String Matching Algorithm for Auto Complete and Auto Correct

Nathania Calista Djunaedi - 13521139
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): Nathania.calista01@gmail.com

***Abstract*— Technology has become unseparatable aspects from human life. Several features are created by programmer to help human achieve their tasks as efficiently as possible. Autocomplete and Autocorrect are two features that have significant impact on human's life. These two features can be seen at many applications in our daily life, for example at Microsoft Word, Google Docs, or IDE such as Visual Studio. These features can be implemented by using string matching with 3 different algorithms, such as Bruteforce, KMP, and BM algorithm. Th**

***Keywords*— KMP, bruteforce, BM, stringmatching, autocorrect, autocomplete**

I. Introduction

These days, technology is an aspect that cannot be separated from human daily life. Technology has been used in various aspects, ranging from education, work, health, economy, and many more. It also come in various shapes, like computer, laptop, smartphone, smartwatch, and many more. Technology is used to accomplish difficult tasks as well as simple tasks that are frequently performed by humans. By using technology, humans are expected to accomplish their tasks efficiently and accurately. Examples of technologies that is commonly used to enhance efficiency and accuracy are autocorrect and autocomplete feature.

Autocorrect is a word processing feature that identifies misspelled words, and use algorithms to identify the words most likely to have been intended. Several companies, like Apple, Google, and Microsoft have used this feature in all of their products. IDE applications usually don't use autocorrect feature. When a user enter words that are not in the program's database, program will notify the user that their words are wrong. Not only that, the program will also provide several recommendations for words that are most similar to the user's input. An example of the use of autocorrect can be seen in the image below

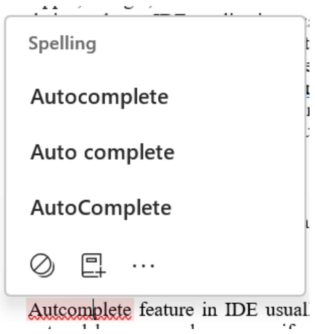
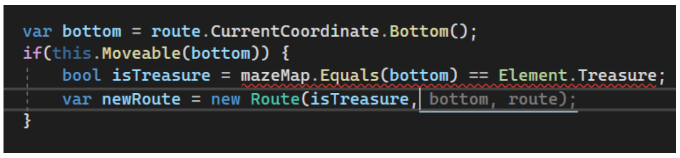


Fig 1 Example of Autocorrect Feature

Source : Author's personal documentation

Autocomplete is a feature that attempts to predict and automatically complete the current word or phrase as it is being entered by the program's user. Different with autocorrect, several IDE, such as visual studio, has this feature. Autocomplete feature in IDE usually works by detecting word entered by user and compare if any words in database have similar patterns with that the user input. If there is a match, the program will show the rest of the code or sentence at the screen and user can use the suggestion by clicking enter. An example of the use of autocomplete can be seen in the image below



```
var bottom = route.CurrentCoordinate.Bottom();
if(this.Moveable(bottom)) {
    bool isTreasure = mazeMap.Equals(bottom) == Element.Treasure;
    var newRoute = new Route(isTreasure, bottom, route);
}
```

Fig 2 Example Autocomplete Feature

Source : Author's personal documentation

In it's implementation, the autocorrect and autcomplete features often lack accuracy and experience frequent failures. Therefore, the author would like to discuss the impleentation of string matching algorithms in both of these features.

II. Theory and Concepts

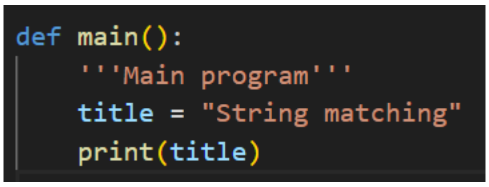
A. *Autocorrect and Autocomplete*

Autocorrect is a word processing feature that identifies misspelled words, and uses algorithms to identify the words most likely to have been intended. When autocorrect was first found, in 1990, program doesn't compare the user's input to any dictionary. Instead, the program compares it to a preprogrammed table of everyday mistakes and their replacement. In the years that followed, autocorrect has become more sophisticated and has the ability to compare user's input to a dictionary based on the language used. These days, people can feel the implementation of autocorrect in almost all applications

Autocomplete, known as word completion, is a software feature that suggests finishing what is being typed by comparing the current text with previously-entered text. Autocomplete was first introduced by Google and was aimed to assist users in typing search queries based on previous query. Later on, this feature is also used at several IDE, like visual studio.

B. *String*

A string is one of the data types in programming that is usually represented as an array of characters capable of storing letters, numbers, punctuation marks, or other valid characters. A string can be formed using static allocation or dynamic allocation methods. One of the factors that determines these methods is the programming language used. In static allocation, the length of a string is declared from the beginning, while in dynamic allocation, the length of a string can vary depending on its usage. Example on how to declare a string in python can be seen on the image below



```python
def main():
    '''Main program'''
    title = "String matching"
    print(title)
```

Fig 3 String in Python

Source : Author's personal documentation

Some important terms related with string matching algorithms are :

1. Prefix

   A prefix refers to one or more letters at the beginning of a word that alter the original meaning of a word. If m is a length of a string (S), then, prefix must be located between S[0…m-1] and must contain the first letter of the word. For example, at the picture above, the prefix of title is "Str", "St", and many more. Meanhwile, "ring" is not a prefix of title.

2. Suffix

   A suffix refers to one or more letters at the end of a word that alter the original meaning of a word. If m is a length of a string (S), then, suffix must be located between S[0..m-1] and must contain the last letter of the word. For example, at the picture above, the suffix of title is "ching", "ng", "matching", and many more. Meanwhile, "Str" is not a suffix of title.

C. *String Matching*

String matching algorithm is an algorithm used to find occurrences of a short string within a long string. In string matching algorithms, the longer string is commonly referred as the "text", while the shorter string is referred to as the "pattern".

One example of an implementation of string matching is in a chatbot, where the program has a database that contains pre-entered information and receive an input from user. In a chatbot, informations in database is considered as the "text", and the user's input is considered as the "pattern". When receiving input from the user, the program will perform string matching algorithm with the entire data in the database until it finds a match.

Generally, there are 3 main algorithms for performing string matching :

1. Brute Force Algorithm

   String matching can be done using the brute force algorithm, however, this algorithm has a very high time complexity. The process of string matching with the brute force algorithm involves the following steps:

   a. Compare the first character of the text with the first character of the pattern
   b. If the first character of the text matches the first character of the pattern, continue the comparison to the next indices of both the text and the pattern
   c. If the character in the text is not the same as the character in the pattern, compare the next index of the text with the first index of the pattern (start checking from the beginning of the pattern)
   d. Repeat this process until iteratively until either the entire text has been traversed or the pattern has been found within the text

   Example of how brute force algorithm work to do string matching can be seen at Figure 4

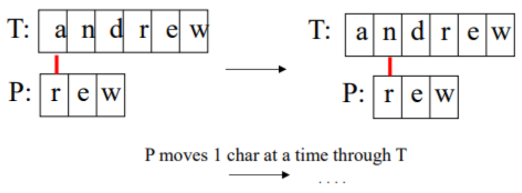


Figure 4 Brute Force Algorithm
Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf .Accessed

The average time complexity for this algorithm is O(mn), where m is the length of the text and n is the length of the pattern.

2. Knuth-Morris-Pratt Algorithm (KMP)
KMP algorithm is one of the most efficient algorithm for string matching. It has lower time complexity compared to brute force algorithm because the shifting of the pattern is done in more efficient way. The process of string matching with KMP algorithm involves the following steps :
   A. Compare the first character of the text with the first character of the pattern
   B. If the first character of the text matches the first character of the pattern, continue the comparison to the next indices of both the text and the pattern
   C. If the character in the text is not the same as the character in the pattern(P) at P[j], then, find the largest prefix of P[0..j-1] that is a suffix of P[1..j-1]
   D. Repeat this process until iteratively until either the entire text has been traversed or the pattern has been found within the text

Example of KMP algorithm can be seen in the figure below

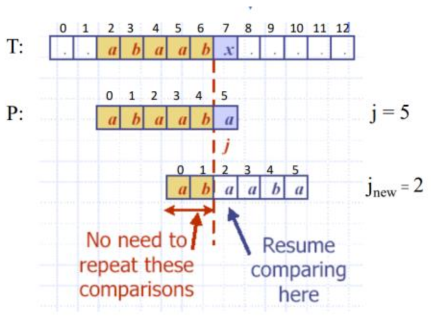
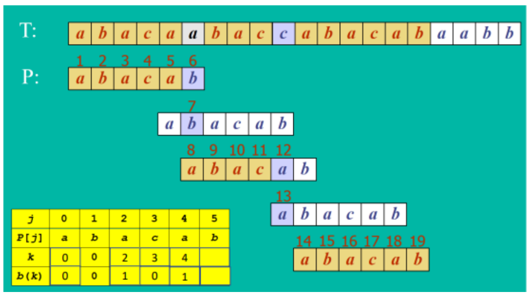


Figure 5 KMP algorithm
Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf .Accessed

To determine how far a pattern should be shifted, KMP algorithm has a border function. Border function b(k) is defined as the size of the large prefix of P[0..k] that is also a suffix of P[1..k]. Where k is the position before the mismatch occurred. Examle of how KMP algorithm works with border function can be seen at figure 5



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

Figure 6 KMP Algorithm with Border Function

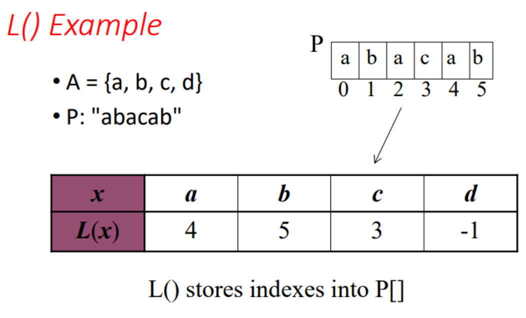
Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf .Accessed

By representing the length of the pattern as m and the length of the text as n, the complexity of the brute force algorithm is as follows :

- Border function: O(m)
- String search: O(n)
- Total complexity: O(m+n)

3. Boyer-Moore Algorithm (BM)
BM algorithm is also one of the most efficient algorithm for string matching. It has lower time complexity compared to brute force algorithm because the shifting of the pattern is done in more efficient way than brute force. By representing the pattern as P and the text as T, BM algorithm is based on 2 main techniques:
   A. The Looking Glass Technique
   Find P in T by moving backwards through P, starting at it’s end.
   B. The Character Jump Technique
   When a mismatch occurs at T[i], for example, the character at T[i] is x, there are 3 possible cases, tried in order :
   - P contains x somewhere
   When this case occurs, try to shift P right to align the last occurrence of x in P with T[i]
   - P contains x somewhere, but a shift right to the last occurrence is not possible
   When this case occurs, shift P right by 1 character to T[i+1].
   - P doesn’t contain x
   Shift P to align P[0] with T[i+1]

To search for last occurrence of a character, BM has a function called last occurrence function also called as L function. This function maps all the letters in P into an integers. By representing x as a letter in P, L(x) is defined as the largest index i such that P[i] equals to x or P[i] equals to -1 if it doesn’t exist. Example for last occurrence function can be seen as figure 6 below

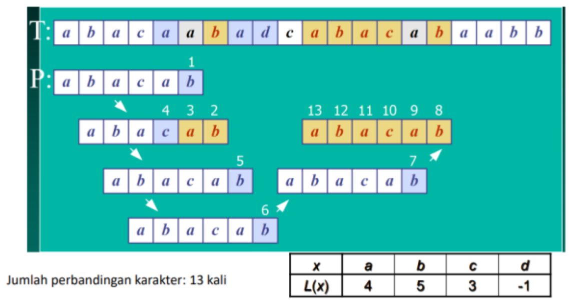


x	a	b	c	d
L(x)	4	5	3	-1

Figure 7 Last Occurrence Function

Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf .Accessed

Example for Boyer Moore algorithm in string matching can be seen at figure 7


T: a b a c a a b a d c a b a c a b a a b b
P: a b a c a b
1
4 3 2
13 12 11 10 9 8
5
7
6
Jumlah perbandingan karakter: 13 kali

x	a	b	c	d
L(x)	4	5	3	-1

Figure 8 BM Algorithm

Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf .Accessed

The worst case time complexity for BM algorithm is O(m+n), where m is the length of a pattern and n is the length of a text.

III. String Matching implementation

In order to create a simple and efficient auto-correct and auto-complete program, author utilized the brute force, KMP and BM algorithms. The author used these three algorithms to compare their efficiency in implementing the auto-complete and auto-correct features. However, it is important to mention that in real life, auto complete and auto correct features can't be created only by string matching algorithms. Most likely, these features will also require a machine learning and more algorithms.

A. Testing mechanism for Auto Correct Feature

In order to test the effectiveness of brute force algorithm, KMP algorithm, and BM algorithm, author creates a program and use database that contains English dictionary. This program is implemented using Python, the algorithm specifications are as follows :

1. Program can access database that has already prepared and contains English words .
2. Program can receive user input .
3. If the user's inputs are not stored in database, the program will consider it as a mismatch. Moreover, the user will be given the most similar word that is stored inside database.
4. If the user's inputs are stored in database, the program will consider it as a match and display a success message.

In this program, author use Kaggle's datasets that contain approximately 334000 English words.

(source : https://www.kaggle.com/datasets/rtatman/english-word-frequency)

B. Testing Mechanism for Auto Complete Feature

In order to test the effectiveness of brute force algorithm, KMP algorithm, and BM algorithm, author creates a program and use database that contains Starbucks location around the world. This program is implemented using Python, the algorithm specifications are as follows:

1. Program can access database that has already prepared and contains several information related with Starbucks's locations around the world.
2. Program can receive user input that is not fully completed.
3. Program do string matching algorithms between the user input and the database by using three algorithms (brute force, KMP, and BM algorithms)
4. If programs find a match between string and database, program will display the output. By doing this, the program can detect and predict what is the next letters that will be typed by the user.
5. If the program doesn't find any match, program won't display anything

In this program, author use Kaggle's datasets that contain approximately 25.700 locations.

(source : https://www.kaggle.com/datasets/starbucks/store-locations)

C. String Mathcing Algorithm's Implementation

Since there are 3 main algorithms for string matching, the author implements 3 algorithms by using Python. Implementation of each algorithm can be seen below :

- Brute Force Algorithm

```
'''Brute Force File'''
def brute_force(text, pattern):
    '''Brute Force Algorithm'''
    len_pattern = len(pattern)
    len_text = len(text)
    if len_pattern > len_text:
        return 0
    elif len_pattern == len_text:
        if text == pattern:
            return 1
        else:
            return 0
    for i in range(0, len_text -
len_pattern + 1):
        j = 0
        while (j < len_pattern and
text[i+j] == pattern[j]):
            j += 1
        # If pattern is found inside a
text
        if j == len_pattern:
            return 1
    return 0
```

- KMP Algorithm

```
def KMP(str,substr,table):
    '''KMP function'''
    len_sub = len(substr)
    len_str = len(str)
    if len_sub>len_str :
      return 0
    elif len_sub == len_str:
        if(str == substr):
            return 1
        else:
            return 0
    i,j = 0,0
    while(i < len_str):
        if str[i] == substr[j]:
            i += 1
            j += 1
        elif j == 0:
            i += 1
        else :
            j = table[j-1]
            if len_sub == j:
                return 1
    return 0
def border_function(substr):
    '''Border function for KMP (used
when a mismatch occurs)'''
    table = [0 for i in range (len
            (substr)+1)]
    table[0] = 0
    j = 0
    i = 1
    while(i < len(substr)):
        if substr[i] == substr[j]:
            table[i] = j + 1
            i += 1
            j += 1
        elif j == 0:
     table[i]= 0
```

```
            i += 1
        else :
            j = table[j-1]
    return table
```

- BM Algorithm

```
def last_occurence(substr):
    table = [0 for i in range(257)]
    for i in range(0, 257):
        table[i] = -1
        for i in range(0,len(substr)):
            if(ord(substr[i]) < 257):
             table[ord(substr[i])] = i
    return table

def BM(string,substr,table):
    len_sub = len(substr)
    len_str = len(string)
    i = 0
    if len_sub > len_str:
        return 0
    elif len_sub == len_str:
        if(string == substr):
            return 1
        else:
            return 0
    for i in range(-1,len_str-len_sub):
        j = len_sub-1
        while(j >= 0 and string[i+1] ==
substr[j]):
            j -= 1
        if j < 0:
            return 1
        if(ord(string[i+j]) < 257):
            slide = j -
table[ord(string[i+j])]
            if slide < 1:
                slide = 1
            i += slide
        else:
            return 0
    return 0
```

D. Main Program for Auto Correct

```
'''
    AutoCorrect.py
'''
def main():
    '''Main program untuk run dan read
file'''
    english_words = []
    with open("unigram_freq.csv","r") as
file :
        csv_file = csv.reader(file)
        for lines in csv_file :

english_words.append(lines[0])

    n = input("Enter your pattern : ")
    pattern = n.split()
    not_found = []
suggestions = []
    is_found = False
    start_time = time.time()
    # Brute force algorithm
    for item in pattern :
        temp = ""
        minimum_distance =
distance(item,english_words[0])
        for word in english_words:
            if(brute_force(word,item) ==
1):
                is_found = True
                break
            else :
                is_found = False
                if(distance(item,word) <
minimum_distance):
                    minimum_distance =
distance(item,word)
                    temp = word

        if(not is_found):
            not_found.append(item)
            suggestions.append(temp)
    kmp_found = False
    kmp_missmatch = []
```

```
    kmp_suggestions = []
    start_time = time.time()
    for item in pattern :
        table = border_function(item)
        temp = ""
        minimum_distance = distance(item,
english_words[0])
        for word in english_words:
            if(KMP(word,item,table) ==
1):
                kmp_found = True
                break
            else :
                kmp_found = False
                if(distance(item,word) <
minimum_distance):
                    minimum_distance =
distance(item,word)
                    temp = word
        if(not kmp_found):
            kmp_missmatch.append(item)
            kmp_suggestions.append(temp)
bm_found = False
    bm_missmatch = []
    bm_suggestions = []
    start_time = time.time()
    for item in pattern :
        table = last_occurence(item)
        temp = ""
        minimum_distance =
distance(item,english_words[0])
        for word in english_words:
            if(BM(word,item,table) == 1):
                bm_found = True
                break
            else :
                bm_found = False
                if(distance(item,word) <
minimum_distance):
                    minimum_distance =
distance(item,word)
                    temp = word
        if(not bm_found):
            bm_missmatch.append(item)
            bm_suggestions.append(temp)
```

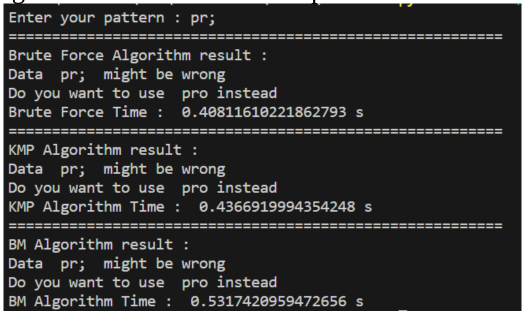
E. Main Program for Auto Complete

```
"""File for auto complete feature"""
def auto_complete():
    """Main function for auto complete
feature"""
    starbucks_location = []
    with
open("directory.csv","r",encoding="utf-
8") as file :
        csv_file = csv.reader(file)
        for lines in csv_file:
           starbucks_location.append
(lines[4])
user_input = input("Enter a location : ")
    pattern = user_input.split()
    is_found = False
    bf_result = ""
    start_time = time.time()
    for word in pattern :
       bf_result = ""
        for text in starbucks_location :
            temp = text.split()
            for str in temp :
                if(brute_force(str,word)
== 1):
                    bf_result = text
                    is_found = True
                    break
            if(is_found):
                break
        if(is_found):
            break
    kmp_found = False
    kmp_result = ""
    start_time = time.time()
    for word in pattern :
        table = border_function(word)
        kmp_result = ""
        for text in starbucks_location :
            temp = text.split()
            for str in temp :
                if(KMP(str,word,table) ==
1):
                    kmp_result = text
                    kmp_found = True
                    break
            if(kmp_found):
                break
        if(kmp_found):
            break
    bm_found = False
    bm_result = ""
    start_time = time.time()
    for word in pattern :
        table = last_occurence(word)
        bm_result = ""
        for text in starbucks_location :
            temp = text.split()
            for str in temp :
                if(BM(str,word,table) ==
1):
                    bm_result = text
                    bm_found = True
                    break
            if(bm_found):
                 break
       if(bm_found):
            break
```

IV. EXPREMIENT

A. Experiment for Auto Correct Feature

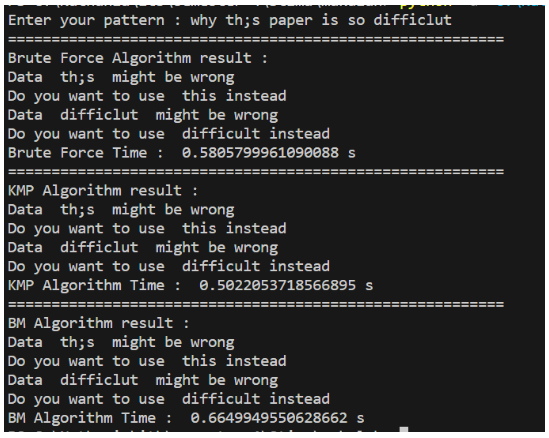
1. Test Case 1
   Program receive an incorrect input from user



```
Enter your pattern : pr;
=========================================================
Brute Force Algorithm result :
Data  pr;  might be wrong
Do you want to use  pro instead
Brute Force Time :  0.40811610221862793 s
=========================================================
KMP Algorithm result :
Data  pr;  might be wrong
Do you want to use  pro instead
KMP Algorithm Time :  0.4366919994354248 s
=========================================================
BM Algorithm result :
Data  pr;  might be wrong
Do you want to use  pro instead
BM Algorithm Time :  0.5317420959472656 s
```

Fig 9 Test Case 1

2. Test Case 2
   Program receive multiple incorrect input from user

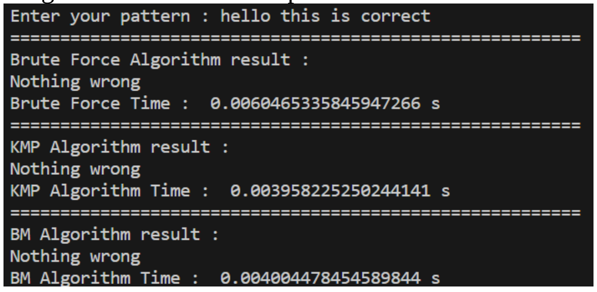


```
Enter your pattern : why th;s paper is so difficlut
=========================================================
Brute Force Algorithm result :
Data  th;s  might be wrong
Do you want to use  this instead
Data  difficlut  might be wrong
Do you want to use  difficult instead
Brute Force Time :  0.5805799961090088 s
=========================================================
KMP Algorithm result :
Data  th;s  might be wrong
Do you want to use  this instead
Data  difficlut  might be wrong
Do you want to use  difficult instead
KMP Algorithm Time :  0.5022053718566895 s
=========================================================
BM Algorithm result :
Data  th;s  might be wrong
Do you want to use  this instead
Data  difficlut  might be wrong
Do you want to use  difficult instead
BM Algorithm Time :  0.6649949550628662 s
```

Fig 10 Test Case 2

3. Test Case 3

   Program receive correct input from user



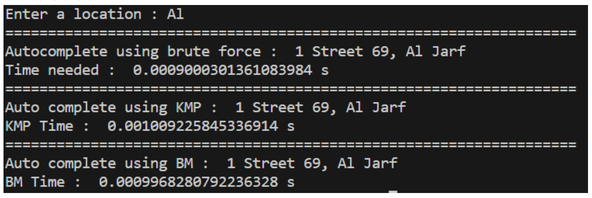
```
Enter your pattern : hello this is correct
=========================================================
Brute Force Algorithm result :
Nothing wrong
Brute Force Time :  0.0060465335845947266 s
=========================================================
KMP Algorithm result :
Nothing wrong
KMP Algorithm Time :  0.003958225250244141 s
=========================================================
BM Algorithm result :
Nothing wrong
BM Algorithm Time :  0.004004478454589844 s
```

Fig 11 Test Case 3

B. *Experiment fot Auto Complete Feature*

1. Test Case 1

   Program receives a location that has been stored in database

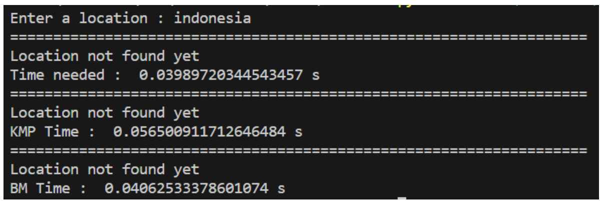


```
Enter a location : Al
=================================================================
Autocomplete using brute force :  1 Street 69, Al Jarf
Time needed :  0.0009000301361083984 s
=================================================================
Auto complete using KMP :  1 Street 69, Al Jarf
KMP Time :  0.001009225845336914 s
=================================================================
Auto complete using BM :  1 Street 69, Al Jarf
BM Time :  0.0009968280792236328 s
```

Fig 12 Test Case 1

2. Test Case 2

   Program receives a location that has not been stored in database



```
Enter a location : indonesia
=================================================================
Location not found yet
Time needed :  0.03989720344543457 s
=================================================================
Location not found yet
KMP Time :  0.05650091171264648 s
=================================================================
Location not found yet
BM Time :  0.04062533378601074 s
```

Fig 13 Test Case 2

V. ANALYSIS

Based on all the tests conducted in the previous section, the string-matching algorithm can be applied and used for simple autocorrect and autocomplete programs. From the three algorithms used in the experiments, it can be seen that the BM algorithm has a longer execution time compared to the other algorithms. This is due to the initialization of the table in the BM algorithm, which increases the required time.

Furthermore, it can be seen that the autocomplete algorithm is faster compared to the autocorrect algorithm. The main reason for result is the dataset for autocorrect is more comprehensive or larger compared to the dataset for autocomplete. Additionally, in autocomplete, the text in the database is no longer an array of strings with only one word. In autocomplete, the text in the database is an array of strings that can have multiple letters or words, for example, "Starbucks Chicago". Therefore, during the matching process, if it is found that the first sentence of the pattern is the same as the first sentence of the text at a certain index, the search will be immediately stopped and continued to the next index.

VI. CONCLUSION

Based on all the theories and experiments conducted, it can be concluded that string matching can be used to create a simple autocorrect and autocomplete feature. The most efficient algorithm for string matching depends on the given case (pattern) and the database (text). String matching algorithms are highly suitable for beginner programmers who want to learn basic auto correct and auto complete.

The autocorrect and autocomplete programs that have already been developed by author can still be further optimized by using machine learning and additional algorithms. High-accuracy autocorrect and autocomplete programs that are also equipped with fast execution time can greatly help the lives of the community, especially programmers or students.

a.

VIDEO LINK AT YOUTUBE

To educate more people, especially students, about string matching algorithms, author has published a video. Video can be seen at https://www.youtube.com/watch?v=USG3X0GGJ5g

ACKNOWLEDGMENT

First and foremost, I would like to express my gratitude to God who has given me strength to finish this paper. Furthermore, I would also like to express my gratitude to the professors of Algorithm Strategy class, Mr. Rinaldi Munir, Mr. Rila Mandala, and Mrs. Nur Ulfa, for their guidance throughout the fourth semester and for providing materials related to algorithm strategy.

REFERENCES

[1] Boyer, Robert S.; Moore, J Strother (October 1977). "A Fast String Searching Algorithm". Comm. ACM. New York: Association for Computing Machinery. 20 (10): 762–772

[2] I Knuth, Donald; Morris, James H.; Pratt, Vaughan (1977). "Fast pattern matching in strings". SIAM Journal on Computing. 6 (2): 323–350.

[3] http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata. Accessed on 21 May 2023

[4] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf .Accessed on 21 May 2023

[5] https://press.rebus.community/programmingfundamentals/chapter/string-data-type/. Accessed on 21 May 2023

STATEMENT

I hereby declare that the paper I have written is my own work and is not a summary or translation or someone else's paper, and it is not plagiarized.

Bandung, 22 May 2023

[illegible]

Nathania Calista Djunaedi
13521139