Nicholas Robishaw

INF-503 HW2

NAU ID: nr768

Student ID: 005841641

**Job script commands for running the program after compiling.**

**Command to run linear search:** srun ./my\_program human.txt human\_reads\_125\_32.fa “-linear”

**Command to run Binary search:** srun ./my\_program human.txt human\_reads\_125\_32.fa “-binary”

Note: I allocated 20 gbs of memory to this program which is a little overkill but it eliminates the crashes due to the memory buffer needed when resizing those large arrays.

**Subproblem A: Linear search**

The logic I used when searching through the genome and query data for matching fragments is very simple and easy to follow. First. Inside the linear search function, I have one main loop that is responsible for iterating through the genome string and creating test fragments from the current index plus the next 31 characters. I have an inner loop that will iterate over each fragment in the query array starting from index 0 and going 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 fragments that have already been found, which will drop the runtime down. This way, I can streamline the fragment search process by not wasting time on fragment that have already been found.

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

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

Unfortunately I couldn’t get my linear search to run with all the data in under a day. Instead I only read in part of the genome which decreased the runtime substantially to allow for results. I only read in the first 100 scaffolds which is about 1/6th of the total amount for the full data which means my runtime would be O(n/6 x q) where n would be the total genome size and q would be the number of the fragments. It was able to find all the fragments in about 10 hours which is really poor compared to binary search. The actual runtime of linear search with the full data would be O(nxq) to which I calculated to take about 63 hours to find the first 1 million fragments. This is masive compared difference between binary search which only needed about 1 minute to find the first 1 million fragments.

**Subproblem B): Binary search.**

Moving onto the 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 order so binary search can divide and conquer its way to hopefully finding the fragments. 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 the right, and finally merge the two sorted arrays together again. This approach has a runtime of O(n log n), which is very quick compared to bubble sort, which has a terrible runtime in the worst case of O(n^2). Once the query array is sorted, it is finally time to search for the fragments. My binary search 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, in which the program will calculate a mid-index and test against 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 alphabetical order. This step will continue until the middle index matches the test fragment, or, in the worst case, there is no match. This would in turn result in my program having an overall runtime of about O (genome size) x O(query size log query size), which, compared to linear search, is far faster.

Now here are the results of the binary search algorithm. As we can see the binary search algorithm ran in a much smaller runtime compared to linear search. There is a big difference in the output since the program is only displaying the first 15 fragments. Since there was a sort that happened before the binary search was able to run the output will also be in alphabetical order from after the search. So, this means that we are essentially seeing different fragments that aren’t present in the first 15 fragments of the unsorted query data.

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 read 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 number of fragments it did indeed sort them correctly.

A screenshot of a computer program

Description automatically generated

I added some extra timestamps into the program to get an accurate time score of how long it took find the first 5k, 100k and 1 million. From the beginning of the search the total time it took to reach 1 million was just under 1 minute. I had my program stop when it hit the 1 million fragments found to save a bit of time. I can speculate that the search function will run for about 125 minutes which equates to 2 hours and 5 minutes. This would be calculated off the fact that it took just about a minute to reach 1 million fragments found. I was a little tight on time, so I was not able to run the binary search to attempt to locate all the fragments.

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 is the compare two 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 comparing 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 statements and find that an empty string was being sent 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 of data from an 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 pointers 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 it was very poor. Instead of just simply setting the pointers in the index, I unfortunately copied each array's 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 characters 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 array memory locations are 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 that copied 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 the strcmp() C function drastically improved readability and the number of code lines 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 program.

Readability was improved when I decided to move larger operations to their own functions, like copyCol, copyQuery, copyString, searchQuery, and sortFragments. Having the ability to make these modular functions that do simple things drastically improves the readability of the driver functions since there aren’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 and testing since the modular functions can be tested individually aside from the driver functions and can really narrow down where a bug is occurring.

The last optimization I made was the day before this project was due. When looking over my program after getting an initial working build, I tried to see what could be simplified and what could be improved. When reading in both files, my function would only be taking the amount it needs to house the next fragment and genome. This is good for space complexity but really takes a toll on the runtime since when a new array is created, it needs to loop through and copy all the elements from the first array to the new array. To make this more efficient, I decided to allocate a large block of memory at the start of the function call so that the function doesn’t have a bunch of looping to go through when adding new data. This way, the program can get a large head start and store most of the file before an array resize needs to happen. The only issue with this method is allocating too much memory, which can waste some of the memory. This was present when I allocated a large block for the query data, of which not all was used. Once the genome file reader began and started to get closer to 600 scaffolds, it would run out of memory and crash the program. This was due to the large memory buffer I would need to copy the old array contents to the new one before freeing the memory of the old array. To fix this, I just added 4 more GBs of memory to the program. This is a good example of determining if space complexity has a hard ceiling and whether I can get away with taking up a lot of space to help runtime.