Where is Waldo and Can We Find Him using String Matching Algorithm?

Julian Caleb Simandjuntak - 13522099
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522099@std.stei.itb.ac.id

***Abstract*—Where's Waldo (Where's Wally) is a puzzle where the reader has to look for a character named Waldo with his unique face and clothes in the puzzle image. There are several ways to solve this puzzle, one of which is by using a string matching algorithm, which utilizes image processing techniques, conversion to binary, and an algorithm to look for patterns in the form of samples of Waldo's face in binary in the puzzle as the text.**

Keywords—Waldo, String Matching, Knuth–Morris–Pratt Algorithm, Boyer-Moore Algorithm

I. Introduction - Who is Waldo?

Where's Wally? is a British series of children's puzzle books created by British illustrator Martin Handford. The books consist of a series of detailed illustrations spread over two pages depicting a dozen or more people doing various funny things in a particular location. Readers are challenged to find a character named Wally and his friends hidden throughout the pages. The books are distributed all over the world and the name of Wally changes based on the country. For this paper, we use the American version, Where's Waldo. Waldo is recognizable by his red-and-white striped shirt, bobble hat, and glasses, but many of the illustrations contain red herrings involving the use of deceptive red-and-white striped objects. The next entry in the long-running book series adds another target for readers to find in each illustration.

In this paper, we try to search for Waldo using the String Matching algorithm, where we will convert the Where's Waldo puzzle into binary and then into Ascii text. Then, several sample photos from Waldo will be used to also be converted into Ascii as a pattern and searched for in Ascii text. A percentage match approach is also used, to prevent total matches not being found, so that Waldo's position in the puzzle can be determined from the part of the text that has the highest match.

Image 1.1. Waldo is here!

II. Basic Theory - How to find Waldo?

A. What is a String Matching Algorithm?

The string matching or pattern matching algorithm is an algorithm for searching for the existence of a "pattern" string in a longer "text" string. For example, a string matching algorithm can search for the word (pattern) "play" in the text "I play basketball" and return the index or which character it starts with, in this case the 3rd character, the 2nd index. The string matching algorithm is widely used in search engines, image analysis, and bioinformatics. There are 3 string matching algorithms, namely brute force, Knuth-Morris-Pratt (KMP), and Boyer Moore (BM).

B. How does it work?

In brute force (general), string matching is done by iterating the character index from left to right of the pattern and from left to right of the text. If the two characters at a particular index match, an increment is performed on both indexes. If a mismatch occurs at the pattern index which is not 0, the pattern index will return to index 0, but if a mismatch occurs at pattern index 0, an increment is made to the text index. The brute force algorithm does not have special handling if a mismatch occurs so it is the slowest algorithm of all string matching algorithms. The brute force algorithm has a complexity of O(mn).

```
Teks: NOBODY NOTICED HIM
Pattern: NOT

  NOBODY NOTICED HIM
1 NOT
2  NOT
3   NOT
4    NOT
5     NOT
6      NOT
7       NOT
8        NOT
```

Image 2.2.1. Brute force string matching algorithm

The Knuth-Morris-Pratt or KMP algorithm performs index iteration from left to right of the pattern and from left to right of the text. Systematic search if both characters at a certain index match using brute force. However, KMP utilizes an array called a border array (also called a failure array), where the array has the length of the pattern minus 1, which contains the largest prefix of border[0..j-1] that is a suffix of border[1..j-1]. During string matching, if a mismatch occurs, you will see the value of the border at index j-1 where j is the pattern index that does not match a particular index in the text. Then, this value will become the new pattern index. If there is a mismatch at the zero pattern index, an increment will be made to the text index. The KMP algorithm is much faster than brute force with an algorithm complexity of O(m+n).

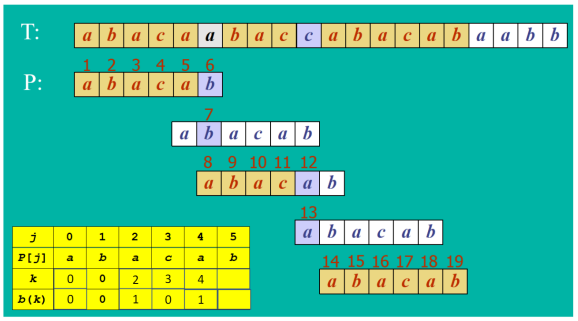

Image 2.2.2. KMP string matching algorithm

The Boyer Moore or BM algorithm is a string matching algorithm with index iteration from left to right of the text, but from right to left patterns, different from previous algorithms. If the two characters match, an index increment is performed on the text and an index decrement is performed on the pattern. The BM algorithm utilizes the last occurrence array, which is an array with the size of the number of unique characters in the text, with each element in the array representing the last index of a character appearing in the pattern or -1 if it does not appear in the pattern. If a mismatch occurs, treatment will be carried out based on 3 cases. The first case, if character c at index i text does not match the character at index j pattern, but is to the left of character index j, index j will jump to the last index where character c is located before index j using the last occurrence array.

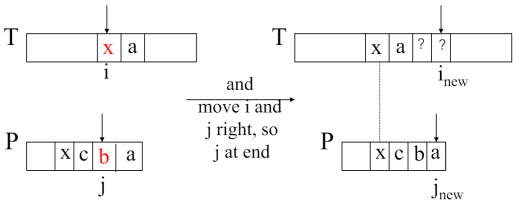

Image 2.2.3. Case 1 of mismatch in BM algorithm

The second case is if character c is to the right of character index j, an increment is carried out at index j (j+1).

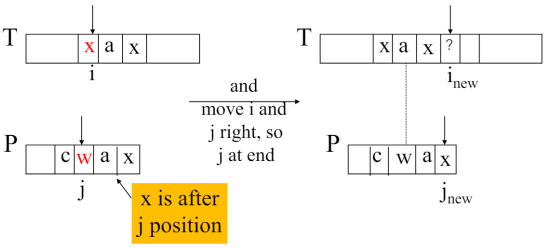

Image 2.2.4. Case 2 of mismatch in BM algorithm

The final case is if the character c is not in the pattern, an increment is made at index i (text) and resets index j to be the last index in the pattern.

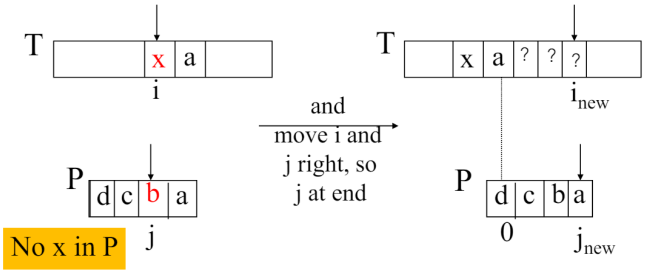

Image 2.2.5. Case 3 of mismatch in BM algorithm

The BM algorithm is also faster than brute force, with an algorithm complexity of O(nm+A).

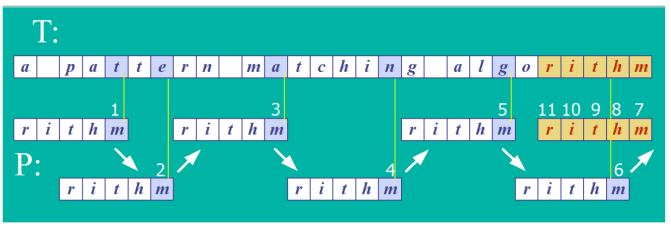

Image 2.2.6. BM string matching algorithm

C. *How to use it to find Waldo?*

Before going into the string matching method, there are several ways to solve the Where's Waldo puzzle:

1. Search for it manually using our eyes.
2. Using machine learning or artificial intelligence.
   Nowadays, technology has become increasingly advanced. Humans can utilize ML or AI that is trained using some Where's Waldo puzzle sample material, with its iconic clothes and faces, which can then be used to solve other Where's Waldo puzzles.
3. Using similarity percentage.
   As mentioned in the previous point, Waldo's face and clothes are quite iconic, so if we search the puzzle based on the color or shape of Waldo's face or clothes, we can get various percentages for several parts of the puzzle. Then, we can take the piece that has the highest percentage as the piece in the puzzle where Waldo is there.
4. Using string matching.
   By using a string matching algorithm, we can find Waldo's position in the image. However, based on the algorithm, text and patterns are needed. Of course, when working on a Where's Waldo puzzle, the puzzle itself can be text, however, there is nothing that can be used as a basis for the pattern. For this paper, a limitation is given, namely that there is a pattern in the form of Waldo's face contained in a puzzle in a certain size. So, the program does not look for Waldo blindly, but looks for the position of Waldo's face which is a pattern in the puzzle which is text. Details of the implementation strategy for the string matching algorithm are explained in the next chapter.

III. Implementation - Time to Find Waldo!

The implementation chapter will include the limitations that will be used and an explanation of the concept and flow of the program along with its implementation using the Python programming language (on CLI) with the OS, time and pillow libraries.

A. *Limitations*

The following are the limitations used in program implementation

1. As explained in the previous chapter, a kind of database will be used in the form of a collection of Waldo faces found in a particular Where's Waldo puzzle as a pattern. The puzzle itself will function as text. This means that if Waldo is not found in an image, there are 2 possibilities, there is no Waldo in the image or the part of the image that contains Waldo is not in the database.
2. Each Waldo face that becomes a pattern will have a bitmap format with a size of 80x80 pixels. Data collection was done by converting the original puzzle into a bitmap first and then cutting the image to the size of 80x80 pixels around Waldo's face.
3. The database will only be a local folder in the program, as well as the puzzle which is the input or test case, and the output from the program will be confirmation of whether Waldo is present, the time for the program to do string matching, and if there is Waldo in the puzzle, there will be a black box around Waldo's face in the original image.

B. *Program Flow*

The program flow is as follows:

1. The program will ask the user to enter a puzzle image that Waldo will look for in the tc folder.
2. The user enters the file name (along with its extension) into the program. If the file is not in the tc folder, the program will ask the user to re-enter it.
3. Once the file is valid, the user will enter the string matching algorithm they want to use, either brute force, KMP, or BM. If the algorithm is invalid, the program will ask the user to re-enter it.
4. The user selected puzzle image will be converted into a bitmap file.
5. From the bitmap file, it will be converted into a binary string, into the text of the algorithm.
6. Iteration is carried out for each bitmap file in the database that becomes the pattern. Each bitmap will be taken in the middle, measuring 80 x 1 pixels or horizontally, then also converted into a binary string, as a pattern.
7. String pattern matching is carried out on the text using the selected string matching algorithm.
8. If found, the time needed for the program to perform string matching will be displayed and the initial puzzle image will be displayed with a black square around Waldo's face.
9. If it is not found, it will only display the time and the words, Waldo not found.

C. *Implementation*

Following is the implementation of the program in Python:

1. Additional algorithm for cropping an image to 80x80 pixels based on certain x and y.

```python
# Crop the image at a specific location to a size of 80x80 pixels.
def crop_image_to_80x80(input_image_path, output_image_path):
    image = Image.open(input_image_path)

    if image.size == (80, 80):
        print("Image is already 80x80 pixels.")
        return

    width, height = image.size
    left = 60
    top = 500
    right = left + 80
    bottom = top + 80
    cropped_image = image.crop((left, top, right, bottom))

    cropped_image.save(output_image_path)
    print("Image cropped successfully to 80x80 pixels.")
```

Image 3.3.1. Brute Force string matching algorithm implementation in Python

2. String matching algorithm

a. Brute Force

```python
# Brute Force Algorithm
# The Brute Force algorithm searches for a pattern in the text from left to right
# without any special handling, so if there is a mismatch, it only advances
# the index in the text and resets the index in the pattern.
def brute_force_search(pattern, text):
    n = len(text)
    m = len(pattern)

    # Iterate through the text
    for i in range(n - m + 1):
        # Check if the pattern matches the substring starting at index i
        match = True
        for j in range(m):
            if text[i + j] != pattern[j]:
                match = False
                break

        # If a match is found
        if match:
            return i

    # If no match is found
    return -1
```

Image 3.3.2.1. Brute Force string matching algorithm implementation in Python

b. KMP

```python
# Knuth-Morris-Pratt (KMP) Algorithm
# The KMP algorithm is a method for searching a pattern within a text
# from left to right but more efficiently than brute force by using
# the border array (Border Function) which helps determine the next
# matching position when a mismatch occurs.

# Function to create the border array
# The border array is an array that for each position in the pattern,
# stores the longest prefix which is also a suffix.
# Also known as the border (Longest Prefix Suffix) or failure function.
# Example: for the pattern ['a', 'b', 'a', 'a', 'b', 'a'], its border is
# [0, 0, 1, 1, 2].
def border_function(pattern: str):
    # Initialize variables
    m = len(pattern) - 1
    border = [0] * m
    length = 0
    i = 1

    # The first element of border is always 0
    border[0] = 0

    # Iterate through each element in the pattern, starting from the second element
    while i < m:
        # If there's a match, set border[i] to length
        # (length of the prefix that matches the suffix up to character index i)
        if pattern[i] == pattern[length]:
            length += 1
            border[i] = length
            i += 1
        # If there's no match, check the length
        else:
            # If there was a previous match, set length to the value of the previous border
            if length != 0:
                length = border[length - 1]
            # If there was no previous match, set it to 0
            else:
                border[i] = 0
                i += 1

    # Return the border array
    return border
```

Image 3.3.2.2.1. Border Function in Python

```python
# KMP Algorithm
# Function to execute the KMP algorithm using the border array
# created previously to improve efficiency in case of mismatches.
def KMP_search(pattern: str, text: str):
    # Initialize variables
    m = len(pattern)
    n = len(text)
    found = False

    # Create the border array
    border = border_function(pattern)

    # Iterate through indices
    i = 0
    j = 0

    # While the text is not exhausted,
    while i < n and not found:
        # If characters match, advance both indices
        if pattern[j] == text[i]:
            j += 1
            i += 1

        # If the end of the pattern is reached, the pattern is found in the text, break out of the loop
        if j == m:
            found = True

        # Otherwise, check if the text is not exhausted and if characters do not match.
        elif i < n and pattern[j] != text[i]:
            # If the index for the pattern j is not 0, shift the pattern index according to
            # the value at index j - 1 in the border, allowing partial matches
            if j != 0:
                j = border[j - 1]
            # If j = 0, iterate i
            else:
                i += 1

    if i == n and not found:
        return -1

    return i
```

Image 3.3.2.2.2. KMP string matching algorithm implementation in Python

c. BM

```python
# Boyer-Moore Algorithm
# The Boyer-Moore algorithm is a string matching algorithm that searches a
# text from left to right, but matching starts from the right end of the pattern
# and moves to the left. The Boyer-Moore algorithm utilizes two techniques:
# the looking-glass technique and the character-jump technique, which will be
# discussed in the program. The Boyer-Moore algorithm also uses the Last
# Occurrence Function which returns an array containing the last index of
# each character (ASCII) appearing in the pattern.

# Function to return an array containing the last index
# of each ASCII character appearing in the pattern.
# The array will have a length of 256, each corresponding to an ASCII character,
# and if the character does not appear in the pattern, it will be filled with -1.
def last_occurrence_function(pattern):
    # 256 ASCII characters
    last = [-1] * 256
    # Fill the array with the last index of each character's occurrence
    for i in range(len(pattern)):
        last[ord(pattern[i])] = i
    return last
```

Image 3.3.2.3.1. Last Occurrence Function in Python

```python
# BMSearch Algorithm
# Function to execute the Boyer-Moore algorithm on a text
# by calling the last_occurrence_function to get the
# array of the last index for each ASCII character (-1 if not present)
# in the pattern, then applying the looking-glass technique
# and the character-jump technique.
@staticmethod
def BM_search(pattern, text):
    # Instantiate variables
    last = last_occurrence_function(pattern)
    m = len(pattern)
    n = len(text)
    found = False

    # Iterate over the index, moving from left to right in the text
    # Check from right to left in the pattern.
    i = m - 1
    j = m - 1

    # Perform the iteration
    while i < n and not found:
        # If there is a match
        if pattern[j] == text[i]:
            # If it is the first character
            if j == 0:
                found = True
            # If not, apply the looking-glass technique
            # Looking-glass technique: searching P in T by moving backward
            # from the end.
            else:
                i -= 1
                j -= 1
        # If there is a mismatch, apply the character-jump technique
        # Character-jump technique: If there is a mismatch, with T[i] = x,
        # and P[j] != x, handle one of the following 3 cases:
        # 1. If P has x to the left of j, shift P to the right to align with the last index of x.
        # 2. If P has x to the right of j, shift P to the right by 1 character to T[i+1].
        # 3. If x is not in P, shift P so that P[0] = T[i+1].
        else:
            lo = last[ord(text[i])]
            i = i + m - min(j, 1 + lo)
            j = m - 1

    # If no match is found
    if i > n - 1 and not found:
        return -1

    return i
```

Image 3.3.2.2. BM string matching algorithm implementation in Python

3. Image manipulation algorithm
   a. Convert image to bitmap format

```python
# Convert the image format to bitmap.
def image_to_bmp(input_image_path, output_image_path):
    image = Image.open(input_image_path)

    bitmap = image.convert('1')

    bitmap.save(output_image_path)
```

Image 3.3.3.1. Algorithm to convert image to bitmap

   b. Convert image to binary string

```python
# Convert the image to binary.
def image_to_binary(input_image_path):
    image = Image.open(input_image_path)

    image = image.convert("L")

    width, height = image.size

    binary_data = ""
    for y in range(height):
        for x in range(width):
            pixel = image.getpixel((x, y))
            binary_data += "1" if pixel < 128 else "0"

    return binary_data
```

Image 3.3.3.2. Algorithm to convert image to binary

   c. Crop to take out the middle of the image

```python
# Crop the image at the center to a size of 1 pixel by the width of the image.
def crop_middle_row(input_image_path, output_image_path):
    image = Image.open(input_image_path)

    width, height = image.size

    row_index = height // 2

    cropped_image = image.crop((0, row_index, width, row_index + 1))

    cropped_image.save(output_image_path)
```

Image 3.3.3.3. Algorithm for cropping the middle of an image

   d. Mark Waldo on the picture by painting a black square

```python
# Draw a square on the image around a specific index that is converted to x and y positions.
def draw_square_around_index(image_path, n):
    image = Image.open(image_path)
    width, height = image.size

    draw = ImageDraw.Draw(image)

    row = n // width
    col = n % width

    square_size = 60
    thickness = 15

    top_left_x = max(0, col - square_size)
    top_left_y = max(0, row - square_size)
    bottom_right_x = min(width - 1, col + square_size)
    bottom_right_y = min(height - 1, row + square_size)

    for i in range(thickness):
        draw.rectangle(
            [top_left_x - i, top_left_y - i, bottom_right_x + i, bottom_right_y + i],
            outline="black"
        )

    image.save('./result/image_with_square.bmp')
    image.show()
```

Image 3.3.3.4. Algorithm to show where is Waldo in the original image

4. Main program

```python
# Main function
if __name__ == "__main__":
    # Introduction
    print("----- Let me help you find Waldo! -----\n")
    print("Put the image you want to find in the tc folder!\n")

    # Input image from the 'tc' folder
    print("Which image do you want to use?")
    while True:
        filename = input(">>> ")
        filepath = os.path.join('tc', filename)

        if os.path.isfile(filepath):
            break
        else:
            print(f"File '{filename}' not found in the 'tc' folder. Please try again.")
    print()

    # Input the algorithm to be used
    print("Which algorithm do you want to use?")
    print("1. Brute Force")
    print("2. Knuth-Morris-Pratt")
    print("3. Boyer Moore")
    algorithm = int(input(">> "))
    while algorithm not in [1, 2, 3]:
        algorithm = int(input(">> "))
    print()

    # Start the timer
    start_time = time.time()

    # Update filepath
    filepath = os.path.join("./tc", filename)

    # Convert the image to BMP format
    image_converter.image_to_bmp(filepath, "./temp/converted_image.bmp")

    # Convert the image to binary format
    file_binary = image_converter.image_to_binary("./temp/converted_image.bmp")
```

Image 3.3.4.1. Main program part 1

```python
# Check each image in the database to see if there is a match
for dataname in os.listdir("./db"):
    datapath = os.path.join("./db", dataname)

    # Crop the middle row
    image_converter.crop_middle_row(datapath, "./temp/converted_db.bmp")

    # Convert the cropped image to binary as well
    data_binary = image_converter.image_to_binary("./temp/converted_db.bmp")

    # Get the length of the binary data
    data_binary_length = len(data_binary)

    # Perform string matching based on the selected algorithm
    if algorithm == 1:
        idx = bf.brute_force_search(data_binary, file_binary)
    elif algorithm == 2:
        idx = kmp.KMP_search(data_binary, file_binary)
    else:
        idx = bm.BM_search(data_binary, file_binary)

    if idx != -1:
        break

# Stop the timer
end_time = time.time()
elapsed_time = end_time - start_time
print("Elapsed time:", elapsed_time, "seconds")

if idx != -1:
    print("Waldo found!")
    image_converter.draw_square_around_index(filepath, idx)
else:
    print("Waldo not found!")
```

Image 3.3.4.1. Main program part 2

IV. Testing and Analysis - Waldo Found!

A. *Example of using the program*

Below is an example of how the program is run to find Waldo in a puzzle image (Image 4.1.1) using the KMP algorithm.

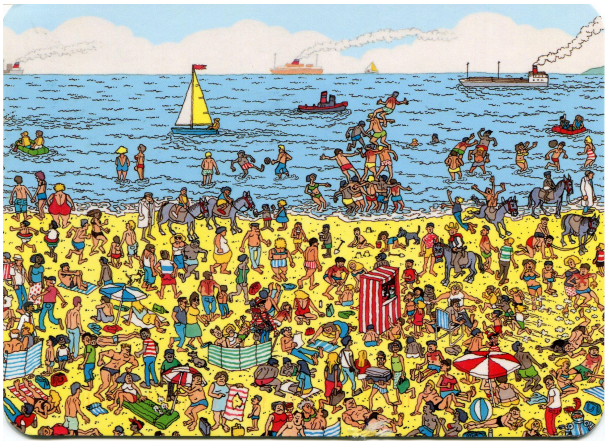

Image 4.1.1. A Where's Waldo puzzle

```
----- Let me help you find Waldo! -----

Put the image you want to find in the tc folder!

Which image do you want to use?
>>> 1.jpg

Which algorithm do you want to use?
1. Brute Force
2. Knuth-Morris-Pratt
3. Boyer Moore
>> 2

Elapsed time: 2.132661819458008 seconds
Waldo found!
```

Image 4.1.2. The program

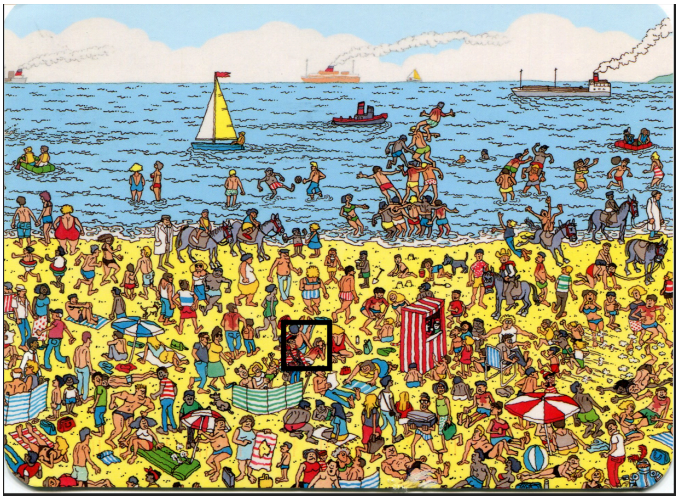

Image 4.1.2. The result

B. *Analysis of each string matching algorithm*

Below is a comparison of the three string matching algorithms with three different puzzles.

Table 4.2.1. Comparison of time needed for each string matching algorithm (in seconds)

Test case	Process time		
	Brute Force	***KMP***	***BM***
1	2.23	2.09	2.12
2	1.19	1.00	1.08
3	1.77	1.46	1.71

Based on the table above, it can be seen that the KMP algorithm is the fastest algorithm, followed by the BM algorithm and the brute force algorithm. However, the time difference between the three algorithms is not much different.

V. Conclusion - With Waldo!

The string matching algorithm can be an alternative for finding Waldo in the Where's Waldo puzzle. However, limitations are needed in the form of Waldo data samples which become patterns and specific image processing techniques so that it can be ensured that Waldo is found in puzzles that definitely contain Waldo.

For Waldo searches with string matching alone, all three algorithms are quite fast, with KMP being the fastest, followed by BM and brute force.

References

[1] Rinaldi Munir, "Pencocokan String (String/Pattern Matching)", 2024, (*https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf*)

[2] Jimmy Lion, "WHERE'S WALLY? OR WHERE'S… WALTER? WALLY'S NAME AROUND THE WORLD", 2023, (https://uk.jimmylion.com/blogs/magazine/wheres-wally#:~:text=Originally%20dubbed%20Wally%20by%20British,with%2C%20hence%20the%20minor%20tweak)

[3] "Where's Waldo?", n. d., (https://waldo.fandom.com/wiki/Where%27s_Waldo%3F)

Attachment

Github Link: Github Link

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Juni 2024

Julian Caleb Simandjuntak - 13522099