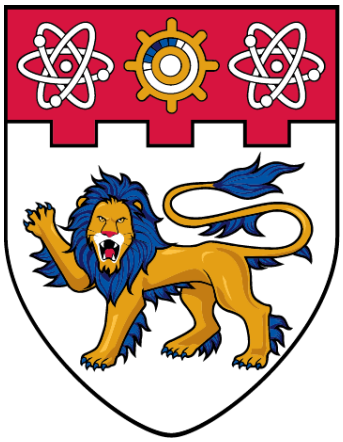

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ2001 Algorithm Lab 1 Report

Group 4

Chang Heen Sunn U1920383H

Luo Wenyu U1922009G

Ooi Wei Chern U1920504J

Saw William Joseph U1921246F

Tan Ching Fhen U1920787D

---

1 Introduction

In bioinformatics research, string-matching methods and algorithms are implemented for DNA and protein sequence searching. Using the target genomic pattern as a substring and the nuclei acid sequence subjected to analysis as the base string, we designed suitable algorithms for the problem while drawing inspirations from existing algorithms and online resources. Searching for a query sequence in a nucleic acid sequence that ranges from a few nucleotides to billions of base pairs is a time-consuming process and because of that, it is important that the algorithm designed must be efficient, precise and quick, coupled with low time complexity.

2 Naive Algorithm

The naïve algorithm, also known as the brute force method, examines each character in the string and compares them to the given pattern one letter at a time. It is an extremely lengthy process that is highly inefficient but simple to code and execute. Figure 1 shows the codes of naive algorithm.

```
def brute_force(self, text, target):
    result = []
    m = len(text)
    n = len(target)
    i, j = 0, 0

    for i in range(0, m-n+1):
        for j in range(0, n):
            if text[i+j] != target[j]:
                break

            if j == n-1:
                result.append(i)

    if result:
        print("Pattern:", target)
        print("Sequence found at position index:- ",result)
    else:
        print("Sequence not found.")
```

Figure 1: Codes of the Naive Algorithm

Time complexity

Assuming the length of text is n and length of the pattern is m, it is trivial to explain that the time complexity of naïve algorithm is O(n) for best case, O(nm) for the worst case. For average case, since there are only four characters (A, C, G, T) appearing in the DNA sequence, we can make a smart assumption that the probability of any character occurring in the genome is independent and equal, that is ¼ for each. We can hence model the probability of getting a mismatch by the Geometric distribution. If we do the calculation, we will find that the expected number of comparisons before a mismatch in each alignment of the pattern to the text occurs to be 4/3, with a constant time complexity of O(1). Since there will be a maximum of n+m-1 number of comparisons, on average the time complexity of the naïve algorithm will be equal to O((4/3)(n-m+1)), which is therefore equivalent to O(n) asymptotically.

3 Knuth-Morris-Pratt Algorithm

Inspired by the Knutt-Morris-Pratt algorithm, we modelled our first algorithm after principles adapted in this string-matching algorithm. The algorithm is implemented in the COVID 19 genome to find the matching sequence of substring.

How it works

In summary, the algorithm operates through two phases. In the first phase, the algorithm constructs a table by considering how many times a certain character has appeared in the previous pattern. The values taken are based on the length of the longest proper prefix (LPS) that is also a suffix. With the values of LPS in the table, the algorithm is able to avoid backtracking and skips precisely the correct number of positions to save time and not miss out on any of the positions with possible matches in the second phase. This unique method of adapting LSP makes KMP the first linear-time algorithm for string-matching. Figure 2 shows how the two phases are done when matching the genome with the given pattern.

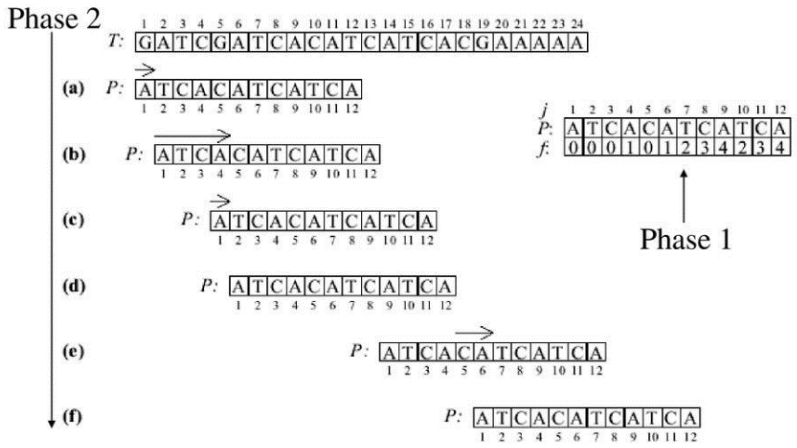

Figure 2: An example of KMP algorithm.

Time complexity

Knuth-Morris-Pratt algorithm involves pre-processing and searching phases. When constructing the LPS, the pointer will move back for a few times, but this will not occur for all the characters in the pattern string. It only occurs after a certain number of matches, so the LPS array can be considered as O(m) time complexity on average, where m is the length of the pattern being searched. During the searching phase, an inner loop and an outer loop are involved. The text pointer that we used never moves backwards, so the inner loop decreases i, which is the pattern string pointer, iteratively to 0 when the outer loop is incremented. This can happen n times at most. The worst case will take 2n loop iterations so the time complexity of searching phase is considered as O(n).

Best case

The best-case scenario occurs when each time the pattern string shifts, a suffix could be found so that maximum skipping could be performed during the searching phase. The time complexity can be considered as O(n).

Worst case

pat	A	B	C	D	E
lsp	0	0	0	0	0
	0	1	2	3	4

Figure 3: Example of LPS table

The worst-case scenario occurs when the LPS cannot be found in the pattern, that is, all the characters in the pattern are distinct. In this case, the LPS table does not store the value of the length of the maximum matching proper prefix which is also a suffix. As such, people will tend to think the operation of KMP algorithm in this case will just act like the brute force algorithm, hence having a time complexity of O(nm), which is in fact, wrong.

Assuming the length of the text is equal to n, there will be only one main loop in the searching process making n number of comparisons. Now we let k be the number of mismatches occurred during the comparison, we will hence have a total number of comparisons be n+k. Due to the design of the KMP algorithm itself, whenever a mismatch happens, we are only able to move index of pattern (pointer j) back by a maximum number of times we incremented it, before we proceed to increment the index of the text (pointer i). Note that the LPS is zero, i.e., there is no prefix which is also a suffix appears in the pattern and hence the algorithm will need to compare the pattern with the next element in the text. Thus, in the worst case, there will be a maximum of n number of mismatched, and hence bringing the total number of comparisons ran by the KMP algorithm to be equal to 2n. The time complexity of the searching process is therefore asymptotically equivalent to O(n), and if we too consider the pre-processing time, it will be O(n+m) in total.

Average case

The average time complexity is O(m+n) which can be considered as a linear time. Moreover, since the pattern we are searching for is usually quite short, we can ignore the time used for building the LPS array. We can consider the total time complexity as O(n) when n is much larger than m (n >> m).

4 Boyer-Moore-Horspool Adaptation

Our second algorithm is inspired by the principles of Boyer-Moore-Horspool (BMH). However, we incorporated a two-way BMH.

How does searching occur?

When the algorithm searches the genome forwards, it checks the target pattern backwards. When it searches the genome backwards, it checks the target pattern forwards.

What is a bad match table(skipper) and how is it made?

‘Skipper’ tells the algorithm how many skips to take, given a character. When the algorithm searches forwards, it uses ‘forward_skipper’. When it searches the genome backwards, it uses ‘backward_skipper’.

‘Forward_skipper’ takes the reversed index of the **last occurrence** of a character in the pattern but ignores the last character. ‘Backwards_skipper’ takes index of the **first occurrence** of a character in the pattern but ignores the first character. Using this guiding rule, given a pattern ‘ATGTC’(Table 1), the resulting bad match tables are shown in Figure 4.

The algorithm decides to go in the direction where the sum of the ‘skipper’ table is lower. If the sum of ‘skipper’ tables is equal, like in Figure 4, it defaults to searching forwards.

Index	0	1	2	3	4
Reversed Index	4	3	2	1	0
Character	A	T	G	T	C

Table 1: Pattern ‘ATGTC’

```
forward_skipper:  {'A': 4, 'T': 1, 'G': 2}
backward_skipper:  {'C': 4, 'T': 1, 'G': 2}
```

Figure 4: Bad match tables for pattern ‘ATGTC’

How does skipping occur?

When searching the genome forwards the corresponding genome character of the last character in the pattern is looked up in ‘forward_skipper’ (Figure 4). When searching the genome backwards, the corresponding genome character of the first character in the pattern (highlighted in Figure 5) is looked up in ‘backward_skipper’(Figure 4). Then, the algorithm skips by that value. If a character is not found in the table, the algorithm skips by the **length of pattern**. For example, in Figure 5, ‘T’ is found in ‘backward_skipper’, so 1 skip occurs. Next, ‘A’ is not found in ‘backward_skipper’, so algorithm skips by length of pattern, 5.

This principle applies whether a mismatch occurs or a pattern is found.

```
       0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
Genome: A  C  G  T  T  G  A  A  C  T  G  G  C  A  T  G  T  C  T
Pattern:                                       A  T  G  T  C

       0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
Genome: A  C  G  T  T  G  A  A  C  T  G  G  C  A  T  G  T  C  T
Pattern:                                    A  T  G  T  C

       0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
Genome: A  C  G  T  T  G  A  A  C  T  G  G  C  A  T  G  T  C  T
Pattern:                       A  T  G  T  C
```

Figure 5: Illustration of a backward skipping

Significance and originality

The motivation to incorporate this novel idea of a two-way BMH is to avoid the worst case of making m(n-m+1) comparisons. By choosing the ‘skipper’ table that is lower in sum, the algorithm avoids the original worst case where it always takes only 1 skip and compares every character in the pattern each time. Because in this scenario, it will have switched over to searching in the other direction.

Time complexity

Best case

The best-case scenario occurs when the mismatch happens at the first letters of comparison and that letter in the virus genome is not found in the dictionary. With each cycle of comparison, the target genome continuously skips along the virus genome by the entirety of its length. Therefore, the time complexity will be $\lfloor \frac{n}{m} \rfloor$ = O(n).

Worst case

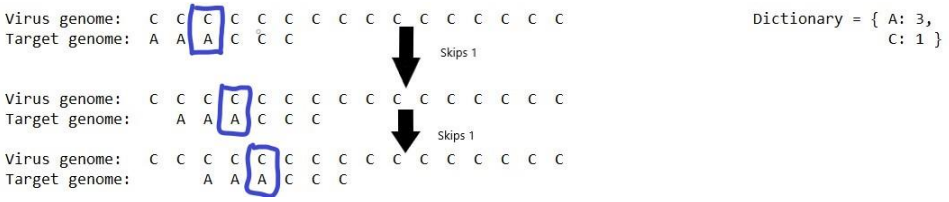

Figure 8: Example of Worst Case of BMH

The worst-case scenario occurs when the $(\frac{m}{2}-1)^{th}$ letter of the target genome from the right mismatches with the corresponding letter of the virus genome. Hence, the former will continuously traverse along the latter by 1 unit. Therefore, the time complexity for the worst-case scenario is *(n-m+1)*$(\frac{m}{2}+1)$ = O(*nm)* because a total of (*n-m*+1) skips will occur with each skip carrying out $(\frac{m}{2}+1)$ number of comparisons.

Average case

As there are *m* number of possible skips with each skip carrying out $(\frac{m}{2}+1)$ possible number of comparisons, there are $(\left(\frac{m}{2}+1\right)m)$ number of cases with each case occurring with a probability of $\frac{1}{(\frac{m}{2}+1)m}$. Therefore, the average time complexity is modelled after this equation,

$$\text{Average time complexity} = \frac{1}{(\frac{m}{2}+1)m}\left(\sum_{j=1}^{1+\frac{m}{2}}\sum_{i=1}^{m}\frac{\lceil n-m+1\rceil}{i}j\right),$$

where i and j are the number of skips and comparisons respectively.

By expanding this equation, we get

$$\frac{1}{(\frac{m}{2}+1)m}\left(\sum_{j=1}^{1+\frac{m}{2}}\sum_{i=1}^{m}\frac{\lceil n-m+1\rceil}{i}j\right)=\frac{\lceil n-m+1\rceil}{(\frac{m}{2}+1)m}\left(\sum_{j=1}^{1+\frac{m}{2}}j\right)\left(\sum_{i=1}^{m}\frac{1}{i}\right)=\frac{\lceil m^2\left(\frac{n}{4}-\frac{5}{4}\right)+n\left(\frac{3}{2}m+2\right)-\frac{m^3}{4}-\frac{1}{2}m+2\rceil}{2m(\frac{m}{2}+1)}(\ln m)$$

Therefore, if n is extremely large, the dominant term would be *nm* and this will result in,

$$\frac{1}{(\frac{m}{2}+1)m}\left(\sum_{j=1}^{1+\frac{m}{2}}\sum_{i=1}^{m}\frac{\lceil n-m+1\rceil}{i}j\right) = O(\frac{\lceil nm\rceil}{8(1+\frac{m}{2})})= O(\lceil nm\rceil).$$

5 Conclusion

In Table 2 below, we have summarised the pre-processing time and matching time complexity of all three algorithms implemented in this project.

Algorithm	Pre-processing Time	Matching Time
Naïve	0	O(nm)
Knuth-Morris-Pratt	O(m)	O(n)
Boyer-Moore-Horspool	O(m)	O(nm)

Table 2: Summary of Algorithm Processing Time

Based on the table above, Knuth-Morris-Pratt (KMP) and Boyer-Moore-Horspool (BMH) outperforms the Naïve algorithm and are able to achieve lower time complexity. Specifically, for this project, we found that the algorithm inspired by Boyer-Moore-Horspool is the most suitable at matching genomic pattern with the given genome.

6 Reference

[1] Stack Exchange Contributors (September 2015). How to simplify the sum over 1/i. Retrieved from https://cs.stackexchange.com/questions/30034/how-to-simplify-the-sum-over-1-i

[2] F. Saldaña. (September 2020). The Boyer-Moore-Horspool Algorithm. Retrieved from https://nearsoft.com/blog/the-boyer-moore-horspool-algorithm/#:~:text=substring%20cannot%20match.-,Complexity,to%20Boyer%2DMoore%20original%20implementation

[3] QianhangSun (Oct 2019). The String Match Algorithm. https://github.com/QianhangSun/StringMatch/blob/master/Ramdom_Project/KMP_string_matching.py

Statement of Contribution

In this project, the workload was divided among the five of us following the breakdown below:

Chang Heen Sunn – Investigated concepts of Knuth-Morris-Pratt and written the code for the KMP algorithm.

Luo Wenyu – Explored the branching ideas of Knuth-Morris-Pratt and analyzed the time complexity of KMP.

Ooi Wei Chern - Studied Knuth-Morris-Pratt, compiled the complete report and slides and written the comparison and conclusion of the project.

Saw William Joseph – Focused on the time complexity of Boyer-Moore-Horspool and calculated it.

Tan Ching Fhen – Written original codes using ideas from Boyer-Moore-Horspool and implemented a simpler interpretation on the codes.