15-110 Hw6 - Protein Sequencing

Hw6 and its check-ins are organized differently from the other assignments. If you haven't already done so, you should read the **Hw6 General Guide** to understand how this assignment works.

Project Description

In this project, you will use data analysis to process and analyze DNA sequences for the gene p53, which is used to suppress cancer in organisms. Specifically, you will compare the p53 genes in humans and elephants, to identify what they have in common and how they are different.

To do this, you will interpret DNA data files from the NIH, convert them to RNA, then convert that to a sequence of generated proteins. You will then compare the protein sequences of the two different genes, automatically generate text reports of your findings, and graph the results.

Note that basic knowledge of DNA/RNA/proteins is helpful for this project, but not required.

Click on the following links to read the instructions for each week's assignment:

Hw6 Check-in 1 - due Monday 11/18

Hw6 Check-in 2 - due Monday 11/25

Hw6 - due Wednesday 12/04

Hw6 Check-in 1 - due Monday 11/18

In the first stage of the project, you will download DNA data from NIH's website, convert it to RNA, and then convert that to a sequence of proteins. During this process you will output how many bases exist in each sequence, how many of those bases are actually used, and how many proteins are synthesized from the sequence.

Step 0: Written Assignment **[45pts]**

In addition to completing the steps described below, there is a short written assignment on the week's material. You can find the written assignment on Gradescope, and linked on the Assignments page of the course website.

Step 1: Get the DNA data **[10pts]**

First, you need to retrieve the NIH data you will analyze in this project. To do this, go to the NIH gene database website, and retrieve the following webpages:

- Human p53: https://www.ncbi.nlm.nih.gov/nuccore/NC_000017.11?report=fasta&from=7668402&to=7687550&strand=true
- Elephant p53: https://www.ncbi.nlm.nih.gov/nuccore/NW_003573467.1?report=fasta&from=11687413&to=11699835&strand=true

Note that DNA is composed of four bases: cytosine (C), guanine (G), adenine (A), and thymine (T). These bases, when read together, produce instructions that the organism can follow in order to create useful things. We represent the connected sequence of bases in a text file with a single letter per base.

Copy the DNA text from each page into a text file in your project's folder. Make sure to ONLY copy in the DNA, not the surrounding text! You should name the file with the human DNA "human_p53.txt", and the file with the elephant DNA "elephant_p53.txt".

Now implement the function readFile(filename) included in the starter file. Given a filename, read the text from that file into a variable. Remove any newlines ('\n') from the text (potentially by using a string method we went over in class), then return it. When called on a DNA file, this will return a string holding all the DNA in the file.

To test this function, run testReadFile(). This function assumes you have named your data files appropriately, and included them in the same directory as the starter file.

Step 2: Convert DNA to RNA **[10pts]**

Next, you need to convert the DNA string to a string of RNA, which will be used to follow the instructions of the DNA. However, this process is not done by reading all of the DNA string. Instead, the organism breaks down the DNA into groups of three bases (called codons). One codon, ATG, signals the Start of an RNA strand; three other codons (TAA, TAG, and TGA) signal the end (Stop).

Implement the function dnaToRna(dna, startIndex) included in the starter file. You can assume that the start point for the RNA has already been found (startIndex); your task is to read codons from that start point until the end point is found, and return a list of all the codons in between. To do this, loop through the DNA sequence and produce codons as you go, by combining together every three DNA bases in a row into a codon. Add each codon to a list as a three-character string. As soon as you have added a Stop codon (or run out of DNA bases to read), return the list of codons.

One final note- for a variety of reasons, RNA uses U as a base instead of T. You'll need to replace every T base you find with a U instead. This means you'll need to check for Stop codons UAA, UAG, and UGA.

To test this function, run testDnaToRna().

Step 3: Make a Codon Dictionary **[10pts]**

To turn RNA into proteins, we'll need to know which amino acid each codon corresponds to. We've provided this information in the file codon_table.json, which came as part of your starter zip file. Unfortunately, this data is not formatted exactly as you need it to be; it maps amino acids to lists of codons, but you need to map each codon to a amino acid.

Implement the function makeCodonDictionary() in the starter file. First, read the contents of the json file into a dictionary (we'll call in aminoD) by calling json.load() on the codon table filename. Then use the loaded dictionary (which maps amino acids to lists of codons) to generate a new dictionary (which will map codons to amino acids). Make sure to change all Ts in the codons to Us as you do this!

To test this function, run testMakeCodonDictionary().

Note: make sure that you store codon_table.json in the same directory as your starter file; we'll expect it to be there for autograding purposes.

Step 4: Convert RNA to Proteins **[10pts]**

To turn RNA into proteins, we'll need to identify each codon in the RNA sequence and add its associated amino acid to the chain. This chain of amino acids will become our protein.

Implement the function generateProtein(rna, codonD) in the starter file. This takes an RNA sequence (a list of three-character strings, the codons) and the codon dictionary, and returns a protein (a list of amino acid strings). To do this, go through each codon in the RNA list, and add its associated amino acid (based on the codonD) to a new list.

Note that you'll need to special-case the first codon in the list. If it is AUG, you should add "Start" to the protein instead of AUG's amino acid (Met), since Met encodes the start of a sequence in this circumstance.

To test this function, run testGenerateProtein().

Note: technically, we aren't producing 100% accurate proteins with this system. We're skipping a step in the translation process, where the RNA is 'spliced' to remove unnecessary portions of the strand, based on introns and exons. Unfortunately, there's no simple rule to detect where an intron or exon is; in fact, there are whole research teams dedicated to this question! We'll just produce slightly-inaccurate proteins for now.

Step 5: Synthesize Proteins **[15pts]**

Finally, we need to put all of the previous steps together in order to synthesize proteins from our data file. This is where we start processing real data!

Implement the function synthesizeProteins(filename) in the starter file. This program should read the DNA from the given filename (using readFile()) and print out the total number of bases in that DNA (the length of the string). It should also produce a codon dictionary by calling makeCodonDictionary().

The program should then identify all of the RNA strands that can be produced from the DNA by iterating through all the indexes in the DNA string, looking for the start code (ATG) at each point. Note that you'll need to keep track of a list of proteins and a count variable outside of the loop.

- If you identify an index in the DNA that corresponds to ATG, call dnaToRna starting from that index to extract the entire RNA sequence, then call generateProtein on the resulting RNA (and codon dictionary) to produce a protein. That protein should be added to an overall protein list. Then update the index in the DNA strand to skip past all the already-checked bases (by adding 3 * the length of the RNA strand).
- If you get to an index that does not correspond to ATG, add one to the index, and also add one to the count variable, as this is an unused base.

When you finish looping, you'll have two useful pieces of information: a list of all the proteins synthesized from the DNA, and a count of all the bases that were not used. Print out the unused-base count, as well as the total number of proteins synthesized. Then return the list of proteins.

To test this function, run the provided function runWeek1(). You should find that the human DNA synthesizes to 119 proteins (with 10560 unused bases), and the elephant DNA synthesizes to 77 proteins (with 6204 unused bases). Cool!

Hw6 Check-in 2 - due Monday 11/25

In the second stage of the project, you will analyze the protein sequences generated in Stage 1, to determine what the biggest commonalities and biggest differences are between the two sequences. You will then automatically produce a text report showcasing these results.

Step 0: Written Assignment **[45pts]**

In addition to completing the steps described below, there is a short written assignment on the week's material. You can find the written assignment on Gradescope, and linked on the Assignments page of the course website.

Step 1: Find Common Proteins **[10pts]**

First, you need to determine if there are any proteins that occur in both genes, and what those proteins are. This will help determine how similar the two genes actually are.

Implement the function commonProteins(proteinList1, proteinList2) in the starter file. This function takes two lists of proteins (where each protein is a list of amino acids) and returns a list of all the unique proteins that occur in both lists. Each protein should only occur once in the result list, even if it shows up multiple times in both genes.

To test this function, run testCommonProteins().

Step 2: Combine Protein Lists **[5pts]**

It turns out that there aren't many proteins in common between our two genes at all. Therefore, it's more interesting for us to consider what the biggest differences between the two genes are. More specifically, we want to compare the amino acids generated by the two genes, to see if anything in particular occurs more often in humans than elephants, or vice versa.

To do this comparison, we first need to collapse the list of proteins into a list of the amino acids that occur across all the proteins. Implement the function combineProteins(proteinList) that takes a list of proteins (where each protein is a list of amino acid strings) and returns a list of all the amino acids that occur across all the proteins, in their original order. In other words, this function inputs a 2D list of strings and outputs a 1D list of strings by ‘flattening’ the original list.

To test this function, run testCombineProteins().

Step 3: Generate Amino Acid Dictionary **[5pts]**

Once we have a list of amino acids, we can use it to generate a dictionary that maps each amino acid in the list to a count of how often it occurs. We'll use this dictionary to determine the frequencies of each amino acid- how common is each type in the gene?

Implement the function aminoAcidDictionary(aaList) in the starter file. This takes a list of amino acids (strings), aaList, and returns a dictionary that maps each amino acid to how often it occurs in the list.

To test this function, run testAminoAcidDictionary().

Step 4: Sort Amino Acids by Frequency **[10pts]**

Now that we know how common each amino acid is, we can start comparing amino acids between genes of different lengths. To do this, we need to be able to generate a list of amino acids ordered by frequency- in other words, a list that has the rarest amino acids at the front and the most common amino acids at the back. If an amino acid occurs in very different places between two frequency list, it's a major difference between the two organisms.

Implement the function sortAminoAcidsByFreq(aaList) in the starter file, which takes a list of amino acids and produces a sorted frequency list (a 2D list). This list should contain a two-element list for each amino acid in the dictionary. The first element should be the frequency of the list (the number of times it occurs divided by the total number of amino acids); the second element should be the amino acid itself. You should use your aminoAcidDictionary() function to make solving this problem easier.

For this analysis, we don't care about how often the Start and Stop codons occur. Make sure those are not added to the list.

To test this function, run testSortAminoAcidsByFreq().

Step 5: Find Amino Acid Differences **[10pts]**

Now we have everything we need to find the biggest differences between the two genes. We'll generate frequency lists for the two genes, identify amino acids that occur at different indexes between the two lists, and return a list of those amino acids (along with their frequencies).

Implement the function findAminoAcidDifferences(proteinList1, proteinList2) in the starter file. This takes two protein lists and returns a list of three-element lists, where the first element in the list is an amino acid, the second element is the frequency of that amino acid in proteinList1, and the third element is the frequency of that amino acid in proteinList2. You should only include amino acids in this returned list if they have sufficiently different frequencies across the two lists, as defined below.

To generate this list, you should first use your combineProteins() and sortAminoAcidsByFreq() functions to generate a frequency list for each protein list. Then go through each amino acid in the lists. An amino acid has sufficiently different frequencies if it A) occurs at different indexes in the two frequency lists, and B) the difference between those two frequencies is greater than some threshold. We'll set the threshold at 0.005 (or 0.5%) for this analysis.

To test this function, run testFindAminoAcidDifferences().

Step 6: Generate Text Report **[15pts]**

Now that we have working functions for finding the major commonalities and differences between genes, we need to display the results to the user. Implement the function displayTextResults(commonalities, differences) in the starter file. This function takes two lists - the list of common proteins between two genes, and the list of most-different amino acids in the two genes - and prints out a textual report of those results.

First, you should print out the common proteins. This part of the report should include a short piece of text stating that these are the common proteins, and should display the proteins in a readable format (not just as printed lists). You should also omit any two-codon proteins in the list, as these are just empty ["Start", "Stop"] proteins.

Then, you should print out the most different amino acids. This part of the report should include a short piece of text stating that these are the amino acids that occurred at the most different rates, and for each amino acid should print out that rate for each sequence in a readable format (again, not just a printed list).

For each part of the report, you may choose to personalize the text, as long as you follow the requirements above. To test your function, run the provided function runWeek2(). Here's what our function produced; yours does not need to look 100% identical, but should have approximately the same results.

```
The following proteins occurred in both DNA Sequences:
Ala
Gly
Lys
Ser-Pro-Leu
Thr

The following amino acids occurred at very different rates in the two DNA sequences:
Tyr : 2.06% in Seq1, 2.6% in Seq2
Met : 2.13% in Seq1, 3.09% in Seq2
Ile : 3.07% in Seq1, 2.46% in Seq2
Phe : 3.98% in Seq1, 5.26% in Seq2
Thr : 4.02% in Seq1, 4.82% in Seq2
Lys : 4.96% in Seq1, 4.2% in Seq2
Arg : 6.01% in Seq1, 4.63% in Seq2
```

Hw6 - due Wednesday 12/04

In the final stage of the project, you will take the analysis results from Stage 2 and use them to produce a bar chart which visually demonstrates the different frequency rates of amino acids in the two p53 genes under investigation. You will use the module matplotlib to do this visualization.

Step 0-A: Complete Check-in 1 **[20pts]**

If you got a perfect score on Hw6 Check-in 1 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check-in 1 and use it to update your Check-in 1 code. This is your chance to implement any features you might have missed before, and fix any code that isn't working.

Step 0-B: Complete Check-in 2 **[20pts]**

If you got a perfect score on Hw6 Check-in 2 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check-in 2 and use it to update your Check-in 2 code. This is your chance to implement any features you might have missed before, and fix any code that isn't working.

Step 1: Install matplotlib and numpy **[0pts]**

In order to use the matplotlib library, you will need to install it on your machine. If you do not have a personal computer, note that the cluster machines on Gates 5 should have matplotlib installed already.

To install matplotlib and one of its dependencies, numpy, we recommend that you use the pip tool included in your Python installation. This tool will manage the installation process for you, which is much easier than trying to install a module manually. To use pip, open Terminal on a Mac/Linux or PowerShell on Windows. Then run the following lines of code:

```
pip install numpy
pip install matplotlib
```

If an error message occurs, try googling it to find a solution. TAs can also help debug installation errors via Piazza or in office hours.

Note that pip is associated with the default version of python on your computer. If you have multiple versions installed (which is often true of Macs, as they come with Python 2.7 installed by default), you will have to run a different command to install the libraries into the version of Python you use. These commands might work:

```
python3 -m pip install numpy
python3 -m pip install matplotlib
```

You can test whether the modules have installed into the correct version of Python by running the following commands in your interpreter. If they do not give you an error, you're good to go!

```
import numpy
import matplotlib
```

Step 2: Generate Chart Labels **[10pts]**

In order to plot our data using matplotlib, we first need to reformat it to fit the input that matplotlib expects. This first part of this will involve making labels for the graph based on all the amino acids that will be graphed.

Implement the function makeAminoAcidLabels(geneList) in the starter file. This takes a list of genes (where each gene is a protein list) and finds all the amino acids that occur across all the genes. This will let us compare more than two genes at a time, and accounts for amino acids that might occur in one gene but not the other. It should return a sorted list of all the amino acids found.

We suggest that you use the previous functions combineProteins() and aminoAcidDictionary() to simplify this function.

To test this function, run testMakeAminoAcidLabels().

Step 3: Generate Chart Data **[15pts]**

Next, we need to process our data into the format that matplotlib will accept for data values. Specifically, we need to generate a list of lists, where each list's index i contains the frequency for the i'th element of the labels list. So if Ala is in the 0th index of the labels list for our two p53 genes and result is our result list, result[0][0] would be the frequency of Ala in the human gene, and result[1][0] would be the frequency of Ala in the elephant gene.

Implement the function setupChartData(labels, geneList) in the starter file. This takes a labels list (produced by makeAminoAcidLabels()) and a list of genes (where each gene is a protein list), and returns a list of frequency lists, as defined in the previous paragraph. If an amino acid does not occur in the gene, set its frequency to 0.

We suggest that you again use the previous functions combineProteins() and aminoAcidDictionary() to simplify this function.

To test this function, run testSetupChartData().

Step 4: Create a Bar Chart **[15pts]**

Now that we have labels and data, we can actually draw the bar graph. For this, we'll need to write a lot of matplotlib code. We recommend that you begin by going over the example bar chart code from the data analysis lecture on visualization, and that you review the following example program from matplotlib's website:
https://matplotlib.org/gallery/lines_bars_and_markers/barchart.html

Implement the bar chart code in the function createChart(xLabels, freqList, freqLabels, edgeList=None) in the starter file. xLabels is the list of amino acid labels (generated by makeAminoAcidLabels()); freqList is the 2D list of amino acid frequencies (generated by setupChartData()); freqLabels is a list of strings, where each string is the name of the gene represented by the corresponding list in freqLists. We'll discuss edgeList in more depth in the next step; for now, you can ignore it.

We want to make a bar chart that has a category for every gene in the geneList, and an x value for every label in the label list. Therefore, you should set

x = numpy.arange(len(xLabels))

To make the graph look nice, we need to set a buffer that will resize based on the number of genes in the freqList. To do this, we can set the bar width with the code:

width = 0.6 / len(freqLists)

Then, instead of repeating a line of code for two different groups (as is done in the matplotlib website example), use a loop to iterate over each of the list indexes in the freqLists input. If i is the list's index, we can define the position for each bar call with

offset = - 0.3 + width/2
ax.bar(x + offset + i*width, …

You should be able to fill in the rest from the examples mentioned above.

To test this function, run testCreateChart(). You should produce a reasonable-looking graph from the example data!

Step 5: Outline Different Edges **[10pts]**

Finally, to make our amino acids chart a little more advanced, we'll incorporate the results of our analysis from Stage 2. We'll set the edge color of the amino acid bars which are sufficiently different to be a different color than the bars of the rest of the amino acids, to make them stand out.

Implement the function makeEdgeList(labels, biggestDiffs) in the starter file. This takes a list of labels (generated by makeAminoAcidLabels()) and a list of biggest-difference amino acids (generated by findAminoAcidDifferences()). This function returns a new list, the same length as the labels list, where each element of the list is "black" if the corresponding amino acid is in the biggestDiffs list, and "white" otherwise.

Recall that biggestDiffs contains three-element lists, and that the first element of each three-element list is the amino acid. You'll have to check that part of the inner list to see if biggestDiffs contains a specific amino acid.

To test this function, run testMakeEdgeList().

Once your makeEdgeList function is working, update your createChart function so that the ax.bar call sets the optional parameter edgecolor equal to edgeList. This will update the edge colors based on your generated list, if you pass it into the function as a parameter!

Step 6: Put It All Together **[10pts]**

You've finally finished implementing all the features of the project. Now, you just need to put them all together.

Implement the function runFullProgram() in the starter file. This function should load the DNA data in your two p53 files, process them both into protein lists, generate a text report comparing the two genes, and then generate a bar chart comparing the two genes (with the sufficiently-different amino acids outlined in black).

When you run this function, you should be able to view the results of all of your analysis. Congratulations- you're done!