Introduction

DNA (Deoxyribonucleic acid) and RNA (Ribonucleic acid) sequence searching has been one of the most fundamental problems faced in bioinformatics research. The genome sequence files generated can be as small as a few kilobytes, or as large as hundreds of gigabytes. To process the larger files would require an immense amount of time. DNA and RNA only consist of 4 unique characters, with DNA consisting of "GACT" and RNA consisting of "GACU", instead of the usual 26 characters.

Aim

The aim of this project is to explore algorithms that speed up the searching process over brute force search. The report discusses both theoretical and practical time complexities between different algorithms (Brute Force, Boyer Moore, Knutt-Morris-Pratt) and compares their respective performances.

Resources

Genome sequences were downloaded from the National Center for Biotechnology Information (NCBI). The team worked on 3 files of different sizes:

File name	Size
COVID.FNA	30 KB
SALMONELLA.FNA	5 MB
INFLUENZA.FNA	1.4 GB

Possible limitations and issues

The file to be referred to would be the file appended with "new". Example: For the file "covid.fna", the file with the joined dna sequence would be named "new_covid.fna". This would contain the matched indexes.

With the file sizes of the genome sequence being so large, it would be impractical to load the entire file directly into the system, as we would run into memory limitations. In our implementation, we strove to avoid this problem by splitting up our data into chunks by a process called chunking. For example, a 1.4 GB file would be split into 14 files, each of size 100 MB (0.1 GB), which would then be processed separately. The size of the chunks can be adjusted accordingly. With chunking, there is no hard limit on the size of the files the team can process. The team has performed tests of files up to 1.4 GB with no issues faced.

A buffer is also implemented in between chunks to ensure that we would not miss out on the sequences that were cut apart during the splitting process, and ensure that any DNA sequences caught in between chunks are accounted for. With regards to the buffer size, it would be initialised to be the length of the genome sequence we are querying for. With this length as the buffer, there would not be any DNA sequences caught in between chunks that would be lost.

Plan of action

For our plan of action, we would first implement the Brute Force algorithm as the baseline. Following that, the team chose a few established algorithms, coupled with some innovative ideas that we came up

with. Next, we performed empirical analysis on the results and finally some comparisons of the results, from which we will draw our conclusion from. In the following section, we would be analysing some of the algorithms that we have implemented. For consistency, our analysis will refer to *m* as the length of the pattern to query, and *n* as the length of the genome.

Brute Force

This is the naive string searching algorithm which uses 2 nested loops. It loops through the entire sequence and in case of a mismatch, it backtracks and shifts the outer-loop variable by 1.

Best Case

The query is not present in the sequence. Therefore, the first character of the query never matches any character of the sequence. The inner loop is never executed.
E.g. Text = "ABCDABCDABCDABCD" and Pattern = "ZZZZ".
Let the compiler perform c1 operations in the outer for loop and c2 operations in the nested inner for loop, where c1 and c2 are constants.
Total number of operations = (n-m+1) * c1
Time complexity is O(n-m) ≈ **O(n)**

Worst Case

When all characters of the genome sequence match the query, or when all characters of the genome sequence and query are the same, except the last.
E.g. Text = "AAAAAAAAAAA" and Pattern = "AA".
E.g. Text = "AAAAAAAAAAAB" and Pattern = "AAAB".
Number of iterations of outer loop = n-m+1
Number of iterations of inner loop = m
Total number of operations = (n-m+1) *(m*c2 +c1)
Time complexity = **O(nm)** ≈ (m^2 << mn, therefore neglected)

Average Case

The number of times the outer loop runs = n - m + 1
The inner loop may run x times where x ∈ [1, m]. Each value of x has an equal probability of 1/m.
Number of iterations of inner loop =

$$\frac{1}{m}\sum_{i=1}^{m} i * C = \frac{1}{m}(C + (2 * C) + (3 * C) + \ldots + (M * C))$$

$$= \frac{C}{m}(\frac{m}{2})(m + 1) = \frac{m+1}{2}C$$

Total number of iterations = $((\frac{m+1}{2})C)$(n-m+1) $= ((\frac{mn-m^2+n+1}{2})$
Time complexity is therefore ≈ **O(mn)**

Boyer Moore

It essentially has three key features: scanning from right to left, generating a bad character table and the good suffix table as well. With these tables there would be predefined values where you would skip searches based on what you see during the search. These tables are only generated with the query sequence that you are looking for and does not require preprocessing on the fna file.

Best Case

The condition for the best case is when the query is completely not present in the sequence.
E.g. Text = "ABCDABCDABCDABCD" and Pattern = "XYZ".
Since the last character of the pattern, "Z", does not match the third character "C" of the pattern, and "C" does not match any character from the pattern "XYZ" (using a bad match table), the "pointer" shifts 3 positions ahead. This process repeats until the end of the text is reached. Hence, number of comparisons = Length of text / Length of pattern. Time complexity = **O(n/m)**, where n = length of text, m = length of pattern.

Worst Case

The condition for the worst case is when every letter in the pattern is present in the text.
E.g. Text = "AAAAAAAAAAAAAA" and Pattern = "AAA".
By the formula, value of char in pattern = length of pattern - index - 1, the (most recent) value of A is 1. Firstly, the pattern is matched against the first 3 characters of the text, which constitutes the first 3 comparisons. Then, the "pointer" shifts one position ahead (owing to the value of "A") and makes the next 3 successful comparisons. This process continues till the end of the text.
Hence, the number of comparisons = length of text * length of pattern, time complexity = **O(m*n).**

Average case

According to available research (Baeza-Yates, R. A., & Régnier, M, 1992), the expected number of text-pattern comparisons c, is linear in the size of the text for Boyer Moore Horspool algorithm. As a similar variant with the same working principles and through mathematical induction, the Boyer Moore algorithm should also have the same time complexity of **O(n)**.

Knuth-Morris-Pratt

The main idea of Knutt-Morris-Pratt (KMP) is to reduce the amount of backtracking when a character of a given pattern (*m*) does not match with a character of the string we are searching in (*n*). This is achieved by making use of previous information of where the last matched character is, and continuing the pattern matching from that position. A table (called *Pi/π table*) will store the prefixes (first occurrence of a character) and suffixes (subsequent occurrence of a character) of the pattern. The *π table* will then be used in the search to find the last matched character when there is a mismatch between m and n, and it will try to match the characters again starting from that position.

Best Case

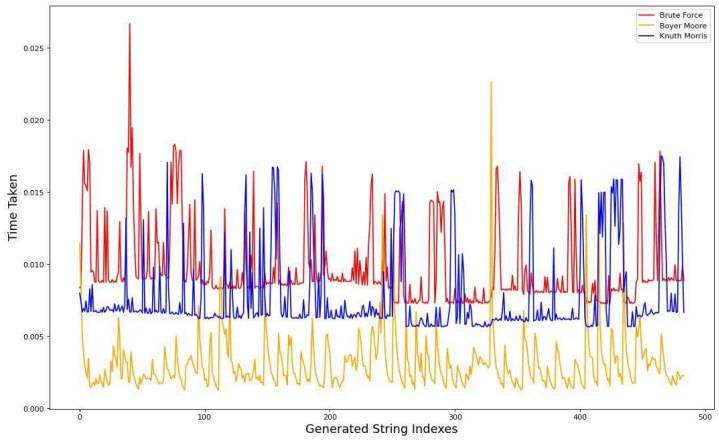
In its best case, KMP will be O(n). This means that there is no mismatch in characters between the pattern and the text. There will only be one traversal of the outer loop (of size n). It will be a linear search without having to traverse the π table.

Average/Worst Case

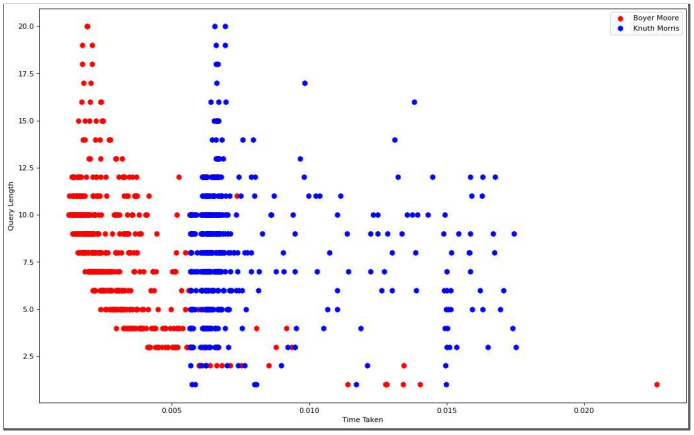
There are 2 loops used in the implementation of KMP, the first being the loop used to generate the π table. This loop follows the length of the pattern (size *m*) we are trying to match, which gives us a time complexity of O(m). Next, we will have to loop through the text (size *n*) we are searching. KMP will only traverse the pi table if there is a mismatch, otherwise it will be similar to linear search. In both average

and worst case, there will always be two loops (one for traversing the pattern, the other for the text), hence, the time complexity is O(m+n)

Results Analysis & Conclusion



This graph shows the performance of the three algorithms as discussed previously. The independent variable "Generated String Index" represents a list of patterns varying in lengths and characters we have generated for testing. From the graph, we can observe that BM generally performs the best, followed by KMP, then Brute Force. Though the difference between the latter two is much smaller. Brute Force generally performs the slowest due to its double loop structure which searches through every single character of the text, which results in it having an average time complexity of O(nm). On the other hand, both BM and KMP algorithms have ways to "escape" unnecessary comparisons (when strings are longer/shorter respectively) using lookup tables as described above.



With deeper analysis into both KMP and BM, we can notice that BM performs better for longer pattern strings, while KMP performs slightly better than BM for shorter pattern strings. The first observation may be explained by the "skip" table for pattern strings, as generated by BM algorithms. Longer pattern strings mean potentially larger skips throughout the genome sequence, hence, shorter time is taken. The second observation can be explained by the smaller "pi" table generated with a shorter pattern string, hence, less time is taken to traverse the smaller table throughout the genome sequence. With these observations, we can come to the conclusion that there is no 'best' algorithm overall, but more on which one best suits our needs. If our query string is short, KMP would be a better choice. Conversely, if the query string is long, BM would best be fitted for the task. Regardless, either one of these algorithms will be a substantial improvement over Brute Force.

References

1) Boyer-Moore good-suffix heuristics. (2013, October 13). Stack Overflow. https://stackoverflow.com/questions/19345263/boyer-moore-good-suffix-heuristicsBoyer-Moore
2) String Matching - Good suffix shift. (2016, April 5). Stack Overflow. https://stackoverflow.com/questions/36426911/boyer-moore-string-matching-good-suffix-shift
3) How does grep run so fast? (2012, September 27). Stack Overflow. https://stackoverflow.com/questions/12629749/how-does-grep-run-so-fast
4) Lang, H. W. (2018, June 4). Boyer-Moore algorithm. W.Inf.Hs-Flensburg.De/. https://www.inf.hs-flensburg.de/lang/algorithmen/pattern/bmen.htm
5) Lengmead, B. (2015, May 19). ADS1: Boyer-Moore basics. YouTube. https://www.youtube.com/watch?v=4Xyhb72LCX4&ab_channel=BenLangmead
6) Size Matters: A Whole Genome is 6.4B Letters. (2017, July 28). VeritasGenetics. https://www.veritasgenetics.com/our-thinking/whole-story/
7) Baeza-Yates, R. A., & Régnier, M. (1992). Average running time of the Boyer-Moore-Horspool algorithm. Theoretical Computer Science, 92(1), 19–31. https://doi.org/10.1016/0304-3975(92)90133-z