Max Hartman and Michelle Ru
Project Proposal
CS 225 Extra Credit Project
KMP Algorithm

Academic Reference:

We will be implementing the Knuth-Morris Pratt (KMP) algorithm. The academic reference we are using is: https://epubs.siam.org/doi/abs/10.1137/0206024?journalCode=smjcat

Algorithm Summary:

This algorithm is used to find if and where a pattern is contained inside a larger string. The algorithm inputs two strings and outputs all of the indices at which a match is found. This algorithm is useful because it beats the runtime of the traditional naive string search algorithms in cases where we search for a^nb strings in a^yb strings, where n << y (e.g. ‘aaaaaab’ in ‘aaaaaaaaaaaaaaaaaab’). Even strings such as palindromes can be found in O(n+m) time, where n is the length of the string and m is the length of the pattern. The reason why this algorithm is so efficient is because it involves pre-processing the input to generate an array that allows us to skip over certain characters when we actually process the string to find matches.

Function I/O:

The expected overall input of our code will be the absolute path of the file we want to run our algorithm on. From there, we will loop through and conduct our search on each strand of DNA (each line of the txt file). We will also need to input the pattern that we would like to search for. The expected output for each line of DNA will be a vector (in numerical order) of all of the indices where a match is found.

In order to implement this algorithm, we will split the overall task of searching for the string into three main “stages".

The first stage will be processing the text file, which will be done by the file_process method. This method will input a string with the absolute path of the file we want to work with. It will output a vector of strings that will then be looped through by the second and third stages of the algorithm in order to conduct the actual searches. Here is the method header for the first method:

```
std::vector<std::string> file_process(std::string filename);

@param filename - the absolute path of the txt file
```

```
@return a vector containing the data
```

The second stage will be creating the pre-processing function, which will input the pattern that we are searching for and output a vector that will be used in the final stage to determine which characters in the string we can skip over during the search. This function (in addition to the next function) will be called on each string of the vector outputted by the file_process method (which in turn corresponds to each line of DNA in the txt file). Here is the method header for the this function:

```
std::vector<int> pre_process(std::string pattern);

@param pattern - the pattern that we want to search for in each line
of DNA

@return the auxiliary vector used in the final part to determine
which characters will be skipped
```

The final stage of the algorithm will be the actual processing of the string in order to find matches. This will be done using a function that inputs the DNA strand we are searching through, the pattern we want to find, and the vector that was outputted by the first stage. The output will be a vector of all the indices of the overall DNA strand that our pattern is found in. This method will be run on each string in the vector returned by stage one. Here is the method header for the main function that is called after the pre-process function:

```
std::vector<int> kmp(std::string dna_strand, std::string pattern,
std::vector<int> pre_process);

@param dna_strand - the DNA strand we are looking through

@param pattern - the pattern we want to search for

@param pre_process - the vector used to determine which characters
will be skipped

@return a vector that contains all of the indices that the pattern is
found in the dna strand
```

To test our code, we wrote a series of 6 tests. The “PROCESS FILE” test case tests stage one of the algorithm, ensuring that we correctly process the absolute filename into a vector of strings, where each string corresponds to one line of the txt file.

Tests 1 through 5 test stages two and three of the algorithm. Each test is run on one line of DNA. The test cases have two “require” statements. The first “require” ensures that stage 2 (the preprocess function) correctly generates the auxiliary vector. The second “require” ensures that stage 3 (the actual search part of the code) correctly outputs a vector containing the indices at which a match is found. If all outputs match their expected results, on a diverse test set, we can safely assume our algorithm works as intended.

Data Description:

We have five text files, each containing a different number of DNA strands differing by one order of magnitude: 5 (xs.txt), 50 (s.txt), 500 (m.txt), 5000 (l.txt), and 50000 (xl.txt). To generate the DNA in the text files, we used an online DNA generator found at https://molbiotools.com/randomsequencegenerator.php. We set a constant GC content of 50% and also made each strand of DNA 100 bases long. We then calculated how many total bases would be in each file by multiplying the number of strands by 100. We then used the formatting tools on the generator in order to insert a line break every 100 bases. For example, for s.txt we wanted 50 lines of DNA, each 100 bases long. To do this, we generated a DNA strand that was 50 * 100 = 5000 bases long, and then inserted a line break every 100 bases. We then repeated this process for each data set.