CSCI-1200 Data Structures — Spring 2017
Homework 9 — MiniBLAST Hash

BLAST (**B**asic **L**ocal **A**lignment **S**earch **T**ool) is an algorithm for fast, efficient searching of databases of proteins, DNA, and RNA sequences. A BLAST search allows a researcher to compare a query sequence against a library of sequences. For example, after sequencing a gene from a newly discovered bacteria, BLAST can be used to determine if the gene is similar to any genes in known bacteria. You can read more about BLAST at https://en.wikipedia.org/wiki/BLAST or try it out at http://blast.ncbi.nlm.nih.gov/Blast.cgi.

Please carefully read the entire assignment before beginning your implementation.

In this homework assignment, we create an extremely simplified version of BLAST. Our version works in a manner somewhat similar to BLAST, but the BLAST algorithm is sophisticated and BLAST software is highly optimized. BLAST is used to regularly search Genbank, a repository of sequence data containing over 1 trillion bases (letters) of sequence data. Online searches are typically returned in less than a minute. We won't try anything quite so ambitious. Our version of BLAST, MiniBLAST, will search small genome files with query strings in a DNA alphabet (A,C,G,T).

The genome files that we will search will consist of a number of lines, each line containing letters from the DNA alphabet. Here are the first few lines of the *genome_small.txt* file:

```
$ head genome_small.txt
TAATGACCTAAATAATCTAAACAAAAGGAAGAGAGATAGTCCGGATTACC
TGGGACATGGAAAACCCTCCTTTCTCTCATCAGCTTCCCACCCCACCTCT
GCCCAGCGCTAATCATGATTTAATAGCCTTCCTTAATCACTTACTCTGTT
TGCTGCTTCATCTAAAAACTTAAGATGCTCTGGGTTAGATCACAGTCTAA
CTCATCACAAATGGATAGAACGACCTGGTAGTTTTCCAGATTTCCATTGT
CCAAACTAATCAGCCAACACACTCAATGATGCACATTATTTTCCACGTAT
ATGGCCTTAGAGATGGGACTAAAAGTCCCGACTGTACTGAGGATGTTTGA
CAGGTTTTGCCATTCTAACTGCTAGTGCTGTGTAATGTGGCTAGGAAGAA
GCAAGGAACAGAGAGATACAGATATGATTTCCTGGGACCAGCTATAGGAG
AGATTCTTCAATTACATCATCTTTGCTCATCCCAAACACCTTGACAAGTA
```

The genome data above is a small region from human chromosome 18.

The queries will be also strings from the DNA alphabet. A typical query could look like:

```
CTCATCACAAATGGATAGAACGACCTGGTA
```

This query can be located at the start of the 5th line.

The strategy that we will employ is to index the genome file with a series of *k-mers*. A *k-mer* is a sequence of k letters from the DNA alphabet, where k is an integer. A given *k-mer* may appear many times within the genome.

We index the genome by building a **hash table** with the *k-mers* as the keys. The hash table is to be implemented with an array or a vector at the top level. A *hash function* will map the *k-mer* to a position in the table. You will chose an appropriate structure to store the *kmers* and genomic locations of the *k-mers*, i.e. the positions where they are found in the genomic sequence. You build the hash table by iterating through the genome sequence with a series of overlapping windows of length k, calculating the the index into the hash table using the hash function. Store the *kmer* and its genomic location in the table. When iterating through

the genome sequence, the first *k-mer* is the genome sequence from 0 to k-1; the second is the sequence from 1 to k, etc.

When searching a biological sequence database, it's often the case that we do not find an exact match to a query string. MiniBLAST will process queries of varying lengths and allow for mismatches between the genome and query string. To search the genome, MiniBLAST uses the first k letters of the query string as a seed. It is important that searching the database for the initial seed be efficient. Thus, the choice of hash function and table implementation is important. If the seed can be found in the table, the program should try to extend the match by adding letters from the indexed genomic position until the full query is matched or the allowed number of mismatches is exceeded. For simplicity we require that the seed be an exact match. The mismatches may occur anywhere after the seed in the match string.

Choice of Hash Function and Table Implementation

The choice of the hash function is up to you. A good hash function should be fast, O(1) computation, and provide a random, uniform distribution of keys throughout the table. You may use one of the hash functions mentioned in lecture, one found on the internet, or one of your own devising. If you choose to download a hash function from the internet, you must provide the URL in your README and include the source code with your submission. If the downloaded file requires a copyright notice, you MUST include that notice. Be sure to observe any copyright restrictions on the use of the code. In your README file, describe your hash function and table implementation.

A typical *k-mer* will be found in several location in the genome. Your hash table implementation should enable efficient retreival of the multiple locations where the *k-mer* is found.

To store the *k-mers* and their genomic positions in the hash table, once the table index of the *kmer* key has been found, you may use any of the data structures that we have covered so far in class. To handle collisions, use one of the open addressing methods described in lecture (linear probing, quadratic probing, or secondary hashing). Linear probing is the simplest of these three methods. You may not use *std::hash*, *std::unordered_map*, *std::unordered_set*, *std::map* or similar STL functions/containers.

When implementing the hash table, set the initial size of the table. As you enter data in the table, calculate the occupancy ratio:

$$occupancy = \frac{number\ of\ unique\ key\ entries}{table\ size}$$

When the *occupancy* >than some fixed level, double the size of the table and rehash the data. Describe your re-sizing method in the hash table section of the README file.

Input/Output & Basic Functionality

The program should read a series of commands from std::cin (STDIN) and write responses to std::cout (STDOUT). Sample input and output files have been provided. You can redirect the input and output to your program using the instructions in the section **Redirecting Input & Output** found at http://www.cs.rpi.edu/academics/courses/spring17/ds/other_information.php

Your program should accept the following commands:

- *genome filename* - Read a genome sequence from *filename*. The genome file consists of lines DNA characters.
- *table_size N* - this is an optional command. N is an integer. It is the initial hash table size. If it does not appear in the command file, the initial table size should be set to 100.
- *occupancy f* - this is an optional command. f is a float. When the occupancy goes above this level, the

table should be resized. If it does not appear in the command file, the initial level should be set to 0.5.

- *kmer k* - *k* is an integer. The genome should be indexed with *kmers* of length *k*.
- *query m query_string* - Search the genome for a match to *query_string* allowing for *m* mismatches.
- *quit* - Exit the program.

Ignore blank lines in the file.

Here is some sample input, showing typical commands:

```
genome genome_small.txt
table_size 100
occupancy 0.5
kmer 10
query 2 TATTACTGCCATTTTGCAGATAAGAAATCAGAAGCTC
query 2 TTGACCTTTGGTTAACCCCTCCCTTGAAGGTGAAGCTTGTAAA
query 2 AAACACACTGTTTCTAATTCAGGAGGTCTGAGAAGGGA
query 2 TCTTGTACTTATTCTCCAATTCAGTCACAGGCCTTGTGGGCTACCCTTCA
query 5 TTTTTTTTTTTTTTTCTTTTTT
quit
```

You may assume that the commands will appear in the input files in the order shown above.

For output, the program will report the query, and if matches are found, the genome position (positions start at 0) of the match, the number of mismatches between the genome and the query string, and the genome sequence matching the query. If a match can't be found, the program will report "No Match".

The corresponding output to the input above should be:

```
Query: TATTACTGCCATTTTGCAGATAAGAAATCAGAAGCTC
504 0 TATTACTGCCATTTTGCAGATAAGAAATCAGAAGCTC
Query: TTGACCTTTGGTTAACCCCTCCCTTGAAGGTGAAGCTTGTAAA
5002 2 TTGACCTTTGGTTAACCAATCCCTTGAAGGTGAAGCTTGTAAA
Query: AAACACACTGTTTCTAATTCAGGAGGTCTGAGAAGGGA
4372 0 AAACACACTGTTTCTAATTCAGGAGGTCTGAGAAGGGA
Query: TCTTGTACTTATTCTCCAATTCAGTCACAGGCCTTGTGGGCTACCCTTCA
No Match
Query: TTTTTTTTTTTTTTTCTTTTTT
4428 0 TTTTTTTTTTTTTTTCTTTTTT
4429 3 TTTTTTTTTTTTTTCTTTTTTG
4430 4 TTTTTTTTTTTTTCTTTTTTGA
4431 5 TTTTTTTTTTTTCTTTTTTGAG
```

You are not explicitly required to create any new classes when completing this assignment, but please do so as it will improve your program design. We expect you to use `const` and pass by reference/alias as appropriate throughout your assignment.

Order Notation

In your `README.txt` file, report the time and space complexity (order) of your implementation for building the index for a genome of length **L**. Does the *k-mer* size, **k**, and the average number of locations, **p**, where the key is found affect your answer? What is the order time notation for matching a query of length **q** in a genome of length **L** when the key size is **k** and the key is found at **p** different genomic positions?

Extra Credit

Add a new command to implement the database using one of the other data structures that we have covered so far in the course: vectors, lists, arrays etc. Compare the performance your alternative method to the homework method by making a table of run times for each of the genomes and query sets provided with the homework and compare the times to build the index and the times to process the queries. Document any new commands you have added in your README file.

Submission

Use good coding style and detailed comments when you design and implement your program. Please use the provided template `README.txt` file for any notes you want the grader to read, including work for extra credit. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity". If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file.