School of Computer Science and Engineering

CZ2001: ALGORITHMS

Semester 1 2020/2021

Project 1: Searching Algorithm

Group members:

AIDE ISKANDAR BIN ABDUL RAHIM

CHEW KIT WAYE ANDREL

CHIEN YONG QIANG

GOH YUN BO, WAYNE

JIN HAN

Lab Group: SSP7 Group 1

Introduction

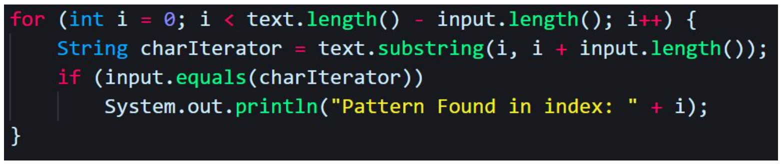
In bioinformatics research, DNA molecule and protein sequence searching are fundamental processes. Nucleic Acids are composed within a molecule, which consist of five-lettered sequences that mark the order of nucleotides. Alleles are formed within each DNA (A,C,G,T) or RNA (A,C,G,U) molecule [1], which are used for searching.

Sequence searching requires time, thus, essential to find the best possible case in any algorithm implemented, through careful analysis. In the times of Covid-19, sequence searching algorithms are crucial for completion at its fastest speed. Sample data from the NCBI repository are used in this project, to analyse the algorithms designed customarily toward virus genome sequences [2].

In order to search within those sequences, three different algorithms have been implemented, each programmed in Java. They will be used for the verdict of the quickest algorithms, in terms of time complexity. Firstly, the Brute Force algorithm [3][4], known for being the easiest, yet slowest algorithm for searching. Second, the Boyer-Moore algorithm [5], boasting a slightly faster performance to Brute Force. And third, the Knuth-Morris-Pratt algorithm(KMP) [6], with similar speed to Boyer-Moore. In this report, comparisons between the algorithms will be shown, to conclude the fastest of the three.

Analysis of Algorithms

Brute Force Sequential Search: (m = pattern, n = text file)



```
for (int i = 0; i < text.length() - input.length(); i++) {
    String charIterator = text.substring(i, i + input.length());
    if (input.equals(charIterator))
        System.out.println("Pattern Found in index: " + i);
}
```

Given a text with length n and pattern of length m, the algorithm will begin at index[0] of the text, and check for a copy of the pattern. The algorithm will then move on to index[1], and check if the pattern is present. This process will repeat itself until the algorithm finishes its last check at index[n-m+1].

Best case scenarios will occur when the text does not contain the 1st character in the pattern. Hence, the algorithm will compare only the 1st character in the pattern, against each character in the text until it reaches index[n-m+1]. The number of comparisons is (n-m+1).

Text: AGTGTTAAGGT
Pattern: CAA

Since the text and pattern is based on RNA/DNA molecules, there are only 4 different possible characters. Hence, for each comparison between a character in the pattern and the text, there is a 1-m/n% possibility of mismatch. On average, it will take less than 1-(m/n) comparisons for a mismatch to occur. Thus, an upper bound of the average number of comparisons is n/m(n-m+1).

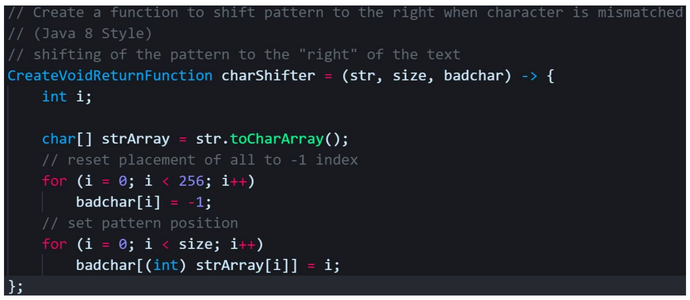
Worst case scenarios will occur when:
1) When all characters of the text and pattern are the same.
Text: GGGGGGGGG
Pattern: GGGG

2) When only the last character is different.
Text: TTTTTTTTTTT
Pattern: TTG

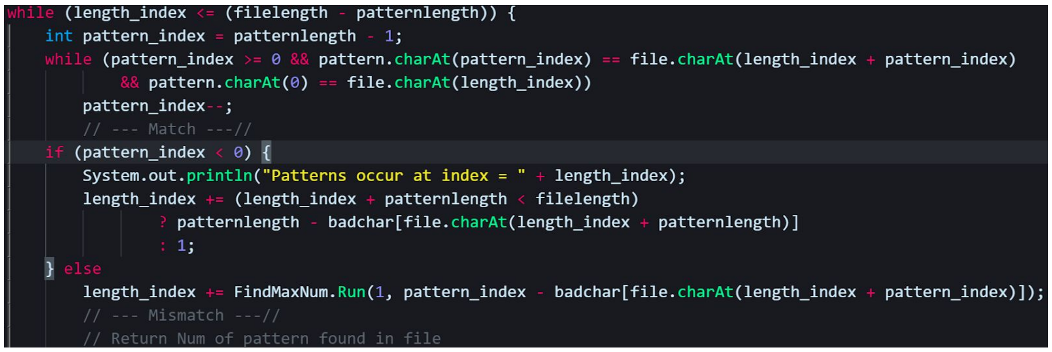
The number of comparisons done in these scenarios are [m(n-m+1)].

Best case time complexity: O(n)
Worst case time complexity: O(mn)
Average case time complexity: O(n)

Modified Boyer-Moore Searching Algorithm: (m = pattern, n = text file)



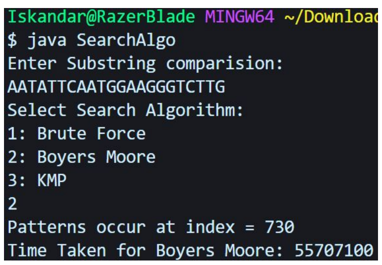
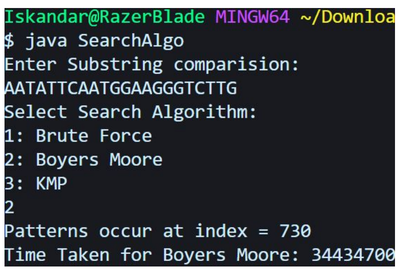
```
// Create a function to shift pattern to the right when character is mismatched
// (Java 8 Style)
// shifting of the pattern to the "right" of the text
CreateVoidReturnFunction charShifter = (str, size, badchar) -> {
    int i;

    char[] strArray = str.toCharArray();
    // reset placement of all to -1 index
    for (i = 0; i < 256; i++)
        badchar[i] = -1;
    // set pattern position
    for (i = 0; i < size; i++)
        badchar[(int) strArray[i]] = i;
};
```



```
while (length_index <= (filelength - patternlength)) {
    int pattern_index = patternlength - 1;
    while (pattern_index >= 0 && pattern.charAt(pattern_index) == file.charAt(length_index + pattern_index)
            && pattern.charAt(0) == file.charAt(length_index))
        pattern_index--;
        // --- Match ---//
    if (pattern_index < 0) {
        System.out.println("Patterns occur at index = " + length_index);
        length_index += (length_index + patternlength < filelength)
                ? patternlength - badchar[file.charAt(length_index + patternlength)]
                : 1;
    } else
        length_index += FindMaxNum.Run(1, pattern_index - badchar[file.charAt(length_index + pattern_index)]);
        // --- Mismatch ---//
        // Return Num of pattern found in file
```

Boyers-Moore does a search by an offset of the 1st index from the searching file, to the last index of the pattern. This searching method does a backwards comparison of the pattern with the searching file. Reducing the front loop few indexes of the searching file to check, only if the last index of the pattern is a mismatch. In cases of mismatch, the letter that causes the mismatch will be the key to search for the next possible pattern.

The algorithm has been modified to include the 1st index. The pattern's key must match in order to do a thorough search of the pattern in numbers of n. However, this only affects the search time within the 2nd loop, when it only occurs in a partial match.

Original:	Modified:
	

```
Iskandar@RazerBlade MINGW64 ~/Downloa
$ java SearchAlgo
Enter Substring comparision:
AATATTCAATGGAAGGGTCTTG
Select Search Algorithm:
1: Brute Force
2: Boyers Moore
3: KMP
2
Patterns occur at index = 730
Time Taken for Boyers Moore: 55707100
```

```
Iskandar@RazerBlade MINGW64 ~/Downloa
$ java SearchAlgo
Enter Substring comparision:
AATATTCAATGGAAGGGTCTTG
Select Search Algorithm:
1: Brute Force
2: Boyers Moore
3: KMP
2
Patterns occur at index = 730
Time Taken for Boyers Moore: 34434700
```

Best Case:

Text: ABCDEFGHIJKLMNOPQRST
Pattern: QRST

Worst Case:

Text: AAAAAAAAAAAAAAAAAAAAA
Pattern: AAAA

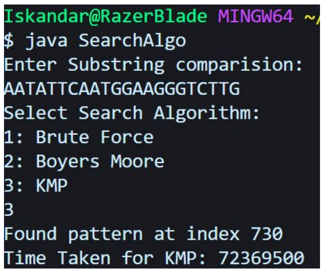
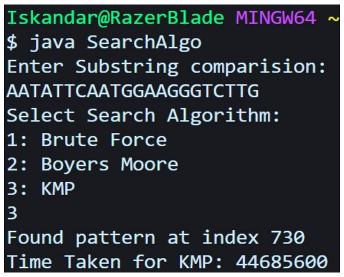
Best case time complexity: O(m/n)
Worst case time complexity: O(mn)

Average case time complexity:
Matching probability with no pattern repetition would be 1/ (nPm) [1/n* 1/(n-1) * 1/(n-2) * ....* 1/1]

Matching probability with pattern repetition would be 1/ (nPm / r!) [1/n* 1/(n-1) * 1/(n-2) * ....* 1/1]
Not matching probability with pattern repetition would be 1 - (1/ [nPm] [1/n * 1/(n-1) * 1/(n-2) *....* 1/1])
Not matching probability no pattern repetition would be 1 - (1/ [nPm / r!] [1/n * 1/(n-1) * 1/(n-2) *....* 1/1])
Average of the algorithms would be 1/ nPm [1/n * 1/(n-1) * 1/(n-2) *.... * 1/1]* n/m + 1 - 1/ nPm [1/n*1/(n-1)*1/(n-2)*....*1/1] * mn (We assume pattern has no repetition letters)
Average => O(m/n+mn)

Modified KMP Searching Algorithm: (m = pattern, n = text file)
The KMP Search algorithm checks from left to right. It stores m number of integers into an array, where m is the length of the pattern. The array is stored in such a way that the sub-pattern can be identified within the pattern. The purpose is to prevent the algorithm from checking those previously checked patterns. For example, a text of “AACAAAB” and a pattern of “AAAB”. An array lps[ ] will be stored as [0,1,2,0]. The first 2 letters match against the pattern, but text ‘C’ and pattern ‘A’ do not match. lps[ ] will be used to identify the sub-pattern. Text ‘C’ will then be compared to the second pattern ‘A’, instead of the first. Since it failed again, it will then move on to compare text ‘A’ with the start of pattern ‘A’.

As seen from the example, it is possible for this algorithm to make useless comparisons. Hence, instead of meaninglessly comparing text to pattern when mismatched, it will first check to see if the letter in text exists in the letter in pattern. If it does not, the pattern will be entirely skipped to compare with the next letter in text. Given the above example, when text ‘C’ is compared against pattern ‘A’, it will first find out if pattern contains C. Since it does not, the whole pattern will be shifted to compare against the next letter from the text.

Original	Modified
	

```
Iskandar@RazerBlade MINGW64 ~
$ java SearchAlgo
Enter Substring comparision:
AATATTCAATGGAAGGGTCTTG
Select Search Algorithm:
1: Brute Force
2: Boyers Moore
3: KMP
3
Found pattern at index 730
Time Taken for KMP: 72369500
```

```
Iskandar@RazerBlade MINGW64 ~
$ java SearchAlgo
Enter Substring comparision:
AATATTCAATGGAAGGGTCTTG
Select Search Algorithm:
1: Brute Force
2: Boyers Moore
3: KMP
3
Found pattern at index 730
Time Taken for KMP: 44685600
```

Best Case:
Text: GTCGTCGTCGTC
Pattern: ATCG
Best case occurs when the first letter of the pattern does not match against any of the letters in text. This means that there will be (n-m+1) comparisons.

Worst Case:
Text: AAAAAAAAAAAA
Pattern: AAAT

Worst case is when all letters in text are contained in pattern, and when the last letter of pattern is a mismatch. The pattern must contain (m-1) number of the same letter.

Time complexity for worst case = $m+2(n-m)$
= $2n-m$
= $O(n)$

Index	0	1	2	3	4	5
Text	A	A	A	A	A	A
Pattern	A	A	A	A		
		A	A	A	A	
			A	A	A	A

A successful pattern found will be indicated as "occurrence at (i+1)th position", where $0 \le i < n - m + 1$. Assume the probability of a successful search at ith index is $\frac{1}{n-m+1}$.

Average time complexity of successful case $A_s(n)$:

$$\frac{1}{n-m+1} \times [m + (m+2) + (m+4) + \ldots + (m + 2(n-m))]$$

$$= \frac{1}{n-m+1} \times \sum_{i=0}^{n-m} 2i + m \;=\; \frac{1}{n-m+1} \times \left[m(n-m+1) + \frac{(n-m+1)(0+2(n-m))}{2} \right] = n$$

Average time complexity of unsuccessful case $A_f(n)$:

$$m + 2(n-m) = 2n - m$$

Probability of one random nucleotide within DNA or RNA being one of its elements is 25%. Hence, we can infer that the success rate of matching can never be greater than 25%.

Therefore, the average-case complexity = Prob(success) $A_s(n)$ + Prob(unsuccessful) $A_f(n)$

$$= \frac{n}{4} + \frac{3(2n-m)}{4} = O(n)$$

Best case time complexity : O(n-m)
Worst case time complexity : O(n)
Average case time complexity : O(n)

Conclusion

After individually analysing and modifying the searching algorithms, it is concluded that the modified Boyer-Moore search algorithm performs the best, based on time complexity. Brute force remains the least effective algorithm. Although both KMP and Boyer-Moore were modified, Boyer-Moore outperforms, evident from its search speed shown in the program runtime, along with the best and worst cases.

Statement of Contribution

The implementation, including design and analysis of the 3 algorithms, documentation of report and creation of presentation slides were collectively contributed by all group members - Iskandar, Yong Qiang, Andrel, Wayne and Jin Han.

References

[1] Lodish H, Berk A, Zipursky SL, "Molecular Cell Biology. 4th edition", 2000. [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK21514/

[2] NCBI, Genome Data Source File. [Online]. Available: https://ftp.ncbi.nlm.nih.gov/genomes/Viruses/

[3] GeeksforGeeks, "Naive Algorithm for Pattern Searching", 2019. [Online]. Available: https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/

[4] T. Jiang, "String". [Online]. Available: http://www.cs.ucr.edu/~jiang/cs141/string.pdf

[5] GeeksforGeeks, "Boyer Moore Algorithm for Pattern Searching", 2019. [Online]. Available: https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

[6] GeeksforGeeks, "KMP Algorithm for Pattern Searching", 2019. [Online]. Available: https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/