: The minimum possible value of $d_{\mathrm{H}}(T)$, followed by a collection of DNA strings to be assigned to the internal nodes of T that will minimize $d_{\mathrm{H}}(T)$ (multiple solutions will exist, but you need only output one).

Sample Dataset

```
(((ostrich,cat)rat,(duck,fly)mouse)dog,(elephant,pikachu)hamster)robot;
>ostrich
AC
>cat
CA
>duck
T-
>fly
GC
>elephant
```

```
-T
>pikachu
AA
```

Sample Output

```
8
>rat
AC
>mouse
TC
>dog
AC
>hamster
AT
>robot
AC
```

Note

Given internal strings minimizing $d_{\mathrm{H}}(T)$, the alignment between any two adjacent strings is not necessarily an optimal global paired alignment. In other words, it may not be the case that $d_{\mathrm{H}}(\overline{s}_i, \overline{s}_j)$ is equal to the edit distance $d_{\mathrm{E}}(s_i, s_j)$.

Problem 38

Completing a Tree

The Tree of Life

"As buds give rise by growth to fresh buds, and these, if vigorous, branch out and overtop on all sides many a feebler branch, so by generation I believe it has been with the great Tree of Life, which fills with its dead and broken branches the crust of the earth, and covers the surface with its ever-branching and beautiful ramifications." **Charles Darwin**, ***The Origin of Species***

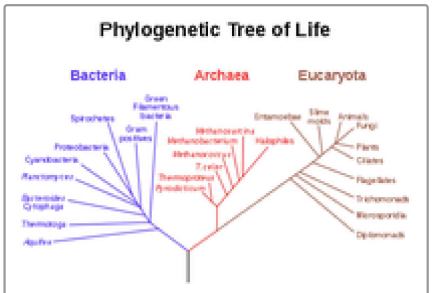

Figure 1. A phylogeny illustrating proposed evolutionary relationships among the three domains of life: Bacteria, Archaea, and Eukaryota.

A century and a half has passed since the publication of Darwin's magnum opus, and yet the construction of a single **Tree of Life** uniting life on Earth still has not been completed, with perhaps as many as 90% of all living species not yet catalogued (although a beautiful interactive animation has been produced by OneZoom).

To get an insight about state-of-art attempts to build this tree, you may take a look at the Tree of Life Web Project - collaborative effort of biologists from around the world to combine information about diversity of life

on Earth. It is a peer-reviewed ongoing project started in 1995, now it holds more than 10,000 pages with characteristics of different groups of organisms and their evolutionary history, and tree still grows.

Instead of trying to construct the entire Tree of Life all at once, we often wish to form a simpler tree in which a collection of species have been clumped together for the sake of simplicity; such a group is called a **taxon** (pl. taxa). For a given collection of taxa, a **phylogeny** is a treelike diagram that best represents the evolutionary connections between the taxa: the construction of a particular phylogeny depends on our specific assumptions regarding how these evolutionary relationships should be interpreted. See **Figure 1**.

Problem

An undirected graph is **connected** if there is a path connecting any two nodes. A **tree** is a connected (undirected) graph containing no cycles; this definition forces the tree to have a branching structure organized around a central core of nodes, just like its living counterpart. See **Figure 2**.

We have already grown familiar with trees in "Mendel's First Law", where we introduced the probability tree diagram to visualize the outcomes of a random variable.

In the creation of a phylogeny, taxa are encoded by the tree's **leaves**, or nodes having degree 1. A node of a tree having degree larger than 1 is called an **internal node**.

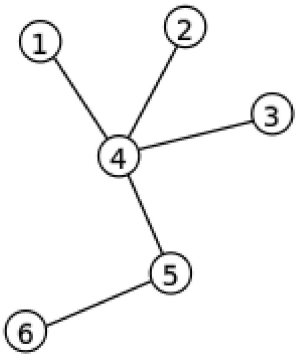

Figure 2. A labeled tree with 6 vertices and 5 edges.

Given: A positive integer n ($n \leq 1000$) and an adjacency list corresponding to a graph on n nodes that contains no cycles.

Return: The minimum number of edges that can be added to the graph to produce a tree.

Sample Dataset

```
10
1 2
2 8
4 10
5 9
6 10
7 9
```

Sample Output

```
3
```

Extra Information

After solving this problem, a standard mathematical exercise for the technically minded is to verify that every tree having 2 or more nodes must contain at least two leaves.

Problem 39

Distances in Trees

Paths in Trees

For any two nodes of a tree, a unique path connects the nodes; more specifically, there is a unique path connecting any pair of leaves. Why must this be the case? If more than one path connected two nodes, then they would necessarily form a cycle, which would violate the definition of tree.

> The uniqueness of paths connecting nodes in a tree is helpful in phylogenetic analysis because a rudimentary measure of the separation between two taxa is the distance between them in the tree, which is equal to the number of edges on the unique path connecting the two leaves corresponding to the taxa.

Problem

Newick format is a way of representing trees even more concisely than using an adjacency list, especially when dealing with trees whose internal nodes have not been labeled.

First, consider the case of a rooted tree T. A collection of leaves $v_1, v_2, \ldots, v_n$ of T are **neighbors** if they are all adjacent to some internal node u. Newick format for T is obtained by iterating the following key step: delete all the edges $\{v_i, u\}$ from T and label u with $(v_1, v_2, \ldots, v_n)u$ This process is repeated all the way to the root, at which point a semicolon signals the end of the tree.

A number of variations of Newick format exist. First, if a node is not labeled in T, then we simply leave blank the space occupied by the node. In the key step, we can write $(v_1, v_2, \ldots, v_n)$ in place of $(v_1, v_2, \ldots, v_n)u$ if the v_i are labeled; if none of the nodes are labeled, we can write $(, , \ldots,)$.

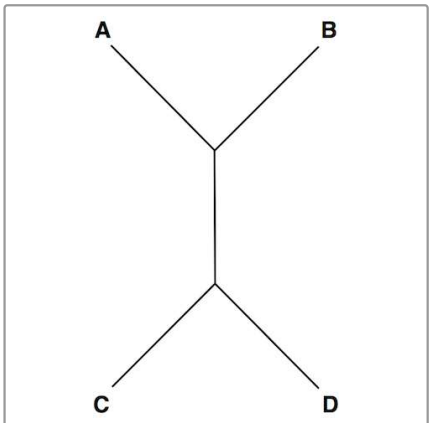

Figure 1. This tree can be represented in Newick format in a number of ways, including (C, D, (A, B)); (A, (D, C), B); and (((A, B), C), D);.

A second variation of Newick format occurs when T is unrooted, in which case we simply select any internal node to serve as the root of T. A particularly peculiar case of Newick format arises when we choose a leaf to serve as the root.

Note that there will be a large number of different ways to represent T in Newick format; see **Figure 1**.

Given: A collection of n trees ($n \leq 40$) in Newick format, with each tree containing at most 200 nodes; each tree T_k is followed by a pair of nodes x_k and y_k in T_k.

Return: A collection of n positive integers, for which the kth integer represents the distance between x_k and y_k in T_k.

Sample Dataset

```
(cat)dog;
dog cat
```

```
(dog,cat);
dog cat
```

Sample Output

```
1 2
```

Problem 40

Counting Phylogenetic Ancestors

Culling the Forest

In "Completing a Tree", we introduced the tree for the purposes of constructing phylogenies. Yet the definition of tree as a connected graph with no cycles produces a huge class of different graphs, from simple paths and star-like graphs to more familiar arboreal structures (see **Figure 1**). Which of these graphs are appropriate for phylogenetic study?

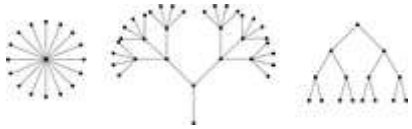

Figure 1. Trees come in lots of different shapes.

Modern evolutionary theory (beginning with Darwin) indicates that a common way for a new species can be created is when it splits off from an existing species after a population is isolated for an extended period of time. This model of species evolution implies a very specific type of phylogeny, in which internal nodes represent branching points of evolution where an ancestor species either evolved into a new species or split into two new species: therefore, one edge of this internal node therefore connects the node to its most recent ancestor, whereas one or two new edges connect it to its immediate descendants. This framework offers a much clearer notion of how to characterize phylogenies.

Problem

A **binary tree** is a tree in which each node has degree equal to at most 3. The binary tree will be our main tool in the construction of phylogenies.

A **rooted tree** is a tree in which one node (the **root**) is set aside to serve as the pinnacle of the tree. A standard graph theory exercise is to verify that for any two nodes of a tree, exactly one path connects the nodes. In a rooted tree, every node v will therefore have a single **parent**, or the unique node w such that the path from v to the root contains $\{v, w\}$. Any other node x adjacent to v is called a **child** of v because v must be the parent of x; note that a node may have multiple children. In other words, a rooted tree possesses an ordered hierarchy from the root down to its leaves, and as a result, we may often view a rooted tree with undirected edges as a directed graph in which each edge is oriented from parent to child. We should already be familiar with this idea; it's how the Rosalind problem tree works!

Even though a binary tree can include nodes having degree 2, an **unrooted binary tree** is defined more specifically: all internal nodes have degree 3. In turn, a **rooted binary tree** is such that only the root has degree 2 (all other internal nodes have degree 3).

Given: A positive integer n ($3 \leq n \leq 10000$).

Return: The number of internal nodes of any unrooted binary tree having n leaves.

Sample Dataset

```
4
```

Sample Output

```
2
```

> **Hint**
>
> In solving "Completing a Tree", you may have formed the conjecture that a graph with no cycles and n nodes is a tree precisely when it has $n-1$ edges. This is indeed a theorem of graph theory.

Problem 41

Enumerating Unrooted Binary Trees

Seeing the Forest

In "Counting Unrooted Binary Trees", we found a way to count the number of unrooted binary trees representing phylogenies on n taxa. Our observation was that two such trees are considered distinct when they do not share the same collection of splits.

> Counting all these trees is one task, but actually understanding how to write them out in a list (i.e., enumerating them) is another, which will be the focus of this problem.

Problem

Recall the definition of Newick format from "Distances in Trees" as a way of encoding trees. See **Figure 1** for an example of Newick format applied to an unrooted binary tree whose five leaves are labeled (note that the same tree can have multiple Newick representations).

Given: A collection of species names representing n taxa.

Return: A list containing all unrooted binary trees whose leaves are these n taxa. Trees should be given in Newick format, with one tree on each line; the order of the trees is unimportant.

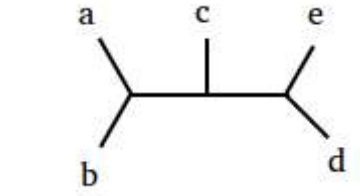

Figure 1. This unrooted binary tree may be represented in Newick format by (((a,b),c),(d,e)); another way of encoding it is ((a,b),(c,(d,e))).

Sample Dataset

```
dog cat mouse elephant
```

Sample Output

```
(((mouse,cat),elephant))dog;
(((elephant,mouse),cat))dog;
(((elephant,cat),mouse))dog;
```

Problem 42

Newick Format with Edge Weights

Weighting the Tree

A vital goal of creating phylogenies is to quantify a molecular clock that indicates the amount of evolutionary time separating two members of the phylogeny. To this end, we will assign numbers to the edges of a tree so that the number assigned to an edge represents the amount of time separating the two species at each end of the edge. More generally, the evolutionary time between any two species will be given by simply adding the individual times connecting the nodes.

Problem

In a **weighted tree**, each edge is assigned a (usually positive) number, called its **weight**. The distance between two nodes in a weighted tree becomes the sum of the weights along the unique path connecting the nodes.

To generalize Newick format to the case of a weighted tree T, during our repeated "key step," if leaves $v_1, v_2, \ldots, v_n$ are neighbors in T, and all these leaves are incident to u, then we replace u with $(v_1 : d_1, v_2 : d_2, \ldots, v_n : d_n)u$ where d_i is now the weight on the edge $\{v_i, u\}$.

Given: A collection of n weighted trees ($n \leq 40$) in Newick format, with each tree containing at most 200 nodes; each tree T_k is followed by a pair of nodes x_k and y_k in T_k.

Return: A collection of n numbers, for which the kth number represents the distance between x_k and y_k in T_k.

Sample Dataset

```
(dog:42,cat:33);
cat dog

((dog:4,cat:3):74,robot:98,elephant:58);
dog elephant
```

Sample Output

```
75 136
```

Problem 43

Counting Unrooted Binary Trees

Counting Trees

A natural question is to be able to count the total number of distinct unrooted binary trees having n leaves, where each leaf is labeled by some taxon. Before we can count all these trees, however, we need to have a notion of when two such trees are the same.

> Our tool will be the split. Recall from "Creating a Character Table" that removing any edge from a tree T separates its leaves into sets S and S^c, so that each edge of T can be labeled by this split $S \mid S^c$. As a result, an unrooted binary tree can be represented uniquely by its collection of splits.

Problem

Two unrooted binary trees T_1 and T_2 having the same n labeled leaves are considered to be equivalent if there is some assignment of labels to the internal nodes of T_1 and T_2 so that the adjacency lists of the two trees coincide. As a result, note that T_1 and T_2 must have the same splits; conversely, if the two trees do not have the same splits, then they are considered distinct.

Let $b(n)$ denote the total number of distinct unrooted binary trees having n labeled leaves.

Given: A positive integer n ($n \leq 1000$).

Return: The value of $b(n)$ modulo 1,000,000.

Sample Dataset

```
5
```

Sample Output

```
15
```

Problem 44

Locating Restriction Sites

The Billion-Year War

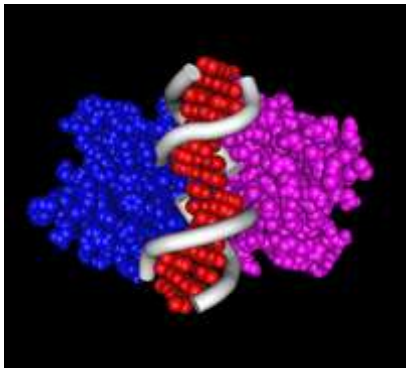

Figure 1. DNA cleaved by EcoRV restriction enzyme

The war between viruses and bacteria has been waged for over a billion years. Viruses called **bacteriophages** (or simply phages) require a bacterial host to propagate, and so they must somehow infiltrate the bacterium; such deception can only be achieved if the phage understands the genetic framework underlying the bacterium's cellular functions. The phage's goal is to insert DNA that will be replicated within the bacterium and lead to the reproduction of as many copies of the phage as possible, which sometimes also involves the bacterium's demise.

To defend itself, the bacterium must either obfuscate its cellular functions so that the phage cannot infiltrate it, or better yet, go on the counterattack by calling in the air force. Specifically, the bacterium employs aerial scouts called **restriction enzymes**, which operate by cutting through viral DNA to cripple the phage. But what kind of DNA are restriction enzymes looking for?

The restriction enzyme is a **homodimer**, which means that it is composed of two identical substructures. Each of these structures separates from the restriction enzyme in order to bind to and cut one strand of the phage DNA molecule; both substructures are pre-programmed with the same target string containing 4 to 12 nucleotides to search for within the phage DNA (see **Figure 1**.). The chance that both strands of phage DNA will be cut (thus crippling the phage) is greater if the target is located on both strands of phage DNA, as close to each other as possible. By extension, the best chance of disarming the phage occurs when the two target copies appear directly across from each other along the phage DNA, a phenomenon that occurs precisely when the target is equal to its own reverse complement. Eons of evolution have made sure that most restriction enzyme targets now have this form.

Problem

A DNA string is a **reverse palindrome** if it is equal to its reverse complement. For instance, GCATGC is a reverse palindrome because its reverse complement is GCATGC. See **Figure 2**.

```
5'...GAT|ATC...3'
3'...CTA|TAG...5'
```

Figure 2. Palindromic recognition site

Given: A DNA string of length at most 1 kbp in FASTA format.

Return: The position and length of every reverse palindrome in the string having length between 4 and 12. You may return these pairs in any order.

Sample Dataset

```
>Rosalind_24
TCAATGCATGCGGGTCTATATGCAT
```

Sample Output

```
4 6
5 4
6 6
7 4
17 4
18 4
20 6
21 4
```

Extra Information

You may be curious how the bacterium prevents its own DNA from being cut by restriction enzymes. The short answer is that it locks itself from being cut through a chemical process called **DNA methylation**.

Problem 45

Creating a Restriction Map

Genetic Fingerprinting

Recall that a restriction enzyme cuts the endpoints of a specific interval of DNA, which must form a reverse palindrome that typically has length 4 or 6. The interval of DNA cleaved by a given restriction enzyme is called its recognition sequence.

A single human chromosome is so long that a given recognition sequence will occur frequently throughout the chromosome (recall from "Expected Number of Restriction Sites" that a recognition sequence would be expected to occur several times even in a short chromosome). Nevertheless, the small-scale mutations that create diversity in the human genome (chiefly SNPs) will cause each human to have a different collection of recognition sequences for a given restriction enzyme.

Genetic fingerprinting is the term applied to the general process of forming a limited picture of a person's genetic makeup (which was traditionally cheaper than sequencing). The earliest application of genetic fingerprinting inexpensive enough to be widely used in common applications, like forensics and paternity tests, relied on a process called **restriction digest**. In this technique, a sample of DNA is replicated artificially, then treated with a given restriction enzyme; when the enzyme cuts the DNA at restriction sites, it forms a number of fragments. A second process called **gel electrophoresis** then separates these fragments along a membrane based on their size, with larger pieces tending toward one end and smaller pieces tending toward the other. When the membrane is stained or viewed with an X-ray machine, the fragments create a distinct banding pattern, which typically differs for any two individuals.

These intervals can be thought of simply as the collection of distances between restriction sites in the genome. Before the rapid advances of genome sequencing, biologists wanted to know if they could use only these distances to reconstruct the actual locations of restriction sites in the genome, forming a **restriction map**. Restriction maps were desired in the years before the advent of sequencing, when any information at all about genomic makeup was highly coveted. The application of forming a restriction map from cleaved restriction fragments motivates the following problem.

Problem

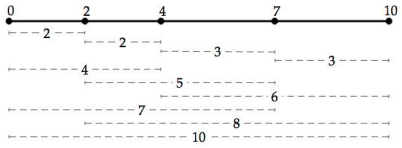

Figure 1. In the simplified figure above, we know that the dashed segments came from a chromosome; we desire a collection of numbers whose differences match the lengths of the dotted lines, which will correspond to the locations of restriction sites on the unknown chromosome. Taken from Jones & Pevzner, An Introduction to Bioinformatics Algorithms.

For a set X containing numbers, the **difference multiset** of X is the multiset ΔX defined as the collection of all *positive* differences between elements of X. As a quick example, if $X = \{2, 4, 7\}$, then we will have that $\Delta X = \{2, 3, 5\}$.

If X contains n elements, then ΔX will contain one element for each pair of elements from X, so that ΔX contains $\binom{n}{2}$ elements (see combination statistic). You may note the similarity between the difference multiset and the Minkowski difference $X \ominus X$, which contains the elements of ΔX and their negatives. For the above set X, $X \ominus X$ is $\{-5, -3, -2, 2, 3, 5\}$.

In practical terms, we can easily obtain a multiset L corresponding to the distances between restriction sites on a chromosome. If we can find a set X whose difference multiset ΔX is equal to L, then X will represent possible locations of these restriction sites. For an example, consult **Figure 1**.

Given: A multiset L containing $\binom{n}{2}$ positive integers for some positive integer n.

Return: A set X containing n nonnegative integers such that $\Delta X = L$.

Sample Dataset

```
2 2 3 3 4 5 6 7 8 10
```

Sample Output

```
0 2 4 7 10
```

Problem 46

Expected Number of Restriction Sites

A Shot in the Dark

In "Locating Restriction Sites", we first familiarized ourselves with restriction enzymes. Recall that these enzymes are used by bacteria to cut through both strands of viral DNA, thus disarming the virus: the viral DNA locations where these cuts are made are known as **restriction sites**. Recall also that every restriction enzyme is preprogrammed with a reverse palindromic interval of DNA to which it will bind and cut, called a **recognition sequence**. These even length intervals are usually either 4 or 6 base pairs long, although longer ones do exist; **rare-cutter enzymes** have recognition sequences of 8 or more base pairs.

In this problem, we will ask a simple question: how does the bacterium "know" that it will probably succeed in finding a restriction site within the virus's DNA? The answer is that the short length of recognition sequences

guarantees a large number of matches occurring *randomly*.

Intuitively, we would expect for a recognition sequence of length 6 to occur on average once every $4^6 = 4,096$ base pairs. Note that this fact does not imply that the associated restriction enzyme will cut the viral DNA every 4,096 bp; it may find two restriction sites close together, then not find a restriction site for many thousand nucleotides.

In this problem, we will generalize the problem of finding an average number of restriction sites to take into account the GC-content of the underlying string being analyzed.

Problem

Say that you place a number of bets on your favorite sports teams. If their chances of winning are 0.3, 0.8, and 0.6, then you should expect on average to win 0.3 + 0.8 + 0.6 = 1.7 of your bets (of course, you can never win exactly 1.7!)

More generally, if we have a collection of events $A_1, A_2, \ldots, A_n$ then the expected number of events occurring is $\Pr(A_1) + \Pr(A_2) + \cdots + \Pr(A_n)$(consult the note following the problem for a precise explanation of this fact). In this problem, we extend the idea of finding an expected number of events to finding the expected number of times that a given string occurs as a substring of a random string.

Given: A positive integer n ($n \leq 1,000,000$), a DNA string s of even length at most 10, and an array A of length at most 20, containing numbers between 0 and 1.

Return: An array B having the same length as A in which $B[i]$ represents the expected number of times that s will appear as a substring of a random DNA string t of length n, where t is formed with GC-content $A[i]$ (see "Introduction to Random Strings").

Sample Dataset

```
10
AG
0.25 0.5 0.75
```

Sample Output

```
0.422 0.563 0.422
```

The Mathematical Details

In this problem, we are speaking of an expected number of events; how can we tie this into the definition of expected value that we already have from "Calculating Expected Offspring"?

The answer relies on a slick mathematical trick. For any event A, we can form a random variable for A, called an **indicator random variable** I_A. For an outcome x, $I_A(x) = 1$ when x belongs to A and $I_A(x) = 0$ when x belongs to A^c.

For an indicator random variable $I_A(x) = 1$, verify that $\mathrm{E}(I_A) = \Pr(A)$.

You should also verify from our original formula for expected value that for any two random variables X and Y, $\mathrm{E}(X + Y)$ is equal to $\mathrm{E}(X) + \mathrm{E}(Y)$. As a result, the expected number of events $A_1, A_2, \ldots, A_m$ occurring, or $\mathrm{E}(I_{A_1} + I_{A_2} + \cdots + I_{A_m})$ reduces to $\Pr(A_1) + \Pr(A_2) + \cdots + \Pr(A_m)$

Problem 47

Local Alignment with Scoring Matrix

Aligning Similar Substrings

Whereas global alignment (see "Global Alignment with Scoring Matrix") can be helpful for comparing genetic strings of similar length that resemble each other, often we will be presented with strings that are mostly dissimilar except for some unknown region of the strings, which may represent a shared gene. To find such genes, we need to modify global alignment to instead search for shared motifs in the form of locally similar regions (recall "Finding a Shared Motif" and "Finding a Shared Spliced Motif").

Using global alignment often fails to find shared motifs hidden in larger strings because (especially if the similar region is found on different ends of the string) aligning the strings causes gap penalties to rack up.

If we are only interested in comparing the regions of similarity, then we would like to have some way of disregarding the parts of the strings that don't resemble each other. The way to do this is to produce alignment scores for all possible pairs of substrings.

Problem

A **local alignment** of two strings s and t is an alignment of substrings r and u of s and t, respectively. Let $\mathrm{opt}(r, u)$ denote the score of an optimal alignment of r and u with respect to some predetermined alignment score.

Given: Two protein strings s and t in FASTA format (each having length at most 1000 aa).

Return: A maximum alignment score along with substrings r and u of s and t, respectively, which produce this maximum alignment score (multiple solutions may exist, in which case you may output any one). Use:

- The PAM250 scoring matrix.
- Linear gap penalty equal to 5.

Sample Dataset

```
>Rosalind_80
MEANLYPRTEINSTRING
>Rosalind_21
PLEASANTLYEINSTEIN
```

Sample Output

```
23
LYPRTEINSTRIN
LYEINSTEIN
```

Problem 48

Local Alignment with Affine Gap Penalty

Building Upon Local Alignments

We have thus far worked with local alignments with a linear gap penalty and global alignments with affine gap penalties (see "Local Alignment with Scoring Matrix" and "Global Alignment with Scoring Matrix and Affine Gap Penalty").

> It is only natural to take the intersection of these two problems and find an optimal local alignment given an affine gap penalty.

Problem

Given: Two protein strings s and t in FASTA format (each having length at most 10,000 aa).

Return: The maximum local alignment score of s and t, followed by substrings r and u of s and t, respectively, that correspond to the optimal local alignment of s and t. Use:

- The BLOSUM62 scoring matrix.
- Gap opening penalty equal to 11.
- Gap extension penalty equal to 1.

If multiple solutions exist, then you may output any one.

Sample Dataset

```
>Rosalind_8
PLEASANTLY
>Rosalind_18
MEANLY
```

Sample Output

```
12
LEAS
MEAN
```

Problem 49

Construct the Burrows-Wheeler Transform of a String

Our goal is to further improve on the memory requirements of the suffix array introduced in "Construct the Suffix Array of a String" for multiple pattern matching.

Given a string *Text*, form all possible **cyclic rotations** of *Text*; a cyclic rotation is defined by chopping off a suffix from the end of *Text* and appending this suffix to the beginning of *Text*. Next — similarly to suffix arrays — order all the cyclic rotations of *Text* lexicographically to form a |*Text*| × |*Text*| matrix of symbols that we call the **Burrows-Wheeler matrix** and denote by *M*(*Text*).

Note that the first column of *M*(*Text*) contains the symbols of *Text* ordered lexicographically. In turn, the second column of *M*(*Text*) contains the second symbols of all cyclic rotations of *Text*, and so it too represents a (different) rearrangement of symbols from *Text*. The same reasoning applies to show that any column of *M*(*Text*) is some rearrangement of the symbols of *Text*. We are interested in the last column of *M*(*Text*), called the **Burrows-Wheeler transform** of *Text*, or *BWT*(*Text*).

Burrows-Wheeler Transform Construction Problem

Construct the Burrows-Wheeler transform of a string.

Given: A string *Text*.

Return: *BWT*(*Text*).

Sample Dataset

```
GCGTGCCTGGTCA$
```

Sample Output

```
ACTGGCT$TGCGGC
```

Extra Dataset

Click for an extra dataset

> **Note**
>
> Although it is possible to construct the Burrows-Wheeler transform in O(|*Text*|) time and space, we do not expect you to implement such a fast algorithm. In other words, it is perfectly fine to produce BWT(*Text*) by first producing the complete Burrows-Wheeler matrix M(*Text*).

Problem 50

Reconstruct a String from its Burrows-Wheeler Transform

In "Construct the Burrows-Wheeler Transform of a String", we introduced the Burrows-Wheeler transform of a string *Text*. In this problem, we give you the opportunity to reverse this transform.

Inverse Burrows-Wheeler Transform Problem

Reconstruct a string from its Burrows-Wheeler transform.

Given: A string *Transform* (with a single "$" sign).

Return: The string *Text* such that *BWT*(*Text*) = *Transform*.

Sample Dataset

```
TTCCTAACG$A
```

Sample Output

```
TACATCACGT$
```

Extra Dataset

Click for an extra dataset

Problem 51

Constructing a De Bruijn Graph

Wading Through the Reads

Because we use multiple copies of the genome to generate and identify reads for the purposes of fragment assembly, the total length of all reads will be much longer than the genome itself. This begs the definition of **read coverage** as the average number of times that each nucleotide from the genome appears in the reads. In other words, if the total length of our reads is 30 billion bp for a 3 billion bp genome, then we have 10x read coverage.

To handle such a large number of k-mers for the purposes of sequencing the genome, we need an efficient and simple structure.

Problem

Consider a set S of $(k+1)$-mers of some unknown DNA string. Let S^{rc} denote the set containing all reverse complements of the elements of S. (recall from "Counting Subsets" that sets are not allowed to contain duplicate elements).

The **de Bruijn graph** B_k of order k corresponding to $S \cup S^{\text{rc}}$ is a digraph defined in the following way:

- Nodes of B_k correspond to all k-mers that are present as a substring of a $(k+1)$-mer from $S \cup S^{\text{rc}}$.
- Edges of B_k are encoded by the $(k+1)$-mers of $S \cup S^{\text{rc}}$ in the following way: for each $(k+1)$-mer r in $S \cup S^{\text{rc}}$, form a directed edge $(r[1:k], r[2:k+1])$.

Given: A collection of up to 1000 DNA strings of equal length (not exceeding 50 bp) corresponding to a set S of $(k+1)$-mers.

Return: The adjacency list corresponding to the de Bruijn graph corresponding to $S \cup S^{\text{rc}}$.

Sample Dataset

```
TGAT
CATG
TCAT
ATGC
CATC
CATC
```

Sample Output

```
(ATC, TCA)
(ATG, TGA)
(ATG, TGC)
(CAT, ATC)
(CAT, ATG)
(GAT, ATG)
(GCA, CAT)
(TCA, CAT)
(TGA, GAT)
```

Problem 52

Genome Assembly Using Reads

Putting the Puzzle Together

In practical genome sequencing, even if we assume that reads have been sequenced without errors, we have no idea of knowing immediately the particular strand of DNA a read has come from.

> Also, our reads may not have the same length. In 1995, Idury and Waterman proposed a way to boost read coverage and achieve uniform read length by breaking long reads into overlapping k-mers for some fixed value of k. For example, a 100 bp read could be split into 51 overlapping 50-mers.

Problem

A **directed cycle** is simply a cycle in a directed graph in which the head of one edge is equal to the tail of the next (so that every edge in the cycle is traversed in the same direction).

For a set of DNA strings S and a positive integer k, let S_k denote the collection of all possible k-mers of the strings in S.

Given: A collection S of (error-free) reads of equal length (not exceeding 50 bp). In this dataset, for some positive integer k, the de Bruijn graph B_k on $S_{k+1} \cup S_{k+1}^{\text{rc}}$ consists of exactly two directed cycles.

Return: A cyclic superstring of minimal length containing every read or its reverse complement.

Sample Dataset

```
AATCT
TGTAA
GATTA
ACAGA
```

Sample Output

```
GATTACA
```

Note

The reads "AATCT" and "TGTAA" are not present in the answer, but their reverse complements "AGATT" and "TTACA" are present in the circular string (GATTACA).