DNA Sequence Matching by Using String Matching Algorithm

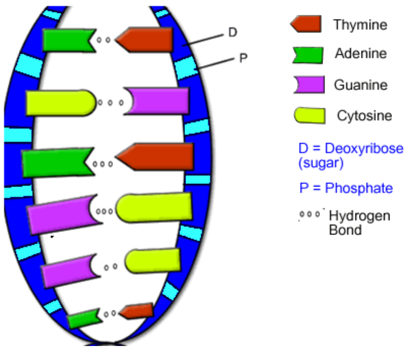
Andre Susanto - 13512028
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
andresusanto@students.itb.ac.id; as@andresusanto.info

***Abstract*—Deoxyribonucleic acid or we know it as DNA is a molecule that encodes the genetic instructions used in the development and functioning of all known living organisms and much kind of viruses. DNA is a long polymer made from repeating units called nucleotides. Scientist have discovered that DNA can be written as a sequence (we'll call it later as DNA Sequence), and this sequence can take up to 3.0×10^9 base pairs (for human genome). When we want to analyze DNA Sequences, especially if they have a big number of pairs, we will want an efficient, yet powerful algorithm to make our analysis done faster. That's why we'll look some string matching algorithm and find the best one to be used for DNA Sequence Analysis.**

***Index Terms*—DNA, DNA Sequence, String Marching, DNA Analysis, DNA Sequence Matching.**

I. Introduction

DNA which is the shorter form of Deoxyribonucleic acid is a molecule that encodes the genetic instructions used in the development and functioning of all known living organisms and many viruses [1]. DNA encodes the genetic instructions as a sequence which we call it the DNA Sequence [1].



Picture 1.1 – DNA Sequence

DNA Sequence only contains four kind of acid types [1], which are the Thymine (T), Adenine (A), Guanine (G) and Cytosine (C). The Thymine is a pair with Adenine, and the Guanine is a pair with Cytosine. In DNA Sequence only one of the strands has the real meaning of the whole sequence. That strand is called '*sense strand*'. The other strand is just a complementary for the first one and called '*anti sense*' [2].

DNA Sequencing process requires advanced technology and sophisticated equipment. They can cost the scientists up to $700,000 per equipment which is around Rp8,400,000,000 and $6,000 per run which is around Rp72,000,000 [3].

Picture 1.2 – DNA Sequencer

Of course with that amount of money, scientists want accurate, yet fast and efficient analysis process. Because of the advancement of technology, scientists can rely some of their works to computers. Even by using supercomputers, scientists will require fast and efficient algorithm to analyze their '*big data*'. Picking the wrong algorithm will cost them some time and of course will affect the future analysis of the data.

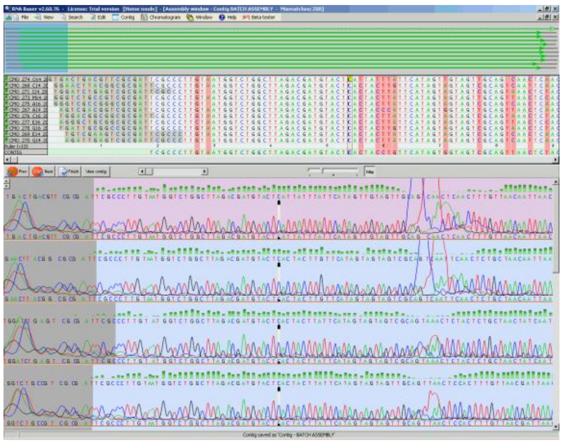
In this paper, we will discuss about the DNA Subsequence Matching which can be solved using *String Matching Algorithm* in computer science. There are some variant of String Matching Algorithm. Three of them are *String Matching by Brute-Force*, Knuth-Morris-Pratt or KMP Algorithm, and Boyer-Moore Algorithm.

Each variant has their own advantages and disadvantages. Even for the simplest one, the String

Matching Algorithm by using Brute-Force, and the '*smart*' variants such as KMP Algorithm and Boyer-Moore Algorithm. We will discover which one is the best algorithm should be applied by Geneticist (Geneticist is scientist who expertise in heredity and DNA Science).

II. Related Theories

A. *DNA Sequence*



Picture 2.1 – The DNA Sequence Analysis Program

The DNA Sequence is formed by four types of acid types which are the Thymine (T), Adenine (A), Guanine (G) and Cytosine (C) [2]. The DNA Sequence contains a pair of strands; one of them contains information about the owner of the DNA, the other one is just a complementary for the other one [2]. The strand that contains the real information contains series of acids. This series of acids can be decoded to some useful information, for example if we want to find out whether the dolphins and orcas are cousins to each other.

B. *String*

In computer science, string is a finite sequence of symbols that can consist of alphabetic, numeric, or special characters [4]. In some programming language, string are represented as a primitive data type, in other programming language string are represented as a class (or object for its instance), and there is also some programming language that not know string at all (they represented as *array of characters*).

The formal definition of string is "*Let Σ be a non-empty finite set of symbols (alternatively called characters), called the alphabet. No assumption is made about the nature of the symbols. A string (or word) over Σ is any finite sequence of symbols from Σ. For example, if Σ = {0, 1}, then 01011 is a string over Σ*" [4].

There are some properties of string as shown in Table below:

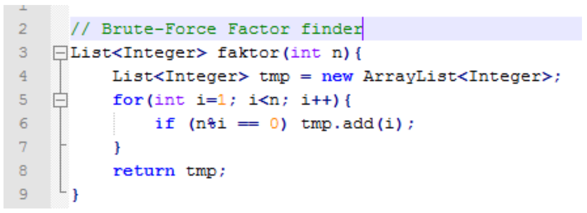
Name	Description
Length	Assume S is a string of size m. $S = x_1x_2x_3.....x_m$ Length is the size of string which in this case is m.
Prefix	Assume S is a string of size m. $S = x_1x_2x_3.....x_m$ k is an integer between 1 and m, string P is a Prefix of S if P is a substring *S[1..k-1]*.
Suffix	Assume S is a string of size m. $S = x_1x_2x_3.....x_m$ k is an integer between 1 and m, string X is a Suffix of S if X is a substring *S[k-1..m]*.
Reverse	Assume S is a string of size m. $S = x_1x_2x_3.....x_m$ String R is a reverse of string S if and only if R contains all of symbols in S in reverse order. $R = x_mx_{m-1}x_{m-2}.....x_1$

Table 2.1 – Some Properties of String

String with length of zero or *S[0]* is called null, the symbol is ∅.

C. *Brute Force Algorithm*

In computer science, brute force is a very general problem solving technique that consists of enumerating systematically all possible candidates of solution to find the solution that satisfy the problem given [5].



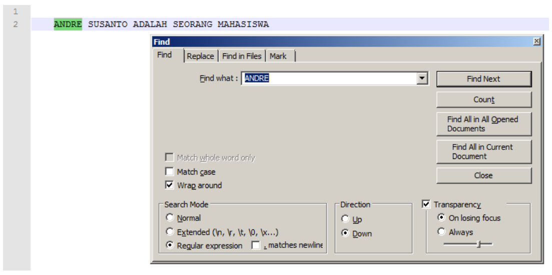
```
// Brute-Force Factor finder
List<Integer> faktor(int n){
    List<Integer> tmp = new ArrayList<Integer>;
    for(int i=1; i<n; i++){
        if (n%i == 0) tmp.add(i);
    }
    return tmp;
}
```

Picture 2.2 – Brute Force Factor finder in Java

For example, a brute force algorithm to find the factors of an integer (i) will enumerate all possible solution from 1 to i and check whether i divides by the number in iteration process is not making a reminder. Another example of brute force algorithm to find an object in a collection will check the collection item each by each from beginning to the end of item.

D. *String Matching*

In computer science, string matching is a technique to find whether a pattern is a match of a given string. If the pattern given is a string, then string matching is a technique to find whether the given string is a subset of another given string (For example, string “ABCDEFGHIJKLM” will matches with pattern “ABC” or “DEF” or “KLM”). If the pattern given is an expression, then string matching will process the pattern to find whether the given string satisfy the pattern or contain a substring that matches it (For example, string “ABCDEF” will matches with regex “/***[A-Z]***+/g” or “/***w***+/g”.


ANDRE SUSANTO ADALAH SEORANG MAHASISWA

Find what : ANDRE

Match whole word only: unchecked; Match case: unchecked; Wrap around: checked

Search Mode: Normal / Extended (\n, \r, \t, \0, \x...) / [✓ Regular expression]; . matches newline: unchecked

Direction: Up / [✓ Down]

Transparency: checked; [✓ On losing focus] / Always

Find
Find
Replace
Find in Files
Mark
Find Next
Count
Find All in All Opened Documents
Find All in Current Document
Close

Picture 2.3 – String Matching Application

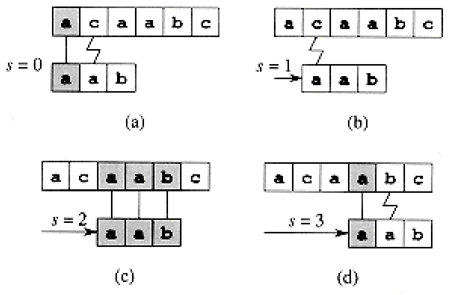
String Matching is widely applied to so much program in the world. This includes the string finder in many variations of text editor, the regular expression (or well known as regex), and even some of Web Analyzer uses string matching algorithm to get the information or data.

E. *Brute Force String Matching Algorithm*

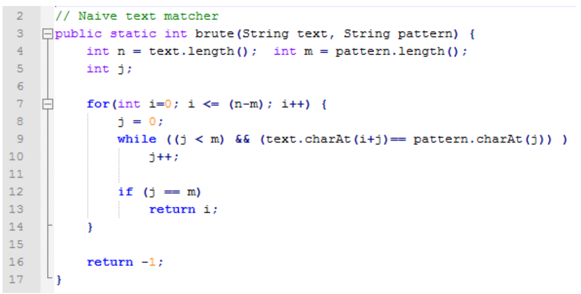
The Brute Force String Matching Algorithm (or also called *Naïve String Matching Algorithm*) is made by using the concept of previous theory (the Brute Force theory). The brute force string matching algorithm can be done with the following steps [6]:

1. Iterates the characters in string given from the beginning of the string to the end of the string minus the length of the pattern and matches it with the beginning of the pattern.
2. If the beginning character of the pattern equals with the current iterated character, compare the next character of the pattern with the next character of the string. Do this step until the end of the pattern or stop if the current iterated character doesn’t match anymore.
3. If the iterated characters match with the whole pattern, then the string matches. Continue the iteration process to find other matches.
4. In the other hand repeat the process until the end of the string minus the length of the pattern.
5. If until the end of string there are no any substring matches with the pattern, then the string is not match with the pattern.

For example we have a string “acaabc” and a pattern “aab”. First, match the first character of the string with the first character of the pattern. It matches; let’s compare the second character of the pattern with the second character of the string. It doesn’t match. So, compare the second character with the first character of the pattern (because it’s in the iteration process). Repeat these process until we figure that the pattern will match the string in the third character until the fifth character. After this matches, do the iteration process until it reaches the end of string minus the length of the pattern.



Picture 2.4 – Example of Naïve String Matching


```
// Naive text matcher
public static int brute(String text, String pattern) {
    int n = text.length();  int m = pattern.length();
    int j;

    for(int i=0; i <= (n-m); i++) {
        j = 0;
        while ((j < m) && (text.charAt(i+j)== pattern.charAt(j)) )
            j++;

        if (j == m)
            return i;
    }

    return -1;
}
```

Picture 2.5 – Brute Force String Matching Implementation in Java

This algorithm seems very simple, but in its worst case this algorithm will have a very high time complexity. For example if we want to match a string “aaaaaaaaaaaaaaab” with pattern “ab”, this algorithm will do $m(n - m + 1)$ comparisons which will make the time complexity become **O(mn)**.

In its best case, this algorithm can process the string with n times comparisons (n is the length of the string) which will make its time complexity become **O(n)**. This case occurs when the first character of the pattern doesn’t match the iterated character in the string except for the matched substring. For example if we want to match a string “We will go to jogja” with pattern “jogja”.

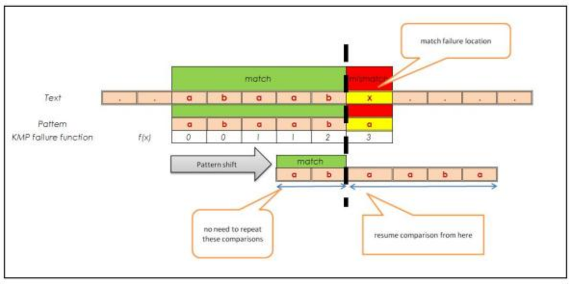
The average case (based from ordinary text matching) of this algorithm take **O(m+n)** which is quite fast. For Example if we want to match a string “a string searching example is standard” with pattern “super”.

This algorithm is fast when the variance of the alphabet of the string (and pattern) is large (this will make the possibility of mismatch occurs in the beginning is high). And its slower if the variance of alphabet is small (for example binary files, image files, etc.).

F. *Knuth-Morris-Pratt (KMP) Algorithm*

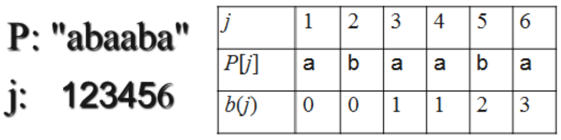
The Knuth-Morris-Pratt (we’ll call it KMP later), is a string matching algorithm that iterates the pattern from the beginning of the string to the end of string (just like the brute force string matching algorithm), but this algorithm shift the pattern more intelligently than brute force string matching algorithm [6].

The KMP Algorithm fundamental idea is based from the following question "*If a mismatch occurs between the text and pattern P at P[j], what is the most we can shift the pattern to avoid wasteful comparisons?*" and an answer "*the largest prefix of P[1 .. j-1] that is a suffix of P[1 .. j-1]*" [6].



Picture 2.6 – Fundamental idea of KMP Algorithm

To use this algorithm, firstly we must process the pattern used to get the KMP Border Function of the pattern. The border function *b(k)* is defined as the size of the largest prefix of P[1..k] that is also a suffix of P[1..k]. With *j* = mismatch position in P[] and *k* = position before the mismatch (k = j-1). The other name of KMP Border Function is *failure function* or we can call it just *fail*



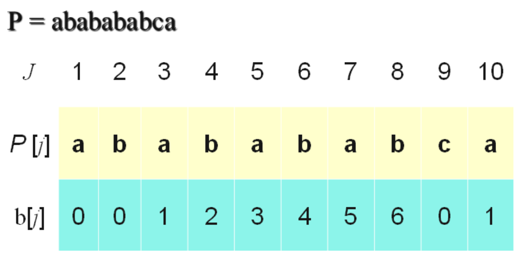
P: "abaaba"

j: 123456

j	1	2	3	4	5	6
P[j]	a	b	a	a	b	a
b(j)	0	0	1	1	2	3

Table 2.2 – Border Function in Table Representation

For example, we want to get the value of border function for the pattern P = "abaaba". To make the process easier, let's put each of characters in P to a table consists of Character position j (as column header) and border function at j (as row header) as shown in Table 2.2. Iterate from 1 to 6, in each iteration process find the largest suffix that is also a prefix in substring P[1..j] where j is the current iterated position. For example, in j = 5, we got the b(j) = 2 because the largest prefix of "*abaab*" which is also a suffix for "*baab*" is "*ab*" and the length of "*ab*" is 2, so we got *b(5) = 2*. Another example of KMP Border Matching determination is presented in the following table (Table 2.3).

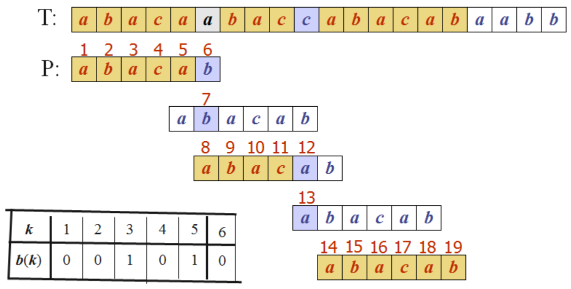


P = abababapca

J	1	2	3	4	5	6	7	8	9	10
P[j]	a	b	a	b	a	b	a	b	c	a
b[*j*]	0	0	1	2	3	4	5	6	0	1

Table 2.3 – Example of Determining Border Function

Now, how to use the Border Function that we get from previous process? The usage of the Border Function will modify the Brute Force String Matching algorithm (in the way of shifting the pattern). When a mismatch occurs at P[j], (i.e. P[j] != T[i]), then k = j-1 and j = b(k) + 1, which will obtain the new j for comparison.



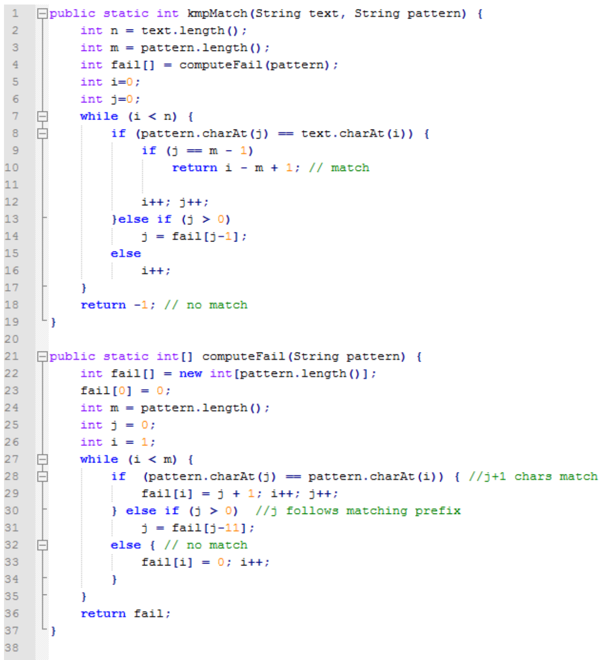
k	1	2	3	4	5	6
b(*k*)	0	0	1	0	1	0

Picture 2.7 – KMP Example

Now, let's try an example. We want to check whether string "*abacaabaccabacabaabb*" matches with pattern "*abacab*". First, we compute the border function of pattern "*abacab*" which will resulting to "001010" border function.

The first five comparisons between pattern and string match each other until the sixth comparison. Because the sixth comparison is fail, check the value of b(j-1) border function, in this case is b(5). The value is 1, so the next comparison (7th) start at *j = b(5) + 1 = 2*. The seventh comparison doesn't match, once again look at the value of b(j-1) which in this case is 0 which will resulting the next comparison (8th) start at j = 1. Repeat the process until we found that the pattern completely matches the string at position 14 util 19.

The implementation of KMP can easily be done by implementing it in two different function. The first one is function to match the string with the pattern and the second is function to preprocess the pattern and generate the border function of it.



```java
public static int kmpMatch(String text, String pattern) {
    int n = text.length();
    int m = pattern.length();
    int fail[] = computeFail(pattern);
    int i=0;
    int j=0;
    while (i < n) {
        if (pattern.charAt(j) == text.charAt(i)) {
            if (j == m - 1)
                return i - m + 1; // match

            i++; j++;
        }else if (j > 0)
            j = fail[j-1];
        else
            i++;
    }
    return -1; // no match
}

public static int[] computeFail(String pattern) {
    int fail[] = new int[pattern.length()];
    fail[0] = 0;
    int m = pattern.length();
    int j = 0;
    int i = 1;
    while (i < m) {
        if  (pattern.charAt(j) == pattern.charAt(i)) { //j+1 chars match
            fail[i] = j + 1; i++; j++;
        } else if (j > 0)  //j follows matching prefix
            j = fail[j-11];
        else { // no match
            fail[i] = 0; i++;
        }
    }
    return fail;
}

```

Picture 2.8 – Implementation of KMP in two functions by using Java

The time complexity for this algorithm can be determined by joining time complexity for determining the border functions (which is ***O(m)***) and time complexity for matching the string and pattern (which is ***O(n)***). So the time complexity for KMP String Matching Algorithm will be ***O(m+n)*** [6]. This time complexity is quite faster compared to the brute force string matching time complexity.

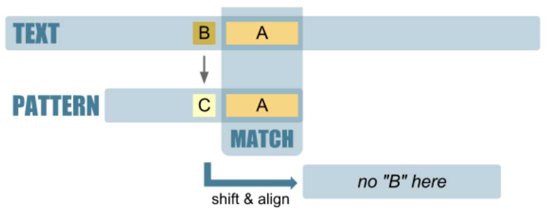
The main advantage of this algorithm is the intelligent shifting method that will prevent the algorithm to move backwards in the matching string. This makes the algorithm is a very good algorithm for processing very large files that are read in from external devices or through a network stream.

Although KMP Algorithm has an intelligent pattern shifting method, it doesn't work so well as the size of the alphabet increases. That because it will increases the chance of mismatches and mismatches tend to occur early in the pattern, but KMP is faster when the mismatches occur later).

G. Boyer-Moore Algorithm

The Boyer-Moore string matching algorithm (we'll call it as BM later) is a string matching algorithm that uses two important techniques in order to prevent redundant comparison [6].

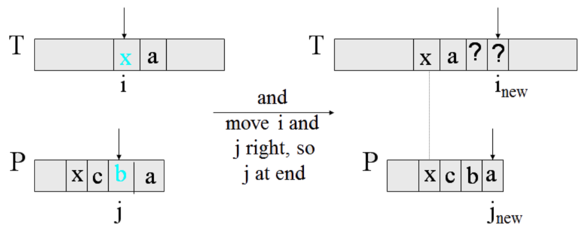
The first technique is the *looking-glass technique*. By using this technique, BM Algorithm will look for pattern in string by moving backwards through the pattern starting at its end. This technique illustrated in Picture 2.9.



Picture 2.9 – Looking glass technique illustration

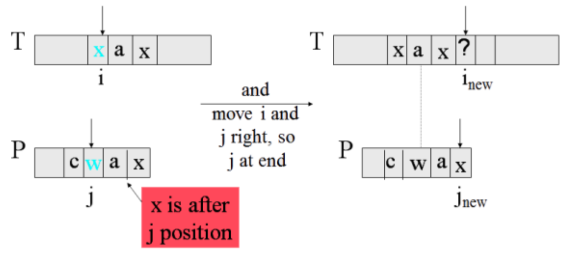
The second technique is *the character jump technique*. By using this technique, the BM Algorithm will jump intelligently by using the current mismatch cases (mismatch occurs when the character in pattern P[j] is not the same as S[i]). There are three mismatch cases, tried in order.

The first cases occurs when P contains ***'x'*** somewhere, then try to shift P right to align the last occurrence of ***'x'*** in P with T[i]. The illustration of this case is presented in the following Picture 2.10.



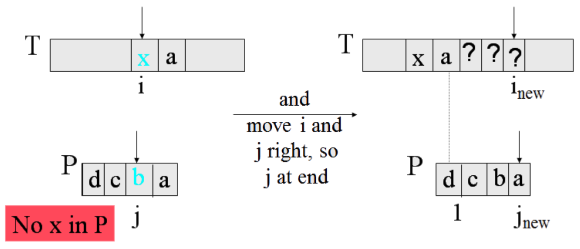
Picture 2.10 – Case 1 character jump technique

The second case occurs when P contains ***'x'*** somewhere, but a shift right to the last occurrence is not possible, then shift P right by 1 character to T[i+1]. The illustration of this case is presented in the following Picture 2.11.



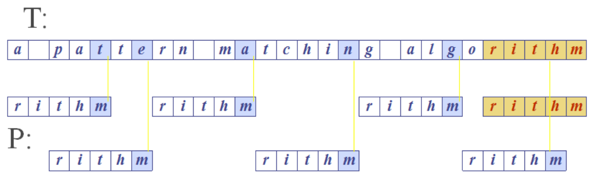
Picture 2.11 – Case 2 character jump technique

The third case occurs when neither of the first case nor the second case occurs. If this case occurs then shift P to align P[1] with S[i+1]. The illustration of this case is presented in the following Picture 2.12.



Picture 2.12 – Case 3character jump technique

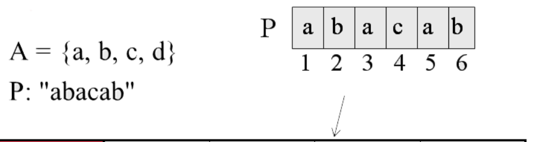
An example which shows us the behavior of the BM Algorithm using the three cases to shift the pattern is presented in the following Picture 2.13.



Picture 2.13 – Example by using these three cases

The last occurrence function in BM Algorithm is determined by preprocess the pattern P and the alphabet A. The last occurrence function (***L()***) maps all the letters in A to integers. L(x) is defined as the largest index i such that P[i] == x, or -1 if no such index exists.

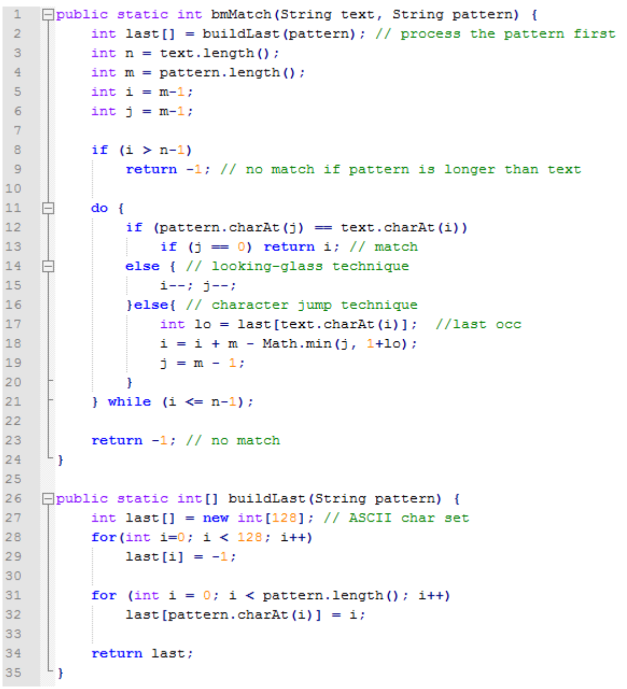
For example, we have a pattern "abacab" and alphabet of the string is A = {a,b,c,d}. So if we process P and find its last occurrence function, we will get the result as presented by Table 2.4 below.



x	*a*	*b*	*c*	*d*
L(x)	5	6	4	-1

Table 2.4 – Last occurrence determination of Pattern P

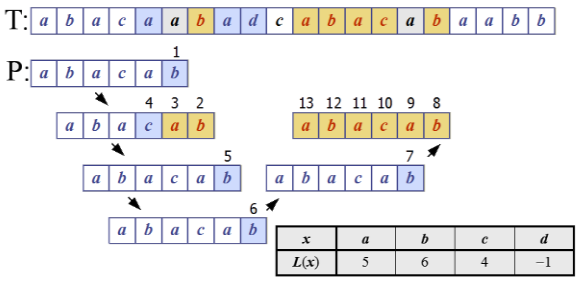
In the implementation process, the BM Algorithm calculates the ***L()*** when the pattern P is read in. Usually, the last occurrence function is stored as an array (just like a table in previous example). The implementation of this algorithm can easily be done by using two functions, the first is the pattern-string matching function and the second is the pattern preprocess function (to generate the last occurrence values). This algorithm's implementation in Java is presented by the following codes in Picture 2.14.



```java
public static int bmMatch(String text, String pattern) {
    int last[] = buildLast(pattern); // process the pattern first
    int n = text.length();
    int m = pattern.length();
    int i = m-1;
    int j = m-1;

    if (i > n-1)
        return -1; // no match if pattern is longer than text

    do {
        if (pattern.charAt(j) == text.charAt(i))
            if (j == 0) return i; // match
        else { // looking-glass technique
            i--; j--;
        }else{ // character jump technique
            int lo = last[text.charAt(i)];  //last occ
            i = i + m - Math.min(j, 1+lo);
            j = m - 1;
        }
    } while (i <= n-1);

    return -1; // no match
}

public static int[] buildLast(String pattern) {
    int last[] = new int[128]; // ASCII char set
    for(int i=0; i < 128; i++)
        last[i] = -1;

    for (int i = 0; i < pattern.length(); i++)
        last[pattern.charAt(i)] = i;

    return last;
}
```

Picture 2.14 – Boyer-Moore implementation in Java

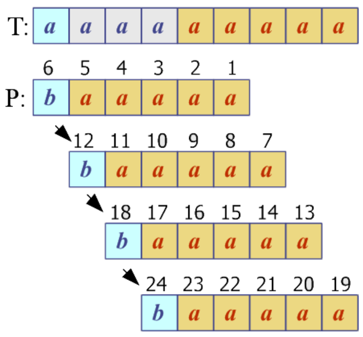
Another example of the BM Algorithm will show each steps taken in comparing and shifting the pattern based on the selected case in each condition. In this example, we have a string T = "abacaabadcabacabaabb" and a pattern P = "abacab". First we compare the last character of the pattern P with the 6th character of the string. It doesn't match, but the character 'a' is in the pattern with last occurrence of 5. So, we shift the pattern so the sixth character is the same alignment with the fifth position of the pattern and start comparing the pattern again. In the 4th comparison, character 'c' in the pattern doesn't match with character 'a' in the string, but it's impossible to shift the pattern to its last occurrence, so in this case, shift the pattern 1 step to the right. In the 6th comparison, the iteration process meets character 'd' in the string which is not exist in the pattern (that resulting last occurrence of -1), so jump the pattern by the pattern length to avoid redundant comparison. Do the processes until it finds a match in 8th comparison until the 13th comparison. This example is illustrated with each step in the following Picture 2.15.



x	a	b	c	d
L(x)	5	6	4	–1

Picture 2.15 – Example with steps taken

The BM Algorithm seems to be fast, but its worst case time complexity is ***O(nm + A)*** which is a very big time complexity to deal with [6]. In fact, the Boyer Moore Algorithm is significantly faster than brute force algorithm for searching English text. Boyer-Moore algorithm tends to be fast when the alphabet (A) is large (the BM algorithm will jump with a long distance when the character in string is not exist in the pattern), and slow when the alphabet is small. This makes the BM Algorithm is not good for searching binary files, images, etc.



Picture 2.16 – The Boyer-Moore worst case

To show us the worst case of the Boyer Moore Algorithm, we run the Boyer-Moore algorithm with a string T = "aaaaaaaaa" and a pattern P = "baaaaa". The Boyer-Moore algorithm will do 24 comparisons to solve this problem (the solving process is shown in Picture 2.16 above).

III. Analysis

A. *Theoretical Analysis*

From the previous chapter, we can get some of the algorithm best and worst cases depend on the input string. The summary of those is listed in the following Table 3.1.

Name	Best	Worst
Naive	Big size alphabet	Small size alphabet
KMP	Small size alphabet	Big size alphabet
BM	Big size alphabet	Small size alphabet

Table 3.1 – Summary of algorithm best and worst cases

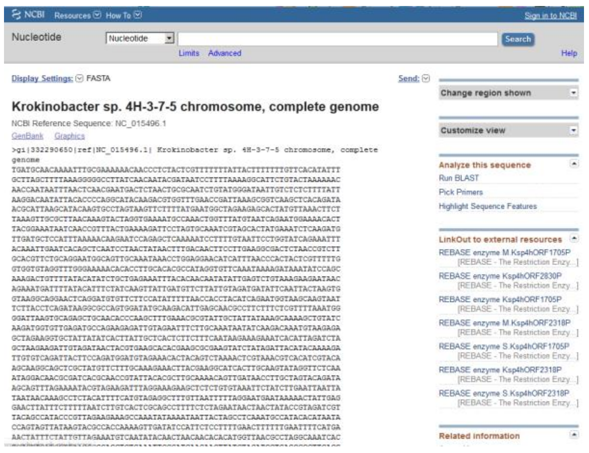
From the previous chapter, we can get the alphabet size of a DNA which is 4 (because the DNA alphabet consist of A = {A,G,T,C}). We can conclude the alphabet size of 4 is a small size alphabet if we compare it to English alphabet that has size of 64 (a-z letters in case sensitive form plus ten numeric values). This size of alphabet tends to make time complexity of Brute Force Algorithm become ***O(mn)***, the Boyer-Moore become ***O(nm + A)*** and the KMP become ***O(m+n)***.

So, theoretically, by using the time complexity value, comparing these algorithms with small alphabet inputs and large string size will make the KMP much faster than Boyer-Moore and Brute Force String Matching algorithm.

B. Practical Analysis

In this practical analysis, we will do some experiments and simulations to determine the best algorithm to be used in DNA Sequence Matching.

In this analysis, we use the piece of codes from the previous chapter (theories chapter) and count each of comparison done by the algorithm in process of solving the given DNA Sequence. The DNA data we used are variations of some species taken from DNA Gen Bank (NCBI-USA). We only use the DNA of micro bacteria which has length 100,000++ characters.


NCBI
Resources
How To
Sign in to NCBI
Nucleotide
Nucleotide
Search
Limits
Advanced
Help
Display Settings:
FASTA
Send:
Change region shown
Customize view
GenBank
Graphics
Analyze this sequence
Run BLAST
Pick Primers
Highlight Sequence Features
LinkOut to external resources
REBASE enzyme M.Ksp4hORF1705P
[REBASE - The Restriction Enzy...]
REBASE enzyme Ksp4hORF2830P
[REBASE - The Restriction Enzy...]
REBASE enzyme Ksp4hORF1705P
[REBASE - The Restriction Enzy...]
REBASE enzyme M.Ksp4hORF2318P
[REBASE - The Restriction Enzy...]
REBASE enzyme S.Ksp4hORF1705P
[REBASE - The Restriction Enzy...]
REBASE enzyme Ksp4hORF2318P
[REBASE - The Restriction Enzy...]
REBASE enzyme S.Ksp4hORF2318P
[REBASE - The Restriction Enzy...]
Related information

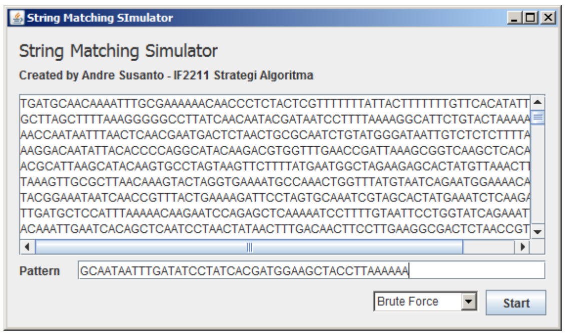
Krokinobacter sp. 4H-3-7-5 chromosome, complete genome

NCBI Reference Sequence: NC_015496.1

```
>gi|332290650|ref|NC_015496.1| Krokinobacter sp. 4H-3-7-5 chromosome, complete
genome
TGATGCAACAAAATTTGCGAAAAAACAACCCTCTACTCGTTTTTTTATTACTTTTTTTGTTCACATATTT
GCTTAGCTTTTAAAGGGGGCCTTATCAACAATACGATAATCCTTTTAAAAGGCATTCTGTACTAAAAAAC
AACCAATAATTTAACTCAACGAATGACTCTAACTGCGCAATCTGTATGGGATAATTGTCTCTCTTTTATT
AAGGACAATATTACACCCCAGGCATACAAGACGTGGTTTGAACCGATTAAAGCGGTCAAGCTCACAGATA
ACGCATTAAGCATACAAGTGCCTAGTAAGTTCTTTTATGAATGGCTAGAAGAGCACTATGTTAAACTTCT
TAAAGTTGCGCTTAACAAAGTACTAGGTGAAAATGCCAAACTGGTTTATGTAATCAGAATGGAAAACACT
TACGGAAATAATCAACCGTTTACTGAAAAGATTCCTAGTGCAAATCGTAGCACTATGAAATCTCAAGATG
TTGATGCTCCATTTAAAAACAAGAATCCAGAGCTCAAAAATCCTTTTGTAATTCCTGGTATCAGAAATTT
ACAAATTGAATCACAGCTCAATCCTAACTATAACTTTGACAACTTCCTTGAAGGCGACTCTAACCGTCTT
GCACGTTCTGCAGGAATGGCAGTTGCAAATAAACCTGGAGGAACATCATTTAACCCACTACTCGTTTTTG
GTGGTGTAGGTTTGGGAAAAACACACCTTGCACACGCCATAGGTGTTCAAATAAAAGATAAATATCCAGC
AAAGACTGTTTTATACATATCTGCTGAGAAATTTACACAACAATATATTGAGTCTGTAAAGAAGAATAAC
AGAAATGATTTTATACATTTCTATCAAGTTATTGATGTTCTTATTGTAGATGATATTCAATTACTAAGTG
GTAAGGCAGGAACTCAGGATGTGTTCTTCCATATTTTTAACCACCTACATCAGAATGGTAAGCAAGTAAT
TCTTACCTCAGATAAGGCGCCAGTGGATATGCAAGACATTGAGCAACGCCTTCTTTCTCGTTTTAAATGG
GGATTAAGTGCAGAGCTGCAACACCCAAGCTTTGAAACGCGTATTGCTATTATAAAGCAAAAGCTGTATC
AAGATGGTGTTGAGATGCCAGAAGAGATTGTAGAATTTCTTGCAAATAATATCAAGACAAATGTAAGAGA
GCTAGAAGGTGCTATTATATCACTTATTGCTCACTCTTCTTTCAATAAGAAAGAAATCACATTAGATCTA
GCTAAGAAGATTGTAGATAACTACGTGAAGCACACGAAGCGCGAAGTATCTATAGATTACATACAAAAGA
TTGTGTCAGATTACTTCCAGATGGATGTAGAAACACTACAGTCTAAAACTCGTAAACGTCACATCGTACA
AGCAAGGCAGCTCGCTATGTTCTTTGCAAAGAAACTTACGAAGGCATCACTTGCAAGTATAGGTTCTCAA
ATAGGACAACGCGATCACGCAACCGTATTACACGCTTGCAAAACAGTTGATAACCTTGCTAGTACAGATA
AGCAGTTTAGAAAATACGTAGAAGATTTAGGAAAGAAGCTCTCTGTGTAAATTCTATCTTGAATTAATTA
TAATAACAAAGCCTCTACATTTTCATGTAGAGGCTTTGTTAATTTTTAGGAATGAATAAAAACTATTGAG
GAACTTATTTCTTTTTAATCTTGTCACTCGCAGCCTTTTCTCTAGAATAACTAACTATACCGTAGATCGT
TACAGCCATACCCGTTAGAAGAAAGCCAAATATAAAATAATTACTAGCCTCAAATGCCATACACATAATA
CCAGTAGTTATAAGTACGCCACCAAAAGTTGATATCCATTCTCCTTTTGAACTTTTTTGAATTTTCATGA
AACTATTTCTATTGTTAGAAATGTCAATATACAACTAACAACACACATGGTTAACGCCTAGGCAAATCAC
```

Picture 3.1 – Source DNA

For the first try, we use the Krokinobacter sp. gen and find whether it matches with pattern "***GCAATAATTTGATATCCTATCACGATGGAAGCTACCTTAAAAAA***". Firstly, we set the test environment and load all data to the simulator.



String Matching SImulator

String Matching Simulator

Created by Andre Susanto - IF2211 Strategi Algoritma

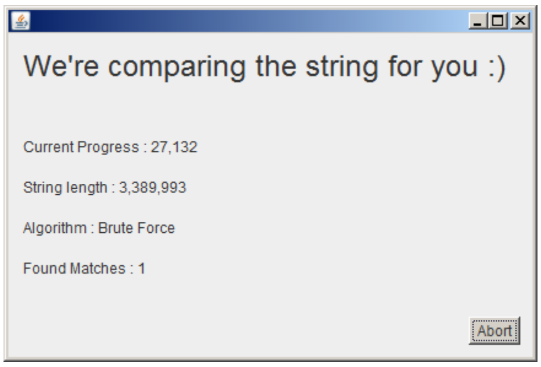
```
TGATGCAACAAAATTTGCGAAAAAACAACCCTCTACTCGTTTTTTTATTACTTTTTTTGTTCACATATT
GCTTAGCTTTTAAAGGGGGCCTTATCAACAATACGATAATCCTTTTAAAAGGCATTCTGTACTAAAAA
AACCAATAATTTAACTCAACGAATGACTCTAACTGCGCAATCTGTATGGGATAATTGTCTCTCTTTTA
AAGGACAATATTACACCCCAGGCATACAAGACGTGGTTTGAACCGATTAAAGCGGTCAAGCTCACA
ACGCATTAAGCATACAAGTGCCTAGTAAGTTCTTTTATGAATGGCTAGAAGAGCACTATGTTAAACTT
TAAAGTTGCGCTTAACAAAGTACTAGGTGAAAATGCCAAACTGGTTTATGTAATCAGAATGGAAAACA
TACGGAAATAATCAACCGTTTACTGAAAAGATTCCTAGTGCAAATCGTAGCACTATGAAATCTCAAGA
TTGATGCTCCATTTAAAAACAAGAATCCAGAGCTCAAAAATCCTTTTGTAATTCCTGGTATCAGAAAT
ACAAATTGAATCACAGCTCAATCCTAACTATAACTTTGACAACTTCCTTGAAGGCGACTCTAACCGT
```

Pattern GCAATAATTTGATATCCTATCACGATGGAAGCTACCTTAAAAAA

Brute Force

Picture 3.2 – Simulator all set and ready to go

After we load all the data to the simulator, run the simulator and wait for the process to be done.



We're comparing the string for you :)

Current Progress : 27,132

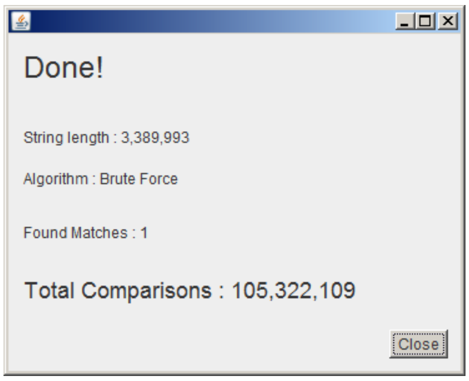
String length : 3,389,993

Algorithm : Brute Force

Found Matches : 1

Picture 3.3 – String matching simulator is working on the DNA

After the process done, we can get how many comparisons done by the algorithm in the process of finding the pattern in the string (in this case in the DNA).



Done!

String length : 3,389,993

Algorithm : Brute Force

Found Matches : 1

Total Comparisons : 105,322,109

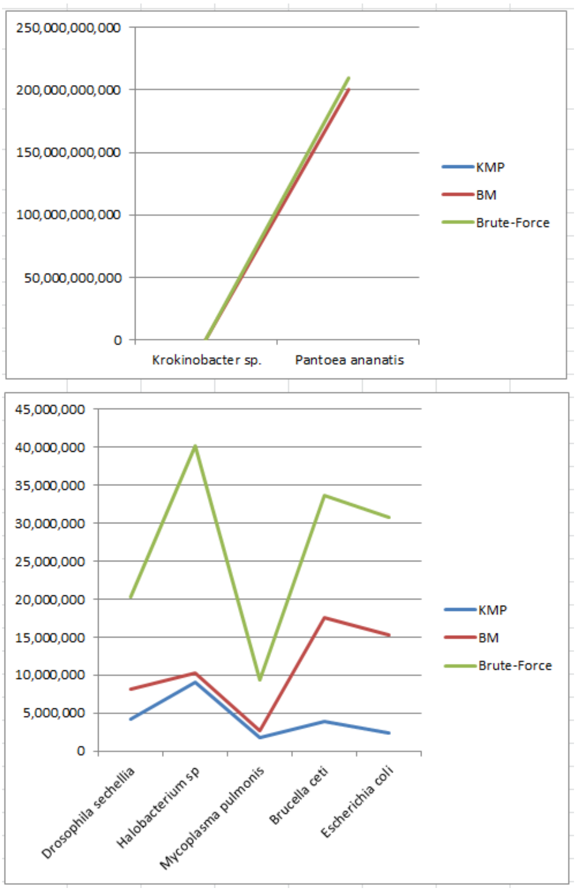
Picture 3.4 – The simulation result

By doing the process to several DNA and Algorithms we may conclude (or prove the theories) the best algorithm to be used in DNA Sequence matching. The following Table 3.2 is the result of several Bacteria's DNA Matching simulation using three string matching algorithm.

DNA	KMP	BM	Brute-Force
Krokinobacter sp.	23,604,303	83,198,247	105,322,109
Pantoea ananatis	61,215,826	200,112,431,564	210,000,341,554
Drosophila sechella	4,167,525	8,137,723	20,332,624
Halobacterium sp	9,113,351	10,281,157	40,132,553
Mycoplasma pulmonis	1,820,115	2,701,458	9,412,765
Brucella ceti	3,912,364	17,545,257	33,635,731
Escherichia coli	2,425,754	15,325,653	30,746,222

Table 3.2 – Comparison of three algorithms in DNA Matching

To make our analysis easier, we may form charts from the data in Table 3.2. Because the first two data contains a very high value, we'll separate those two into different chart with the rest of the data. The charts formed by those data are presented in the following Picture 3.5.



Picture 3.5 – Result of simulation in Chart View

From the comparison charts, we can look that the KMP is always below the other algorithm (that means do less comparison significantly rather than other two string matching algorithm). These data also prove our *Theoretical Analysis* before this sub chapter. So, we may conclude that KMP is the best algorithm to be used in DNA Sequence Matching process.

IV. Conclusion

From the previous chapter (the Analysis chapter), in the Theoretical Analysis, we get that KMP Algorithm's time complexity to analyze DNA Sequences is much smaller than Boyer-Moore or Brute-Force String Matching Algorithm with the same condition. By using the time complexity, we conclude that KMP is theoretically the best solution for DNA Sequence Matching process.

In the Practical Analysis, we have done some experiments and simulations that also prove the previous theoretical analysis. The result of the Practical Analysis by using DNA Samples showed that KMP done less comparisons process rather than two other algorithms.

So, we conclude that KMP is the best algorithm to be used in DNA Sequence Matching rather than other two algorithms (which are Boyer-Moore and Brute Force Algorithm).

V. Acknowledgment

Andre Susanto, as the author of this paper, want to express his deepest gratitude to Dr. Ir. Rinaldi Munir, M.T. and Dr. Masayu Leylia Khodra, ST., MT. as the lecturers of IF 2211 – "Strategi Algoritma". Special thanks to all of my family, my friends in Informatics 2012, and other people that give any form of support to me to finish this paper.

References

[1] A. Berry, J. D. Watson, *DNA: The Secret of Life*, New York: Arrow Books; 2004, pp. 25–44.

[2] Champoux J (2001). "DNA topoisomerases: structure, function, and mechanism". Annu Rev Biochem 70: 369–413.

[3] Basu H, Feuerstein B, Zarling D, Shafer R, Marton L (1988). *Recognition of Z-RNA and Z-DNA determinants by polyamines in solution: experimental and theoretical studies*. J Biomol Struct Dyn 6 (2): 299–309.

[4] Barbara H. Partee; Alice ter Meulen; Robert E. Wall (1990). *Mathematical Methods in Linguistics*. Kluwer.

[5] Munir, Rinaldi, *Diktat Kuliah Strategi Algoritma*, 2nd ed. pp. 22–40.

[6] Davidson, Andrew, *Introduction to Pattern Matching*, 2nd ed. pp. 1–50.

[7] A Gentle Introduction to Pattern. May 10, 2014 (12:00 PM). <http://www.haskell.org/tutorial/patterns.html>

[8] The Matchematica Books. Chapter 2.3-Pattern. May 10, 2014 (12.10 PM). <http://documents.wolfram.com/mathematica/book/section-2.3>

[9] Pure pattern calculus. Cambridge Press. May 11, 2014 (07.13 AM). http://journals.cambridge.org/repo_A45US65o/

Statement

I hereby stated that this paper is copyrighted to me, neither a copy from other's paper nor a translation of similar paper.

Bandung, 17 May 2014

[illegible]

Andre Susanto
13512028