1.Introduction

1.1 Objectives

DNA and protein sequence searching is one of the fundamental problems in bioinformatics research. Our objective for the project is to propose algorithms to solve searching problems in genome sequence and to study and compare the efficiency of several algorithms performed to search for exact occurrences of a query sequence in genome sequences.

1.2 Overview

In this report, we will discuss the design and implementation of three different algorithms, namely Brute-force search algorithm, Knuth-Morris-Pratt (KMP) algorithm and a modified Boyer-Moore (BM) algorithm when searching query sequence in the source genome sequence, followed by the time and space complexity analysis of each algorithm. Conclusion will be drawn through comparison of these three algorithms.

2.Algorithms and analysis

For analysis purposes, we let n denote the length of the genome sequence, and m be the length of the query sequence, where n>>m.

2.1 Brute force Algorithm

2.1.1 Design and implementation

The Brute force search algorithm is a naive approach that uses nested loops to search all the substrings of length m in the genome string starting from indexes 0 to n-m. It checks the characters in a window from left to right and stops the attempt at the first mismatch or when the search is successful. After each attempt, it shifts the search window to the right by exactly 1 position.

2.1.2 Complexity analysis

2.1.2.1 Pre-processing time complexity

In this algorithm, there is no pre-processing of strings needed as no constructing of additional arrays or tables to store information is performed. Thus, there is no pre-processing time complexity.

2.1.2.2 Time complexity analysis

To analyze the time complexity of the brute force algorithm, we will be looking at the number of character comparisons.

Best Case: O(n)

Text: AAAAAAAAAAAAA....
Pattern: CCCCC

Every outer-loop character comparison results in a mismatch, as the first character of the query sequence is not present in the genome sequence. Hence, the inner loop is never executed. The total number of comparisons is only determined by the number of outer-loop iterations, which is (n-m+1), with each iteration having only one comparison giving a total of (n-m+1) comparisons. Since n>>m, best case time complexity can be estimated to **be O(n)**.

Worst Case: O(mn)

Text: AAAAAAAAAAAAA....
Pattern: AAAAA OR AAAAC

For either of the above patterns, m comparisons take place in each inner loop. In the first case, all characters must be matched, and in the second case, all characters until the last one match and the final character must still be checked for the mismatch each time. Thus, the total number of comparisons are m(n - m + 1).

In both cases, since n>>m, worst case time complexity can be estimated to be **O(mn)**.

Average case time complexity O(mn): First, we consider the average number of comparisons that the outer loop is executed, since all possible substrings should be checked along the genome string, the outer loop will be always executed for n-m+1 times. Second, the number of comparisons in the inner loop ranges from 1 to m before exiting the inner loop. Assuming each number of comparisons is equally likely to be chosen with probability of 1/m. The expected number of comparisons is:

E(Number of comparisons of inner loop) = $1mi=1mi = 1m(m(m+1)2) = m+12$

Therefore, average number of comparisons = $m+12(n-m+1)$ ≈ **O(mn)** (assuming n>>m)

2.1.2.3 Space complexity analysis

There is no pre-processing of strings needed as no constructing additional arrays or tables to store information is performed. Therefore, the space complexity is constant, which can be represented by O(1).

2.2 Knuth-Morris-Pratt Algorithm

2.2.1 Design and implementation

In KMP, an array containing the lengths of the longest prefix which is also a suffix (LPS) for each prefix substring is created. When a mismatch is found, this is used to skip forward such that the initial few characters of the search window need not be compared again.

2.2.2 Complexity analysis

2.2.2.1 Preprocessing complexity

Preprocessing uses a while loop and two pointers (i and j) which pass through the pattern. j starts at 0 and i starts at 1, and the loop iterates while i<m for (m-1) total iterations. In each iteration, if the characters pointed by i and j match, both are incremented, else, j backtracks through the LPS array until the characters match or j=0. Hence, j can only backtrack as many steps as i is incremented, for a maximum total of (m-1) comparisons. This gives a total of (2m-2) comparisons, so we can say that preprocessing time complexity is O(m).

2.2.2.2 Time complexity analysis

The search function of the KMP algorithm consists of a while loop which iterates through the entire text string. Let i and j be the control variable and index of pattern respectively. Then the loop iterates until i<n, giving a total of n iterations for the while loop.

The loop body consists of an if-else statement, when the input sequence and the pattern match, both i and j are incremented until there is a mismatch. If there is a mismatch, the LPS array is used to set the value of j depending on whether j!=0. In the worst case, the loop will be executed another n-m+1 times, i.e, the value of j is decreased with the LPS array without changing the value of i. Thus, the total number of iterations including preprocessing is 2n-m+2m-2 i.e, 2n+m-2. Assuming that n>>m, the time complexity for the function will be linear to the size of the text string with value O(n).

Worst Case: O(n)

Text: AAAABAAAA
Pattern: AAAAA
LPS array: [0,1,2,3,4]

After the first 4 matches the value of the i and j will be 4. Since there is a mismatch at the 5th position, j will become LPS[j-1] which is 2. The position of j is shifted by two places, pointing at the 3rd character where the matching starts again. There is a mismatch at the 3rd character again so the pointer shifts backward by 1, j becomes LPS[j-1]. The process continues and we see that the maximum number of comparisons for search algorithms will be 2n-m and time complexity of **O(n)**.

The best case occurs when there is no mismatch. So i and j are incremented at each step without moving j backwards. Thus, the maximum possible iterations will be n-m.

So the total number of iterations including pre-processing will be n-m+m, i.e, n. Thus, the time complexity will be a linear function of the length of the pattern with value O(n).

Best case: O(n)
Text: AAAAAAAAAAAAA....
Pattern: AAAAA
LPS array: [0,1,2,3,4]

Since all the characters of the text string match the pattern, both i and j are incremented. j = length of the pattern, i.e, the entire pattern is found in the text string and so the index of the 1st character of the text string is returned where the sequence was found. j then becomes LPS[j-1] which is 2. j shifts two places backward and the comparison starts from the 2nd character and the process repeats once again.

We see that there are exactly (n-m+1) comparisons for the searching algorithm and the time complexity of **O(n)**.

2.2.2.3 Space complexity analysis

The KMP algorithm only requires as much space as the pattern to store the values of the LPS array, so the space complexity is O(m).

2.3 Boyer Moore Algorithm (Modified)

2.3.1 Design and implementation

Boyer Moore's search window moves from left to right, but characters are compared from right to left. When a mismatch is found, two types of shifting of the window can occur - a bad character shift, or a good suffix shift. A bad character shift always ensures that the character in the text that the mismatch occurred at is now matched, whereas a good suffix shift ensures that the previously matched characters continue to remain matched after the shift. The greater of the two shifts is always taken. Our original modification takes inspiration from KMP and keeps track of the indexes of previously matched letters, so they don't have to be matched again. This significantly improves performance, as seen in the examples later.

2.3.2 Complexity analysis

2.3.2.1 Preprocessing complexity

Two tables are created - a bad character table and a good suffix shift table.

The bad character table is an array of length σ, i.e. the number of characters in the alphabet used (in this case, 4). This step iterates over the pattern once, so time complexity is O(m) and space complexity is O(σ).

The good suffix shift table is an array of length m. The steps involved in its creation are very similar to that of the LPS table in KMP, with the best and worst cases both being O(m) for time complexity.

Thus, overall space complexity is O(m+σ) and overall time complexity is O(m).

2.3.2.2 Time complexity analysis

Let us look at a few cases to see how the algorithm performs.

Best Case: O(n/m)
Text: AAAAAAAAAAAAA....
Pattern: CCCCC

In this case, bad character matching shines - as A is not present in the pattern string, each mismatch causes the search window to shift forward by m=5. As the mismatch always occurs on the first character, only one in every m characters gets compared. Hence, the total number of comparisons are floor(n/m), and time complexity in this case is **O(n/m)**.

Original BM Worst case:
Text: AAAAAAAAAAAAA....
Pattern: AAAAA

In the basic Boyer Moore algorithm, this was the worst case - after the initial m comparisons confirming a match, a good suffix shift of 1 would take place and the process would repeat. This is identical to brute force and would be O(nm).

However, our modification significantly improves on Boyer Moore's character re-comparison.
With our modification, the matched letters of a good suffix shift or the original mismatch of a bad character shift are not compared again, meaning in this case, only each additional letter after a shift are compared, leading to exactly n comparisons and time complexity of **O(n)**.

Theoretical Worst Case: With the above modification, the only letter re-comparison that happens is on the mismatched character after a good suffix shift. Theoretically, if it were possible to have a text and pattern combo such that each comparison is a mismatch, with a good suffix shift of exactly 1, and the mismatched character needing to be re-compared despite the shift, then exactly 2n comparisons would take place (each letter being matches successfully once and mismatched once). Although such a combination of text and pattern cannot exist, it proves that the worst case is bounded by 2n, with the actual worst case being hard to compute and somewhere between n and 2n. Hence, **worst case is O(n)**.

Average case: With good suffix shift and bad character shift both involved, the character comparisons will generally be <n. Although this is hard to prove mathematically, it can be clearly seen from the fact that usually (except in specific cases) each letter is compared at most once, and the time complexity is bounded by Ω(n/m) and O(n). Hence we can say the **average case is O(n)**.

2.3.2.3 Space complexity analysis
Search time does not require any additional memory. Hence, as explained under preprocessing complexity, the total space complexity is O(m+σ).

2.4 Empirical Analysis, Comparison and Conclusion

Algorithm	Number of Sequences	Run Time (in seconds)		
		TTAG (4)	CATCGGAA (8)	TAGCAGTGTGCATTGC (16)
Brute Force	10	3.354	2.682	3.354
	20	4.119	3.309	4.089
	40	4.442	3.553	4.475
Boyer - Moore (Modified)	10	1.289	0.835	0.761
	20	1.427	0.934	0.839
	40	1.477	0.995	0.881
Knuth - Morris - Pratt	10	3.000	2.582	3.152
	20	3.807	3.205	3.876
	40	4.007	3.502	4.222

From the above analysis we can infer that KMP and Boyer-Moore algorithms always perform better than brute force. Both KMP and Boyer-Moore use techniques to reduce the number of comparisons required by efficient shifting. Although the above table makes it seem like KMP is only marginally better than brute force, it is likely because the genome being checked is relatively small (a few hundred MBs) and the preprocessing is still a significant constant factor. As the genome sizes increase to human-level, the difference will become more noticeable.
The table also clearly shows that our modified Boyer-Moore algorithm performs the least comparisons and performs several times faster in all cases, with the difference in time only increasing with genome size. Hence we can conclude that our modified Boyer-Moore would be the best of these three, even with such a small alphabet.

3.1 Notes

- Max size our code was tested against ~211MB, but theoretically it should be able to handle ~ 4GB (max length of string in C++). Also note that the code uses dynamic allocation of memory in the heap, so limitations of CPU and RAM will apply.
- Genomes were obtained from: https://www.ncbi.nlm.nih.gov/assembly/GCF_003254395.2
- No external libraries used, only standard C++ libraries.

3.2 References

- Charras, Christian & Lecroq, Thierry. (2004). Handbook of Exact String Matching Algorithms. https://www.researchgate.net/publication/220693416_Handbook_of_Exact_String_Matching_Algorithms
- https://www.inf.hs-flensburg.de/lang/algorithmen/pattern/kmpen.htm
- https://www.inf.hs-flensburg.de/lang/algorithmen/pattern/bmen.htm