Project 1 – DNA Profiling
CS 251, Fall 2021

Collaboration Policy: By submitting this assignment, you are acknowledging you have read the collaboration policy in the syllabus for projects. This project should be done individually. You may not receive any assistance from anyone outside of the CS 251 teaching staff. Also, please see copyright at the bottom of this description. Do not post your work for this class publicly, even after the course is over.

Late Policy: You are given a total of 5 late days to use at your discretion (for all projects, not each project). You may use the late days in 24-hour chunks (either 1 day at a time or all five at once). You do not need to alert the instructor to ask permission to use late days. You can manage your use of late days on Mimir directly.

Early Bonus: If you submit a finished project early (by Wednesday, September 1st at 11:59 pm), you can receive 10% extra credit. In order to receive the early bonus: (1) your submission needs to pass 100% of the tests cases; (2) you may not have any submissions after the early bonus deadline.

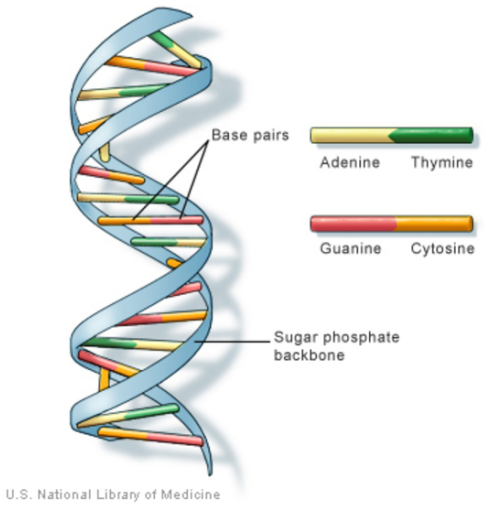
Test cases/Submission Limit: Unlimited submissions.

What to submit: main.cpp and ourvectorAnalysis.txt. Your main.cpp should include your own Creative Component with a comment at the top clearly describing how to run it and what it does. Your ourvectorAnalysis.txt should include your analysis of your usage of ourvectors.

.pdf starterCode

Project Background

Before getting started, please read/scan this entire document. You will notice at the bottom, there are some sections like "Getting started in Mimir", that some students may find helpful. There are other sections that are optional but you might find helpful on the IDE, C++, using ourvector, etc.



Source: https://medlineplus.gov/genetics/understanding/basics/dna/

DNA, the carrier of genetic information in living things, has been used in criminal justice for decades. But how, exactly, does DNA profiling work? Given a sequence of DNA, how can forensic investigators identify to whom it belongs?
Well, DNA is really just a sequence of molecules called nucleotides, arranged into a particular shape (a double helix). Each nucleotide of DNA contains one of four different bases: adenine (A), cytosine (C), guanine (G), or thymine (T). Every human cell has billions of these nucleotides arranged in sequence. Some portions of this sequence (i.e. genome) are the same, or at least very similar, across almost all humans, but other portions of the sequence have a higher genetic diversity and thus vary more across the population.

One place where DNA tends to have high genetic diversity is in Short Tandem Repeats (STRs). An STR is a short sequence of DNA bases that tends to be repeated back-to-back numerous times at specific locations in DNA. The number of times any particular STR repeats varies a lot among different people. In the DNA samples below, for example, Alice has the STR `AGAT` repeated four times in her DNA, while Bob has the same STR repeated five times.

Alice: CT|AGAT|AGAT|AGAT|AGAT|GACTA

Bob: CT|AGAT|AGAT|AGAT|AGAT|AGAT|T

Using multiple STRs, rather than just one, can improve the accuracy of DNA profiling. If the probability that two people have the same number of repeats for a single STR is 5%, and the analyst looks at 10 different STRs, then the probability that two DNA samples match purely by chance is about 1 in 1 quadrillion (assuming all STRs are independent of each other). So if two DNA samples match in the number of repeats for each of the STRs, the analyst can be pretty confident they came from the same person. CODIS, The FBI's DNA database, uses 20 different STRs as part of its DNA profiling process. More on this later.

Ultimately, you are going to write a DNA profiling app. The app will be able to take a dna strand and determine who the DNA belongs to (using a provided database). There are lots of ways you could write an app like this, but for this project you will be required to solve it using only the provided **ourvector** class.

Project Summary

You are going to write an app that is able to build DNA strands using the provided **ourvector** implementation and then determine who the DNA matches to in a database. The app will have a menu, which is how the user communicates with your program. It is really important that write your code in a module way and test as you go. The menu is designed specifically to encourage this type of code develop. Let's look at a sample input/output of the app (red text is user input, everything else is output):

```
Welcome to the DNA Profiling Application.
Enter command or # to exit: load_db small.txt
Loading database...
Enter command or # to exit: display
Database loaded:
Alice 2 8 3
Bob 4 1 5
Charlie 3 2 5

No DNA loaded.

No DNA has been processed.
Enter command or # to exit: load_dna 1.txt
Loading DNA...
Enter command or # to exit: display
Database loaded:
Alice 2 8 3
Bob 4 1 5
Charlie 3 2 5

DNA loaded:
AAGGTAAGTTTAGAATATAAAAGGTGAGTTAAATAGAATAGGTTAAAATTAAAGGAGATCAGATCAGATC
AGATCTATCTATCTATCTATCTATCAGAAAAGAGTAAATAGTTAAAGAGTAAGATATTGAATTAATGGAA
AATATTGTTGGGGAAAGGAGGGATAGAAGG

No DNA has been processed.
Enter command or # to exit: process
Processing DNA...
Enter command or # to exit: display
Database loaded:
Alice 2 8 3
Bob 4 1 5
Charlie 3 2 5

DNA loaded:
AAGGTAAGTTTAGAATATAAAAGGTGAGTTAAATAGAATAGGTTAAAATTAAAGGAGATCAGATCAGATC
AGATCTATCTATCTATCTATCTATCAGAAAAGAGTAAATAGTTAAAGAGTAAGATATTGAATTAATGGAA
AATATTGTTGGGGAAAGGAGGGATAGAAGG

DNA processed, STR counts:
AGATC: 4
AATG: 1
TATC: 5

Enter command or # to exit: search
Searching database...
Found in database!  DNA matches: Bob
Enter command or # to exit: #
```

Milestone #0 – Design your app

This app has five required commands that you will need to implement:

load_db
display
load_dna
process
search

It is recommended that you read through the handout and spend some time planning before you start coding. For example, some sections like "Learning C++ and more about ourvector", "Programming Environment", "Coding Style", and "Getting Started in Mimir" are all at the bottom of this project description. Each milestone will guide you through the implementation of each command, however, your design decisions (for example, how you will store your data) will affect the entire design. Your design will change as you develop, however, you will be most efficient if you spend some time thinking before coding. If you'd like, you can write the menu first and then proceed to the milestones below to develop the functionality of each command (this is the only way you will be able to test against our test cases as you go). Or you can develop the functionality of all the commands first, and build in the menu later (and only test against our test cases once you have the entire app done and tested). Either way, keep the overall design in mind.

DISCLAIMER: *In CS 251, we require you to write code that is functional (i.e. behaves exactly like described in this spec sheet and as the test cases show),likely just like previous classes you have taken. If your code does not pass our funcitonal test cases, you will not get any credit for your work (it doesn't matter how close to correct it is). However, we also require that your code is (1) efficient; (2) written with clean and consistent style. The efficiency will be evaluated via the ourvector analysis. The style will be graded using one style test on Mimir, but mostly will be graded manually by the graders. Your code must be properly commented, but the most essential part of your code style is that your code is functionally decomposed. As a rule of thumb, you should have no function (including main) that is longer than 30 lines of code (ignore blank lines, comment lines). Some students write all their code in main and try to decompose later. This is strongly discouraged. Instead, modularize your code now. As you work on each milestone, make a function (and likely many helper functions). That way, you can test each of those functions as you go. The test cases are set up to help you test your milestones before moving on. Remember, function decomposition is not only useful for testing, it is also used for code clarity. As you work on the first two milestones, the string parsing and file reading can get messy, properly decomposing that work into functions will make your life easier while coding. Check out the last requirement at the bottom of this handout called "cyclomatic complexity", that will help guide you in when to decompose. Some students find the first 251 assignment overwhelming because it requires them to write functional, efficient, AND properly styled code. However, don't be, you are here because you are ready! Take your time and do it right from the start. If you start hacking at it (writing all your code in main, not testing as you go, not commenting your code as you go, not planning before you start), you will probably find it frustrating and likely spend way more time on this assignment than necessary.*

Milestone #1 – Load in the database – load_db

The first command to work on is "load_db" (loading the database). What might such a DNA database look like? The database is provided in the form of a text file, wherein each row corresponds to an individual, and each column corresponds to a particular STR, for example let's say you have a file that looks like this:

```
name,AGAT,AATG,TATC
Alice,28,42,14
Bob,17,22,19
Charlie,36,18,25
```

The data in the above file would suggest that Alice has the sequence AGAT repeated 28 times consecutively somewhere in her DNA, the sequence AATG repeated 42 times, and TATC repeated 14 times. Bob, meanwhile, has those same three STRs repeated 17 times, 22 times, and 19 times, respectively. And Charlie has those same three STRs repeated 36, 18, and 25 times, respectively. Your first milestone is to get this text file read in and to save the data in an appropriate container(s). For this first project, you are restricted to saving DNA of any kind (STR sequences or full DNA sequences) in an ourvector. No other containers are allowed. Also, you may not save the STR or DNA sequences in string containers, they must be saved as ourvectors, i.e., you must save DNA sequences nucleotide base by nucleotide base in an ourvector. You may save them as **ourvector <string>** or **ourvector<char>.** For example, this ***is not*** allowed:

```
string str = "AGAT";
```

However, this ***is*** allowed:

```
ourvector<char> str;
str.push_back('A');
str.push_back('G');
str.push_back('A');
str.push_back('T');
```

Another example of what ***is not*** allowed:

```
ourvector<string> str;
str.push_back("AGAT");
```

However, this ***is*** allowed:

```
ourvector<string> str;
str.push_back("A");
str.push_back("G");
str.push_back("A");
str.push_back("T");
```

Warning: These are just examples, you should be doing file reading and writing general code in your project. You may use strings in your program, obviously, but they cannot be used to store STR or DNA sequences. Your job on this milestone is to get all the database file read in and stored in appropriate container(s) (this will include the individual people and their STR counts AND the actual STR sequences in the first line). For this milestone, you are going to want to do a lot "cout"-ing to make sure the file is being read as intended and stored in the container as intended. While you are restricted to using ourvectors only, there are a lot of choices to be made. Do you want to use a nested ourvector, e.g. ourvector<ourvector<…>>? Or do you want to use a struct and then use an ourvector of that struct? There are lots of options, but that design choice is completely up to you. Do not post or discuss with others your design choice in anymore detail than what is described and suggested here. We are happy to discuss the pros and cons of this choice with you in office hours, however, do not come to office hours asking for us to "approve" your choice.

It won't really help to submit this against our test cases quite yet, but you will be able to after you finish milestone #2.

Error handling: If the user enters an invalid filename for the database file, your program should not try to read the file and should display this error message `Error: unable to open 'invalid_db_file.txt'` where 'invalid_db_file.txt' is the filename that was invalid.

Milestone #2 – Display your data -- display

The next command to work on is the "display" command. This command displays to the screen the database, the dna, and STR count. At this point, you have only loaded in the database, so if you check the sample input/output above, the display command should show this (for small.txt):

```
Database loaded:
Alice 2 8 3
Bob 4 1 5
Charlie 3 2 5

No DNA loaded.

No DNA has been processed.
```

It is a good idea to implement this now, so you can make sure your database was loaded properly (and so you can keep testing it as you add more to your code). You should also test the large.txt database. After you have run tested this on the IDE for lots of different inputs, if you have your menu implemented, you can submit against the test cases on Mimir. You should pass the test cases that test load_db and display only.

Milestone #3 – Load in the dna – load_dna

The next command to work on is the "load_dna" command. The dna files are 1.2-.txt. The small.txt file goes with 1-4.txt and large.txt goes with 5-20.txt. Check out the sample

input/output above and open up the 1-20.txt files to see the format. First, write the function to reads the dna file and saves it. Second, add to your display command/function to now also display the dna data. Remember, you may only save the DNA data using an ourvector container, no other containers or strings are allowed.

Error handling: If the user enters an invalid filename for the dna file, your program should not try to read the file and should display this error message `Error: unable to open 'invalid_dna_file.txt'` where 'invalid_dna_file.txt' is the filename that was invalid.

After you have run tested this on the IDE for lots of different inputs, if you have your menu implemented, you can submit against the test cases on Mimir. You should pass the test cases that test load_db, load_dna, and display only.

Milestone #4 – Process the DNA – process

The next command to work on is the "process" command. Given a sequence of DNA, how might you identify to whom it belongs? Let's look at this sample database again:

```
name,AGAT,AATG,TATC
Alice,28,42,14
Bob,17,22,19
Charlie,36,18,25
```

Well, imagine that you looked through the DNA sequence for the longest consecutive sequence of repeated AGATs and found that the longest sequence was 17 repeats long. If you then found that the longest sequence of AATG is 22 repeats long, and the longest sequence of TATC is 19 repeats long, that would provide pretty good evidence that the DNA was Bob's. Of course, it's also possible that once you take the counts for each of the STRs, it doesn't match anyone in your DNA database, in which case you have no match. For this milestone, you are going to write the code that analyzes the DNA that has been loaded and counts up the longest consecutive repeated STR sequence for each STR listed in the database file. You will search the database for a match in the next milestone.

You might be wondering: what algorithm should I use to find the longest consecutive sequence of repeated STRS for a particular DNA sequence? That is up to you to figure out. And remember, you cannot use any additional containers.

Once you have counted up the longest consecutive sequence of repeated STRS for the loaded DNA and have saved it (NOTE: you are only allowed to use an ourvector container for this), time to test! Check out the sample input/output above and add your STR counts to the display command. Test a few DNA files on your own. The small.txt file goes with 1-4.txt and large.txt goes with 5-20.txt. When you are ready, test against the test cases to see if your STR counts are correct. There might be some edge cases you haven't considered in your algorithm that will need to be reworked. To figure them out, just open up the DNA file and use ctrl+F to see what combination of repeated STRs your algorithm is not accounting for. If you need help with this, we are happy to help you in office hours or on the discussion board. However, do not post/ask "I am not passing test case #, what am I missing?" You need to go into that test case, see what file

is being tested and figure out for yourself what you are missing. If you are having trouble or are stuck, please let us know! However, you should have done significant investigating prior to asking for help. All the information you need to figure this out is provided.

Lastly, note that process depends on the data from load_db and load_dna. Make sure this data is present before processing. If the database is not loaded, output `No database loaded.` if the database is loaded but the DNA is not output `No DNA loaded.`

Milestone #5 – Search the database– search

The final command to work on is the "search" command. The functionality of this command is to use the STR counts you found in the previous milestone to search the database and find the match. See the sample input/output above for more details on this. If there is no match, output `Not found in database.` Note that this command relies on the data from load_db, load_dna, and process, so make sure this data is present before searching. If the database is not loaded, output `No database loaded.` if the database is loaded but the DNA is not, output `No DNA loaded.` if both the database and the DNA are loaded, but the DNA has not been processed, output `No DNA processed.` Please note you must write this search algorithm yourself, you may not use any library functions for this.

Milestone #6 – Creative component

In our 251 projects, we like you think about about design and app development on your own. Having every detail of an app spec'ed out, like what you have completed so far, is important. It definitely is a skill that you will need to contribute to large code bases and existing products. However, we also want you to develop your own ideas. For the creataive component, you will first create your own command. It can be anything, other than the five commands you already implemented and of course, it can't be "#". When your custom command is typed by the user, it should initiatve the code for your creative component. What functionality do you wish this app had? Here is your chance to add it. The details (What is the user input? What is the output? Etc.) are your choice. However, you will be graded on this choice.

Ideas for your creative component:

- Add a command identify multiple DNA files at once.
- Add a command that can count all STR instances instead of only the longest, consecutive repeated STR sequence.
- Add a command that helps in testing. Perhaps you write a command that can run and test many or all of the functions at once?
- Add a command that adds a new individual to the database. Perhaps use STR counts for one of the DNA files that does not currently exist.
- Preferrably, think of something on your own. These are just a few ideas to get you started brainstorming. We prefer when you come up with something on your own.

Requirements for your creative component:

- You must have a comment at the top of main.cpp that describes what your creative component does and how to run it. If this is missing, you will get no credit and no

regrade requests are accepted. The TAs grading must be able to run your creative component to test it. So all the information must be included that allow them to do this. If other commands must be called first in order for your command to work, make sure to include that information.

- It must be a similar difficulty and amount of work/code as the other five commands you wrote.

Ourvector Analysis

Open ourvector.h and go to the member function _stats. Make sure this portion of the code is not commented out:

```
cerr << "*******************************************************" << endl;
cerr << "ourvector<" << name << "> stats:" << endl;
cerr << " # of vectors created:   " << Vectors << endl;
cerr << " # of elements inserted: " << Inserts << endl;
cerr << " # of elements accessed: " << Accesses << endl;
cerr << "*******************************************************" << endl;
```

Run your code (make sure to run all commands, including to quit) with this version of ourvector.h and you should see some vector stats print at the bottom of your code:

```
**********************************************************
ourvector<T> stats:
 # of vectors created:   <X>
 # of elements inserted: <Y>
 # of elements accessed: <Z>
**********************************************************
```

Submit a file named "ourvectorAnalysis.txt". Copy and paste what you get for the report above into your text file (include all input, output, and the ourvector report). In it you must write approximately 250 words to justify why, when, and how your code has created X number of vectors, has inserted Y elements, and accessed Z elements. Your analysis should be specific and quantitative. For the vectors created, you should identify all line numbers of your code where those vectors are created. You should be specific enough that your counts add up to exactly the amount of vectors. For the number of elements inserted and number of elements accessed, you can estimate more approximately based on the sizes of the vectors and the types of operations you are doing.

Your code should minimize number of vectors created, elements inserted, and elements accessed. One way to do this is to make sure you are passing by reference in all functions, particularly for the large containers of data. Another way is avoid unnecessary searching and to properly design your app such that you are storing your data in a efficient way. Each piece of data should only be stored exactly once, and you should justify that this is the case. In this project, there should be ZERO extra copies of the database and the DNA. If you are using for-each loops, be careful that you are using a reference to avoid extra copies being made each time you loop through your containers. Structs are encouraged!

Learning C++ and more about ourvector

C++ Strings: You'll need some string processing, namely finding characters within a string, and extracting a substring. While you can certainly write these functions yourself, it's expected that you'll use the .find() and .substr() functions built into the **string** class provided by C++: http://www.cplusplus.com/reference/string/string/. Don't forget to #include <string>.

ourvector: Your solution is required to store all data in a **vector<T>** class --- to be precise, in a vector-compatible implementation we are providing: **ourvector<T>**. For the purposes of this assignment, always start with an empty vector, and then add data to the vector by calling the **push_back()** member function. When you need to access an element of the vector, use the **.at()** function, or the more convenient and familiar **[ ]** syntax. To empty a vector, use the **.clear()** function. For more info on vector<T>: http://www.cplusplus.com/reference/vector/vector/. Don't forget to #include "ourvector.h" (it is already in the starter code).

File streams: Finally, to work with files, #include <fstream>. To read from a file, use an **ifstream** object, and use the **>>** operator when inputting a single value (e.g. integer or single word). When you need to input 1 or more words into a single string variable, use **getline(infile, variable)**. There is also an optional third parameter defining a delimeter you can use if you wish to put only a segment of the input into the variable, **getline(infile, variable, ',')** , for example, would put part of the line up until the first ',' into variable. A subsequent call to getline would start its input where the last call ended.

String streams: Finding characters within a string and extracting a substring using the built in string functions can be rather tiring, and lead to more messy code. An optional alternative to this is provided by using a **stringstream** object. String streams are like file streams except they work on… strings! Meaning you can use getline or the >> operator to extract parts of a string without having to mess with built in string functions. To initialize a stringstream object named s you would write **stringstream s(some_string)**. Where some_string is the string you wish to initialize the stringstream object on. From there you could use the stringstream object just like a ifstream object with lines such as **getline(s, variable, ',')** or **s >> variable**. Don't forget to #include <sstream>.

Programming Environment

You are required to program your project in Mimir environment. Mimir allows us to all work in the same space and allows all teaching staff to best assist you. It also minimizes messing with cross platform problems. Mimir tracks your progress as you write the code, which will assist us when grading your work.

Mimir is set up to automatically use your highest scoring submission.

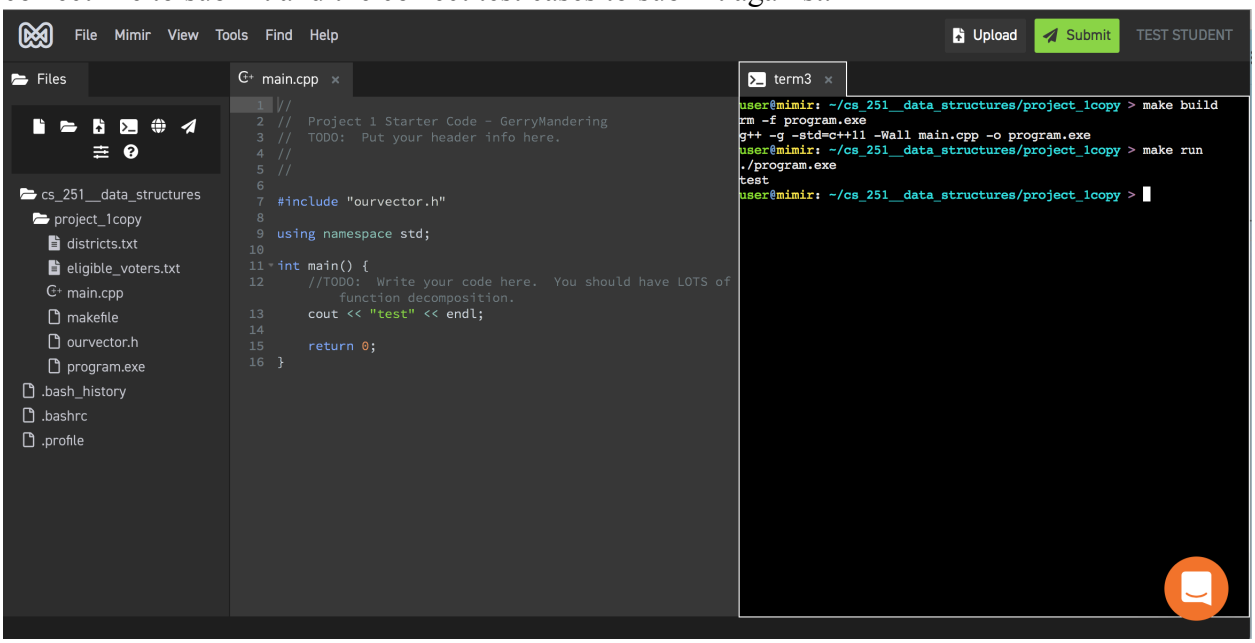
On Mimir, we are compiling via **g++** with **-std=C++11**. Do not ask us to change the C++ version; we are compiling against C++ 11.

See the section below called "Getting Started in Mimir" if you need help…getting started in Mimir. If you are new to a Linux environment (most students are), please pay special attention to what directory (folder) you are in. Lots of students struggle with this.

Getting Started in Mimir

Click on “Open in Mimir IDE”.

Click on the course, then the project folder (yours will be called project1_dna_profiling, below it shows project1_copy). Inside of the Project 1 folder, open the main.cpp file and add a few cout statements to the code. Look at how the terminal below AND the directory on the left from which we opened main are the same. Don't forget to save your code, it does not save automatically. To compile, type “make build” in command window. To run, type “make run”. You will need to save the file, compile, and run each time you make changes to your code. To test your code against the test cases and submit, click submit. Make sure to click the correct file to submit and the correct test cases to submit against.



NOTE: Once you have the IDE open, you may want to adjust the settings. Click File->Settings. We recommend unchecking the box, “use spaces instead of tabs”.
TIP 1: Ctrl+C will cancel a run command if you are stuck in a stream/elsewhere.
TIP 2: You can set the max character width on screen, which helps with coding style requirement: file->settings-> print margin column = 80, file->settings-> show print margin line = on. This is what the style test for check for.
TIP 3: File -> Settings -> *Uncheck* Show intercom help bubble.

Coding Style

You will be asked adhere to the Google C++ coding style guidelines, which are described here:

https://github.com/google/styleguide/blob/gh-pages/README.md

This is autograded using our style test. You should also review the style guidelines from lecture and the style rubric items on Mimir. Your code will be graded for style and any violations to our

style criteria will be a deduction on your final score. Notice our rubric items relating to style (see Mimir rubric, it is released with the project) are mostly deductions. Therefore, while you might have turned in a code that received 180 pts out of 200 pts based on functionality, your code may receive additional deductions based on efficiency, function decomposition, etc.

Requirements

1. ***Only uses ourvector containers:*** You must use **ourvector<T>** ("ourvector.h") for storing all data. No other containers may be used. All DNA sequences, STR sequences, and STR counts must be stored in a ourvector type, you may not store DNA sequences in arrays, strings, or any other type of container.
2. ***No extra libraries allowed:*** You are ***NOT*** allowed to use or add any libraries not included in the starter code or mentioned in the **Learning C++ and more on ourvector** section of this handout.
3. ***Files can only be read exactly once***: Each input file may be opened and read exactly once; store the data in some combination of ourvector<T> (nested ourvectors and structs are encouraged). All files should be opened and read only when the load_db and load_dna commands are called.
4. ***Write your own algorithms:*** The algorithms for counting STRS and searching must be written by you, no library functions allowed.
5. ***ourvector Analysis:*** You should be able to identify and count all ourvectors created. The # of inserts and the # of accesses must be reasonable. You should determine that yourself and justify in your submission (in "ourvectorAnalysis.txt"). See Mimir rubric to see how you will be graded for the ourvector analysis.
6. ***Commenting***: Your main.cpp program file must have a header comment with your name and a program overview. Each function must have a header comment above the function, explaining the function's purpose, parameters, and return value (if any). Inline comments should be supplied as appropriate; comments like "*declares variable*" or "*increments counter*" are useless. Comments that explain non-obvious assumptions or behavior *are* appropriate. See Mimir rubric to see how you will be graded on commenting.
7. ***Function decomposition:*** Each command must be implemented using a function; this implies a complete solution must have at least as many functions as there are commands. However, a good solution will have many more functions to decompose your code properly. A good rule of thumb: all functions should be <= 30 lines of code (blank lines and comment lines don't count). See Mimir rubric to see how you will be graded on function decomposition.
8. ***No global variables:*** use parameter passing and function return. Defining a global struct type is okay.
9. ***No pointers and no heap memory allocation:*** Do not use pointers and do not allocate memory on the heap (i.e. you should not be calling new/delete or

malloc/free). Both will result in style deductions. Neither are appropriate for this project.

10. The **cyclomatic complexity** (CCN) of any function should be minimized. In short, cyclomatic complexity is a representation of code complexity --- e.g. nesting of loops, nesting of if-statements, etc. You should strive to keep code as simple as possible, which generally means encapsulating complexity within functions (and calling those function) instead of explicitly nesting code. Here's an example of simpler code with low CCN:

```
while (…) {
   if (searchFunctionFindsWhatWeNeed(…))
     doSomething();

   next value;
}
```

Here's an example of complex code with high CCN:

```
while (…) {
   for (…) {  // loop to do search
      if { (search condition is met)
         for { (…) // now do something:
           …
         }
      }
   }

   next value;
}
```

As a general principle, if we see code that has **more than 2 levels** of explicit looping --- an example of which is shown above --- that score will receive grade penalties. The solution is to move one or more loops into a function, and call the function.

Citations/Resources

Assignment is inspired by Brian Yu and David Malan from Harvard University. Also, thanks to Kai Bonsol for his contributions.