NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CZ2001 Algorithms
Project 1: Searching Algorithm

Lab Group: SS4

Submitted By:

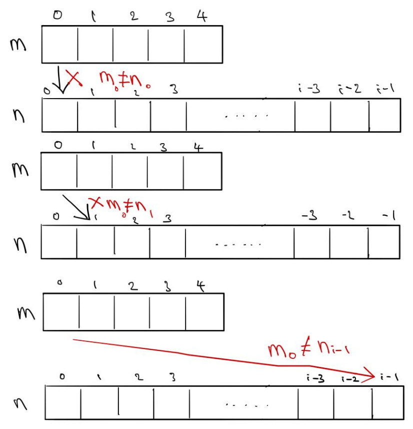
Aaron Tay Han Yan (U1921247B),
Yeong Wei Xian (U1921382J),
Samantha Tan Swee Yun (U1921074B),
Goh Ting Qi (U1920306H),
Song Wei Tyan (U1921107H)

Algorithm 1: Brute Force Sequential Search

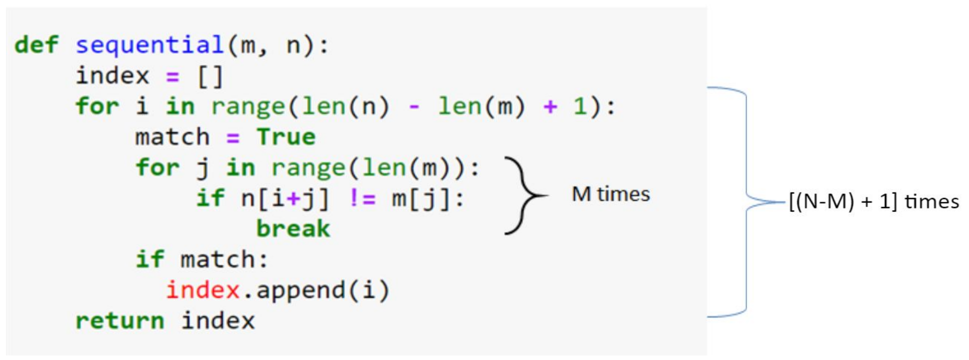
Let m and n be the length of the pattern and the length of the genome sequence respectively.

Best-Case Scenario

For best case scenario, when the first element of m is not present in the search text at all. The time complexity would be O(n).



Worst-Case Scenario



```
def sequential(m, n):
    index = []
    for i in range(len(n) - len(m) + 1):
        match = True
        for j in range(len(m)):
            if n[i+j] != m[j]:
                break
        if match:
          index.append(i)
    return index
```

The worst-case scenario occurs when all characters of the text and pattern are the same (e.g. pattern is "AAA", genome sequence is "AAAAAAAAA") or when the first m-1 elements of the pattern is in the genome sequence but the last element of the pattern is not in the genome sequence (e.g. pattern is "AAAAAC", genome sequence is "AAAAAAAAAAAAAAAAAAAAT").

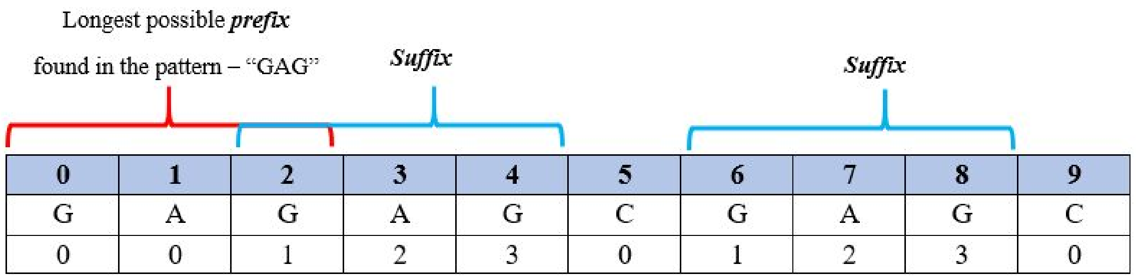
Hence, the outer loop executes [(n-m)+1] times while the inner loop executes m times. Hence, the time complexity is O([(n-m)+1]m) which can also be written as O(nm).

Algorithm 2: Knuth-Morris-Pratt Search

The Knuth-Morris-Pratt (KMP) is an efficient linear time searching algorithm that examines and pre-processes the pattern before the search begins. The goal of examining the pattern is to ensure that every time a successful (or failed) match happens, the pattern itself can provide the information to where the new search should begin. This gives advantage over a naïve algorithm as the search does not necessarily begin from the next index.

Pre-Processing of a Pattern

The KMP algorithm consists of a segment of code to pre-process the query pattern entered by the user. This segment of code aims to find the longest possible ***prefix*** and ***suffixes*** in the pattern, where ***prefix*** denotes the characters from the start of the pattern and its occurrence in the rest of the pattern which is known as ***suffix***. The result of this segment of code generates a table (array) which provides the value that will help to indicate where the next search should begin from when a match or mismatch happens.
The example below depicts the generated array after studying the pattern "GAGAGCGAGC":


Longest possible *prefix* found in the pattern – "GAG"
Suffix
Suffix

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
G	A	G	A	G	C	G	A	G	C
0	0	1	2	3	0	1	2	3	0

Table 1 – Generated Array

Search Function of KMP Algorithm

The KMP search algorithm will then exploit the information generated from studying the pattern to carry out its search function. The array of values generated from the pre-processing prevents the unnecessary continuation of the search at the very next index. At the point of a mismatch, the value at the Pattern index indicates where the next search should begin by adding the value to the Genome Index where the previous match was made. This is possible and legitimate as there is no need to backtrack past the parts of the pattern that is known to be matched. Hence, time complexity and efficiency of the search is increased significantly as will be mentioned in the time complexity analysis below.

The following example depicts the KMP Search Function that exploits the values generated from the pre-processing table:

1. Search begins, and a match is found at Genome Index[0]. Hence, an iteration of the Pattern begins to see if the rest of the pattern matches. However, a mismatch happens at Index[3] of the Genome Sequence and the Pattern as denoted by the boxes highlighted in yellow *in **figure 1***.

Genome Sequence

0	1	2	3	4	5	6	7	8	9
G	A	G	T	G	A	C	G	G	A

Pattern

0	1	2	3	4
G	A	G	A	T
0	0	1	2	0
✓	✓	✓	X	■

Figure 1: First Match

2. Since a mismatch happens at Pattern Index[3]. The next search will begin at Genome Sequence[***i***+3] where ***i*** denotes the index of the previous match which was [0]. Also, the value of Pattern Index[j-1] will be the next starting match, which in this case Pattern Index[3-1] = 1. Hence, the next matching pattern will begin at Genome Index[0+3] and Pattern Index [1] as depicted in ***figure 2***. The Genome Index [1] and [2] is skipped as the pre-processing provides us with the information that the next suffix matched is 'G' located at Genome Index [2].

Genome Sequence

0	1	2	3	4	5	6	7	8	9
G	A	G	T	G	A	C	G	G	A

Pattern

0	1	2	3	4
G	A	G	A	T
0	0	1	2	0
✓	X	■	■	■

Figure 2: Next Match

Time Complexity of KMP

1. Pre-Processing: Suppose m is the length of the pattern. There is a requirement to iterate through the pattern of m length to find all and longest ***prefix suffix***. Hence, the time complexity taken to generate the ***prefix*** and ***suffix*** table is computed as O(m).

2. Search: To search through a Genome Sequence (n), there will be no backtracking in the iteration of Genome Sequence (n) as the prefix suffix table provides us with the information about any matched

parts, resulting in a need of only one iteration through the Genome Sequence. Instead, backtracking may only occur in the iteration of pattern (m). Hence, the search function is computed to be bounded by O(n).

3. Overall: The time complexity of KMP Search algorithm is computed to be O(m+n).

Algorithm 3: Rabin-Karp Search

Let m and n be the search pattern and the genome sequence respectively.

Best-Case and Average-Case Scenario

O(n) step is taken for the rolling hash function since each of the indexes of the genome will stop once, calculate the hash value and then compare with the hash value of the pattern. Then, if it is a potential match, it will verify each letter to check if the match is true and not just a hashing collision. Hence, the total time complexity will be O(n+m).

Worst-Case Scenario

The worst-case scenario occurs if the hash function is made badly which results in false positives at every step. Thus, the m letters in the pattern will be checked n times. Hence, the running time will be O(mn). However, as our hash function is using base 5, false positives will rarely occur. Thus, the worst time complexity is largely avoided.

Space Complexity Issue

Since we added the power of 5 in the hash rolling function, there is a space complexity restraint. The Rabin-Karp search will fail if the pattern string is bigger than 24 letters. Adding a modulus to the rolling hash function might help to resolve this issue but since it will lower the accuracy of the search and hence the time complexity, we have chosen not to implement that.

References

[1]Dept of Research and Development University - Internal Journal of Scientific Engineering and Technology Research. (2014, Nov). Comparison of Three Pattern Matching Algorithms using DNA Sequences. Ijsetr. http://ijsetr.com/uploads/612543IJSETR2868-162.pdf
[2]Tushar Roy - Coding Made Simple. (2015, June 13). Knuth–Morris–Pratt(KMP) Pattern Matching(Substring search). YouTube. https://www.youtube.com/watch?v=GTJr8OvyEVQ
[3]RK: Abdul Bari. (2018, March 30). 9.2 Rabin-Karp String Matching Algorithm. YouTube. https://www.youtube.com/watch?v=qQ8vS2btsxI&ab_channel=AbdulBari

Group Participation

Ting Qi: Helped in formulating the Rabin-Karp and Knuth–Morris–Pratt search algorithm and the analysis of the Rubin Karp algorithms
Aaron: Helped in the analysis, Rabin-Karp algorithm and Python coding interface
Wei Xian: Helped in the analysis of the algorithms and Rabin-Karp search algorithm
Samantha: Helped in Brute-force sequential search algorithm and the analysis
Wei Tyan: Helped in implementation of User Interface, Sequential and Knuth-Morris-Pratt Search algorithm and the analysis of KMP Search Algorithm