

# **LAPORAN TUGAS BESAR 3**

**IF2211 Strategi Algoritma**

**“Pemanfaatan Pattern Matching dalam Membangun Sistem Deteksi Individu Berbasis Biometrik Melalui Citra Sidik Jari”**



**Dosen:**

Ir. Rila Mandala, M.Eng., Ph.D.

Monterico Adrian, S.T., M.T.

**Oleh:**

Rizqika Mulia Pratama (13522126)

Ahmad Thoriq Saputra (13522141)

Muhammad Fatihul Irhab (13522143)

**PROGRAM STUDI TEKNIK INFORMATIKA**

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**

**INSTITUT TEKNOLOGI BANDUNG**

**SEMESTER II TAHUN 2023/2024**

## DAFTAR ISI

<b>DAFTAR ISI</b>	<b>1</b>
<b>BAB I</b>	
<b>DESKRIPSI TUGAS</b>	<b>3</b>
1.1 Skema Implementasi	3
1.2 Identifikasi Biodata	3
1.3 Penanganan Data Korup	4
1.4 Implementasi Program	4
1.5 Penggunaan Program	4
<b>BAB II</b>	
<b>LANDASAN TEORI</b>	<b>5</b>
2.1. Algoritma Knuth-Morris-Pratt (KMP)	5
2.2. Algoritma Boyer-Moore	5
2.3. Regular Expression	6
2.4. Enkripsi Caesar Cipher	6
2.5. Algoritma Levenshtein Distance	6
<b>BAB III</b>	
<b>ANALISIS PEMECAHAN MASALAH</b>	<b>8</b>
3.1 Langkah Pemecahan Masalah	8
3.1.1 Algoritma Knuth-Morris-Pratt (KMP)	8
3.1.2 Algoritma Boyer-Moore	8
3.2 Fitur Fungsional dan Arsitektur Aplikasi Desktop	9
3.3 Contoh Ilustrasi Kasus	9
3.3.1 Ilustrasi Algoritma Knuth-Morris-Pratt (KMP) :	9
3.3.2 Ilustrasi Algoritma Boyer-Moore (BM):	10
<b>BAB IV</b>	
<b>IMPLEMENTASI DAN PENGUJIAN</b>	<b>12</b>
4.1 Implementasi Program	12
4.1.1 Algoritma Knuth-Morris-Pratt (KMP)	12
4.1.2 Algoritma Boyer-Moore	14
4.1.3 Image Processing	16
4.1.4 Regular Expression	18
4.1.5 Menentukan Nilai Batas Persentase Kemiripan dalam Pencarian	21
4.2 Tata Cara Penggunaan Program	21
4.3 Hasil Pengujian	22
4.4 Analisis Algoritma	27
4.4.1 Algoritma Knuth-Morris-Pratt (KMP)	27
4.4.1.1 Kelebihan	27
4.4.1.2 Kekurangan	27
4.4.2 Algoritma Boyer Moore	27

4.4.2.1 Kelebihan	27
4.4.2.2 Kekurangan	28
<b>BAB V</b>	
<b>KESIMPULAN DAN SARAN</b>	<b>29</b>
5.1 Kesimpulan	29
5.2 Saran	29
<b>LAMPIRAN</b>	<b>30</b>
Tautan Repository : <a href="https://github.com/thoriqsaputra/Tubes3_SakedikKasep">https://github.com/thoriqsaputra/Tubes3_SakedikKasep</a>	30
Tauran Video : <a href="https://youtu.be/FkbXe3dQaAM">https://youtu.be/FkbXe3dQaAM</a>	30
<b>DAFTAR PUSTAKA</b>	<b>31</b>

# BAB I

## DESKRIPSI TUGAS

Pada tugas ini, Anda diminta untuk mengimplementasikan sebuah program yang dapat melakukan identifikasi biometrik berbasis sidik jari. Proses implementasi dilakukan dengan menggunakan algoritma Boyer-Moore dan Knuth-Morris-Pratt, sesuai dengan yang diajarkan pada materi dan salindia kuliah.

### 1.1 Skema Implementasi

Penggunaan algoritma pattern matching dalam mencocokkan sidik jari terdiri atas tiga tahapan utama dengan skema sebagai berikut:

1. **Pengambilan Citra Sidik Jari:** Gambar yang digunakan pada proses pattern matching kedua algoritma tersebut adalah gambar sidik jari penuh berukuran  $m \times n$  pixel yang diambil sebesar 30 pixel setiap kali proses pencocokan data. Anda dibebaskan untuk menentukan jumlah pixel yang diambil, asalkan didasarkan pada alasan yang masuk akal dan penanganan kasus ujung yang baik (misalnya jika ukuran citra sidik jari tidak habis dibagi dengan ukuran pixel yang dipilih). Selanjutnya, data pixel tersebut akan dikonversi menjadi binary.
2. **Konversi Data Pixel ke Binary:** Karena binary hanya memiliki variasi karakter satu atau nol, proses pattern matching akan membuat proses pencocokan karakter menjadi lambat karena harus sering mengulangi proses pencocokan pattern. Cara yang dapat dilakukan untuk mengatasi hal tersebut adalah dengan mengelompokkan setiap baris kode biner per 8 bit sehingga membentuk karakter ASCII. Karakter ASCII 8-bit ini akan mewakili proses pencocokan dengan string data.
3. **Pencocokan Sidik Jari:** Pada tahap ini, serangkaian karakter ASCII 8-bit yang merepresentasikan sebuah sidik jari akan dijadikan dasar pencarian sidik jari yang sama dengan daftar sidik jari yang terdapat pada basis data. Pencarian dilakukan dengan algoritma Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM). Jika tidak ada satupun sidik jari pada basis data yang cocok dengan sidik jari yang menjadi masukan melalui algoritma KMP ataupun BM, maka digunakan sidik jari paling mirip dengan kesamaan di atas nilai tertentu. Anda diberikan kebebasan untuk menentukan nilai batas persentase kemiripan ini, silakan melakukan pengujian untuk menentukan nilai tuning yang tepat dan jelaskan pada laporan. Metode perhitungan tingkat kemiripan juga dibebaskan kepada Anda, asalkan dijelaskan di laporan. Asisten sangat menyarankan untuk menggunakan salah satu dari algoritma Hamming Distance, Levenshtein Distance, ataupun Longest Common Subsequence (LCS).

### 1.2 Identifikasi Biodata

Fungsi dari deteksi biometrik adalah mengidentifikasi seseorang. Oleh sebab itu, pada proses implementasi program ini, sebuah citra sidik jari akan dicocokkan dengan biodata seseorang yang terdiri atas data-data pada KTP, seperti: NIK, nama, tempat/tanggal lahir, jenis kelamin, golongan darah, alamat, agama, status perkawinan, pekerjaan, dan kewarganegaraan. Relasi ini dibuat dalam sebuah basis data dengan skema detail yang telah disediakan.

### **1.3 Penanganan Data Korup**

Pada tugas besar kali ini, kita akan melakukan simulasi implementasi data korup yang hanya mungkin terjadi pada atribut nama di tabel biodata (asumsikan kolom lain pada setiap tabel tidak mengalami korup). Data korup yang dimaksud adalah bahasa alay Indonesia, dengan kombinasi variasi huruf besar-kecil, penggunaan angka, dan penyingkatan. Contoh kasus bahasa alay dijelaskan dengan detail sebagai berikut:

- Kombinasi huruf besar-kecil: bintanG DwI mArthen
- Penggunaan angka: B1nt4n6 Dw1 M4rthen
- Penyingkatan: Bntng Dw Mrthen
- Kombinasi ketiganya: b1ntN6 Dw mrthn

Cara yang dapat digunakan untuk menangani ini adalah dengan menggunakan Regular Expression (Regex). Setelah menggunakan Regex, lakukan pattern matching antara nama yang bersesuaian dengan algoritma KMP dan BM.

### **1.4 Implementasi Program**

Sistem yang dibangun akan diimplementasikan dengan basis desktop-app menggunakan bahasa C#. Masukan yang diberikan oleh pengguna adalah sebuah citra sidik jari, dengan basis data SQL yang memiliki struktur relasional yang telah dijelaskan. Tampilan layout dari aplikasi harus mencakup komponen-komponen berikut:

- Judul aplikasi
- Tombol Insert citra sidik jari, beserta display citra sidik jari yang ingin dicari
- Toggle Button untuk memilih algoritma yang ingin digunakan (KMP atau BM)
- Tombol Search untuk memulai pencarian
- Display sidik jari yang paling mirip dari basis data
- Informasi mengenai waktu eksekusi
- Informasi mengenai tingkat kemiripan sidik jari dengan gambar yang ingin dicari, dalam persentase (%)
- List biodata hasil pencarian dari basis data, dengan semua nilai atribut dari individu yang dirasa paling mirip

### **1.5 Penggunaan Program**

1. Pengguna memasukkan citra sidik jari yang ingin dicari biodatanya.
2. Pilih opsi pencarian menggunakan algoritma KMP atau BM.
3. Tekan tombol search, program kemudian akan memproses dan mencari citra sidik jari yang memiliki kemiripan dengan masukan.
4. Program akan menampilkan list biodata jika ditemukan citra sidik jari yang memiliki kemiripan dengan batas persentase tertentu, atau informasi bahwa tidak ada sidik jari yang mirip jika tidak ditemukan.
5. Terdapat informasi terkait waktu eksekusi program dan persentase kemiripan citra.

## **BAB II**

### **LANDASAN TEORI**

#### **2.1. Algoritma Knuth-Morris-Pratt (KMP)**

Algoritma Knuth-Morris-Pratt (KMP) adalah salah satu algoritma pencocokan string yang efisien untuk menemukan pola dalam teks. Algoritma ini dikembangkan oleh Donald Knuth, Vaughan Pratt, dan James H. Morris pada tahun 1977. KMP memperkenalkan pendekatan yang lebih efisien dengan memanfaatkan informasi dari pola yang sedang dicari untuk menghindari pencocokan ulang karakter yang sudah dibandingkan sebelumnya.

KMP menggunakan informasi dari pola untuk mempercepat proses pencarian. Algoritma ini menggunakan tabel yang dikenal sebagai “Longest Prefix Suffix” (LPS) atau tabel “failure action” untuk menyimpan panjang awalan terpanjang yang juga merupakan akhiran untuk setiap posisi dalam pola. Tabel LPS ini membantu menentukan berapa banyak karakter yang bisa dilompati ketika ketidakcocokan ditemukan selama proses pencocokan.

Langkah-langkah KMP:

1. Preproses Pola
  - Buat tabel LPS yang berisi panjang dari awalan terpanjang yang juga merupakan akhiran untuk setiap posisi dalam pola.
  - Tabel LPS membantu menentukan posisi berikutnya dalam pola untuk melanjutkan pencocokan setelah ketidakcocokan terjadi.
2. Pencocokan Pola dalam teks
  - Mulai dari awal teks dan pola
  - Gunakan tabel LPS untuk menghindari perbandingan ulang karakter yang sudah cocok.
  - Jika ada ketidakcocokan, gunakan nilai dari tabel LPS untuk melompat ke posisi berikutnya yang mungkin cocok dalam pola tanpa memeriksa kembali karakter yang sudah dibandingkan sebelumnya.

#### **2.2. Algoritma Boyer-Moore**

Algoritma Boyer-Moore (BM) adalah salah satu algoritma pencocokan string yang sangat efisien, terutama ketika digunakan untuk menemukan pola dalam teks yang panjang. Algoritma ini dikembangkan oleh Robert S. Boyer dan J Strother Moore pada tahun 1977. BM memperkenalkan beberapa teknik yang memungkinkannya untuk melompati bagian dari teks yang sudah diproses, sehingga mengurangi jumlah perbandingan karakter yang diperlukan dibandingkan dengan metode pencocokan string sederhana.

Algoritma Boyer-Moore (BM) yang menggunakan tabel "last" adalah salah satu cara implementasi *Bad Character Heuristic*. Tabel "last" menyimpan posisi terakhir dari setiap karakter dalam pola, yang kemudian digunakan untuk menentukan seberapa jauh pola harus digeser jika terjadi ketidakcocokan.

Langkah-langkah BM:

1. Preproses Pola

- Buat tabel last yang menyimpan posisi terakhir dari setiap karakter dalam pola.
  - Jika karakter tidak ada dalam pola, maka nilai tabel last untuk karakter tersebut adalah -1.
2. Pencocokan Pola dalam teks
- Mulai dari awal teks dan pola
  - Bandingkan karakter dari kanan ke kiri dalam pola dengan karakter dalam teks.
  - Jika ada ketidakcocokan, gunakan tabel last untuk menentukan pergeseran pola.
  - Lanjutkan proses pencocokan sampai seluruh teks telah diproses atau pola ditemukan dalam teks.

### **2.3. Regular Expression**

Regular Expression (Regex) adalah sekumpulan karakter yang membentuk pola pencarian yang digunakan untuk pencocokan teks. Regex sering digunakan dalam pemrograman untuk validasi input, pencarian teks, dan manipulasi string. Regex memberikan cara yang efisien dan fleksibel untuk menangani teks dengan pola yang kompleks.

Regex terdiri dari karakter biasa dan metakarakter. Karakter biasa mencocokkan dirinya sendiri, sedangkan metakarakter memiliki makna khusus yang memungkinkan pembentukan pola yang kompleks.

### **2.4. Enkripsi Caesar Cipher**

Caesar Cipher adalah salah satu metode enkripsi tertua dan paling sederhana yang digunakan dalam kriptografi klasik. Metode ini dinamai berdasarkan Julius Caesar, yang konon menggunakan untuk berkomunikasi secara rahasia dengan para komandannya. Caesar Cipher adalah jenis cipher substitusi di mana setiap huruf dalam teks asli digantikan oleh huruf lain yang terletak beberapa posisi di bawahnya dalam alfabet. Pergeseran ini biasanya diukur dengan jumlah tetap, yang disebut sebagai kunci.

Secara matematis, jika kita misalkan huruf-huruf dalam alfabet dilambangkan dengan A, B, C, dan seterusnya sampai Z, serta diberi indeks 0, 1, 2, dan seterusnya sampai 25, maka enkripsi Caesar Cipher dapat dirumuskan sebagai:  $E(x) = (x + n) \text{ mod } 26$ , di mana  $E(x)$  adalah hasil enkripsi dari huruf  $x$ ,  $x$  adalah indeks dari huruf dalam alfabet, dan  $n$  adalah jumlah pergeseran (kunci). Deskripsi dilakukan dengan rumus:  $D(x) = (x - n) \text{ mod } 26$ .

### **2.5. Algoritma Levenshtein Distance**

Levenshtein Distance adalah sebuah algoritma yang digunakan dalam ilmu komputer untuk mengukur perbedaan antara dua buah string. Ini juga dikenal sebagai “edit distance” dan dinamai menurut ahli matematika Rusia, Vladimir Levenshtein, yang memperkenalkan konsep ini pada tahun 1965. Algoritma ini sangat berguna dalam berbagai aplikasi seperti koreksi ejaan otomatis, pengenalan suara, dan pencocokan pola.

Levenshtein Distance mengukur jumlah operasi yang diperlukan untuk mengubah satu string menjadi string lainnya. Operasi yang diperbolehkan meliputi penyisipan satu karakter, penghapusan satu karakter, dan substitusi satu karakter dengan karakter lain. Setiap operasi memiliki bobot satu, sehingga Levenshtein Distance antara dua string adalah jumlah minimum operasi yang diperlukan untuk mentransformasi satu string menjadi string lainnya.

Secara formal, jika misalkan dua string sebagai  $s_1$  dan  $s_2$  dengan panjang  $|s_1|$  dan  $|s_2|$  masing-masing, maka Levenshtein Distance  $d(s_1, s_2)$  dapat dihitung menggunakan algoritma dinamis. Matriks dua dimensi  $d$  digunakan di mana elemen  $d[i][j]$  menyatakan Levenshtein Distance antara substring  $s_1[1..i]$  dan  $s_2[1..j]$ . Algoritma ini diinisialisasi dengan  $d[i][0] = i$  dan  $d[0][j] = j$  untuk semua  $i$  dan  $j$ , karena mengubah string kosong menjadi string dengan panjang tertentu memerlukan sejumlah operasi penyisipan yang setara dengan panjang string tersebut.

Algoritma ini kemudian mengisi matriks  $d$  dengan aturan rekursif berikut:

$$d[i][j] = \min \{ d[i - 1][j] + 1, d[i][j - 1] + 1, d[i - 1][j - 1] + cost \}$$

dengan  $cost = 0$  jika  $s_1[i] = s_2[j]$  dan  $cost = 1$  jika  $s_1[i] \neq s_2[j]$ .

## **BAB III**

### **ANALISIS PEMECAHAN MASALAH**

#### **3.1 Langkah Pemecahan Masalah**

##### **3.1.1 Algoritma Knuth-Morris-Pratt (KMP)**

Algoritma Knuth-Morris-Pratt (KMP) adalah algoritma pencocokan string yang efisien untuk menemukan sebuah substring dalam string utama. Algoritma ini bekerja dengan memanfaatkan informasi dari pola itu sendiri untuk menghindari pencocokan ulang karakter yang telah diketahui cocok atau tidak cocok. Langkah-langkah dalam KMP meliputi:

- Pengumpulan dan Persiapan Data: Mengumpulkan dataset dari sidik jari dan juga identitas
- Konversi dari gambar sidik jari ke binary: Mengubah gambar sidik jari ke format binary dengan ukuran 5 pixel kali lebar gambar tersebut (ini hanya untuk pattern)
- Konversi biner ke ASCII: Mengubah binary setiap 8 bit menjadi karakter ASCII
- Pembangunan tabel LPS: Tabel ini digunakan untuk mencatat panjang dari awalan terpanjang yang juga merupakan akhiran untuk setiap posisi dalam pola.
- Proses pencocokan: Algoritma menggunakan tabel LPS untuk menggeser pola dengan cara yang optimal ketika terjadi ketidakcocokan.
- Perhitungan kemiripan: Jika pola ditemukan maka tingkat kemiripan 100%, jika pola tidak ditemukan, maka tingkat kemiripan dihitung berdasarkan *Levenshtein Distance*.
- Penanganan data yang korup: Data yang korup terkait nama dari identitas seseorang dilakukan penanganan dengan menggunakan regex.

##### **3.1.2 Algoritma Boyer-Moore**

Algoritma Boyer-Moore merupakan salah satu algoritma pencocokan string yang paling efisien, terutama untuk teks yang panjang. Salah satu komponen penting dalam algoritma Boyer-Moore adalah penggunaan tabel last, yang dikenal juga sebagai Bad Character Heuristic. Tabel last membantu menentukan sejauh mana pola harus digeser ketika terjadi ketidakcocokan karakter. Langkah-langkah Algoritma Boyer-Moore dengan Tabel Last meliputi:

- Pengumpulan dan Persiapan Data: Mengumpulkan dataset dari sidik jari dan juga identitas
- Konversi dari gambar sidik jari ke binary: Mengubah gambar sidik jari ke format binary dengan ukuran 5 pixel kali lebar gambar tersebut (ini hanya untuk pattern)
- Konversi biner ke ASCII: Mengubah binary setiap 8 bit menjadi karakter ASCII
- Pembangunan Tabel Last: Tabel ini mencatat posisi terakhir dari setiap karakter dalam pola. Jika suatu karakter tidak muncul dalam pola, maka nilai tabel untuk karakter tersebut adalah -1.
- Algoritma memulai pencocokan dari akhir pola dan dibandingkan dengan teks dari kiri ke kanan. Jika terjadi ketidakcocokan, pola akan digeser ke kanan berdasarkan tabel last dari karakter yang tidak cocok tersebut.

- Perhitungan kemiripan: Jika pola ditemukan maka tingkat kemiripan 100%, jika pola tidak ditemukan, maka tingkat kemiripan dihitung berdasarkan *Levenshtein Distance*.
  - Penanganan data yang korup: Data yang korup terkait nama dari identitas seseorang dilakukan penanganan dengan menggunakan regex.

## 3.2 Fitur Fungsional dan Arsitektur Aplikasi Desktop

Fitur fungsional yang akan diimplementasikan meliputi:

- Konversi dari gambar ke binary
  - Konversi dari binary ke ASCII
  - Implementasi pencarian pola menggunakan algoritma KMP dan BM.
  - Penanganan data yang korup
  - Perhitungan tingkat kemiripan
  - Antarmuka pengguna untuk memasukkan gambar sidik jari yang akan dicari.
  - Tampilan hasil pencarian yang identitas dari sidik jari yang ditemukan, tingkat kemiripan, dan waktu yang dibutuhkan untuk melakukan pencarian tersebut.

## Arsitektur Aplikasi Desktop:

- Dibangun menggunakan bahasa C#
  - Menggunakan SQL untuk menyimpan data identitas dan sidik jari

### **3.3 Contoh Ilustrasi Kasus**

Diambil contoh yang hendak mencari sidik jari yang mirip dengan sebuah sidik jari dengan karakter hasil konversi ABACAB, program akan mencari citra sidik jari yang memiliki kecocokan dengan hasil konversi sidik jari yang dimasukkan pengguna dengan menggunakan algoritma KMP atau BM. Jika kemiripan tidak 100%, maka akan dicari citra sidik jari di database dengan kemiripan tertinggi, apabila kemiripan tidak mencapai 60%, maka sidik jari yang cocok tidak ditemukan.

### 3.3.1 Ilustrasi Algoritma Knuth-Morris-Pratt (KMP) :

T:		A	B	A	C	A	A	B	A	C	C	A	B	A	C	A	B	A	A	B	B
		1	2	3	4	5	6														
P:		A	B	A	C	A	B														
								7													
									A	B	A	C	A	B							

### **3.3.2 Ilustrasi Algoritma Boyer-Moore (BM):**

Ilustrasi tersebut berlaku untuk kedua jenis algoritma yaitu Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM). Kami juga melampirkan lima hasil uji coba pada program yang dapat

dilihat lebih lanjut pada subbab 4.4 mengenai hasil pengujian. Perbedaan mendasar dari kedua algoritma tersebut adalah arah pencarinya dan cara pergeserannya.

## BAB IV

### IMPLEMENTASI DAN PENGUJIAN

#### 4.1 Implementasi Program

##### 4.1.1 Algoritma Knuth-Morris-Pratt (KMP)

```
function KmpMatch(text: String, pattern: String) -> (Float,  
Integer)
```

**Kamus:**

```
n, m, i, j, bestPosition: Integer  
lps: Array of Integer  
maxSimilarity: Float  
potentialMatches: List of Integer
```

**Algoritma:**

```
n <- len(text)  
m <- len(pattern)  
  
if m = 0 or n = 0 then  
    return (0, -1)  
  
lps <- ComputeLPSArray(pattern)  
i <- 0  
j <- 0  
maxSimilarity <- 0  
bestPosition <- -1  
potentialMatches <- []  
  
while i < n do  
    if pattern[j] = text[i] then  
        j <- j + 1  
        i <- i + 1  
  
    if j = m then  
        maxSimilarity <- 1.0  
        bestPosition <- i - j  
        return (maxSimilarity, bestPosition)  
    else if i < n and pattern[j] != text[i] then  
        if j != 0 then  
            j <- lps[j - 1]  
        else  
            i <- i + 1  
  
        if j > 0 and j < m then  
            potentialMatches.append(i - j)
```

```

        for start in potentialMatches do
            end <- min(start + m, n)
            similarity <-
CalculateLevenshteinSimilarity(text[start:end], pattern)
            if similarity > maxSimilarity then
                maxSimilarity <- similarity
                bestPosition <- start

        return (maxSimilarity, bestPosition)

```

**function** ComputeLPSArray(pattern: String) -> Array of Integer

**Kamus :**

m, len, i: Integer  
lps: Array of Integer

**Algoritma:**

```

m <- len(pattern)
lps <- new Array of Integer[m]
len <- 0
i <- 1
lps[0] <- 0

while i < m do
    if pattern[i] = pattern[len] then
        len <- len + 1
        lps[i] <- len
        i <- i + 1
    else
        if len != 0 then
            len <- lps[len - 1]
        else
            lps[i] <- 0
return lps

```

**function** CalculateLevenshteinSimilarity(str1: String, str2: String) -> Float

**Kamus :**

len1, len2, i, j, cost: Integer  
dp: 2D Array of Integer

**Algoritma:**

```

len1 <- len(str1)
len2 <- len(str2)
dp <- new 2D Array of Integer[len1 + 1, len2 + 1]

```

```

for i from 0 to len1 do
    dp[i, 0] <- i

for j from 0 to len2 do
    dp[0, j] <- j

for i from 1 to len1 do
    for j from 1 to len2 do
        cost <- if str1[i - 1] = str2[j - 1] then 0 else 1
        dp[i, j] <- min(dp[i - 1, j] + 1, dp[i, j - 1] + 1,
dp[i - 1, j - 1] + cost)

maxLen <- max(len1, len2)
similarity <- 1.0 - (float(dp[len1, len2])) / maxLen
return similarity

```

#### 4.1.2 Algoritma Boyer-Moore

```
function BmMatch(text: String, pattern: String) -> Float
```

**Kamus:**

```

last: Dictionary of (Char, Integer)
n, m, i, j, startIndex, lo: Integer
maxSimilarity, similarity: Float

```

**Algoritma:**

```

last <- BuildLast(pattern)
n <- len(text)
m <- len(pattern)
i <- m - 1
maxSimilarity <- 0

if i > n - 1 then
    return -1

j <- m - 1

do:
    if pattern[j] = text[i] then
        if j = 0 then
            return 1
        else
            i <- i - 1
            j <- j - 1
    else
        lo <- if last.contains(text[i]) then

```

```

last[text[i]] else -1
    i <- i + m - min(j, 1 + lo)
    j <- m - 1

    startIndex <- i - m + 1
    if startIndex >= 0 and startIndex + m <= n then
        similarity <-
CalculateLevenshteinSimilarity(text[startIndex:startIndex +
m], pattern)
        if similarity > maxSimilarity then
            maxSimilarity <- similarity

    while i <= n - 1

    return maxSimilarity

```

**function** BuildLast(pattern: String) -> Dictionary of (Char, Integer)

**Kamus:**

```

last: Dictionary of (Char, Integer)
i: Integer

```

**Algoritma:**

```

last <- new Dictionary of (Char, Integer)

for i from 0 to len(pattern) - 1 do
    last[pattern[i]] <- i

return last

```

**function** CalculateLevenshteinSimilarity(str1: String, str2: String) -> Float

**Kamus:**

```

len1, len2, i, j, cost: Integer
dp: 2D Array of Integer

```

**Algoritma:**

```

len1 <- len(str1)
len2 <- len(str2)
dp <- new 2D Array of Integer[len1 + 1, len2 + 1]

for i from 0 to len1 do
    dp[i, 0] <- i

for j from 0 to len2 do

```

```

dp[0, j] <- j

for i from 1 to len1 do
    for j from 1 to len2 do
        cost <- if str1[i - 1] = str2[j - 1] then 0 else
1
        dp[i, j] <- min(dp[i - 1, j] + 1, dp[i, j - 1] +
1, dp[i - 1, j - 1] + cost)

    maxLen <- max(len1, len2)
    similarity <- 1.0 - (float(dp[len1, len2])) / maxLen
    return similarity

```

#### 4.1.3 Image Processing

**function** GetCenterCrop(originalImage: Bitmap, cropWidth: Integer, cropHeight: Integer) -> Bitmap

**Kamus:**

originalWidth, originalHeight: Integer  
startX, startY: Integer  
croppedImage: Bitmap

**Algoritma:**

```

originalWidth <- originalImage.Width
originalHeight <- originalImage.Height

startX <- (originalWidth - cropWidth) / 2
startY <- (originalHeight - cropHeight) / 2

startX <- max(0, startX)
startY <- max(0, startY)

cropWidth <- min(cropWidth, originalWidth)
cropHeight <- min(cropHeight, originalHeight)

croppedImage <- new Bitmap(cropWidth, cropHeight)

using Graphics g <- Graphics.FromImage(croppedImage) do:
    g.DrawImage(originalImage, new Rectangle(0, 0,
cropWidth, cropHeight), new Rectangle(startX, startY,
cropWidth, cropHeight), GraphicsUnit.Pixel)

return croppedImage

```

**function** ResizeImage(sourcePath: String, width: Integer,

```
height: Integer) -> Bitmap
```

**Kamus:**

```
    originalImage, resizedImage: Bitmap
```

**Algoritma:**

```
    using originalImage <- new Bitmap(sourcePath) do:  
        resizedImage <- new Bitmap(width, height)  
  
        using Graphics graphics <-  
Graphics.FromImage(resizedImage) do:  
            graphics.InterpolationMode <-  
InterpolationMode.HighQualityBicubic  
            graphics.SmoothingMode <-  
SmoothingMode.HighQuality  
            graphics.PixelOffsetMode <-  
PixelOffsetMode.HighQuality  
            graphics.CompositingQuality <-  
CompositingQuality.HighQuality  
  
            graphics.DrawImage(originalImage, 0, 0, width,  
height)  
  
        return resizedImage
```

```
function ConvertToAscii(original: Bitmap, threshold: Integer  
= 128) -> String
```

**Kamus:**

```
    binaryImage: Bitmap  
    binaryStringBuilder, asciiArt: StringBuilder  
    grayScale: Integer  
    originalColor, binaryColor: Color  
    pixelColor: Color  
    binaryString, byteString: String  
    asciiByte: Byte
```

**Algoritma:**

```
    binaryImage <- new Bitmap(original.Width,  
original.Height)  
  
    for y from 0 to original.Height - 1 do:  
        for x from 0 to original.Width - 1 do:  
            originalColor <- original.GetPixel(x, y)  
            grayScale <- (originalColor.R * 0.3) +  
(originalColor.G * 0.59) + (originalColor.B * 0.11)
```

```

        binaryColor <- if grayScale < threshold then
Color.Black else Color.White
        binaryImage.SetPixel(x, y, binaryColor)

        binaryStringBuilder <- new StringBuilder()
asciiArt <- new StringBuilder()

        for y from 0 to binaryImage.Height - 1 do:
            for x from 0 to binaryImage.Width - 1 do:
                pixelColor <- binaryImage.GetPixel(x, y)
                binaryStringBuilder.Append(if pixelColor.R = 0
then '0' else '1')

        binaryString <- binaryStringBuilder.ToString()

        for i from 0 to len(binaryString) - 1 step 8 do:
            if i + 8 <= len(binaryString) then:
                byteString <- binaryString.Substring(i, 8)
                asciiByte <- Convert.ToByte(byteString, 2)
                asciiArt.Append((char)asciiByte)

        return asciiArt.ToString()

```

#### 4.1.4 Regular Expression

```

function ConvertAlay(fullName: String, useNumberSymbol:
Boolean = true, useCaseMix: Boolean = true, useVowelRemoval:
Boolean = true) -> String

```

**Kamus:**

```

modifiedName: String
removalProbability: Float
vowels: Set of Char
random: Random

```

**Algoritma:**

```

modifiedName <- fullName.ToLower()

if useNumberSymbol then
    modifiedName <- new String(Select(c in modifiedName,
                                         if
replacements.contains(c.toString())
                                         then
replacements[c.toString()] [random.Next(len(replacements[c.toS
tring()]))] [0]
                                         else c).ToArray())

```

```

    if useCaseMix then
        modifiedName <- new String(Select(c in modifiedName,
                                         if random.Next(2) =
0
                                         then char.ToUpper(c)
                                         else c).ToArray())

    if useVowelRemoval then
        removalProbability <- 0.5
        vowels <- {'a', 'e', 'i', 'o', 'u'}

        modifiedName <- new String(Where(c in modifiedName,
                                         not
                                         vowels.contains(c)
                                         or
                                         random.NextDouble() > removalProbability).ToArray())

    return modifiedName

```

**function** RevertAlay(alayName: String) -> String

**Kamus:**

    regex: Regex  
     modifiedName: String

**Algoritma:**

```

    regex <- new Regex(regexPattern, RegexOptions.IgnoreCase)
    modifiedName <- regex.Replace(alayName, function(match)
                                         key <- match.Value.ToLower()
                                         if
                                         reverseReplacements.contains(key)
                                         then return
                                         reverseReplacements[key]
                                         return match.Value)

    return
    CultureInfo.CurrentCulture.TextInfo.ToTitleCase(modifiedName.
    ToLower())

```

**Static Inisialisasi**

**Kamus:**

    reverseReplacements: Dictionary of (String, String)  
     sortedKeys: List of String

```

    regexPattern: String

Algoritma:
    reverseReplacements <- new Dictionary of (String, String)

        foreach pair in replacements do
            foreach val in pair.Value do
                if not
                    reverseReplacements.contains(val.ToLower()) then
                        reverseReplacements[val.ToLower()] <-
                            pair.Key

            sortedKeys <-
        reverseReplacements.Keys.OrderByDescending(len).ToList()
        regexPattern <- String.Join("|", Select(Regex.Escape,
        sortedKeys))

```

### **Dictionary replacements**

```

"{"a", {"4", "A", "a"}}
>{"b", {"B", "b", "8", "13"}}
>{"c", {"C", "c"}}
>{"d", {"D", "d"}}
>{"e", {"3", "E", "e"}}
>{"f", {"F", "f"}}
>{"g", {"G", "g", "6", "9"}}
>{"h", {"H", "h"}}
>{"i", {"!", "1", "I", "i"}}
>{"j", {"J", "j"}}
>{"k", {"K", "k"}}
>{"l", {"L", "l", "1"}}
>{"m", {"M", "m"}}
>{"n", {"N", "n"}}
>{"o", {"O", "o", "o"}}
>{"p", {"P", "p"}}
>{"q", {"Q", "q", "9"}}
>{"r", {"R", "r", "12", "i2", "I2"}}
>{"s", {"S", "s", "5"}}
>{"t", {"T", "t", "7"}}
>{"u", {"U", "u"}}
>{"v", {"V", "v"}}
>{"w", {"W", "w"}}
>{"x", {"X", "x"}}
>{"y", {"Y", "y"}}
>{"z", {"Z", "z", "2"}}

```

#### **4.1.5 Menentukan Nilai Batas Persentase Kemiripan dalam Pencarian**

Dalam tugas besar ini, kami menetapkan batas kemiripan sebesar 60%. Kami memilih nilai ini karena sesuai dengan karakteristik dan distribusi pola sidik jari yang kami analisis. Dalam proses perbandingan, kami hanya mengekstrak 5 piksel dari kedua sidik jari. Mengingat bahwa piksel-piksel ini tidak diambil dari area yang identik, persentase kemiripan yang lebih tinggi tidak realistik. Oleh karena itu, dengan mempertimbangkan variasi alami pada pola sidik jari, batas 60% adalah nilai yang tepat untuk memastikan hasil yang akurat dan dapat diandalkan. Batas ini memungkinkan kami untuk membedakan dengan jelas antara sidik jari yang mirip dan yang tidak, sambil mempertahankan keandalan analisis kami.

#### **4.2 Tata Cara Penggunaan Program**

Terdapat beberapa prasyarat/dependencies yang harus diunduh dan/atau diinstal sebelum menjalankan program FingerMeScanner yang berbasis GUI. Prasyarat/dependencies tersebut adalah:

1. Visual Studio (<https://visualstudio.microsoft.com/downloads/>)

Selain memenuhi dependencies dari program, perlu juga dilakukan beberapa konfigurasi untuk program FingerMeScanner berbasis GUI. Berikut adalah konfigurasi yang diperlukan:

1. Clone Repository

Clone repository program FingerMeScanner dengan mengeksekusi perintah di bawah pada terminal.

```
git clone https://github.com/thoriqsaputra/Tubes3_SakedikKasep
```

2. Menjalankan Program

- Jika Anda memiliki dataset sidik jari sendiri, tempatkan dataset tersebut di folder test dan simpan database bio di folder src. Jika tidak, gunakan dataset yang sudah tersedia di repository ini.
- Jalankan perintah di bawah untuk mengubah directory ke src. Pastikan directory awal adalah project directory dari program ini.

```
cd src
```

- Jalankan GUI dengan mengeksekusi perintah di bawah.

```
dotnet run
```

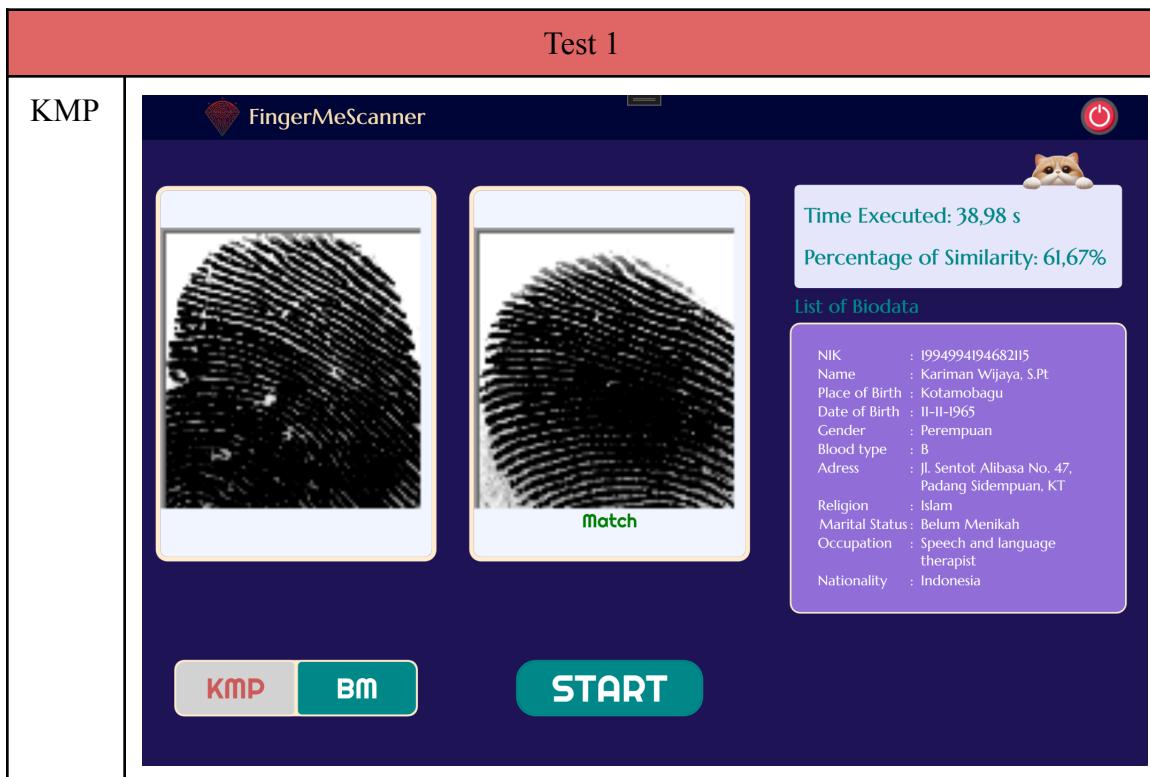
- GUI akan muncul dan siap untuk dicoba.

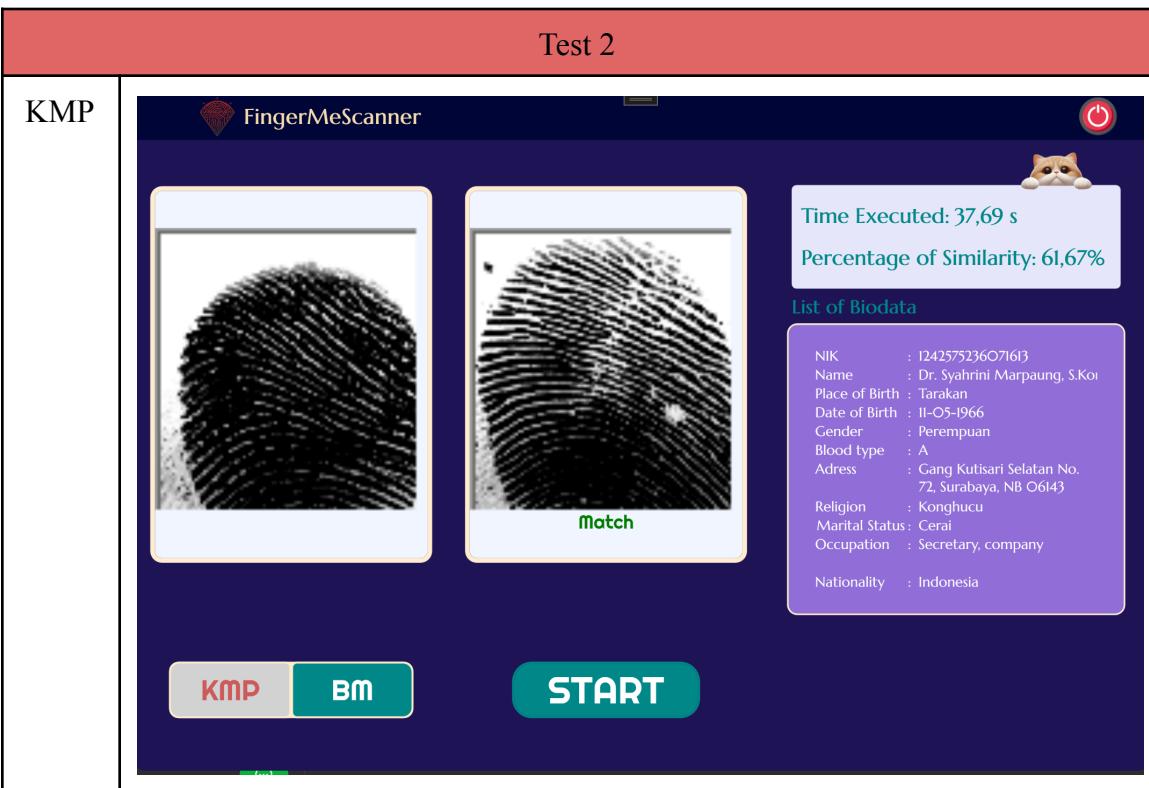
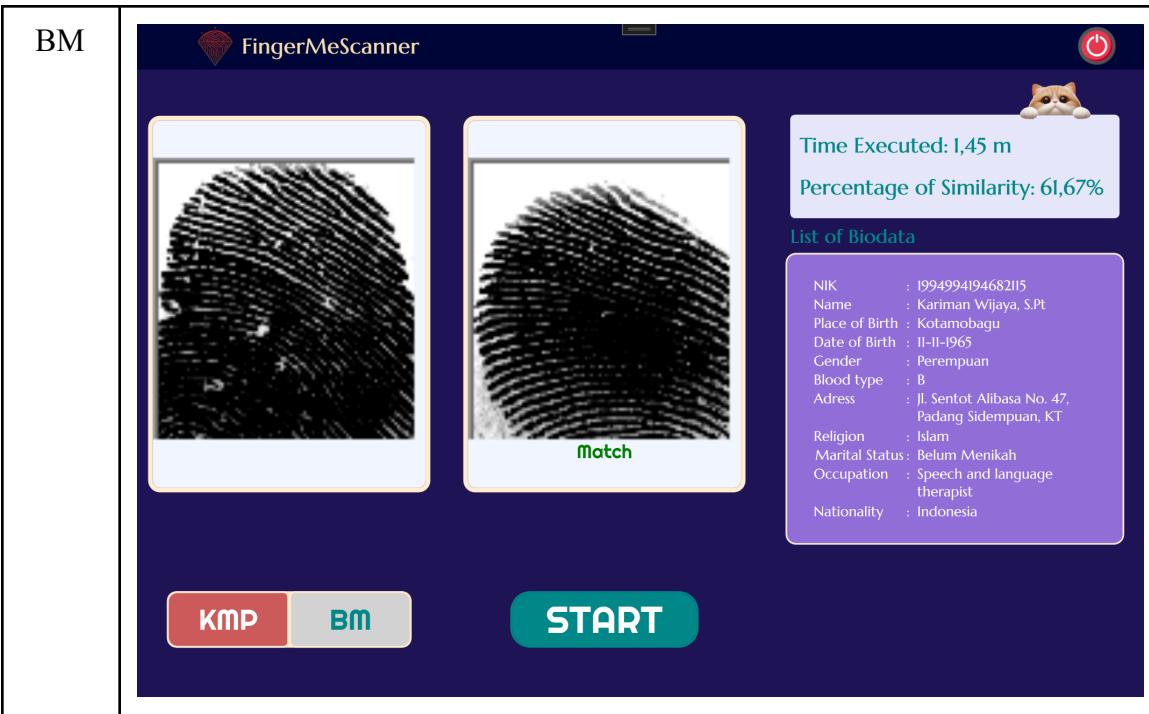
3. Menggunakan Program

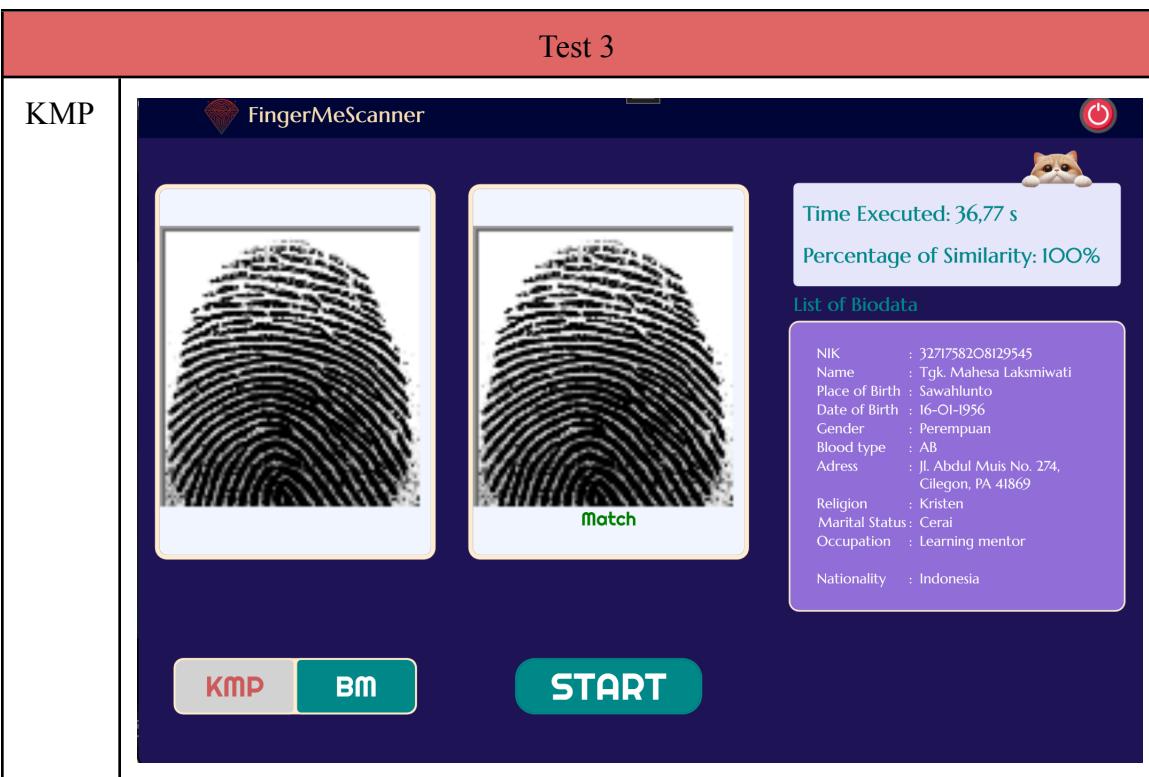
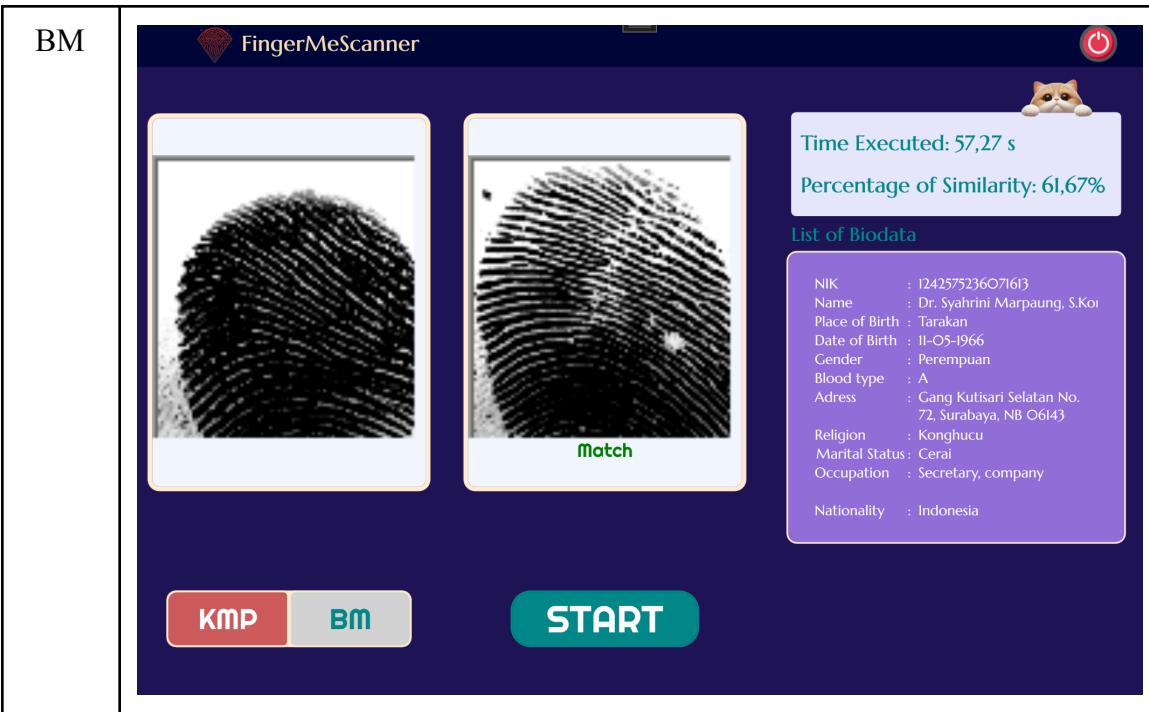
- Unggah gambar sidik jari Anda terlebih dahulu dengan mengklik gambar sidik jari bagian kiri dan pastikan gambar yang diunggah jelas serta sesuai dengan format yang diizinkan.
- Setelah gambar sidik jari berhasil diunggah, pilih salah satu dari dua algoritma yang tersedia untuk mempengaruhi proses pencarian dan hasil akhir.

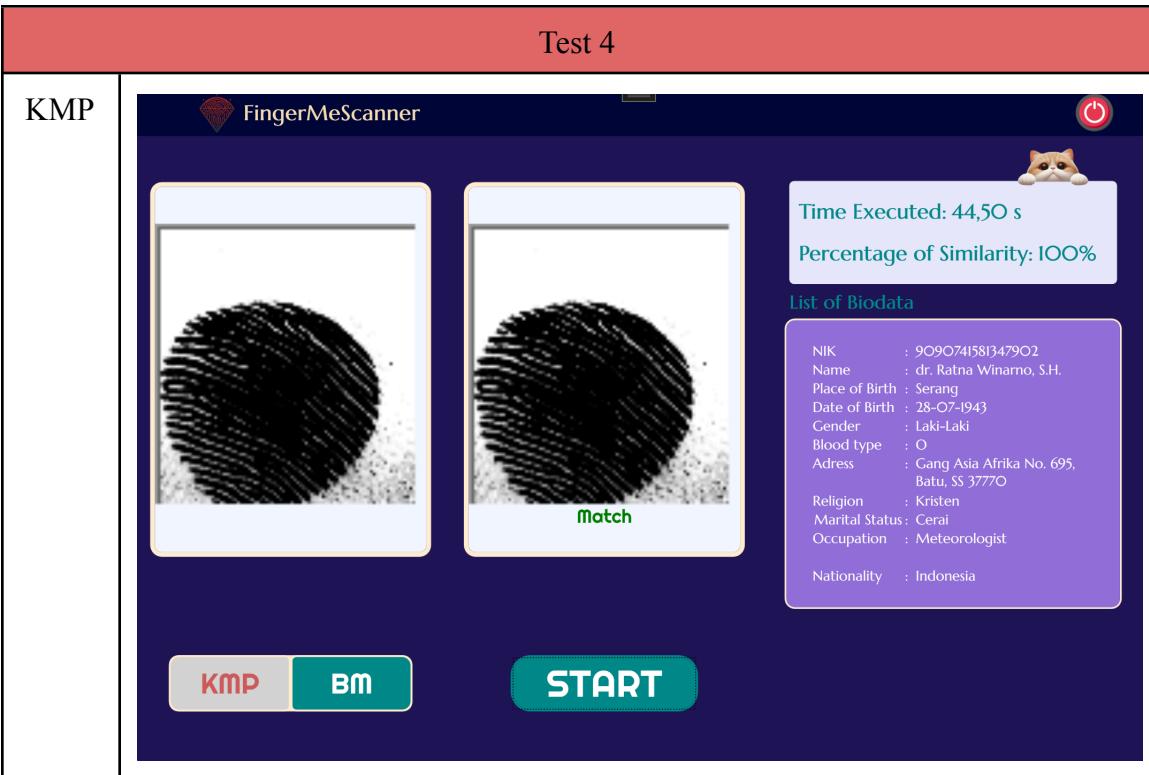
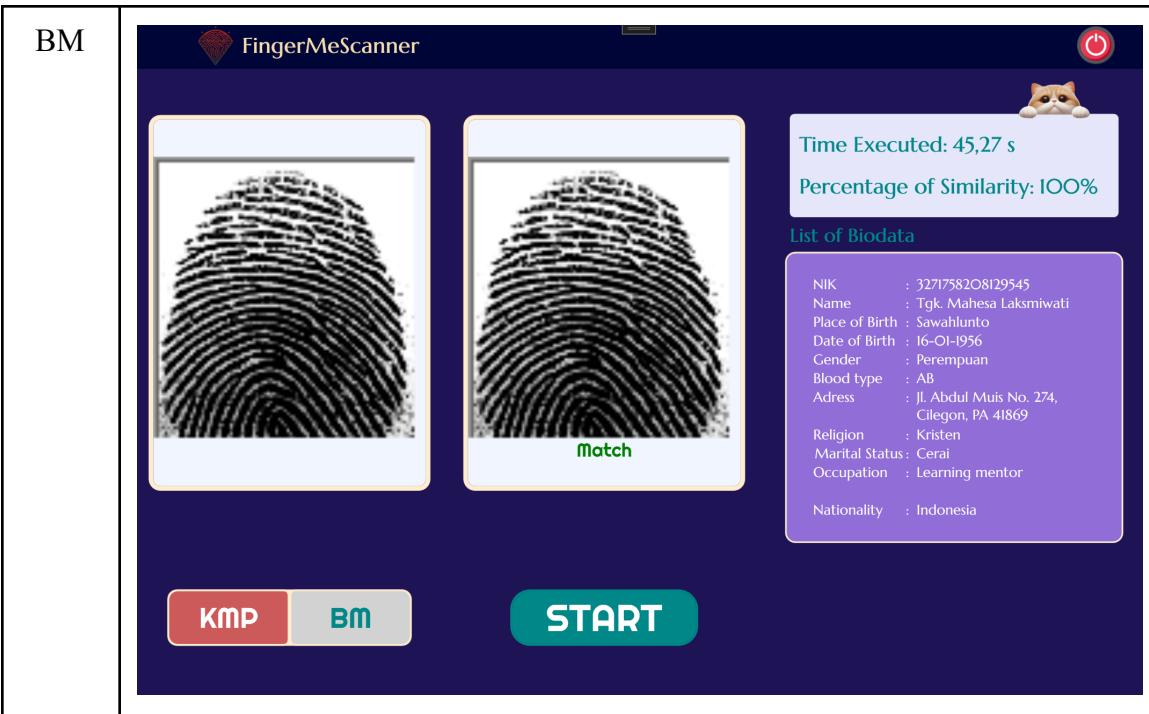
- Jalankan program dengan menekan tombol "Start" setelah memilih algoritma untuk memulai pencarian sidik jari.
- Tunggu beberapa saat hingga program selesai memproses gambar sidik jari dan hasil pencarian akan ditampilkan di layar.

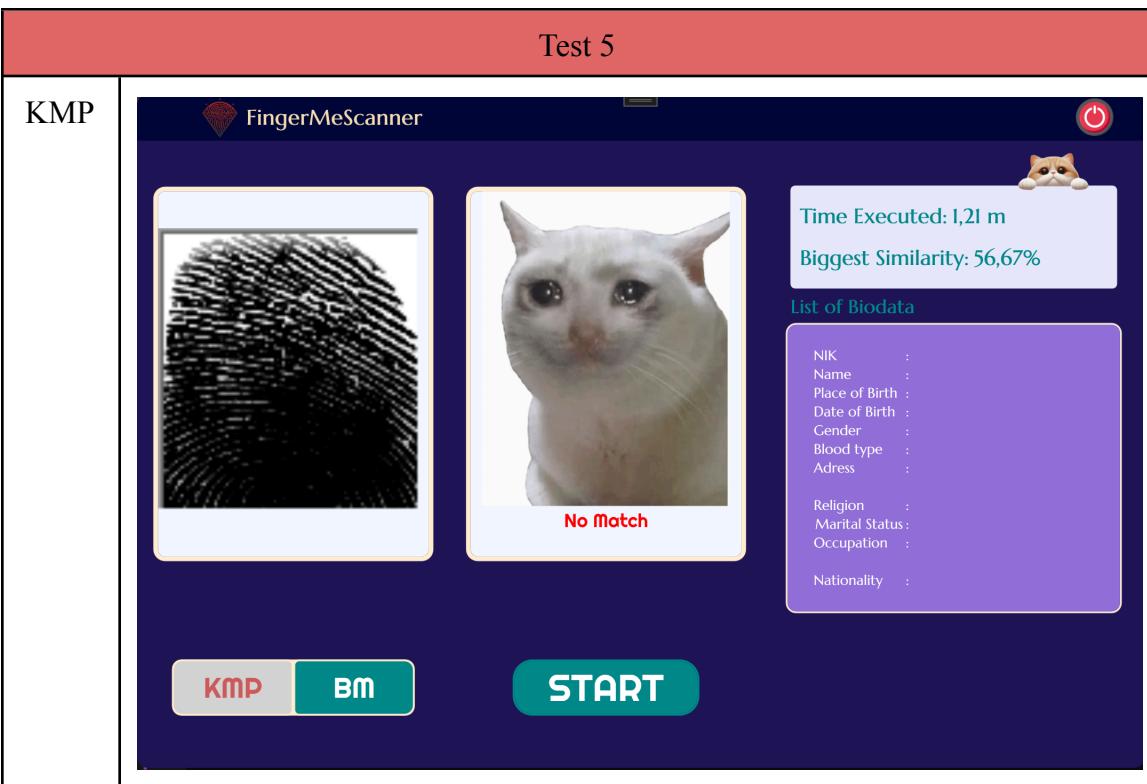
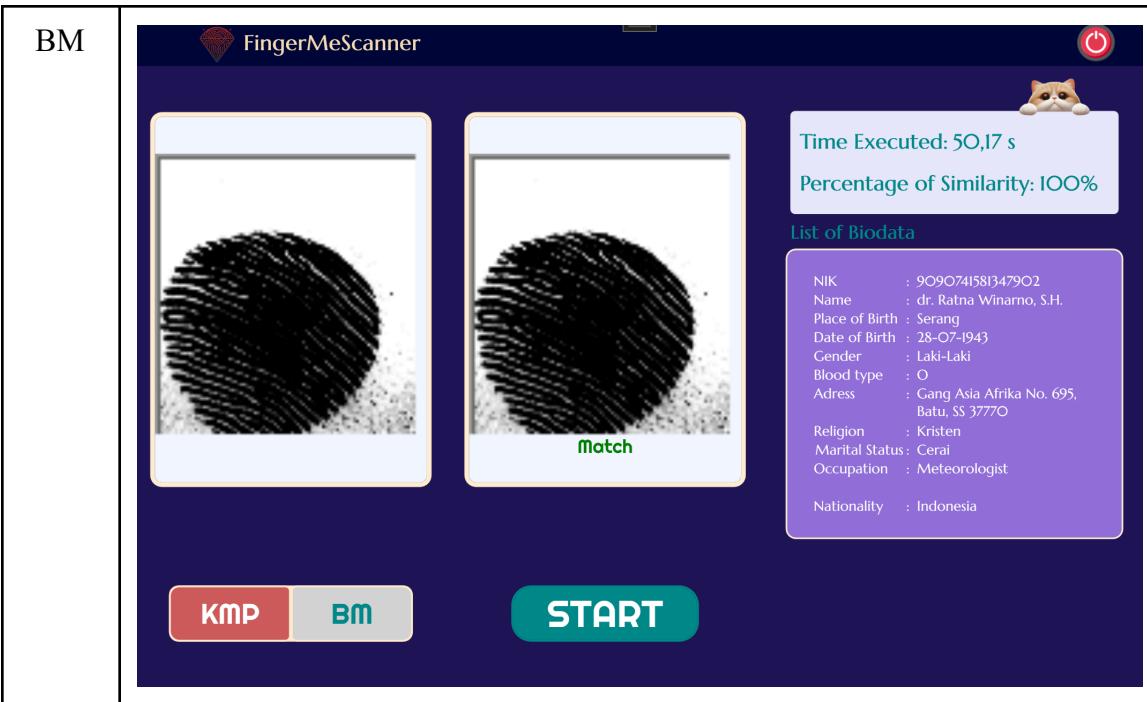
### 4.3 Hasil Pengujian

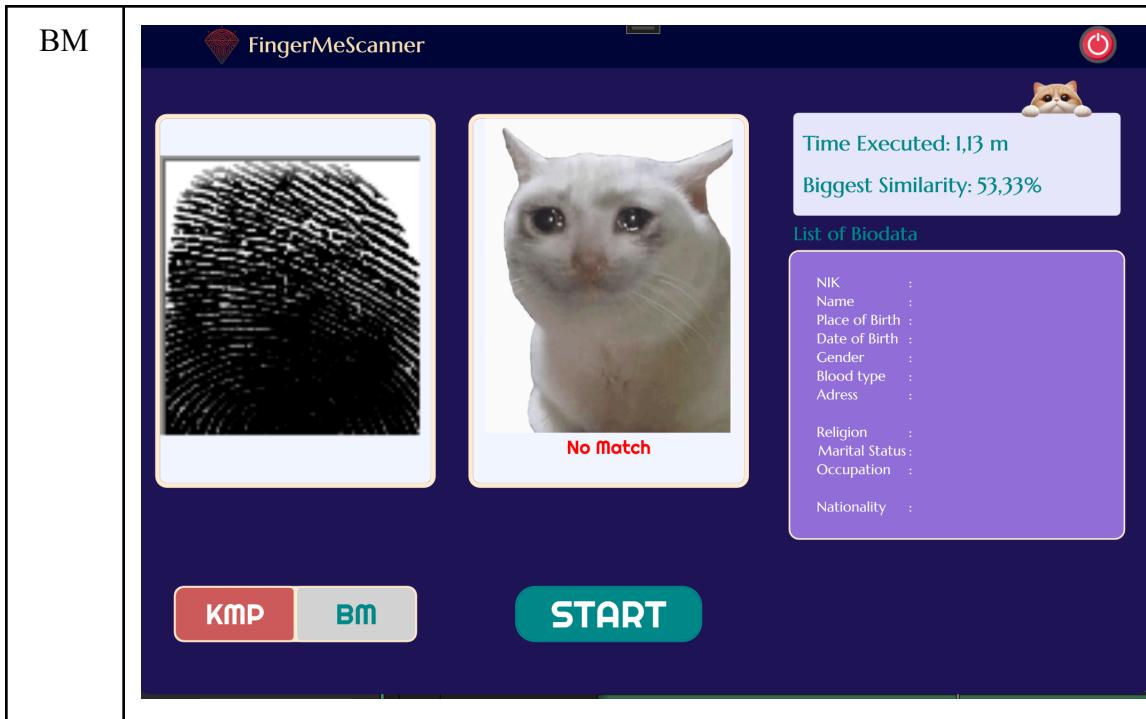












## 4.4 Analisis Algoritma

### 4.4.1 Algoritma Knuth-Morris-Pratt (KMP)

#### 4.4.1.1 Kelebihan

- Efisiensi Waktu: KMP memiliki waktu pencarian linear  $O(n + m)$ , di mana  $n$  adalah panjang teks dan  $m$  adalah panjang pola, sehingga sangat efisien dalam kasus rata-rata dan terburuk.
- Prefiks Efisien: Dengan menggunakan array LPS (Longest Prefix Suffix), KMP dapat menghindari pencocokan ulang karakter yang sudah dibandingkan sebelumnya

#### 4.4.1.2 Kekurangan

- Preprocessing yang Kompleks: Algoritma KMP memerlukan waktu preprocessing  $O(m)$  untuk membangun array LPS, yang bisa menjadi overhead tambahan jika pola berubah-ubah atau banyak pola yang harus diproses.
- Tidak optimal untuk Pola Pendek: Untuk pola yang sangat pendek, overhead preprocessing mungkin tidak sebanding dengan keuntungan dari pencarian yang lebih cepat.

### 4.4.2 Algoritma Boyer Moore

#### 4.4.2.1 Kelebihan

- Efisiensi pada Kasus Umum: Boyer-Moore sering kali lebih cepat dalam praktik karena dapat melewati banyak karakter pada setiap langkah, terutama jika pola dan teks memiliki sedikit kemiripan.

- Heuristik Berbasis karakter: Dengan menggunakan heuristik seperti Bad Character Rule dan Good Suffix Rule, Boyer-Moore dapat dengan cepat menemukan pola dalam teks yang panjang.

#### 4.4.2.2 Kekurangan

- Worst-case yang Buruk: Dalam kasus terburuk, Boyer-Moore bisa bekerja selambat algoritma Brute Force, yaitu  $O(n * m)$ , jika teks dan pola memiliki banyak karakter yang sama.
- Preprocessing yang Rumit: Preprocessing untuk membangun tabel heuristik memerlukan waktu  $O(m + |\Sigma|)$  di mana  $\Sigma$  adalah karakter dalam alfabet, yang bisa menjadi overhead tambahan.
- Kurang Efektif untuk Pola Pendek: Sama seperti KMP, Boyer-Moore mungkin tidak seefisien pada pola yang sangat pendek atau data yang tidak memiliki banyak variasi karakter.

## **BAB V**

### **KESIMPULAN DAN SARAN**

#### **5.1 Kesimpulan**

Pada tugas besar 3 IF2211 Strategi Algoritma Semester 2 Tahun Ajaran 2023/2024 ini, kami diminta untuk membuat sistem deteksi individu berbasis biometrik melalui citra sidik jari. Persoalan yang harus diselesaikan dalam sistem ini adalah pencocokan citra sidik jari yang diberikan dengan citra sidik jari dalam basis data untuk mengidentifikasi individu. Kami diminta untuk mengimplementasikan algoritma Boyer-Moore dan Knuth-Morris-Pratt (KMP) untuk pencocokan pola. Algoritma program dibuat dengan menggunakan bahasa pemrograman C#. Basis data yang digunakan adalah SQL, yang menyimpan informasi biodata individu serta citra sidik jarinya. Selain itu, kami juga menggunakan Regular Expression (Regex) untuk mengatasi masalah data korup dalam bentuk bahasa alay. Program ini memungkinkan pengguna untuk memilih algoritma pencocokan yang ingin digunakan (KMP atau BM), dan menampilkan biodata individu yang paling sesuai dengan citra sidik jari yang diberikan. Sebagai bonus, kami juga melakukan enkripsi terhadap semua data pada basis data menggunakan Caesar Cipher untuk meningkatkan keamanan data pribadi.

#### **5.2 Saran**

Selama proses penggerjaan tugas besar 3 IF2211 Strategi Algoritma, kami menghadapi beberapa kendala seperti pemrosesan citra yang memerlukan konversi ke format string atau biner, yang dapat menambah overhead dalam waktu pemrosesan. Selain itu, algoritma Boyer-Moore memiliki waktu kasus terburuk yang kurang baik jika pola dan teks memiliki banyak karakter yang sama. Untuk pengembangan lebih lanjut, kami menyarankan agar dilakukan optimasi pada proses konversi citra sidik jari agar lebih efisien. Selain itu, dapat dipertimbangkan penggunaan algoritma pencocokan pola lain yang mungkin lebih sesuai untuk data citra biner, seperti algoritma Hamming Distance untuk perhitungan kemiripan. Pengembangan sistem ini juga bisa mencakup peningkatan antarmuka pengguna agar lebih user-friendly dan penambahan fitur-fitur tambahan untuk meningkatkan fungsionalitas dan keamanan sistem. Sebagai tambahan, meskipun penggunaan Caesar Cipher memberikan lapisan keamanan dasar, kami menyarankan untuk mempertimbangkan metode enkripsi yang lebih kuat dan modern untuk melindungi data pribadi yang sensitif.

## **LAMPIRAN**

**Tautan Repository : [https://github.com/thoriqsaputra/Tubes3\\_SakedikKasep](https://github.com/thoriqsaputra/Tubes3_SakedikKasep)**

**Tauran Video : <https://youtu.be/FkbXe3dQaAM>**

## DAFTAR PUSTAKA

- [1]“KMP Algorithm for Pattern Searching,” *GeeksforGeeks*, Apr. 03, 2011. <https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>
- [2]“Regular Expression (Regex) Tutorial,” [www3.ntu.edu.sg](https://www3.ntu.edu.sg/home/ehchua/programming/howto/Regexe.html).
- [3]GeeksforGeeks, “Caesar Cipher in Cryptography - GeeksforGeeks,” *GeeksforGeeks*, Jun. 02, 2016. <https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/>
- [4]E. Nam, “Understanding the Levenshtein Distance Equation for Beginners,” *Medium*, Feb. 27, 2019. <https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0#:~:text=The%20Levenshtein%20distance%20is%20a> (accessed Jun. 08, 2024).
- [5]Maureen, “Behind Fingerprint Biometrics: How It Works and Why It Matters,” *1Kosmos*, Oct. 31, 2023. <https://www.1kosmos.com/biometric-authentication/behind-fingerprint-biometrics-how-it-works-and-why-it-matters/>
- [6]R. Munir, “Pencocokan String (String/Pattern Matching).” Accessed: Jun. 08, 2024. [Online]. Available: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf>