****

**CZ2001 Algorithms**

**Lab Project 1**

**Searching Algorithms**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Index Group: SS4

Group: 2

Members: Selvira Junita Melia (U1921926L)

Glenda Hong Zixuan (U1921784A)

Chia Jia Tian (U1921632K)

Ernest Ang Cheng Han (U1921310H)

Tan Wee Li (U1922935K)

**Table of Contents:**

[Introduction](#_p8h9g8gqueb8) 2

[Base Case Algorithm: Brute-Force Sequential Search](#_9poayqlckfmq) 2

[Algorithm 1:](#_h8kwr6d91alu) 3

[Algorithm 2:](#_h663rpogbtte) 4

# 

# Introduction

In this project, we analysed and contrasted various algorithms to understand their space and time complexities. Using brute-force searching as a baseline, we proposed 2 searching algorithms with higher efficiency.

**For the analysis of all algorithms in this report, n will refer to the length of the DNA sequence while m will refer to the length of the query sequence.**

# Base Case Algorithm: Brute-Force Sequential Search

The brute force algorithm searches index 0 to n-m in the DNA sequence and checks whether the DNA sequence starting at that particular index matches with the query sequence. After each attempt, it increments the index by 1 and does the checking again starting at the new index. If the number of characters that match between the DNA and query sequence at index i is equal to the length of the query sequence, a match has been found and position (i + 1) is returned.

## Pseudocode:

Program bf\_search(query, dna)

**DECLARE** i, j **AS INTEGER**

**FOR** i = 0, 1, 2, 3, 4, … (n - m) **DO**

**SET** j **AS** 0

**WHILE** j is lesser than n **AND** dna[i + j] is equal to query[j] **DO**

**INCREMENT** j

**END WHILE**

**IF** j is equal to m **THEN**

**RETURN** i+1

**END IF**

**END FOR**

## Time Complexity:

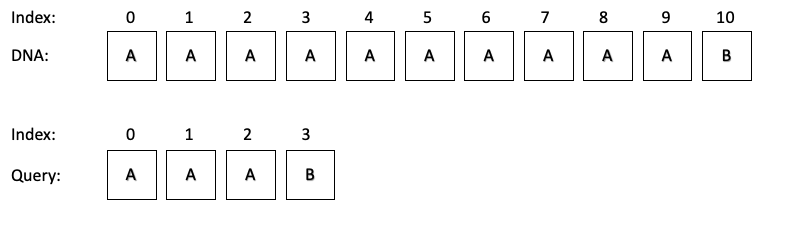
|  |  |  |
| --- | --- | --- |
| Best Case: **O(n)** | Worst Case: **O(nm)** | Average Case: **O(n+m)** |
| DNA = “CTG**AGT**CT”  Query = “**AGT**”  The query sequence only appears once in the DNA sequence and the first character of the query also only appears once. The for loop will be iterated (n - m + 1) times and the while loop only will be iterated m times. Hence, the total number of key comparisons made will be (n - m + 1) + m. Therefore, the best case time complexity is O(n). | DNA = “AAAAAAA**AAT**”  Query = “**AAT**”  The query sequence excluding the last character appears repeatedly in the DNA. The for loop still iterates (n - m + 1) times but the while loop will now iterate m times in each iteration of the for loop. Hence, the total number of key comparisons made will be (n - m + 1)(m) = nm - m2 + 1 times. Therefore, the worst case time complexity is O(nm). | The algorithm at least has to check the first (n - m + 1) characters of the DNA sequence and on top of that there would be both partial and complete matches which means there will be A\*m, where A is some constant, number of additional comparisons. Hence, the total number of key comparisons is (n - m + 1) + A\*m where A > 1. Hence the average case time complexity is O(n + m). |

## Space Complexity:

The algorithm only uses 2 integer variables which means it uses constant space, hence, space complexity is O(1).

## 

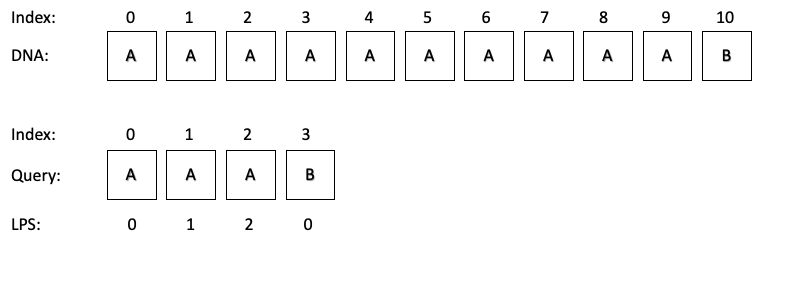
## Limitation:

****

This algorithm does not analyse the pattern of the querying and/or DNA sequence. For instance, after a partial match index 0 of the DNA sequence, the algorithm starts the next round of comparisons at index 1 of DNA and index 0 of the query but could have instead started at index 3 of the DNA and index 2 of the query. This search algorithm uses raw computing power to search for a solution. It is simple to implement albeit inefficient. It is very useful for searching solutions through a small data sample. However, as DNA sequences usually contain extensive amounts of data, brute force algorithms may not be ideal in this case, due to the high number of iterations. Thus, we implemented 2 other algorithms which seek to tackle the limitations faced by brute force algorithms.

# 

# Algorithm 1:



This algorithm is based on the Knuth-Morris-Pratt (KMP) String Search Algorithm which analyses the query for similar patterns by creating and using a LPS (Longest proper Prefix which is also a Suffix) table. Referencing the image above, after unsuccessfully searching the DNA sequence for the query at index 0, instead of having to start at index 1 of the DNA and index 0 of the query, the algorithm uses the LPS table and recognizes that the DNA and query sequence have the same pattern for its first 3 indexes, allowing it to start searching the DNA sequence at index 3 and comparing it to index 2 of the query sequence.

## Pseudocode:

|  |  |
| --- | --- |
| Program create\_lps\_table(query, lps[ ])  **DECLARE** FP, SP **AS INTEGER**  **SET** FP **AS** 0  **SET** SP **AS** 1  **WHILE** second\_pointer is less than length of query **DO**  **IF** query[FP] is equal to query[SP] **THEN**  **INCREMENT** FP  **SET** index [SP] of lps **AS** FP  **INCREMENT** SP  **ELSE**  **IF** FP is not 0 **THEN**  **SET** FP **AS** index [FP - 1] of lps  **ELSE**  **SET** index [SP] of lps **AS** 0  **INCREMENT** SP  **END IF**  **END IF**  **END WHILE** | Program kmp\_search(query, dna)  **DECLARE** lps[ ] **AS INTEGER ARRAY**  **DECLARE** dna\_index, query\_index **AS INTEGER**  **SET** lps size **AS** m  **SET** dna\_index **AS** 0  **SET** query\_index **AS** 0  create\_lps\_table(query, lps)  **WHILE** query\_index is less than m **DO**  **IF** query[first\_pointer] is equal to query[second\_pointer] **THEN**  **INCREMENT** dna\_index **AND** query\_index  **ELSE**  **IF** query\_index is not 0 **THEN**  **SET** query\_index **AS** index [query\_index - 1] of lps  **ELSE**  **INCREMENT** dna\_index  **END IF**  **END IF**  **END WHILE** |

## Time Complexity:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Best Case: **O(m + n)** | Worst Case: **O(m + n)** | Average Case: **O(m + n)** |
| create\_lps\_table() | Query = “**AGT**”  Every character in the query sequence is different.The function will reiterate through the while loop for the length of the query sequence (m). Therefore, the best case time complexity for create\_lps\_table() is O(m). | Query = “**AAT**”  Every character, except for the last character, in the query sequence is the same. For the first (m -1) characters, the function will reiterate through the while loop (m - 1) times. For the last character, the function will iterate through the loop m times.  Time Complexity  = O(m - 1) + O(m)  = O(m)  Therefore, the worst case time complexity for create\_lps\_table() is O(m). | Since the best and worst case time complexities are both O(m), it can be assumed that the average case time complexity must also be O(m) |
| kmp\_search() | Query = “**AGT**”  DNA = “**AGT**CTCTG”  The query sequence sequence appears in the DNA sequence only once at the start of the DNA sequence. The function will iterate through its while loop for the length of the query sequence (m). The function then checks through the remaining pattern by iterating the while loop (n - m) times.  Thus, the function does a total of n loops.Therefore, the best case time complexity for kmp\_search() is O(n). | Query = “**ACT**”  DNA = “GGGGGGGGG**ACT**”  If there is only one match at the end of the DNA sequence, then the KMP algorithm becomes no different than the Brute Force String Search, where the algorithm has to search the DNA sequence index to index with every position of the query sequence. Over each iteration, there can be at most 1 false comparison, hence there will be n false comparisons over n DNA sequence. Thus having 2n comparisons which takes O(n). Therefore, the worst case time complexity for kmp\_search() is O(n). | Since both the time complexity in the best and worst case are O(n), it can be assumed that the average case time complexity must also be O(n). |
| Total Time Complexity | Total Time Complexity = O(m) + O(n) = O(m + n)  Therefore, the time complexity for all 3 cases for the KMP algorithm is O(m + n) | | |

## Space Complexity:

The amount of space required would hence be m + 4 since the LPS table will be the size of query and our code will also use 4 variables to temporarily store values and help with the construction of the LPS table, resulting in space complexity O(m).

# 

# Algorithm 2:

We based our second algorithm on the Rabin-Karp algorithm but we made modifications to better optimise the code to our problem. For DNA and RNA sequences, there are only 5 different letters used and each will be mapped to a unique single digit number. Hence, we are able to create a specific hash function that will produce hash values that retain information about the position of each letter in the sequence.

E.g. A → 0, C → 1, G → 2, T → 3, U → 4

Hash value of “CAGT” = 1 x 53 + 0 x 52 + 2 x 51 + 3 x 50 = 10235 = 3810

↑ ↑ ↑ ↑

1 0 2 3

Therefore, there would never be a case of false positives as each unique sequence will produce its own unique hash value, hence there is no need to individually double check each letter in the query with the DNA using an additional while loop. Using a rolling hash function, the new hash values of the DNA sequence can be updated while iterating through the sequence. If the hash values are equal, no additional comparisons need to be made and the position is returned. As comparing hash values will be quicker, we are able to improve overall time complexity.

## 

## Pseudocode:

|  |  |
| --- | --- |
| Program rk\_search(query, dna)  **DECLARE** pattern\_list, text\_list **AS LISTS**  **DECLARE** i **AS INTEGER**  **CONVERT** query, dna **TO INTEGER**  **APPEND** query **TO** pattern\_list  **APPEND** dna **TO** text\_list  **DECLARE** pattern\_hashes, text\_hashes **AS INTEGER**  **FOR** i = 0, 1, 2, 3, 4, … (n - m + 1) **DO**  **IF** i is not equal to 0 **THEN**  **UPDATE** hash\_values  **IF** text\_hashes is equal to pattern\_hashes **THEN**  **RETURN** i  **END FOR** | |
| Program create\_hash(string, base , end)  **DECLARE** hash\_value as **INT**  **FOR** i = 0, 1, 2, 3, 4, … end **DO**  **INCREMENT** hash\_value **BY** string[i] \* base^( end - i - 1)  **RETURN** hash\_value  **END FOR** | Program recalculate\_hash(string, old\_index, new\_index, old\_hash, m, base)  **SET** new\_hash **AS** old\_hash - (string[old\_index] \* base^(m - 1))  **SET** new\_hash **AS** new\_hash \* base  **INCREMENT** new\_hash **BY** string[new\_index] |

## Time Complexity:

|  |  |  |
| --- | --- | --- |
| Best / Worst / Average Case: **O(n + m)** | | |
| The preprocessing time includes the time taken to calculate the hash values of the query and the DNA sequence, to be used for comparison. This requires a for loop to run the length of the query, m, to convert the letters to numbers which takes O(m). It also includes converting the entire DNA sequence to numbers, matching each letter to a unique integer which takes O(n). Hence, the preprocessing time complexity is O(n + m).  In every case, regardless if there are any matches, the rolling hash function goes through the entire length of the sequence, n. Even when the hash values match, no further checking is required as they represent a unique sequence of letters. Hence, time complexity in every case is O(n).  Hence, the total time complexity is O(n + m) + O(n) = O(2n + m) = O(n + m) for all 3 cases. | | |
|
|

## Space Complexity:

The algorithm uses constant space to store counter variables, and lists containing converted formats of the sequence and query. The hash value of the current text is updated throughout the length of the entire sequence to be compared to the query’s hash value. Due to the constant space use, space complexity is O(1).

# 

# Program limitation:

Our program could handle large .fna files (~1GB), though it took on average approximately 7 minutes to run.

It was still able to run for files that were smaller (~10MB) with relative ease.

# 

Additionally, the query input size limit is 23 characters for Algorithm 2, due to the limit size of the hash values.

# 

# 

# References

1. Geeksforgeeks (2020). *Naive Algorithm For Pattern Searching*. Retrieved 10 September 2020, from <https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/>
2. Brilliant.org (2020). *Rabin-Karp Algorithm | Brilliant Math & Science Wiki*. Retrieved 10 September 2020, from <https://brilliant.org/wiki/rabin-karp-algorithm/>
3. Bari,A. [Abdul Bari]. (2018, May 30). 9.2 Rabin-Karp String Matching Algorithm [Video]. YouTube. Retrieved 10 September 2020, from <https://www.youtube.com/watch?v=qQ8vS2btsxI>
4. Geeksforgeeks (2020). *KMP Algorithm for Pattern Searching.* Retrieved 10 September 2020, from <https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>

# 

# Statement of Contribution

|  |  |
| --- | --- |
| Name | Contribution |
| Ernest Ang Cheng Han | * Created the main code for Knuth-Morris-Pratt algorithm * Contributed to the lab report * Contributed to the slides |
| Glenda Hong Zixuan | * Created the main code for interface * Contributed to the lab report * Contributed to the slides |
| Selvira Junita Melia | * Created the main code for Rabin-Karp algorithm * Contributed to the lab report * Contributed to the slides * Presenter |
| Chia Jia Tian | * Created the main code for Brute-Force algorithm * Modified the code for Rabin-Karp algorithm * Contributed to the lab report * Contributed to the slides |
| Tan Wee Li | * Contributed to the lab report * Contributed to the slides * Presenter |