Nicholas Robishaw

INF503 HW-4

NAU ID: nr768

Student ID: 005841641

Zoom Recording URL:
https://nau.zoom.us/rec/share/y5Wa0mXm5fmkm9azN805ePBJVUJzR5Iit0od0xf0S041oNhJ9f66aWG5r_cfdeJT.UFPb6pO2e6Ye4YBJ?startTime=1721944050000

Cmd line input

Format for all parts: srun ./my_program human.txt human_reads_125_32.fa "<search size>" "<Problem #>" "<Part Letter>"

```
1A) ./my_program human.txt human_reads_125_32.fa "10000" "1" "A"
1B) ./my_program human.txt human_reads_125_32.fa "10000" "1" "B"
2A) ./my_program human.txt human_reads_125_32.fa "10000" "2" "A"
2B) ./my_program human.txt human_reads_125_32.fa "10000" "2" "B"
```

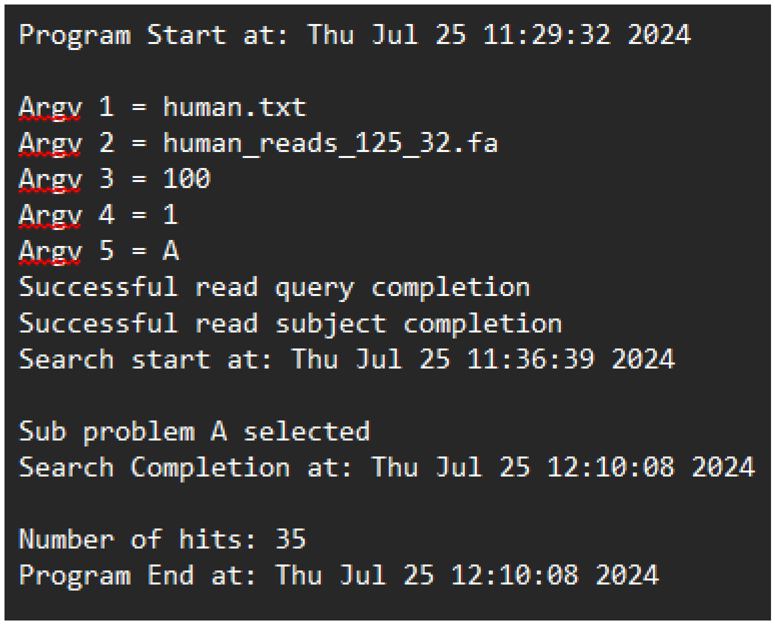
Problem 1)

Problem 1, Part A, was the hardest part of the assignment, in my opinion. This was mainly because of how I was going to incorporate the Needleman-Wunch algorithm into my code. It took a bunch of planning and rewatching the lectures, as well as looking at online visualization tools, to really understand the main process of the algorithm. I started out the program by first making the NW function generate a score matrix and returning it back to the function call. This score matrix would compare the two strings and generate a similarity score based on the match, mismatch, and gap scores. Now these results will differentiate based on what the scoring is for the three of those parameters. For this program, we needed to use a match score of 2, a mismatch score of -1, and a gap score of -1. At the bottom right of the matrix, the final similarity score would be generated, and that number is what the program would need to determine if we have a hit or not. Once that score was returned to the search function, I would then calculate the floor needed to determine if it was a hit or not. In this case, we were aloud at most two mismatched characters to be called a hit. To determine this floor value, I simply did the calculation: (fragment sizes x match score) – (3 x match score). I used 3 in this case just so I could determine if the similarity score was greater than the floor in my if statement. Now that the biggest part of the program was finished, I could then move on to the main parts of the program. To begin, I would read the query file into a 2D array where each index would hold a fragment of size 32. Then I would read in the whole genome into a character array, which would be about 3 billion characters. Now for Part A. I would generate a random index from 0 to the size of the genome array minus the test string length, which in this case would just be the fragment size of 32 characters (n-mers). Depending on the input parameters, this search function would loop until the amount of random index was met by the input parameter. During each loop, the test fragment would be created from the random index, and then another loop would be used to iterate through the query array and test each fragment n times, where n is the size of the query array. Each query index would be tested against the test fragment in the Needleman Wunch function, and then the score returned would be used to determine if the current test fragment was found in the query array. Now this would result in the big O runtime of: O (query size) x O (genome size) x O (loop iteration parameter) x O (query size) x O (fragment size^2), which is a very long runtime just for this homework assignment. Now, when I calculated the runtime for problem 1, I estimated that it would take about 128 billion instructions to do just one loop. For this homework, we needed to test 10k, 100k, and 1M loops and random indexes. When I broke this down and tested it even further, I found that it took just under 1 hour to complete just 1 loop iteration or random index test. Multiplying this by 1,000, I can easily say that this program would need about 41 days to run. Unfortunately, this would not be possible with the current homework due date, so what I did in my testing was decrease the number of loops and decrease the number of query fragments read initially. This would not give an accurate hit count since some queries would be missing, but I would be able to generate more random indexes to test, which I can show the hit amounts for in an appropriate amount of time.

Outputs

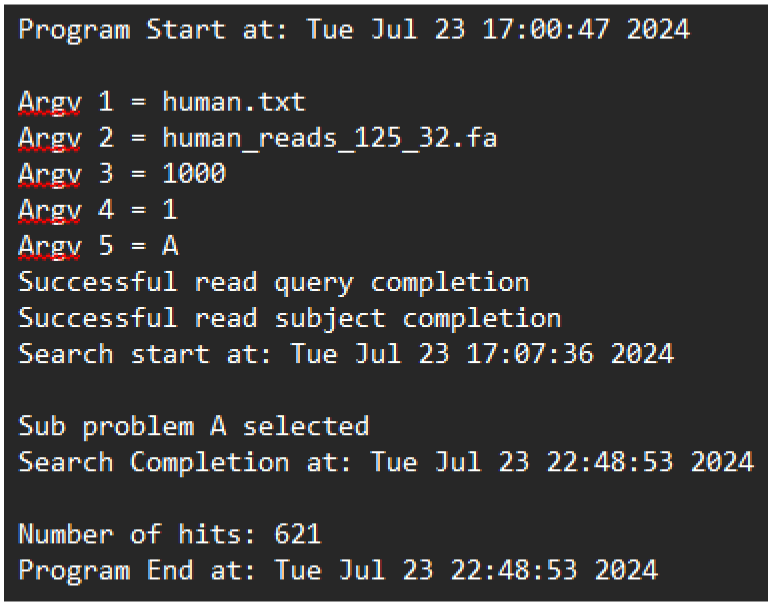
Now due to the exhaustion process of problem 1 I shrunk the search loops down to 100, 1000 and 2000 to get results within a couple days.

Output for 100 random indices' (48 minutes)



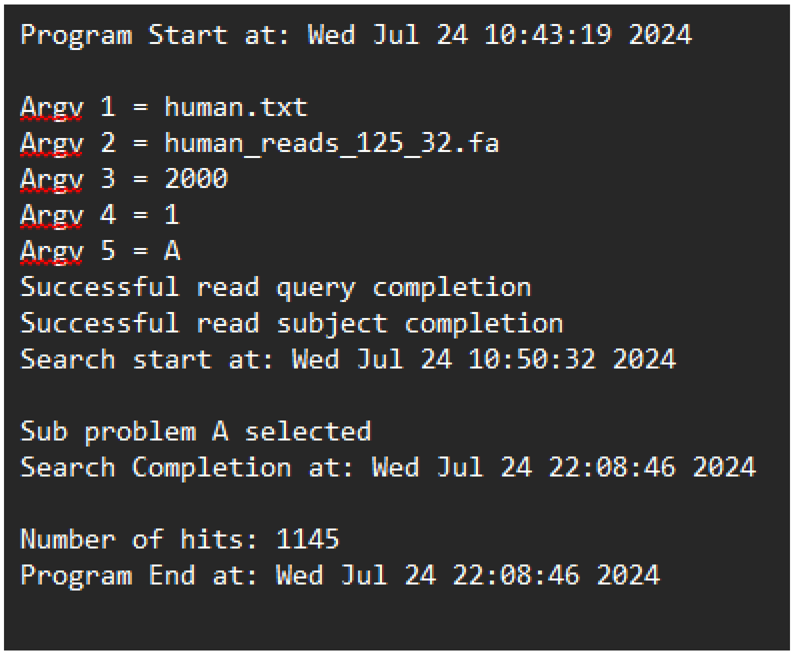
```
Program Start at: Thu Jul 25 11:29:32 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 100
Argv 4 = 1
Argv 5 = A
Successful read query completion
Successful read subject completion
Search start at: Thu Jul 25 11:36:39 2024

Sub problem A selected
Search Completion at: Thu Jul 25 12:10:08 2024

Number of hits: 35
Program End at: Thu Jul 25 12:10:08 2024
```

Output for 1000 random index's (6 hours)



```
Program Start at: Tue Jul 23 17:00:47 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 1000
Argv 4 = 1
Argv 5 = A
Successful read query completion
Successful read subject completion
Search start at: Tue Jul 23 17:07:36 2024

Sub problem A selected
Search Completion at: Tue Jul 23 22:48:53 2024

Number of hits: 621
Program End at: Tue Jul 23 22:48:53 2024
```

Output for 2000 random index's (12 hours)



```
Program Start at: Wed Jul 24 10:43:19 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 2000
Argv 4 = 1
Argv 5 = A
Successful read query completion
Successful read subject completion
Search start at: Wed Jul 24 10:50:32 2024

Sub problem A selected
Search Completion at: Wed Jul 24 22:08:46 2024

Number of hits: 1145
Program End at: Wed Jul 24 22:08:46 2024
```

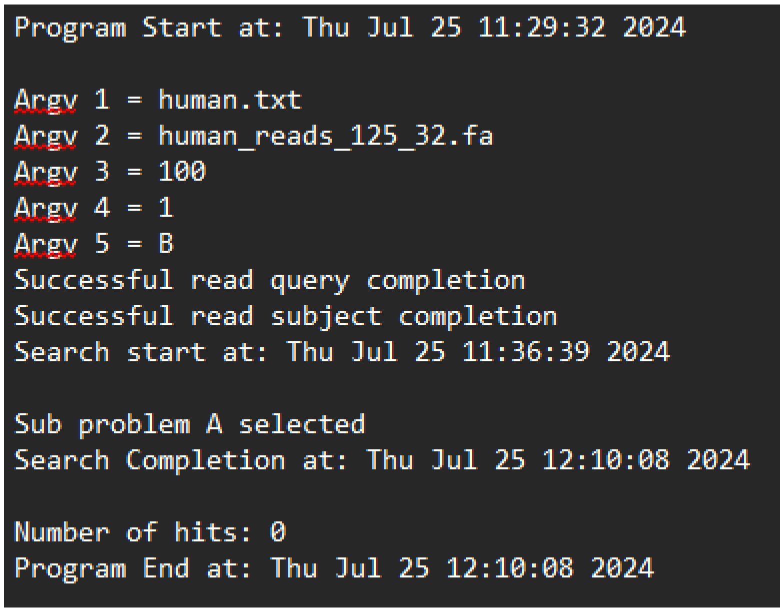
As we can see from the output in Problem 1 Part A, the number of hits is low, and the completion time is low as well since I had to decrease the input number of queries to save days on the runtime. Now this was a good way to test if the program was working and to see how many matches happen with the lowered query count, and I can say that the number of hits will scale with the number of queries thrown in. That is, I suspect that since I lowered the number of queries by about ¾ the total amount, I can say the number of hits would scale by a multiplier of 3. This deduction was a calculated measure to complete the homework in a reasonable amount of time due to the problem of queuing with all the other students. I figured that a runtime over a day was a little overkill, and I could scale it back to get reasonable results and just approximate what the results would look like with the actual data. I suspect that it would take around 3 billion hours to complete the whole subject dataset. This is most likely still a low number, but I calculated this with the big O runtime stated in the explanation of part A.

Problem 1 part B was almost the exact same as part A, but instead of generating random indexes in the genome array to create the test fragments, I would simply generate a completely random test fragment from scratch, which means I would not need to read in the genome array, which would save a couple minutes at the beginning of the program. Now, generating the random test fragments would happen at the top of the search loop, and I would create a modular function to return a string of a given length. This string would consist of As, Cs, G, T, and N's.

Then the process from there would happen the same, where there would be a nested loop inside the main search loop that would iterate through the query array and calculate the similarity score of the test fragment and the current query string in the Needleman Wunch function. I would follow the same logic for determining if it was a hit or not, and the runtime would be just about the same, but I would save resources and runtime in the beginning since I would not need to store the whole genome of 3 billion characters. So, part B would have a cheaper space complexity of O (genome size). The runtime would come out to O (query size) x O (loop iteration parameter) x O (query size) x O (fragment size^2).

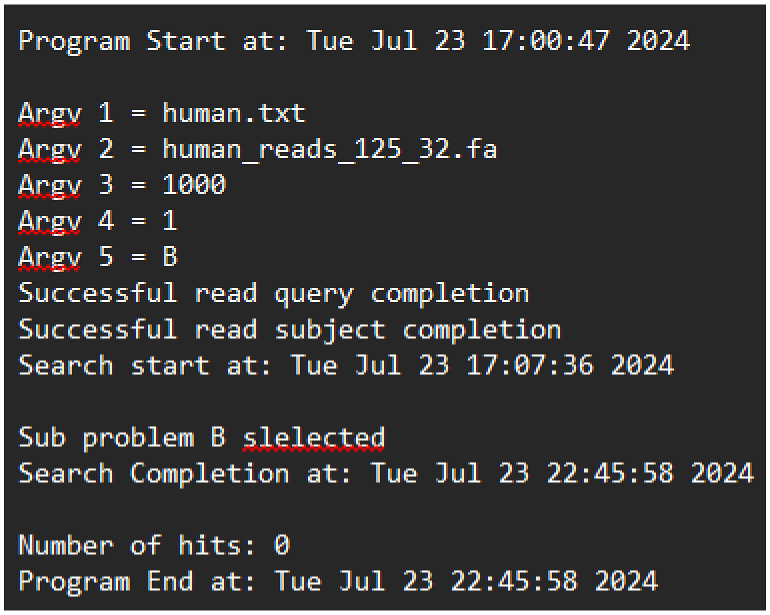
Outputs

Output for 100 random indices' (45 minutes)



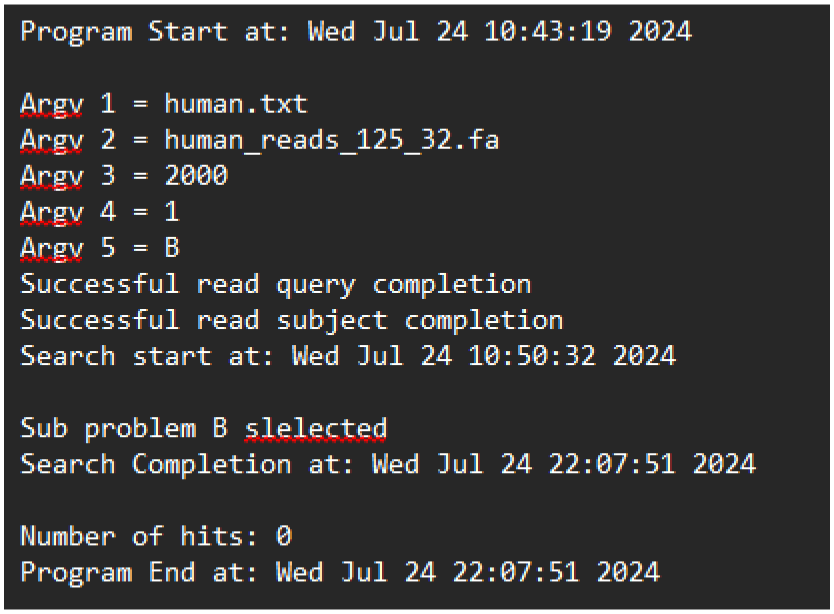
```
Program Start at: Thu Jul 25 11:29:32 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 100
Argv 4 = 1
Argv 5 = B
Successful read query completion
Successful read subject completion
Search start at: Thu Jul 25 11:36:39 2024

Sub problem A selected
Search Completion at: Thu Jul 25 12:10:08 2024

Number of hits: 0
Program End at: Thu Jul 25 12:10:08 2024
```

Output for 1000 random index's (6 hours)



```
Program Start at: Tue Jul 23 17:00:47 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 1000
Argv 4 = 1
Argv 5 = B
Successful read query completion
Successful read subject completion
Search start at: Tue Jul 23 17:07:36 2024

Sub problem B slelected
Search Completion at: Tue Jul 23 22:45:58 2024

Number of hits: 0
Program End at: Tue Jul 23 22:45:58 2024
```

Output for 2000 random index's (12 hours)



```
Program Start at: Wed Jul 24 10:43:19 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 2000
Argv 4 = 1
Argv 5 = B
Successful read query completion
Successful read subject completion
Search start at: Wed Jul 24 10:50:32 2024

Sub problem B slelected
Search Completion at: Wed Jul 24 22:07:51 2024

Number of hits: 0
Program End at: Wed Jul 24 22:07:51 2024
```

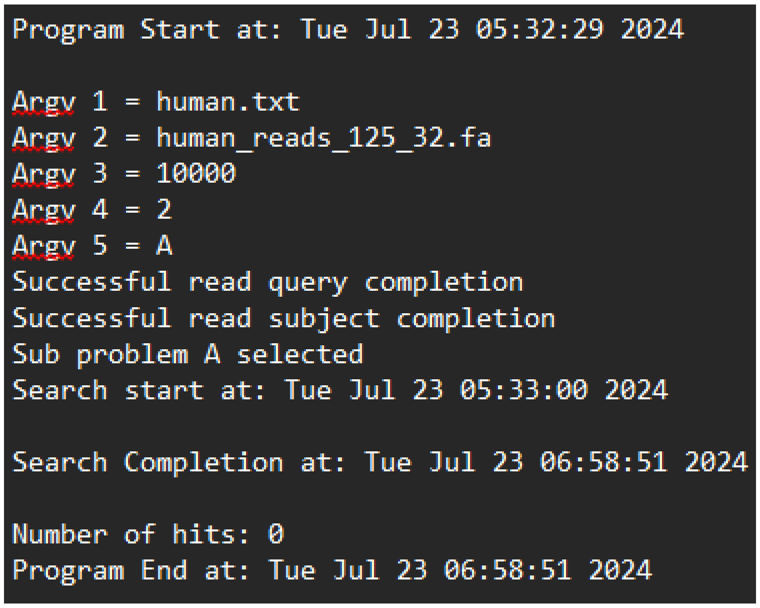
Looking over the outputs for part B of problem 1, we can see the runtime is very similar to part A since they are technically identical except the input data was random in part B. The interesting part is that the number of hits was 0 in this case. I suspect that this is mostly due to the shorted number of queries read in due to runtime. I also suspect that the randomness of the subject also played a part in the lack of hits. When testing the random function, I did see that it created correctly formatted random strings that would register as a hit if found in the query. I think if I had the time to run with the full query amount, I could have gotten a small number of hits, but it still wouldn’t be close to the number of hits in Part A.

Problem 2)

For problem 2, I was tasked with creating a similar program to problem 1 in the sense of using a Needleman-Wunch algorithm to determine hits in our query array. For this problem, I will be incorporating the BLAST algorithm to help with the runtime for comparison. To begin this program, I would read in the query data into a 2D character array where each index is a query fragment of size 32. I would then read in the whole genome data into a character array, which was about 3 billion characters. Now that all the data is read in, I would call a search function, which will generate a completely random n-mer string from the genome array. Now the size of this n-mer string would come from the input parameters, which for this homework assignment were 10k, 100k, and 1M. So, I would call a function that would generate a completely random index from the character array ranging from 0 to the genome size (n-mer size). Once that n-mer string was created, it would then be passed to the BLAST function, where it would be dissected. The first thing to happen when landing in the BLAST function would be to break up the n-mers into seeds and store the seeds inside a hash table to create an easy way to search for fragments. A seed is a small portion of the fragments we want to test. For this homework, we initially use a seed size of 11. The hash table would store these seeds, which would account for a search time of O (1) x O (linked list length). Now that the n-mers seeds were loaded into the hash table, I can now iterate through the query array and at each query index in the query array. I would take that fragment and break it into seeds as well. I would then look for the query fragment seeds in the hash table. If a seed was found inside the hash table, then my program would transition into the comparing sequence. To start the comparing sequence, I would grab the location of the seed in the hash table and extract the n-mer index from the seed. This index would be where the seed was in the n-mer string. The algorithm would then expand both the left and right of the index in the n-mer string, along with expanding the query fragment left and right. These expansion strings would then be passed into the Needleman Wunch function to determine the similarity score. Just like in problem 1, I would only be allowed up to two-character mismatches to increment the hit count. And then the process would repeat for the remaining seeds in the current query fragment. Then that whole process would repeat for the remaining query indexes. This runtime would equate to O (query size) x O (genome size) x O (query size) x O (n^3).

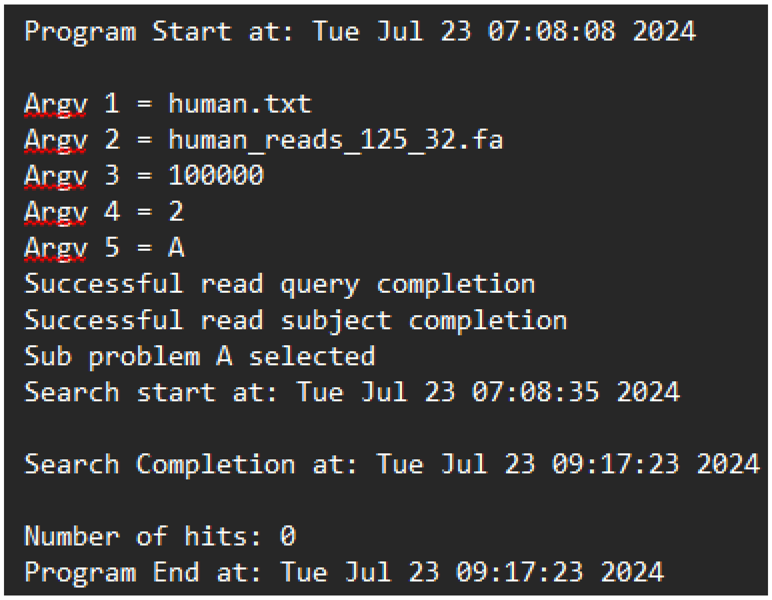
Outputs

10000 length n-mer (45 minutes)



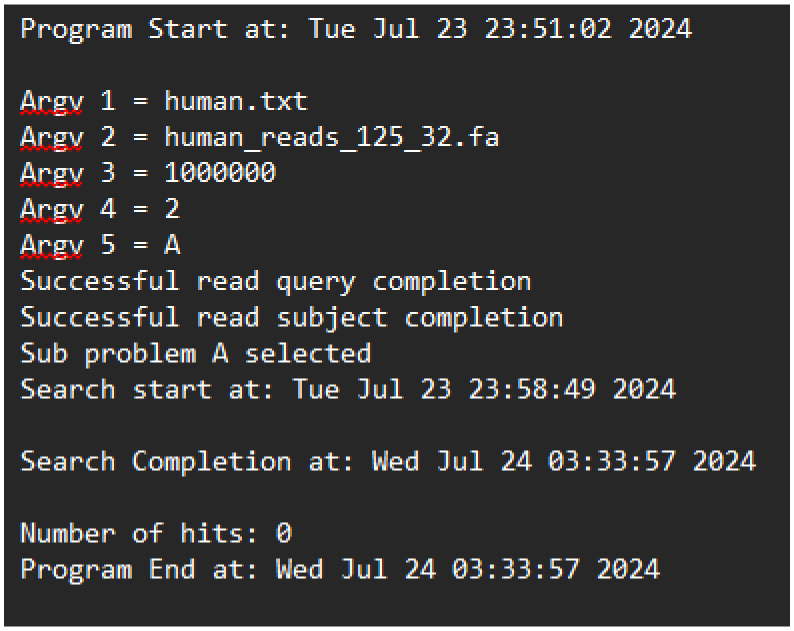
```
Program Start at: Tue Jul 23 05:32:29 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 10000
Argv 4 = 2
Argv 5 = A
Successful read query completion
Successful read subject completion
Sub problem A selected
Search start at: Tue Jul 23 05:33:00 2024

Search Completion at: Tue Jul 23 06:58:51 2024

Number of hits: 0
Program End at: Tue Jul 23 06:58:51 2024
```

100000 length n-mer (2 hours)



```
Program Start at: Tue Jul 23 07:08:08 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 100000
Argv 4 = 2
Argv 5 = A
Successful read query completion
Successful read subject completion
Sub problem A selected
Search start at: Tue Jul 23 07:08:35 2024

Search Completion at: Tue Jul 23 09:17:23 2024

Number of hits: 0
Program End at: Tue Jul 23 09:17:23 2024
```

1000000 length n-mer (4 hours)



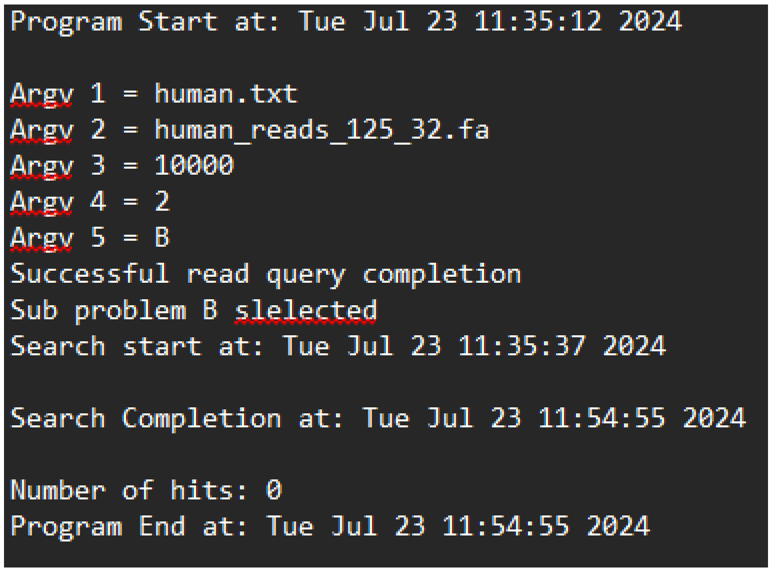
```
Program Start at: Tue Jul 23 23:51:02 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 1000000
Argv 4 = 2
Argv 5 = A
Successful read query completion
Successful read subject completion
Sub problem A selected
Search start at: Tue Jul 23 23:58:49 2024

Search Completion at: Wed Jul 24 03:33:57 2024

Number of hits: 0
Program End at: Wed Jul 24 03:33:57 2024
```

Looking over the outputs for Part A of Problem 2, we can see that the runtime was much faster than Problem 1, and I was able to run with the full query amount. This is due to the BLAST algorithm's method of breaking down the fragments into seeds, testing them at their simplest form, and then only testing them further if they're a potential match. The only issue was that there were no hits found, which was strange. I used the same method for calculating hits in problem 1, but since this is a heuristic algorithm, it does not guarantee a hit. I think this was mainly due to the seed size of 11, since it is about 1/3 of the whole fragment length, which speeds up the runtime but decreases the accuracy of the hits. I tried turning the seed size down to 7 and didn’t see any changes to the hit count. If I wasn’t tight on time, I would ultimately try a seed size of about 3, but I suppose it would take longer than a day to complete for all test sizes. To search for the whole subject data set I would suppose it would take about 1 week to complete due to how quickly it was able to run the 1 million test fragments. The only issue would be the accuracy of the blast algorithm. Now this runtime also depends on the seed length and if the seed length goes even smaller than it could possibly double to 2 weeks to read through the entire subject dataset.

Finally, in problem 2, part B I confidently say it was basically the exact same as part A, but our input data for the subject data was randomly generated. This means that instead of reading in the subject data to create our large n-mer character segment, the program would just create its own n-mer character segment with random As, Gs, Cs, Ts, and Ns. Which means that the number of hits will be a bit more random compared to the human.txt input data. It would follow the same steps for searching as stated in Part A and would conduct the same hit test as Part A as well. The only change was the input data, which, as we can see in the outputs below, was not the best due to the heuristic approach, which does not guarantee a hit.

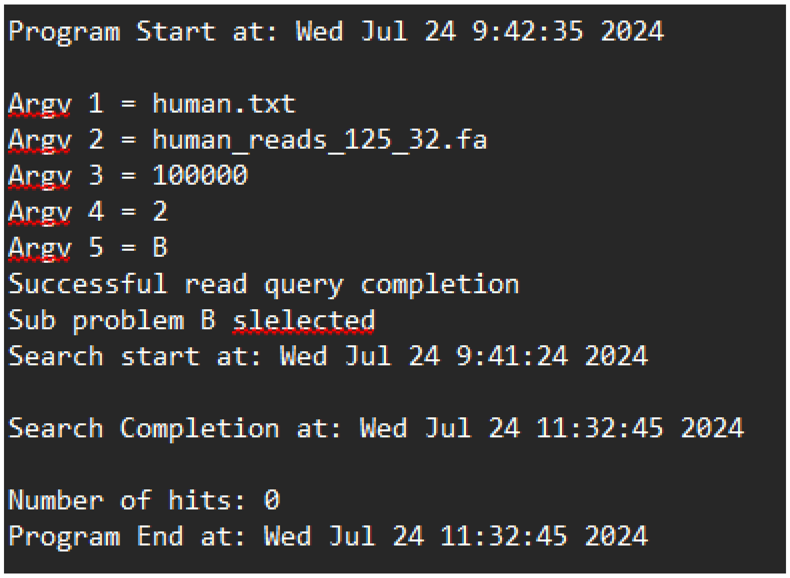
Outputs

10000 length n-mer (45 minutes)



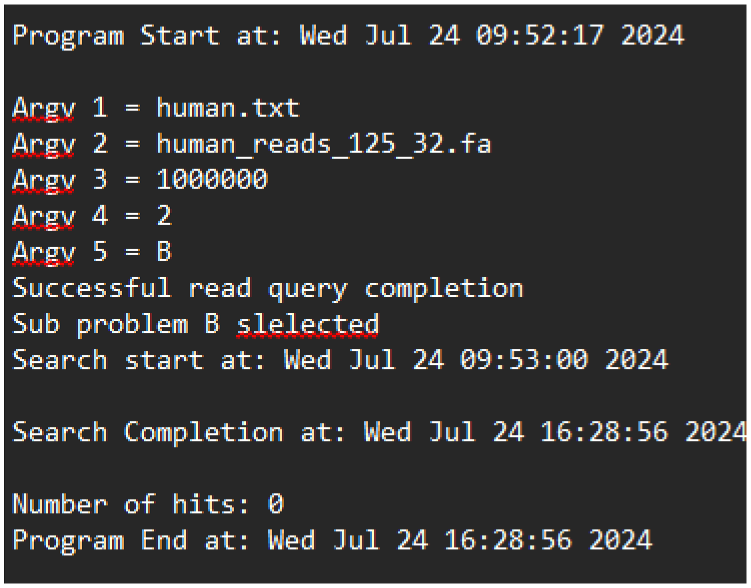
```
Program Start at: Tue Jul 23 11:35:12 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 10000
Argv 4 = 2
Argv 5 = B
Successful read query completion
Sub problem B slelected
Search start at: Tue Jul 23 11:35:37 2024

Search Completion at: Tue Jul 23 11:54:55 2024

Number of hits: 0
Program End at: Tue Jul 23 11:54:55 2024
```

100000 length n-mer (2 hours)



```
Program Start at: Wed Jul 24 9:42:35 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 100000
Argv 4 = 2
Argv 5 = B
Successful read query completion
Sub problem B slelected
Search start at: Wed Jul 24 9:41:24 2024

Search Completion at: Wed Jul 24 11:32:45 2024

Number of hits: 0
Program End at: Wed Jul 24 11:32:45 2024
```

1000000 length n-mer (4 hours)



```
Program Start at: Wed Jul 24 09:52:17 2024

Argv 1 = human.txt
Argv 2 = human_reads_125_32.fa
Argv 3 = 1000000
Argv 4 = 2
Argv 5 = B
Successful read query completion
Sub problem B slelected
Search start at: Wed Jul 24 09:53:00 2024

Search Completion at: Wed Jul 24 16:28:56 2024

Number of hits: 0
Program End at: Wed Jul 24 16:28:56 2024
```

Now for the results for part B of problem 2, we can see that it is about the same as part A in the sense that there were no hits found. Like I stated in the breakdown of Part A, I thought this was strange, and I directly tied it to the seed size since I knew the functionality for testing hits was the same as in Problem 1. Now the results compared to problem 1, part B, were the same for the number of hits, which was comical, but that was due to the limited queries read in for problem 1. Now we can see that the runtime is much better than problem 1, part B, which is the only thing that I can directly compare to in this document. The program was able to read in all the queries and look at even more test fragments in a much shorter amount of time.

Key Differences

The biggest difference between Part A and Part B for both problems is the difference between the hits. In part A, we were using part of the human genome, and in homework's past, I knew we had matches between the subject data and the query data. Now in Part B, we would be creating our own subject data from randomness, which means there was a larger possibility for less or more matches depending on how well the randomness was. In my case, the program couldn't find matches very well with the randomness compared to the actual genome. The randomness did help the program's runtime and space complexity since it wouldn't have to rely on data that would be read in since it would just create its own data when it needed it. The key difference between problem 1 and problem 2 was the search style. In problem 1, we are strictly relying on the exhaustive fuzzy matching Needleman Wunch algorithm, which does guarantee a match if it is present, but the runtime was larger by O (n). Problem B, on the other hand, used the

BLAST algorithm along with the Needleman Wunch, so it was technically a heuristic fuzzy matching algorithm, which meant that it wouldn’t guarantee a match if it existed, but if it did, it would complete much faster. The BLAST algorithm was very similar to a divide and conquer algorithm, which in the past has been very effective compared to linear-based algorithms. Problem 1 I would compare closely to linear search since it would be testing one full fragment at a time throughout the whole process. The blast algorithm also had some extra steps, which made it a little more confusing in the development stages due to the seeds, hash table, and expansion steps.

Bugs encountered

One large bug that I encountered with the blast section of the homework was the expansion of the seeds in the subject fragment. After the seeds are created from the subject data, they are inputted into the hash table for easy lookup. When the program iterates through the query array and tests the current fragment seeds against the hash table seeds, there will only be two types of returns. The first being that if there’s a match, it will return the subject index where the seed is in the subject data. The other would be if there was no match, and it would simply return -1. Now the problem came when the results were returned to the blast function, and there would be an if statement check to see if the returned subject index was greater than 0. Now, for some reason, if the statement was allowing the -1 to pass through, when the test strings would be created for the Needleman Wunch function, the test fragment string would have a random garbage character at the beginning. This led to incorrect scoring and really slowed down the development process. I fixed this by just making sure the value returned from the seed search was not -1, which made sure that the expansion for the left wouldn’t go over the left side of the array.

Optimization

The only optimization I made in this program was in both problems, part B. Since part B in both problems will just be their own random data, I would essentially not need to read the subject data since it is not being used. At the start of development, I was always reading the subject data but soon saw that it was redundant and was just a waste of runtime and space. This way, I can save a bit of time, and more importantly, I wouldn’t need to use as much memory since I wouldn’t have to store the 3 billion characters in memory.