Create a class called Queries\_AR. The purpose of the class will be to contain a dataset of genomic  
sequences (queries) and all of the functions needed to operate on this set. Use the 2D array  
data-structure to store the genomic sequences of the dataset. For this assignment, you can completely disregard the headers of the sequence fragments. At minimum, the class must contain (15pts):

● A default constructor (that zeroes everything out)  
● At least one custom constructor (e.g. one taking a file path or ifstream as input)  
● A function to read the query dataset file  
● A search function designed to find a sequence fragment within class’s data  
● A function to sort the fragments of the Queries\_AR object  
● A destructor

A. (20 pts) Read in the entire query dataset and store it in an instance of the Queries\_AR class. Read in the entire subject dataset into a single, concatenated character array (same way you did it in HW#1). Implement a search function which would search for 32-character fragments of the subject sequence within the Queries\_AR object. The search function should return the location (index) of the match OR a negative value if a ‘hit’ was not found. Iterate through 32-character long fragments of the subject dataset, searching for each one in the query dataset.

● How long did it take you to search for the first 5k, 10K, 100K, and 1M 32-character long  
fragments of the subject dataset within the query dataset?  
● How long would it take to search for every possible 32-character long fragment of the  
subject dataset within the query dataset? Please note that depending on the efficiency of  
your algorithm, this step may take a long time. If the total time is greater than 24 CPU hours,  
provide an estimate rather than an exact number.  
● Print the first 15 fragments of the subject dataset along with it’s indices that you found  
within the Query AR object (if any).

B. (20 pts) Read in the entire query dataset and store it in an instance of the Queries\_AR class. Sort all character fragments in alphabetic (lexicographic) order. Any sorting algorithm will do. Read in the  
entire subject dataset into a single, concatenated character array (same way you did it in HW#1).  
Implement a search function which would search for 32 character fragments of the subject sequence within the Queries\_AR object. The search function you implement should be optimal in time compared to the search function implemented in Part A and should return the location (index) of the match OR a negative value if a ‘hit’ was not found. Iterate through 32-character long fragments of the subject dataset, searching for each one in the query dataset.

● How long did it take you to search for the first 5k, 10K, 100K, and 1M 32-character long  
fragments of the subject dataset within the query dataset?  
● How long would it take to search for every possible 32-character long fragment of the  
subject dataset within the query dataset? Please note that depending on the efficiency of your algorithm, this step may take a long time. If the total time estimate is greater than 24  
CPU hours, provide an estimate rather than an exact number.  
● Print the first 15 fragments of the subject dataset along with its indices that you found  
within the Query AR object (if any).

Explain the logic used in implementing the search algorithm in Part A for searching the first 5k, 10K, 100K, and 1M 32-character long fragments of the subject dataset within the query dataset. Clearly state the motive of this function and detail the steps taken in its implementation.

Subproblem A) Linear search

The logic I used when searching though the genome and query array for matching fragments is very simple and easy to follow. First, I will begin with linear search since its far simpler due to it not needing a sorted query array to function. To being the linear search function I have 1 main loop that is responsible for iterating through the genome string and create test fragments from the current index + next 31 characters. I have an inner loop that will iterate over each fragment in the query array starting from index 0 and go to the last item in the query array. I can assume that the runtime will be O(Genome Size) x O(query Size). This is the worst-case scenario though because I have some further logic in the query loop that will skip over fragment that have already been found which will drop the runtime. This way I can fix the issue with duplicate fragments being able to be found and stop the program for rechecking data that has already been found.

Now the results for the linear search can be seen in the screenshot below.

INSERT SCREEN SHOT OF OUTPUT HERE

Subproblem B) Binary search.

Moving onto binary search strategy where it gets a little more complicated due to the nature of the algorithm. Before the binary search strategy can be started my program needs to sort the query array into alphabetical ordering so binary search can divide and conquer its way to hopefully finding the fragment. I used the merge sort strategy to effectively sort the query array into alphabetical order in a very time efficient manner. Merge sort also falls under the divide and conquer strategy where it first breaks down the array into a left and right side recursively until it reaches one value in either array. Then as it recurses back up it will sort the left side first, then right and finally merge the two sorted arrays together again sorting them in sorter. This approach has a runtime of O(n log n) which is very quick compared to bubble sort which has a terrible runtime at worst case of O(n^2). Once the query array is sorted it is finally time to search for the fragments. My binary sort function is set up very similar to the linear search function structure where there is an initial loop that will iterate through the entire genome character array and create test fragments (index to index + 31 ). It will then go to an inner loop which is responsible for iterating through the query array. The only difference is the divide and conquer approach to which the program will calculate a mid-index and test the fragment in the query array at the mid index to the test fragment from the genome. It will then determine if the fragment is less than (comes before alphabetically) or greater than (comes after alphabetically) in the query data which is already sorted in alphabetically order. This step will continue until the middle index is a match with the test fragment or if at worst case there was not match. This would in turn result in my program having an overall runtime of about O(genome size) x ( query size log query size ) which compared to linear search is far faster.

Now here are the results of the binary search algorithm. Keep in mind that I had issues with running the program within a full day due to the sheer size of the files. To get these outputs I only read in the first 1 million query fragments. This resulted in a total runtime of just under 1 hour. I’m not sure exactly how many query fragments there are in the file but I can assume that there are millions of lines that my program would be dealing with. I can estimate that my program would take about 1 to 1 and a half days to complete the whole scope.

Insert output here.

A screenshot of a computer

Description automatically generatedA screen shot of a computer screen

Description automatically generated

As seen in the screenshots above the program was able to reach in the query fragments and sort them in alphabetical order. Now it’s hard to tell since there were multiple all “A” protein fragments but when running this program on an even smaller amount of fragment it did indeed sort them correctly. Now the only issue that was present in this screen shot was that there were 15 duplicate all “A” fragments which in turn means that we are only going to see the results of 1 fragment. Unfortunately, it seems that there were not all “A” fragments found as seen in the screen shot below. A screenshot of a computer program

Description automatically generated

Now during this test with only the first 1 million fragments being read in, only about 5 thousand fragments were able to be found in about that 1 hour runtime.

ii. Describe the specific bugs and issues you encountered while solving this  
assignment. These bugs could be from any part of your code for this homework.  
Provide detailed explanations of these challenges, avoiding trivial errors such as  
"missing a semicolon in the code."  
 When coding this project, I ran into a good number of bugs which plagued my development time. The first one being the compare 2 fragments functionality in both linear and binary search functions. I used the same strcmp() C function that will compare the left and right parameters which will output -1 if the left side is larger, 0 if identical and finally 1 if the right side is larger. This was a very easy function to set up in an if() check but when it came to compare the test fragment string and the query string it really had issues since they were technically of different types. I was passing in the pointer to the query array which in turn was not able to correctly use the string at that pointer. I was able to figure out this program with some simple print statement and find that an empty string was being send in place of the pointer so it would always be returning -1.

Another issue that was found later in the testing stage was the copying data from and old query array to a newly resized one. This function operated similarly to the resize character array which is responsible for resizing the genome array. This query constructor though was just dealing with an array of pointer instead of a whole character array. When the query constructor function was called to resize the array after the first fragment it would allocate a new array of pointers with an updated size to house the new fragment pointer. Once the new query array of pointers was allocated the program would need to copy the data from the old array to the new one but the way I did was very poor. Instead of just simply setting the pointers in the index I unfortunately copied each arrays character array in manually which really hit the runtime and was unnecessary since we are dealing with an array of pointers. The difference was that the genome array was a character array which needed a contiguous block of memory which the newly resized genome array would need to copy the character over manually. The query array was an array of pointers meaning that the data does not need to be stored in contiguous memory since the array of pointers will act like contiguous memory since all the individual arrays memory location is stored in a main array.

iii. Highlight at least one specific optimization you made to improve the code's  
efficiency or readability.

One optimization that was made was discussed in the section above with the query constructor function copying data from the old query array to the newly resized one. That was by far the biggest optimization in terms of runtime and space complexity. The use of strcmp() C function drastically saved on readability and the number of code lines that were in the program. Once I added this function to my program, I cut down on about 50 lines of code since this comparing operation happens multiple times within the code.

Readability was improved when I decided to move larger operations to their own function like copyCol, copyQuery, copyString, searchQuery and sortFragments. Having the ability to make these modular functions that do simple this drastically improves the readability the driver functions since there isn’t a bunch of logic operators, loops and if/else statements all jumbled together. This will also help the entire structure of the code and can even improve debugging/testing since the modular functions can be tested individually aside from the driver functions and can really narrow down where a bug is occurring.