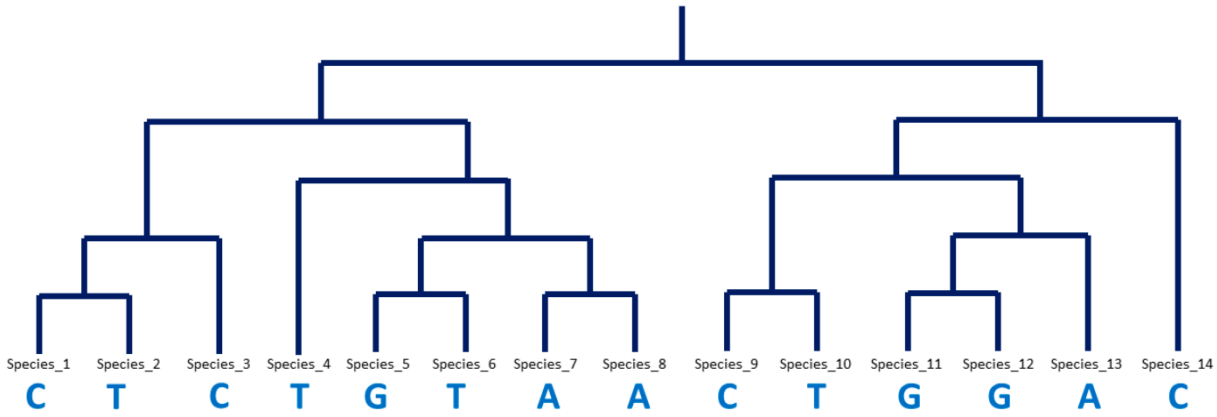
PROBLEM SET 7

Due Friday, February 28, by 3:00PM through Canvas. Assignments turned in more than 5 minutes after the beginning of class will be penalized 10 points, with an additional 10 points every 24 hours thereafter.

PART A. ALGORITHMS (20 POINTS)
1. (20 points)

1a. (10 points) Apply Fitch's algorithm to the following tree (with the shown nucleotide assignment for the tips). In your answer show the results of both the bottom-up phase (i.e., the set of possible states in each internal node) and the top-down phase (i.e., an actual assignment for each of the internal nodes). Finally, what is the parsimony score for this tree?



1b. (10 points) If you were now given an additional species (species_15) with some nucleotide assignment and were told that this species is known to be an outgroup (first diverging species) to the above 14 species, by how much could the parsimony score change if you will add this species to the tree? Can the parsimony score go down? Would you have to apply Fitch's algorithm from scratch to get the new parsimony score?

PART B. PROGRAMMING (80 POINTS)

2. (40 points) Parts a-d build upon one program. Please turn in a single program, **snp_print_rsIDs.py,** for problem 2. The others are intermediates which will be incorporated in this final program.

SNPs (single nucleotide polymorphisms) are locations in the genome where more than one base commonly occurs within the population. For example, at position 6204892 on chromosome 1, 60% of the population may have an A, while the other 40% have a G. The file **SNPs_chr19.txt** was generated from a database of commonly occurring SNPs. The file is shown below. Each common SNP has an SNP ID assigned to it, and the major (more common) and minor (less common) base(s) at that position are shown.

```
chr     pos     SNP_ID  Major   Minor
19      62935   rs534193774     CCT     C
19      64705   rs559839262     C       T
19      69984   rs372156287     G       A,C
19      80840   rs201816663     CCT     C
19      80849   rs564694216     G       C,T
19      89304   rs376221143     G       A
19      89430   rs570626439     C       G
```

2a. (10 points) You have generated RNA-seq data from a cell line, counted the number of A's, C's, G's, and T's at each position on chromosome 19, and printed a subset of these counts into a file called **data_file_chr19.txt**. You would like to count how many genomic locations in this file correspond to locations of common SNPs annotated in **SNPs_chr19.txt.** Please write a program (count_SNP_positions.py – will not be turned in) which counts the genomic positions in data_file_chr19.txt which are also present in SNPs_chr19.txt, and prints the count as a percent of total sequences in the data_file_chr19.txt. (Don't worry about the bases at these positions just yet.) You can assume for this part of the problem that only positions on chr19 are shown in both files.

HINT: Start by making a dictionary of known SNP positions from SNPs_chr19.txt, and use it to look up positions in the data file. Think about why a dictionary is a better choice than a list.

INPUT:
> python count_SNP_positions.py SNPs_chr19.txt data_file_chr19.txt

OUTPUT: (Please show percent to 2 decimal places)
2.31 % of positions in data file are common SNPs

A note on dictionaries (since I've seen some overly complicated dictionary notation):

```
>>> my_dict = {}
>>> my_key = "GEN559"
>>>my_dict[my_key] = "Intro_coding_class"  #this adds the key "GEN559" to the dictionary
>>> my_dict[my_key] #this is how to access a value associated with my_key.
"Intro_coding_class"
```

Since my_dict.keys() is just a list, you can iterate through it like any other list:

```
>>> for item in my_dict.keys():
        print(item)
```

2b. (10 points) Oh no! Your PI just handed you a data file which includes more than one chromosome (data_file_multi_chr.txt). Modify your program from 2a to work on files which containing genomic locations from multiple chromosomes. Notice, the chromosome name format is different in the data and reference file (SNPs_multi_chr.txt). This is a very common problem in genomics pipeline development. You can assume that chromosome names are the same except for the added prefix "chr" in the data file, but don't assume you know which chromosomes are included ahead of time. Write a program called **count_SNP_pos_multiple_chrs.py.**

HINT: Modify your dictionary key to contain both chromosome and position info as a single string.

INPUT:
> python count_SNP_pos_multiple_chrs.py SNPs_multi_chr.txt data_file_multi_chr.txt

OUTPUT:
0.50 % of positions in data file are common SNPs

2c. (10 points) Now you would like to know whether the base most commonly observed at each location in your data file corresponds to the major allele for that position in the reference SNP file. Look for exact matches only – don't worry about parsing lists of SNPS (i.e. "A,C").

Your program should contain a function called get_base() which takes as input the list of values associated with each base from a single line of the data file and returns the base with the highest count associated with it. If there is a tie, return the first instance.

HINTS:
- Modify your dictionary to contain information about the major allele
- You are welcome to use the max() function to find the maximum value.

INPUT:
> python count_major_alleles.py SNPs_multi_chr.txt data_file_multi_chr.txt

OUTPUT:
0.50 % of positions in data file are common SNPs
64.29 % of these match the major allele

2d. (10 points) Lastly, we want to keep track of the snp_IDs (rs###) associated with the 64.29% of SNPs we identified above, and write just these snp_IDs in a single column to a file called major_rs_ids.txt Add a few lines of code to your program to keep track of the snp_IDs associated with chromosomal positions and print the relevant ones to an output file.

HINT:
Dictionaries values can be lists! You can include both major allele and snp_ID info in your dictionary value and access them just like you would access elements in a list.

Example of dictionaries with lists as values:
>>> my_dict = {}
>>>my_dict[my_key] = ["A", "B","C"] #the value here is this whole list
>>> my_dict[my_key][1]
"B"

INPUT:
> python snp_print_rsIDs.py SNPs_multi_chr.txt data_file_multi_chr.txt rs_ID_output_file.txt

OUTPUT:
0.50 % of positions in data file are common SNPs
64.29 % of these match the major allele
(rs_ID_output_file.txt should now contain a list of all the SNP_IDs associated with the major alleles found in the data. There should be 18).

Turn in the program **snp_print_rsIDs.py.**

3. (40 points) For this problem, we will extract information from a file called **single_cell_RNA_seq.txt.** This file contains mapped RNA-seq reads for many individual cells. Each column is described below:

Column 1: A cell identifier (sort of -- you'll deal with this in part a)
Column 2: Strand information. A "0" means indicated this RNAseq read maps to the + strand; a "16" indicates it maps to the - strand.
Column 3: chromosome
Column 4: location on chromosome
Column 5: sequence

3a. (10 points) The cell identifier in column 1 actually contains some extra information which we would like to remove. Write a function called **generate_cell_name()** which takes as input a string of the format "A10_ATCCATCT_AATCATACGG" and returns "A10_AATCATACGG", removing the letters between the underscores. (Don't forget to test that your function works!)

Use this function in a program called **read_counts.py** which counts the number of sequences associated with each cell and prints them to the terminal in the format shown. Use a dictionary in your program!

INPUT:
> python read_counts.py single_cell_RNA_seq.txt

OUTPUT:
A10_AATCATACGG : 2591
A10_ATAGGAGTAC : 1993
A10_CGAATTCGTT : 1525
A10_GGCTGGCTAG : 1060
A10_ACGGTCTTGC : 2193
A10_GCTCCATTCG : 4983
A10_ACCATAGCGC : 2040
A10_CTCTTAGCGG : 8024

A10_TGATTCAACT : 2248
A10_AGAGGTCGCA : 900
A10_TCGCGTACTT : 2296
A10_AATAATAATG : 2013
A10_AACTACGGCT : 1701
A10_CGCAATATCA : 2070
A10_CCTACGGCAA : 6573
A10_TTCAAGAATC : 2348
A10_ATGCTCGCAA : 1356
A10_TCCTTACCAA : 1613
A10_CATCGCGCTC : 1235

3b. (10 points) We would like to compare the base composition at the beginning and end of each sequencing read. Specifically, we think there might be an enrichment of "A" at the ends of reads and want to know if this is the case. A quick way to check is to calculate the number of A's in the last n bases of each read and compare that to the number of A's in the first n bases.

Write a function called **count_As()** which returns the number of A's in a substring of an input sequence either at the start or end of the sequence. It should take in three arguments:
(1) a sequence
(2) the word "start" or "end"
(3) number of bases to grab from the start or end of the sequence in which to count A's (we'll call this substring length)

If the substring length is longer than the sequence, your function should return a warning which includes the sequence itself (such as "Warning: substring length longer than string ATCAAATAA") and exit (you can just add sys.exit() after the warning statement to do this). You are welcome to add other checks, like whether your function call has the right number of arguments, but don't have to.

Turn in a program called **test_count_A_func.py** containing the function count_As() and some test code you used to make sure it works. (We'll apply this function to our data in part d).

3c. (5 points) We would like to reverse complement the subset of these sequences which are on the minus strand (16 in the second column) before applying the function from part 3b. We have already written a function to do this in class called get_rc(). We can just use this function directly into our program, but let's instead make a module that we can import into our program. Modify the function slightly to accommodate "N's" in the sequence (which should stay N's in the reverse complement) and paste it into a file called dna_fun.py. Save this file in the folder from which you will run your program. Turn in **dna_fun.py** and a (very short) program called **test_your_module.py** in which you import this module and test that you can run get_rc() from within another program. (My program is two lines)

3d. (15 points) Now let's use the functions from parts b and c to write a program which calculates the total number of A's at the beginning and end of the sequences in the file **single_cell_RNA_seq.txt**, taking care to reverse complement the sequences on the minus strand before counting A's. Import the module containing get_rc() into your program. Turn in a program called **count_A_enrichment.py**.

INPUT: (Second argument is the number of bases to at the beginning and end of the sequence to consider)
> python count_A_enrichment.py single_cell_RNA_seq.txt 10

RETURN:
Number of A's at start: 139920
Number of A's at end: ####