

CS4414 Fall 2024: Homework 3, Parts 1 and 2.

4 weeks of homework. One (long) assignment document.

Due dates:

- ☐ **Part 1: Oct 11. (~2 weeks)**
- ☐ **Part 2: Oct 27. (2 weeks)**

Start early... if you finish early, that will be great and you can relax. But this is way better than starting late and not finishing at all. We will not be providing “slack days”. When it is time to turn something in, you should upload what you have at that point. Our TAs will share sample solutions you can use for the next step if you didn't finish the prior one, at a penalty of 10pts.

Part 1: Due October 11

The broad goal of Homework 3 is to gain experience with C++, Linux files, and the development and debugging experience using Visual Studio Code. The actual task arises from something real: given a collection of DNA samples from individual animals, construct an evolutionary tree that minimizes the species-to-species distance, then use a standard display program to visualize the tree that starts at some specified ancestral species¹. We will do this one small step at a time over a period of a few weeks.

For Homework 3 you will be working with a collection of data files that you can download from Canvas and extract into your Linux system. Create a project folder and in it put a subfolder SpeciesData, and extract the files there. This year we are providing some starter code; note that you're expected to **modify and add to these starter files**, according to how you want to complete the assignment (for example, by adding “getter” functions that return private member variables in your Animal class). In addition, you will find files A0.dat-A39.dat and a collection of .png files with names like Jelly_Belly.png and Darwins_Totle.png. Inside a typical data file you would find something like this:
























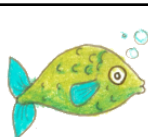







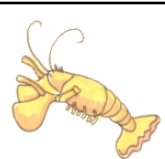



```
Name="Fuzzy Tribble"
LatinName="Tribulum Fuzzius"
ImageFilename="Fuzzy_Tribble.png"
DNA="IAYT... ..IYTYI"
```








The DNA string will be pretty long (about 45,000 characters each). DNA is a string of the characters {I,T,Y,A}. *Genes* are substrings of DNA, starting the special sequence IAY and

¹ An *animal* is an instance of some *species*, so we use both terms here. In your code Species is probably the more appropriate class name because one could imagine having two animals from the same species.

terminated by TYI. The initial IAY and the trailing TYI are not part of the gene. Regions of DNA that are not in a gene are called *non-coding*, and can be discarded. Note that IAY can appear inside a gene (if so, it is part of the gene). An unpaired IAY or TYI can appear in the junk DNA.

Here is our collection of critters:

						
Sprats Butterfly	Bard's Star	Ballard's Hooting Crane	Common Mudfly	Common Snat	Darwins Tortle	Elephant Snark
						
Frlly Sea Sprat	Fuzzy Tribble	Gilligans Squimp	Globe Floater	Gray Floop	Green Herring	Green SnapDragon
						
Hairy Rock Snot	Hallucigenia	Jelly Belly	Larval Treenymph	Leaping Lizard	Long-Snouted Squirk	Munckles Mouse
						
Nocturnal Mourningbird	Nocturnal Plexum	Paradise Rockfish	Biscuit	Pink Ziffer	Policle	Pompous Snarks
						
Nocturnal Mourningbird	Nocturnal Plexum	Paradise Rockfish	Biscuit	Pink Ziffer	Policle	Pompous Snarks

Sextopus	Shy Frecklepuss	Armored Snapper	Asian Lobster	Boxing	Ballards Protoduck	Waldo?	Translucent Tridle
							
Toothy Balloonfish	Swamp Slime	Striped Salamander	Strats Squirrel	Spotted Ghila	Snuffling Blat	Parmesianian	

We will start by reading the 40 files and creating 40 objects, one per file, holding: the name, the latin name, the name of the image file and a list of Gene objects. You'll define the C++ class for animals and genes, and the associated methods, and you will write the code that performs this scanning operation and builds the list. Notice that we are not going to be reading the PNG files – they have nice little pictures of the animals, but most people won't use them in Homework 3.

You may be wondering why we have such nice images and aren't making use of them. This is a side-effect of people using such a variety of computers to log into our remote servers. Portable graphics is a whole art of its own, and in CS4414 we won't be tackling that topic. If you get bored and want to do more than the basic homework, though, do consider adding a feature to generate a web page (an HTML file) for your output data, as an ungraded add-on to the main assignment. If you do that, you can include the icons on the page. People can discuss the HTML aspects on Ed Discussions.

Next, we want you to create a *single* sorted list of all the genes, such that any single gene appears just once in the list, and with the list sorted by length, shortest gene first, longest last. If two genes have the same length, use alphabetical sort order to order them. Once you have this list, give the sorted genes id numbers: 0..n. Thus, each animal will have an object, and one field will be a list of its genes. None of our animals has the same gene multiple times – in each DNA sequence each gene occurs once. These gene objects will each have an id and a genetic code. As a last step run through the animals one by one. For each animal, sort its genes in id-order (smallest to largest).

Now add a printing method to generate output like the one shown below. We won't be deducting for layout, so don't worry about the exact number of spaces or tabs between each field.

Example Input and Output

If you compile your hw3.cpp main file to an executable called "hw3", then your code should run as follows:

At first, it is very unlikely that your code will compile or run correctly. Moreover, you will be learning about new `std::` classes, from the documentation in cppreference.com (and by using Google, if you like). This is all part of learning C++ and we expect you to do this without a huge amount of hand-holding from us, although the TAs are there for you if you get totally stuck. But C++ is a large and sophisticated system, with many tools. Learning to find and use them is all an aspect of programming. One small bit of advice: when you compile and see 500

errors, just focus on fixing the very first one the compiler printed. Often one mistake causes many error messages, so once you fix that first issue, the 500 might drop to 425...

You will need to test your code carefully to make sure that it works properly. Testing is also an important part of the assignment. Most programs start out with bugs. Recognizing that a bug is present, using gdb to pin it down, and then fixing your logic is also part of the task.

Great news! You can reuse some of the code we shared as part of lecture 1!

A number of the features of this program arose in word-count! We recommend that you look at Sagar's wc++ program and "adapt" it into your Homework 3 solution. Use of his code is not required, but it will make life easier. Homework 3, like word-count, involves file scanning, string splitting, `std::vector` for lists of genes (`std::list`, if you prefer – both are options), and `std::map` to create our mapping from gene DNA str to gene-names. Sagar's wc++ can be downloaded [here](#).

Sagar's program also included logic that opens a series of files in a folder (`utils.hpp` and `utils.cpp`). `find_all_files` is very useful (and we've included it in the starter code for `hw3.cpp`). When invoked with the directory in which the search should start as a first argument, and with a function that looks at a file name (a `std::string`) and returns true or false: true if the file should be included in the scan, false if not. It returns a `std::vector<fs::path>` that holds the list of file names that were found, and that matched the pattern. You'll need to write the pattern function, and you'll use a for loop to scan through this list, opening and parsing the files one by one, and remembering to close each file when finished with it. If you read about `std::filename`, you can learn about the `fs::path` type that is being used here. Always consult cppreference.com for information of that kind.

Your program will thus need to use Sagar's file finder to find all the .dat files, load them, and then at the end print out the contents of your set of objects (if you don't, you won't know if your scan operated correctly), and for each animal, the list of genes you found. *Note:* In the future, we will begin using file name arguments to `main()`, but on this homework we will just scan every file in the working directory that ends in .dat.

Don't test on A0-A39 as your very first action. Start by inventing your very own testing data file, TEST.dat, and have a few hand-created genes in it. Use your test file as a tool for debugging your gene parsing code. Debug first, and only test on A0-A39 (with 45000 characters of DNA each) after you are confident that your logic is working correctly!

What could you put in a test file? How about a DNA string with no junk DNA, and just one gene containing one coding letter. For example, with coding letter A, your DNA might be IAYATYI. Verify that your program extracts the gene ("A"). Now try some longer genes. Try a gene with IAY in the coding portion. Next try a DNA sequence with two genes, then four genes. Try an empty gene (IAYTYI). In every case your code needs to give the correct output.

It should never crash or output any sort of garbage. Now try just one animal, perhaps A0.dat. Then try three, as we used in our example.

How would you run these tests? We suggest that you have one folder with all the SpeciesData files, but a second working folder for code development and testing. Sagar's way of scanning files will let you find all the .dat files in the working folder, so you can copy test files in (or remove them) as needed during development. And you can edit your own data files by hand. DNA doesn't need to have 45,000 characters. Test with examples that have 20 or 30 characters in one or two genes.

What to upload for Part 1:

Compiling your code should be similar to Homework 2. Upload a zip file containing your cpp and hpp files, as well as hw3.cpp, which is responsible for generating output like the example given above. No need to include any other files in your zip file.

Part 2: Due October 27

Part 2 is about calculating *gene distance*. The closer two genes are, the more likely it is that one evolved from or into the other, or that they share a recent common ancestor. In our algorithm, matching sections in the two genes contribute nothing to the distance calculation, and a heuristic is used to determine a score for differing sections. The higher the total score, the more distant the two genes are considered to be.

This assignment builds on your solution to Part 1. If you were unable to complete a high-quality solution to Part 1, contact Noam Benson-Tilsen and he will let you have his solution as a starting point for Part 2. *However, we will automatically deduct 10 points from your final score if you base Part 2 on Noam's code.*

Short description

In a nutshell, Part 2 involves three algorithms. We provide a simple rule for computing gene-to-gene distances, and you implement it. Then we provide a simple rule for computing species-to-species distances, and you implement it as well. Finally, we use a famous shortest-paths tree algorithm called Prim's to build an evolutionary tree, starting at some designated root node.

Gene to gene distances are computed this way. Given gene A and gene B, find a region that differs, and compute a letter-by-letter score according to a rule given below. We explain exactly how to find a region that differs below, too.

Animal-to-animal distances are based on the same idea, but now focused on genetic distance. Again, we give a rule for this.

As for Prim's algorithm, there is a simple way to implement it using a `std::priority_queue`. Then you print out the evolutionary tree.

We break this up into sections. It's a good idea to plan out a few deadlines for yourself in advance, so that you have a benchmark you can use to track yourself. Below we provide extremely detailed descriptions of each step so that even if you are very new to writing longer stretches of code, you will be able to follow the step-by-step instructions and get the job done!

Section 1:

A Simplified Gene Distance Algorithm

For ease of exposition, let us begin with an oversimplified algorithm—not the one you will finally implement—that provides some intuition for the more sophisticated approach. Consider two genes, G_a and G_b , where G_a is of the form $G_{\text{prefix}}G_0G_{\text{suffix}}$ and G_b is of the form $G_{\text{prefix}}G_1G_{\text{suffix}}$. G_{prefix} and G_{suffix} are the same for both genes, and may even be zero-length. Thus G_0 and G_1 differ only in their middle sections G_0 and G_1 . Assume the prefix and suffix are maximal, such that G_0 and G_1 do *not* have any characters in common at either end. *We explain exactly how to find maximum prefixes and suffixes below. When developing your program, do some test runs to make sure your implementation of this logic works!*

Scoring

To determine the score for these two gene fragments G_0 and G_1 , we first count the instances of each character. For example, if $G_0 = \text{IAAIYTTITAYAA}$ and $G_1 = \text{AYTTAAATTAAAAAT}$ then we count for G_0 3 I's, 2 Y's, 2 T's and 5 A's. Likewise, for G_1 we have 0 I's, 1 Y's, 5 T's and 9 A's.

Now we compare these numbers. We use three constants—COMMON_COST, DIFF_NUM_COST, and ONLY_IN_ONE_COST—in our score calculation.

- If a character is in one fragment but not the other, add ONLY_IN_ONE_COST for each instance of that character. Example: There are three I's in G_0 but none in G_1 , so we add ONLY_IN_ONE_COST*3 to the score.
- If a character appears in both fragments, add COMMON_COST for each common occurrence, and DIFF_NUM_COST for each difference. Character T appears twice in G_0 and five times in G_1 . Thus, T contributes COMMON_COST*2 + DIFF_NUM_COST*3 to the total score. Likewise, Y contributes COMMON_COST + DIFF_NUM_COST and A contributes COMMON_COST*5 + DIFF_NUM_COST*4.

The nominal constant values are COMMON_COST=15, DIFF_NUM_COST=8, and ONLY_IN_ONE_COST=25, giving a total score of $\text{COMMON_COST} * 8 + \text{DIFF_NUM_COST} * 8 + \text{ONLY_IN_ONE_COST} * 3 = 259$ for the example.

Tip: It's a good idea to store the constants as *const* variables in your *Gene* class instead of hard-coding "magic numbers" into your code.

```
// Score for each common instance of a character
```

```
const int COMMON_COST = 15;
```



```
// Score for difference in number of instances
const int DIFF_NUM_COST = 8;

// Score for characters appearing only in one gene fragment
const int ONLY_IN_ONE_COST = 25;

// Sliding window minimum match length
const int MIN_MATCH = 4;
```

This scoring function is a **heuristic**: a rule of thumb that will suffice for our purposes in CS4414. It may seem a bit ad-hoc, but in fact the Smith-Wasserman and BLAST algorithms use similar heuristics. The reason for using heuristics of this sort is that we can't know how one gene evolved into another—there are too many possibilities to explore, and in general, more than one possibility might lead to the right place. Worse, one can't really know that evolution followed the shortest path. So this is a case in which there isn't any way to know the "true" answer. A good rule of thumb may be the best that can be done!

In a normal C++ project you would make your own design and coding choices, but we'll walk you through this first project in considerable detail to help you get used to object oriented programming in C++. Just the same, we won't be dictating the design of your objects or the methods they should implement. CS 4414 is a capstone course and you need to become quite independent as a developer. We know that the first steps can be a struggle – it isn't easy for anyone! But the more you do, the easier it will get.

The Complete Gene Distance Algorithm

The heuristic scoring algorithm given above is too simple, because there might be large parts of G_0 and G_1 that are identical. The algorithm should detect those parts and only score the parts that are *actually different*. Adding this feature results in a recursive algorithm that breaks the overall genes G_a and G_b down into matching and non-matching regions, ultimately calling the scoring function above for each of the non-matching regions. The final distance will be the sum of all of those scores.

Step 1: Trim Common Affixes

As in the simplified algorithm, we will ignore common prefixes and suffixes in G_a and G_b . That is, the first thing to do with your two input genes is find the longest strings G_{prefix} and G_{suffix} such that G_a is of the form $G_{\text{prefix}}G_0G_{\text{suffix}}$, and G_b is of the form $G_{\text{prefix}}G_1G_{\text{suffix}}$. Throw away G_{prefix} and G_{suffix} ; the result of the trimming operation is just G_0 and G_1 . Note that they necessarily differ in both first and last characters, or the prefix or suffix could have been longer. Either or both of G_0 and G_1 can be empty as well—if both are empty, it is because the two genes are identical, with distance zero.

In some cases, the order in which you trim the prefix and suffix matters. For this assignment, trim the prefix first.

Step 2: Sort the Trimmed Genes

In order to ensure that the algorithm described below is deterministic, and everybody gets the same results, it is important to make sure the computation always happens the same way. We will define a sort order for genes as follows:

1. If G_0 is shorter (has fewer characters) than G_1 , it comes first.
2. If G_0 and G_1 are the same length, and G_0 comes first alphabetically, it comes first.
3. Otherwise, G_1 comes first.

In other words, sort first by length (increasing), and sub-sort alphabetically. Notice that we are just sorting 2 genes here as part of a step in gene comparison. This is unrelated to the sorting we did when we gave id numbers to the (entire) genes loaded from the files in Homework 3, even though it uses the same sort ordering rule.

After trimming the common prefix and suffix from our inputs, we compare the genes using this sort order. If G_0 is greater than G_1 , swap them.

Suggestion: If your gene objects are "comparable", and you store gene fragments as gene objects too, the sort order becomes automatic. And this suggestion means the sorting needed in Homework 3 and 4 actually would both be done by that one comparison operator. The TAs will show you how to implement comparison methods for objects.

Step 3: Sliding Window Matching

Given two trimmed, sorted genes G_0 and G_1 , our distance algorithm will recursively identify and compare segments that differ between the genes. Although we have trimmed common affixes, there might be internal fragments of DNA that are identical; this "sliding window" algorithm is designed to identify identical segments of DNA.

The first step is finding a match. We will use the constant $MIN_MATCH = 4$.

We'll take a "window" of MIN_MATCH characters within G_0 and "slide" it from left to right along G_1 looking for a match.

To illustrate:

```
G0 TYTYYAIITTAITTYAYTAIYYITITITAYIIAIAIYYAT
G1 AAYTYTYYAIITTATIYAATYAYAYIAIYAIITAAAATIAAIIIT
Slide ...
G0 TYTYYAIITTAITTYAYTAIYYITITITAYIIAIAIYYAT
G1 AAYTYTYYAIITTATIYAATYAYAYIAIYAIITAAAATIAAIIIT
Slide ...
G0 TYTYYAIITTAITTYAYTAIYYITITITAYIIAIAIYYAT
G1 AAYTYTYYAIITTATIYAATYAYAYIAIYAIITAAAATIAAIIIT
Slide ...
G0 TYTYYAIITTAITTYAYTAIYYITITITAYIIAIAIYYAT
G1 AAYTYTYYAIITTATIYAATYAYAYIAIYAIITAAAATIAAIIIT
Match!
```

If you reach the end of G_1 and still haven't found a match, slide the window in G_0 to the right by one character and repeat. This is just a nested loop, comparing all possible four-character substrings of G_0 and G_1 until a match is found; the reason you must make the sliding G_1 window the inner loop and G_0 the outer is to ensure that everyone implements the same algorithm and thus gets the same answers.

If a match is found, you know that G_0 and G_1 have four characters in common at that point—but they might have more. "Grow" the match to include all other characters that match in a contiguous segment. In the illustration below, the window is expanded to include five additional matching characters before it reaches non-matching characters.

```
G0  T Y T Y Y A I I T A I T T Y A Y T A I Y Y I T I T I T A Y I I A I A I Y Y A T
G1  A A Y T Y T Y Y A I I T T A T I Y A A T I Y A Y A Y I A I Y A I I T A A A T I A A I I I T
```

Growing the match

Step 4: Using the Match to Compute Distance

If there is no match, then the two genes (really, fragments of the top-level genes G_a and G_b) have no significant portions in common, and the distance between them is determined directly by the scoring algorithm described in the simplified algorithm.

Otherwise, we will split each gene into three segments—everything to the left of the match, the match itself, and everything to its right. The left and right segments are possibly empty. We first run the score calculation (as described in the simplified algorithm) on the left side segments of G_0 and G_1 and add this to our total distance. Second, the match segment is discarded. Third, we run the distance algorithm recursively on the right segment.

```
I A A Y T Y A I I T A I T T Y A Y T A I Y Y I T I T I T A Y I I A I A I Y Y A T
A A Y I Y I T A T A I T T A T I Y A A T I Y A Y A Y I A I Y A I I T A A A T I A A I I I T
```

Green segments are scored. The distance algorithm is run recursively on the red segments. Yellow (matching) segments are discarded. The total distance is the sum of the green score and the recursive red score.

The final distance is the sum of all of the scores computed for the non-matching regions. Here, at last, is pseudo-code for the distance algorithm:

```
int distance( $G_a, G_b$ ):
     $G_0, G_1$  = sort (trim ( $G_a, G_b$ ))
    return distanceRecursive( $G_0, G_1$ )

int distanceRecursive( $G_0, G_1$ ):
     $G_{0left}, G_{1left}, G_{0right},$ 
     $G_{1right}$  = findMatch ( $G_0, G_1$ )
    return simpleScore( $G_{0left}, G_{1left}$ ) + distanceRecursive( $G_{0right}, G_{1right}$ )
```

Note: The trim and sort steps are **not** included in the recursion.

Step 5: Recursive gene distances.

Step 5 extends the algorithm from step 4. Start with the gene comparison algorithm from step 4. Recall that a key step in that assignment was to take the two genomic strings and to put them in a specific order: sorted by size and then with ties broken alphabetically. After that we had G_0 and G_1 and ran a matching algorithm that broke each into three parts: a non-matching gene sequence, a matched section 4 or more characters in length, and then the remainders of G_0 and G_1 respectively. In assignment 2 we employed a "simple distance" computation counting the letters in the non-matching sections on the left, ignored the matching sections, and repeated the comparison procedure on the remainders on the right. But we kept the genes in the same order!

The change we want you to make is to compute the *minimum* genetic distance where you run this computation first on G_0 and G_1 but then a second time on G_1 and G_0 (with the longer string "on top" and the shorter one "underneath"). That is, while we still use a sorting order to *number* the genes, we *don't use a sorting step at all in the distance computation*. Instead, we try both ways and just take the smaller of the computed distances.

As was the case for step 4, the computation itself is repeated. Given our two genes, we trim off any identical prefix and suffix. Now treat the remaining strings as a sequence consisting of codons that match followed by codons that differ, perhaps repeated many times. For each of these sections, we break off and discard the matching portion, compute the distance score on the differing pieces ($G_{0\text{left}}$ and $G_{1\text{left}}$), and then repeat the calculation on the remaining portions which come after the section that matches. Again, you need to try this two ways: $G_{0\text{right}}$, $G_{1\text{right}}$ and then $G_{1\text{right}}$, $G_{0\text{right}}$ taking the minimum computed distance.

Here's pseudo-code for how that might look:

```
distance( $G_a$ ,  $G_b$ ):
     $G_0$ ,  $G_1$  = trim ( $G_a$ ,  $G_b$ )
    return min(distanceRecursive( $G_0$ ,  $G_1$ ), distanceRecursive( $G_1$ ,  $G_0$ ))

distanceRecursive( $G_0$ ,  $G_1$ )
     $G_{0\text{left}}$ ,  $G_{1\text{left}}$ ,  $G_{0\text{right}}$ ,  $G_{1\text{right}}$  = findMatch ( $G_0$ ,  $G_1$ )
    scoreLeft = simpleScore( $G_{0\text{left}}$ ,  $G_{1\text{left}}$ )
    scoreRight = min(distanceRecursive( $G_{0\text{right}}$ ,  $G_{1\text{right}}$ ), distanceRecursive( $G_{1\text{right}}$ ,  $G_{0\text{right}}$ ))
    return scoreLeft + scoreRight
```

The computation lends itself to a recursive procedure. With respect to the version you created in step 4, you can keep the genetic character counting code and the window-matching code, which won't change, but you'll replace the distance and distanceRecursive methods you coded for step 4 with code that looks somewhat similar to the code above. Obviously, the code above isn't real C++. We left a bit of work for you to do!

This recursive procedure will give you different (better) genetic distances than you obtained with the initial, overly simplified version.

Section 2

Step 6: Species-to-species distances

Whew! Now we can compute the distance between any pair of genes. Our next step will be to compute distances for entire species! But the idea, in fact, is very similar.

Recall that we numbered genes as G0, G1, G2... based on our rules for sorting genes (I am adding the G to avoid confusion... internally, G3 is just id number 3). You'll do the same with species: Species will be numbered S0, S1, S2... sorted alphabetically by animal name ("Armored Snapper", "Asian Boxing Lobster", etc.) such that S0 is the first animal alphabetically, S1 is the next, and so on.

From our previous steps, each animal has DNA and by now we have a list of the genes that each animal's DNA contained and a distance rule, gene to gene. Thus if we have animal A and animal B, we can make lists of the genes that each animal had in its DNA: perhaps G1, G7 and G8 for A, and G7, G8, G9 and G11 for B.

Our basic scheme will be to first compare the genes in A to the genes in B, and then do the reverse, comparing the genes in B with the genes in A, and then to average the two results.

To compare two species (A and B), we'll maintain a running sum initialized to zero. For each gene in animal A, find the closest matching gene in animal B, and add the distance to the running score (exact matches will have genetic distance zero, and hence won't contribute to the distance score). We'll do this twice: first iterating over the A genes and comparing them against the ones in B, and then iterating over the B genes and comparing these against the genes in A. The computed score will be the average of the two. For example, perhaps A contains genes G1, G2 and G3 and B contains G4 and G5. So first you'll look at G1 relative to G4 and G5 and figure out which it was closest to, perhaps G4. Next you'll look at G2, perhaps this is also closest to G4. Add the score to the score for G1 versus G4. Finally, you'll look at G3. Maybe this is closest to G5; again, add the score. That gives a total score for A versus B; now repeat for B versus A, average the two, and that's our answer.

Note: We want species-to-species distances to be integers, like the gene-to-gene distances. No need to use floating point for these averages (yes, the distances will round down if the total happened to be odd, but that's just fine).

Output. You will print a list of all the genes and a gene-to-gene comparison matrix exactly like in step 4, but using the recursive gene distance computation from step 5. Then, you will print an animal-to-animal comparison matrix. First, print a line that describes each species and its genes:

```
S0=Armored Snapper: Genes [0, 1, 2, 3, 12, 13, 17]
```

Then print an animal-to-animal comparison matrix using your species distance calculations.

Note that in the matrix, elements within a row should be separated by a tab character ('\t').

Example Output

```
// File(s): A0.dat, A1.dat, A2.dat
```

G0=ITIIIIITYYYYATTAAAIYTAIITAATIAIYAIYTYTAYYAIATAYYIYITIAIYIIITTTAYTTAITYYYAY
G1=IYIIIAAYAYAATIAAAAYAIYTYIYITAIIYTAIAITIIYAAIATITTYAYYYAAAITIITTYAIYTTAYAITTYAY
G2=ITAYTYIYIATTTYAYYYAIYAITTAIYIIIAITYAIIAAATYTYAYYTIATAYYYTYTYYYTYYYAYIYIIYTYAYY
G3=IAIYIIAIYATIIIIYIIYIYTYTYIATTIYTAITIIYTYIATAAIIITTYAYYYTTITTYAYIYIYTYTAYTITTYATY
G4=ITIIIIITYYYYAIYYATYTAAAIYTAIITAATAIIATAYYIYITIAIYIIITTTIYIIYAYTIIAIYTAITYYYAY
G5=ITIIIIITYYYYATTAAAIYTAIIIAATAYTAATIAIYAIYTYTAYYAIATAYYIYITIAIYIIITTTAYTTAITYYYAY
G6=IYIIIAAYAYAATIAAAAYAIYTYIYIYIITTYTAIYTTAIAITIIYAAIATITTYAYYYAAAITIITTYAIYTTAYAITTYAY
G7=IYIIIAAYIAAAAYAYAATIAAAAYAIYTYIYTAIYTTAYAAAIATITTYAYYYAYYYIYTAAYYAAAIIYTTAYAITTYAY
G8=ITAYITTAAYTYTYAYYYAIYAITTAIAATIIYIYAIIAAAIAIYIYTYTAYYTYTYTYTYYYTYYYAYIITTYYYYIYTYAYY
G9=ITAYTYTYIATTTYAIYTIITYYYAIYAITTAIYIIIAITYAIIAAATYTYAYYTIATAYYYTYTYYYTYYYAYIYIIYTYAYY
G10=IAIYIIAIYATIIIIYIIYTYTYTIATTIYTAITIIYTYIATAAIIITTYAYYITTYAYYTTIYIYAIYIYTYTAYTITTYATY
G11=IAITATYIYIIIAITYAIYATIIYIYTYTYTATIIYTYTIIIIITYATAAIIITTYIYTYIYTYIYTYTYTTIYTTAYTITTYATY
G12=ITYTYTYAIIITTYAYTAIYYITITYYAAAYAYTTAITIAIYIYTAIYTYAYYTTAYIATYTYTIIYTAIIATYTYTAIYYTYAYIIAAYIAIAIY
YATIY
G13=IAAYIYIAATIIITTAIATITTYTIAAYAYYTTAAIIIIYIYIAIAAAITIAAAAYYYAATTYYIAIAITYAYITYAIYAIITYAAAYTIAAAAYYYA
AIATAAIYY
G14=ITYTYTYAIIITTYAYTAIYYITITYYAAAYAYTTAITIAIYIYIIIAITYAAIYTYAYYTTAYIATYTYTIIYTAIIATYTYTAIYYTYAYIIAAY
IAIAIYYATIY
G15=ITYTYTYAIIITAIYIYTYAYTTAIIITIAIYTYIAAIYIATTAYTIIITAYYAYYTTAYIATYTYTIIYTAIIATYTYTAIYYTYAYIIIAIYIYAA
IAIAIYYATIY
G16=IAAYIYIAATIIITTAIATITTYTIAAYAYYTTAAIIIIYIYIAIAAAITIAAAAYYYTYTYAAATTYYIAIAITYAYITYAIYAIITYAAAYTIAAA
YAYYAAIATAAIYY
G17=IYYAATAATAITTTTAAIYTIIAAYIYITATAITTTATIIYAAITIAIYAIIAAATIAAIIITAAYYTTYATATIIYATAYAYIIYATYTIIAAYYYATIY
TIYYAYAYYIAIY
G18=IYAAIYIATAIITTYTIAAYAYYTTAAIIIIYIYIAAAAYIIYIYIYIAAIIATIAAAAYYAAATTYYIAAYITYAIYAIITYYAAAYTIAAAAY
YAAITAYIYIATAAIYY
G19=IYYAATAATAITTTTAAIYTIIAAYIYITATAITTTATIIYAAITIIYAYYIAIYAIIAAATIAAIIITAAYYTTYATATIIYATAYAYIIYATYTIIAAY
YYATIYTIYYAYAYYIAIY
G20=IYYAATAITTYAYATAIITTTTAAIYTIIAAYIYITATAITTTATIIYAAITIAIYAIIAAATIAAIIITAAYYTTYATATIIYATAYAYIIYATYTIIAAY
YYATIYTIYYAYAYYIAIY

[illegible]

```

1315 2220 2188 2224 1388 1663 2319 2866 1835 2232 2149 1887 150 2248 0 1115 2738 2339 2049 2338
1653 // G14
2030 1307 2118 1647 1538 2146 1458 1384 1256 2162 1788 1983 965 2443 1115 0 2933 2010 1868 2289
1648 // G15
1975 1824 1761 2053 2087 2074 2256 1563 1780 1843 2101 1911 2827 125 2738 2933 0 2240 825 2339
2390 // G16
1959 1851 1580 1765 2212 1947 1893 2062 1529 1662 1829 1801 2179 2372 2339 2010 2240 0 3054 150
150 // G17
2072 1869 1637 1683 2549 1772 2081 1695 2078 1719 1731 2091 1984 700 2049 1868 825 3054 0 2732
2769 // G18
2092 1914 1628 1813 2345 2080 1956 1732 2315 1710 1877 1819 2223 2450 2338 2289 2339 150 2732 0
300 // G19
2058 2001 1616 1847 2311 2046 2043 2212 1527 1698 1911 1934 1605 2522 1653 1648 2390 150 2769 300
0 // G20

```

```

S0=Armored Snapper: Genes [0, 1, 2, 3, 12, 13, 17]
S1=Asian Boxing Lobster: Genes [5, 6, 9, 10, 14, 16, 19]
S2=Ballards Hooting Crane: Genes [4, 7, 8, 11, 15, 18, 20]

```

```

S0 S1 S2
0 1025 4909 // S0
1025 0 6062 // S1
4909 6062 0 // S2

```

Step 7: Species Distance Algorithm Improvements

In the species distance table from step 6, the animal closest to some selected animal will often be its evolutionary brother or sister. For example, you might imagine some ancestral kind of horse, and two modern-day descendants: a Budweiser Clydesdale and a Tapir. The Tapir is closely related to a horse, and hence the animal distance between Clydesdale and the Tapir might be smaller than the distance between either animal and the ancestral species from which it evolved!

To build our evolutionary tree, we need to push that Clydesdale away from the Tapir. In our data set, it turns out that these sibling relationships always involve at least one **exactly shared gene**. For example, the Tapir and the Clydesdale might both have identical copies of gene G17. In contrast, over evolutionary periods of time between generations of species, no gene remains totally unchanged. As a result, even if a modern animal is quite similar to its ancestor, there will always be some changes to every gene it inherited.

Modifying the Species Class

Modify your animal-to-animal distance computation so that if two animals share a gene, the distance between the two is infinite (or, as computer scientists define infinite, a number so large that any two animals which are not siblings will definitely have a smaller score, in our case 100,000... a number small enough to print without using too much space, but big enough to exceed any possible animal-to-animal distance).

Note: Recall that a species had 0 distance against itself. Yet, a species is certainly its own sibling, so under these new rules it's unclear whether a species should have 0 or 100,000 distance from itself. You may resolve this conflict either way: A species may have 0 or 100,000 distance from itself. However, a species should *definitely* be its own sibling, as a sibling certainly shares a gene with itself.

Note: A good object-oriented design rule is to encapsulate logic in methods. We recommend adding an *is_sibling* method to the species class, that returns true for siblings and false if not).

Section 3:

A Phylogenetic Tree

Now we can finally construct an evolutionary tree. The user who runs the program will need a way to specify which animal should be used as the root of the tree. With this information, you can build your evolutionary tree. Each node will contain an animal as well as children which are its evolutionary descendants. Note that unlike a binary tree, in this more general tree, there could be more than two children for a given node.

The constructor for our `PhylogenyTree` class will take two arguments, a set of all the animals (including the root animal), and then the root animal itself. After the constructor runs, you should have the root node of a fully generated phylogenetic tree. We'll be explaining how to build this tree in the following sections of this specification. Also, **you should not modify the set of animals passed into the constructor.**

In addition, our class will have three more methods, which return the animal stored in this node, the parent node, and a list of child nodes. Finally, there will be one more method, `find()`, which takes an animal as input. It searches the node and all its descendants and returns the node containing that animal, or null if no such node is found.

Constructing the Phylogenetic Tree

Here is a high-level explanation of the basic algorithm we will use to build the tree.

We'll start with the root animal as the only animal in the tree, then add descendant animals one by one until we've found the complete evolutionary tree. There are many possible evolutionary trees we could generate this way, but we want to get the *best one possible*, so each step of the way, we'll have to add the animal that is *closest to an animal already in the tree*. In other words, if we went through each animal not yet in the tree and examined the distance from that animal to each of the animals already in the tree, we would select the pair of animals that minimizes the distance, over all the cases examined. But if we did this with a brute-force search every time, we'd be performing a lot of repeated distance computations, so we want to be slightly smarter about this process. To go into detail about how we do that, we'll first need to discuss two new concepts: *Prim's Algorithm* and *Priority Queues*. If you recall Dijkstra's algorithm for shortest paths, you will realize that our solution is a simplified version of Dijkstra's approach. Prim's algorithm, which predates Dijkstra's, is exactly what we need.

This problem can be viewed as a graph, where the nodes or vertices are the animals, and weighted edges exist between any two animals which are not siblings. The weight of the edge is the distance between the two animals.

Essentially, this problem is constructing a [minimum spanning tree](#) for this graph. This is the tree with the smallest total weight on its edges that connects every node in the graph. [Prim's algorithm](#) is a greedy algorithm that efficiently finds this tree. The algorithm starts with one node already in the tree (the root, in our case), and slowly adds other nodes and edges to the tree. In each step, it chooses the cheapest edge that connects to a node not yet connected to the tree. More technically, it chooses the cheapest edge e connecting vertices u and v such that u is in the tree and v is not. It adds that edge to the tree, and then repeats the process until all the nodes are in the tree.

Priority Queues

Prim's algorithm uses a priority queue (`std::priority_queue`, from the `std::queue` library). In C++, such a queue has an element type (for us, `Edge`), a "container type" (for us, `std::vector<Edge>`), and a comparator method. Obviously, you will want to sort the queue based on animal-to-animal distance.

Putting it all together

We'll have a `PriorityQueue` of `Edges` that could possibly be added to the tree. Initially, our tree should have just one node – the root – corresponding to the first `Animal` you insert into it. And initially the queue should hold edges from the root animal to every other animal.

The central loop is as follows: as long as there are still `Animals` that have not yet been added to the tree, consider the edge with the smallest distance within the queue (it will be at the top of the queue). It has two endpoints: the first will be some animal already in the tree, and the second some other animal. If this second animal is in the tree, ignore (pop) the edge.

If the second animal is not in the tree, add it to the child list of the first animal, and update your "animals in the tree count". Now, add edges to the queue for all the distances from this second animal to all other animals.

The algorithm terminates when every animal is in the tree.

To ensure that everyone produces consistent answers, we will need to do two things: First, if two pairs of animal are tied for their distance, we will tie-break these pairs based on a `String` comparison of the names (e.g. "Biscuit" or "Armored Snapper") of the animals in `nodesToAdd`, adding the animal with the lexicographically smallest name. Also, the children of each node will be stored as a list. When adding a new child, add it to the end of the list of children.

Also, for any run of your program, we expect you to print *only the level of the tree that we tell you to print*. You'll see what this looks like in a moment when we describe the two command line arguments.

Input: Command Line Arguments

Your program will need to be told which animal should be used as the root. This will be supplied as the first argument to `main` and will always be the first line you print in the tree's portion of the output. Suppose your program is called **ptree**. When you run the program, e.g. `./ptree 0 1`, it will compute a tree rooted at animal 0.

The "0" shows up as a string argument to `main`: `argv[1]`. It will be in `c_string` format (null terminated `ascii`). The method `atoi` can then be called to convert this string to an `int`.

Your program will also need to be told which level of the tree to print. This is the second argument. If you run `./ptree 0 1`, your program will construct the tree rooted at animal 0, but only print it and its children (level-1 children). If you run `./ptree 0 2`, you'll just print animal 0

and the grandchildren (level-2 children). (And note that “./ptree 0 0” should just print the root animal.)

Output: Printing the Tree

Note: this is just an example of printing a tree; the example tree is not the correct tree.

Finally, our program will need to print out the distance matrices but also a phylogeny tree, so that it will have something to show. When we display the tree, we first start out with the root animal, which has no tabs.

```
Spotted Ghila
```

For each level that you descend in the tree, indent by one tab before printing the descendants of this node. In these next two examples, Spotted Ghila has one child, Striped Salamander, which also has its own child, Gray Floop.

Level 1:

```
Spotted Ghila
```

```
    Striped Salamander
```

Level 2:

```
Spotted Ghila
```

```
    Gray Floop
```

(Note that Gray Floop is twice as indented as Striped Salamander.)

After printing the output from step 7, you will print this additional evolutionary tree output.

What to hand in

Make a zip file with your hpp and cpp source files (but NOT your compiled object files or executable). Include a “cmake” file so that we can (1) extract your files, (2) run cmake to build your program. Name the executable **ptree**, (3) We will create a subfolder TestData, extract our own selection of data files into this folder (they will be from the A0-A39 set, but not all of them, and we won’t say which subset we plan to use), (4) Pick our own root, and (5) run your program as “./ptree <root animal> <level>”. Note that ptree is built in our build folder but runs in the folder where the data files are stored. We’ll check that the output matches what we expected; if not, we’ll try and see what went wrong and will deduct points accordingly.

Grand prize! Among the solutions that are correct, we will do a bake-off to find the fastest one. The winner will get 10 points of extra credit towards the homework part of their score. The next three fastest will get 5 points.

Bored?

Why not build a web page like this? Or a “genuine” phylogeny tree?



In fact it is easy to display your output as a genuine phylogenetic tree. There are free tools for doing so (Wikipedia has an [index](#)), and it isn't hard to reformat our output into the input format these programs expect. You can ask these tools to display the PNG icons, too. In the past we had excellent experience using [Dendroscope](#), which takes input in [Newick](#) format.

Another fun “enrichment” task would be to make the web page you created in Homework 3 into a “live” web page, like the one used for the screenshot seen above. You will need help from someone who has taken CS 1300, but since this isn't graded and is purely to stay busy, there is absolutely no issue with forming a little group.

Your page should allow the user to click one of the animal icons, and should highlight the closest relative. In addition, as shown below, you might consider changing the background color of the cells to reflect their species' distance to the selected cell's species. You may choose your own color scheme, but please explain your color scheme in the comments of your ComparisonGUI class. The example uses red to indicate the selected cell, yellow to indicate closely-related species, and green to indicate more distantly related species. But any color scheme you like will be just fine. Our png files have a transparent background, so that the standard web page feature for setting a background color will give the same behavior shown in the example below. **There is no extra credit for doing extra tasks. Don't hand them in.**

Note: the data used for this example was deliberately incorrect: the closest relative of Darwin's Turtle is not actually Fuzzy Tribble, nor are the other distances real--the coloring is nonsense.