CZ2001 LAB PROJECT 1: Searching Algorithms

Done By (SSP2): Lim Bing Hong, Jasper (U1922783B), Chan Zhao Hui (U1921662D),
Leow Wei Thou, Samuel (U1922978H), Clarence Hong (U1922950G), Wong Chin Hao (U1921712L)

Overview

In this project, we will be studying the implementation of several string searching algorithms to search for exact occurrences of a query sequence in a nucleic acid sequence. We will be using Brute Force, Knuth Morris Pratt and Boyer Moore Horspool searching algorithms. We will determine which is the better algorithm through the analysis of their respective time complexity in our implementation.

Let length of text to search pattern in = ***n***
Let length of pattern to search = ***m***
Where **n > m**

Brute Force (Naive Approach)

A general problem-solving technique that systematically enumerates all possible candidates for the solution and checking whether each candidate satisfies the problem's statement. It is simple to implement and will always find a solution if it exists, however it's cost is proportional to the number of candidate solutions, thus making it slow.

Time Complexity

Best case: No matches found

```
for(int i = 0; i < (fileString.length() - pattern.length() + 1); i++)        ---- (n-m) + 1
{
    if(fileString.charAt(i) == pattern.charAt(0))                            ---- (n-m)C1
    {
        for(int j = 1; j < pattern.length(); j++)
        {
            if(!(fileString.charAt(i + j) == pattern.charAt(j)))
                statement
(n-m)
            else if(j == pattern.length()-1)
                statement
        }
    }
}
```

Does not run since no matches

Time complexity: $f(n) = (n-m+1) + (n-m)C1$
$= O(n)$ (n > m)

Worst case : All characters in pattern and text are the same

```
for(int i = 0; i < (fileString.length() - pattern.length() + 1); i++)        ---- (n-m) + 1
{
    if(fileString.charAt(i) == pattern.charAt(0))                            ---- (n-m)C1
    {
        for(int j = 1; j < pattern.length(); j++)                            ---- (n-m)m
        {
            if(!(fileString.charAt(i + j) == pattern.charAt(j)))
                statement
(n-m)
 (m-1)      else if(j == pattern.length()-1)                                 ---- (n-m)(m-1)C2
                statement
        }
    }
}
```

Time complexity: $f(n) = (n-m+1) + (n-m)(C1 + m + (m-1)C2)$
$= (n-m+1) + (n-m)C1 + (nm - m^2) + (nm - n - m^2 - m)C2$
$= O(nm)$

Average case: Equal probability of any subset of m matching

```
for(int i = 0; i < (fileString.length() - pattern.length() + 1); i++)        ——— (n-m) + 1
{
    if(fileString.charAt(i) == pattern.charAt(0))                            ——— (n-m)C1
    {
        for(int j = 1; j < pattern.length(); j++)                            ——— (n-m)m
        {
            if(!(fileString.charAt(i + j) == pattern.charAt(j)))
                statement
            else if(j == pattern.length()-1)                                 ——— (n-m)(i)C2
                statement
        }
    }
}
```

(n-m) — (i)

Time complexity: $f(n) =$

$$\frac{1}{m}[(n-m+1)+(n-m)(C1+m+((0)C2)\ +(n-m+1)+(n-m)(C1+m+((1)C2)\ +\ (n-m+1)+(n-m)(C1+m+((m-1)C2)]$$

$$= \frac{1}{m}\sum_{i=0}^{m-1}[(n-m+1)+(n-m)(C1+m+((i)C2)] = \frac{1}{m}\sum_{i=1}^{m}[(n-m+1)+(n-m)(C1+m+((i)C2)]$$

$$= \frac{1}{m}[\sum_{i=1}^{m}(n-m+1)\ +\sum_{i=1}^{m}(nC1+nm-mC1-m^2+i(nC2-mC2)]$$

$$= \frac{1}{m}[m(n-m+1)\ +m(nC1+nm-mC1-m^2)+\sum_{i=1}^{m}i(nC2-mC2)]$$

$$= \frac{1}{m}[m(n-m+1)\ +m(nC1+nm-mC1-m^2)+(nC2-mC2)\sum_{i=1}^{m}i]$$

$$= \frac{1}{m}[m(n-m+1)\ +m(nC1+nm-mC1-m^2)+\frac{m(m+1)}{2}(nC2-mC2)]$$

$$= (n-m+1)\ +(nC1+nm-mC1-m^2)+\frac{m+1}{2}(nC2-mC2)]$$

$$= 2n-2m+2\ +(2nC1+2nm-2mC1-2m^2)+(m+1)(nC2-mC2)]$$

$$= 2n-2m+2\ +(2nC1+2nm-2mC1-2m^2)+\ mnC2-m^2C2-nC2-mC2$$

$$= (2+2C1-C2)n-(2+2C1+C2)m+(2+C2)nm-(2+C2)m^2+2$$

$$= (C)n-(C)m+(C)nm-(C)m^2+C$$

$$= O(nm)$$

Brute-Force (Improvement) - New Algorithm the team created

Pre-processing

```
while(index<fileString.length()) // create index-array table
{
    int startindex = 0;
    startindex  =  fileString.indexOf(startCodon, index); //
    if(startindex != -1) {
        startCodonArray.add(startindex);
        index = startindex + 1; // jump to next index
    }
    else // when startindex == -1
    {
        break;
    }
}
```

Processing

```
for(int i=0; i<startCodonArray.size(); i++) // Compare all the start codon
{
    int fileIndex = 0; // start codon start index
    int fileIndex2 = 0; // end codon start index
    fileIndex = startCodonArray.get(i); // set fileindex to startindex of startcodon
    fileIndex2 = fileIndex + jump;  // set fileindex2 to startindex of endcodon
    int fileIndex3 = startCodonArray.get(i) + jump;
    for(int j=(0+jump); j<(0+jump+3); j++)
    {

    }
}
```

```
if(fileString.charAt(fileIndex3) != pattern.charAt(j))
{
    break;
}
else if(j == patternLength -1) // due to index
{
    String comparisonString = fileString.substring(fileIndex+3, fileIndex2);
    int subpatternstart = 3; //patternstart
    int textpatternstart = fileIndex+3;
    int subpatternend = jump-1;

    if(comparisonString.length() == 0)
    {
        position += fileIndex + 1;
        position += " ";
        total ++;
    }
    for(int k = 0; k<comparisonString.length(); k++)
    {
        if(fileString.charAt(textpatternstart) != pattern.charAt(subpatternstart))
        {
            break;
        }
        else if(fileString.charAt(k) == pattern.charAt(subpatternstart))
        {
            subpatternstart++;
            textpatternstart++;
        }
        else if(k == comparisonString.length()-1)
        {
            position += fileIndex + 1;
            position += " ";
            total ++;
        }
    }
}
else if(fileString.charAt(fileIndex3) == pattern.charAt(j))
{
    fileIndex3++;
    continue;
}
```

Based on the understanding of how the genome works, we understand that there will always be a start and stop codon (in terms of size 3) and each pair of nucleic acid / proteins comes in a pair of size 3. Thus, a typical DNA / RNA will always be in terms of size 3.

Thus, we develop an algorithm that first finds all the occurrences of start codon in the text. Then based on the start codon's occurrences, we compare the stop codon and if matches then we will try to match the substring = pattern - start - stop codon. If everything mathes, we will find the occurrence of the pattern.If any of the above steps fail, the algorithm will jump to the next occurrence of the start codon and skip and text in between. In general, the improvement made to brute force is instead of matching all the text, we only apply brute force to the relevant areas.

Best case (no matches found)**: O(n);** *Only runs the preprocessing as StartCodonArray.size() == 0*
Worst case (all matches found)**: O(n) + (O(n/m) * O(3) * O(m-6)) = O(n);** *O(n) >>> (O($\frac{m-6(n)}{m}$)) (dominant)*
Average case (equal probability)**: O(n);** *O(n) >>> other terms (dominant)*

Conclusion for Brute Force (Improvement):
Using a divide-and-conquer approach, a huge improvement can be seen in terms of time complexity as instead of comparing all the text with pattern, we just check the necessary matches based on start + end codon and skip other irrelevants components. However, this only works if *pattern % 3 == 0 AND pattern.size()* ≥ *6*. Time improvement is directly proportional to the pattern size.

Knuth–Morris–Pratt (KMP) Algorithm

The basic idea for KMP when a mismatch is detected (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will match.

Pre-processing(LPS Array)

```
for(int i = 1; i < pattern.length(); i++)          ——— m
{
    if(pattern.charAt(j) == pattern.charAt(i))
        statement

    else if(j != 0)                                 ——— (m-1)C1
        statement                                   (m-1)

    else
        statement
}
```

Time complexity: $f(n) = m + (m-1)\,C1$
$= O(m)$

Time Complexity
Best / Average / Worst case
Best scenario: No matches found
Worst scenario: Pattern matches at least once
Average scenario: Equal probability of any subset of m matching

```
for(int i = 0; i < fileString.length(); i++)       ——— n+1
{
    if(fileString.charAt(i) == pattern.charAt(j))
        statement

    else if(j != 0)                                 ——— nC2
        statement                                   n

    if(j == pattern.length())
        statement
}
```

Time Complexity: $f(n) = (n+1) + nC2 + pre-processing$
$= O(n) + O(m)$
$= O(n)$ *as n > m*

Conclusion of KMP:
Since the *for* loop for KMP iterates a fixed number of times (*n times*), the best, average and worst cases will all have the same time complexity (*O(n)*). This makes it a much better algorithm to use compared to brute force if time is the key factor for comparison.

Boyer–Moore–Horspool Algorithm

The Boyer–Moore–Horspool algorithm is a simplification of the Boyer–Moore string search algorithm. It makes use of a bad character heuristic to determine how to skip unnecessary characters. The idea of the bad character heuristic is that upon mismatch of a character, we shift the pattern until the mismatch becomes a match or the pattern P moves past the mismatched character.

Pre-processing

```
for (int i = 0; i < 128; i++)                          129
    statement                                          128

for (int i = 0; i < pattern.length(); i++)             m+1
    statement                                          m
```

Time complexity: $f(n) = 129 + 128 + (m+1) + m$

$= O(m)$

Time Complexity

Best case: None of the characters in the pattern (m) occurs in the text to search in (n)

Explanation: Since no characters matches, shift will always increment by m

```
while(shift <= (fileString.length() - pattern.length()))        (n-m)/m + 1
{
    int j = pattern.length() - 1;

    while (j >= 0 && pattern.charAt(j) == fileString.charAt(shift + j))
        statement                                  Does not run since no matches
    if(j < 0)                                      (n-m)/m C1
        statement
    else if(1 > (j - badchar[fileString.charAt(shift + j)]))
        statement

    else
        statement
}
```

(n-m)/m

Time complexity: $f(n) = \frac{n-m}{m}(1 + C1) + 1 + pre-processing$

$= O(\frac{n}{m}) + O(m)$

$= O(\frac{n}{m})$ as $\frac{n}{m} > m$

Worst case: All the characters in the pattern and the text are the same.

```
while(shift <= (fileString.length() - pattern.length()))        (n-m)+1
{
    int j = pattern.length() - 1;

    while (j >= 0 && pattern.charAt(j) == fileString.charAt(shift + j))     (n-m)(m+1)
        statement                                                           (n-m)m
    if(j < 0)                                                               (n-m)C1
        statement
    else if(1 > (j - badchar[fileString.charAt(shift + j)]))
        statement

    else
        statement
}
```

(n-m)

Time complexity: $f(n) = (n-m+1) + (n-m) + (mn+m^2) + C1(n-m) + pre-processing$

$= 2n - 2m + 1 + mn + m^2 + nC1 - mC2 + O(m)$

$= O(mn) + O(m)$

$= O(mn)$ as mn > m

Average case: Equal probability of any subset of m matching

```
while(shift <= (fileString.length() - pattern.length()))          (i)+1
{
    int j = pattern.length() - 1;

    while (j >= 0 && pattern.charAt(j) == fileString.charAt(shift + j))          (i)(m+1)
        statement                                                                (i)m

    if(j < 0)
        statement
(i)
    else if(1 > (j - badchar[fileString.charAt(shift + j)]))                     (i)C1
        statement

    else
        statement
}
```

Time complexity:

$f(n) = \frac{1}{m}[\frac{(n-m)}{1}((m+1)+m+C1+1) + \frac{(n-m)}{2}((m+1)+m+C1+1)+...+\frac{(n-m)}{m}((m+1)+m+C1+1)\,] + preprocessing$

$= \frac{1}{m}\sum_{i=1}^{m}[\frac{1}{i}(n-m)(2m+1+C1)+1] + O(m) = \frac{1}{m}\sum_{i=1}^{m}[\frac{1}{i}(n-m)((m+1)+m+C1)+1] + O(m)$

$= \frac{1}{m}[(n-m)((m+1)+m+C1)\sum_{i=1}^{m}\frac{1}{i} + \sum_{i=1}^{m}1] + O(m) = \frac{1}{m}[(n-m)((m+1)+m+C1)\frac{2}{m(m+1)} + m] + O(m)$

$= (n-m)((m+1)+m+C1)\frac{2}{(m+1)} + 1 + O(m) = (n-m)(\frac{2(m+1)}{(m+1)} + \frac{2m}{(m+1)} + \frac{2C1}{(m+1)}) + 1 + O(m)$

$= O(n) + O(m)$

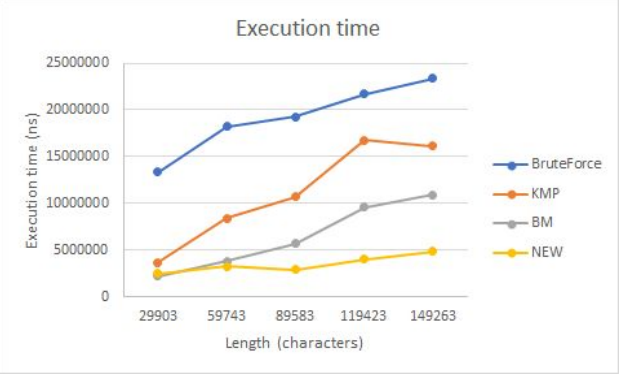
$= O(n)$ as n > m

Conclusion for Boyer-Moore-Horspool:

Overall, Boyer-Moore-Horspool is one of the best algorithms as it has the best time complexity in terms of the best case and average case as it increases linearly. However, in the worst case scenario, it will have a sharper increase as the time complexity is O(nm).

Summary

With the smallest best and average case time complexity out of the other two, Boyer-Moore-Horspool performs better as problem size increases. Based on the graph we derived, we can also see that it outperforms the other 2 algorithms in terms of execution time as the problem size increases (as can be seen from the picture). Although, the algorithm that we implement seems to perform better but due to the limitations, it might not work in other scenarios. (e.g. if scientists discover that DNA/RNA does not always come in a pair of 3). KMP will perform more consistently in different scenarios considering the Time complexity is the same for all the different cases.



Thus, Boyer-Moore-Horspool is better for larger problem sizes while KMP performs more consistently.

APA Citations:

Brute-force search. (2020, August 14). Retrieved September 13, 2020, from https://en.wikipedia.org/wiki/Brute-force_search

Knuth–Morris–Pratt algorithm. (2020, August 29). Retrieved September 13, 2020, from https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

KMP Algorithm for Pattern Searching. (2019, May 20). Retrieved September 13, 2020, from https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

Boyer–Moore–Horspool algorithm. (2020, April 04). Retrieved September 13, 2020, from https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore%E2%80%93Horspool_algorithm

Boyer Moore Algorithm for Pattern Searching. (2019, May 20). Retrieved September 13, 2020, from https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

Statement of Contribution

There were equal contributions by everyone in the team. We worked on all aspects of the project together from implementation, the analysis of different cases, to the report and presentation.

Name	**Focus Area**
Clarence Hong (U1922950G)	BF/BM/KMP
Chan Zhao Hui (U1921662D)	KMP/BF
Lim Bing Hong, Jasper (U1922783B)	BM/BF
Leow Wei Thou, Samuel (U1922978H)	BM/BF
Wong Chin Hao (U1921712L)	KMP/BF