

## **LAPORAN TUGAS BESAR 3**

### **IF2211 Strategi Algoritma**

#### **Pemanfaatan Pattern Matching untuk Membangun Sistem ATS (Applicant Tracking System) Berbasis CV Digital**



Disusun oleh:

Andi Farhan Hidayat (13523142)

Nathanael Rachmat (13523142)

Andrew Tedjapratama (13523148)

**PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
2025**

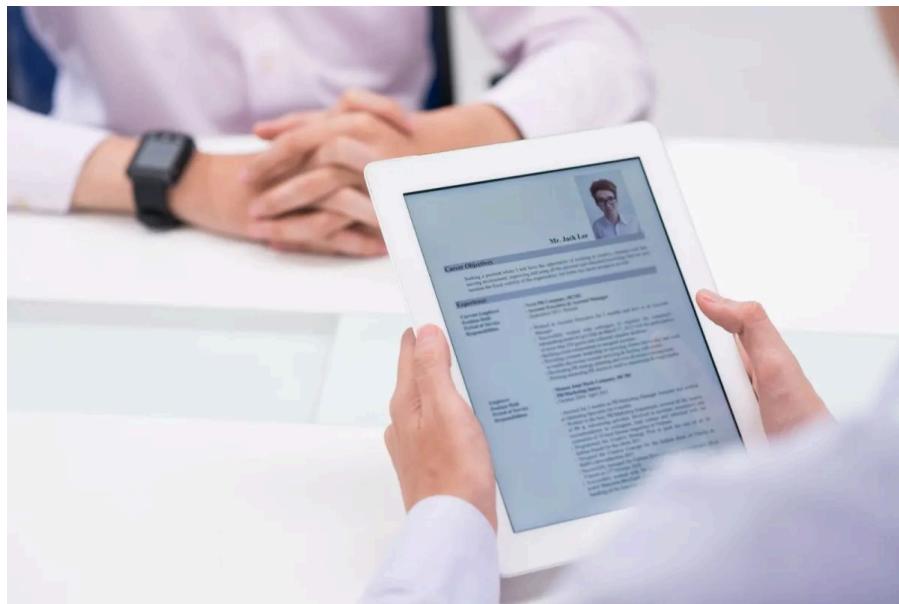
# DAFTAR ISI

Pemanfaatan Pattern Matching untuk Membangun Sistem ATS (Applicant Tracking System) Berbasis CV Digital.....	1
<b>DAFTAR ISI.....</b>	<b>2</b>
<b>BAB I.....</b>	<b>1</b>
<b>Penjelasan Implementasi.....</b>	<b>2</b>
<b>BAB II.....</b>	<b>4</b>
2.1 Konsep Sistem Applicant Tracking System (ATS).....	4
2.1.1 Pengertian ATS dan Peranannya dalam Rekrutmen.....	4
2.1.2 Pemanfaatan Pattern Matching dalam ATS.....	4
2.2 Penjelasan Algoritma String-Matching.....	5
2.2.1 Algoritma Knuth-Morris-Pratt (KMP).....	5
2.2.2 Algoritma Boyer-Moore (BM).....	6
2.2.3 Algoritma Aho-Corasick (BONUS).....	8
2.3 Levenshtein Distance.....	9
2.4 Regular Expression (Regex).....	10
<b>BAB III.....</b>	<b>11</b>
3.1 Langkah-langkah Pemecahan Masalah.....	11
3.2 Proses Pemetaan Masalah menjadi Elemen Algoritma KMP dan BM.....	17
3.2.1 Abstraksi Masalah ATS.....	17
3.2.2 Pemetaan Elemen Masalah ke Komponen Algoritma.....	17
3.3 Fitur Fungsional dan Arsitektur Aplikasi.....	18
3.3.1 Basis Data.....	18
3.3.2 Seeding.....	18
3.3.3 Backend (Python).....	19
3.3.4 Flet.....	19
3.4 Contoh ilustrasi kasus.....	19
3.4.1 Pencarian Single Keyword dengan KMP (Exact Match).....	19
3.4.2 Pencarian Multiple Keywords dengan Boyer-Moore (Exact & Fuzzy).....	20
3.4.3 Pencarian dengan Typo menggunakan AHO-Corasick.....	20
3.4.4 .....	20
<b>BAB IV.....</b>	<b>21</b>
4.1 Spesifikasi Teknis Program.....	21
4.2 Hasil Pengujian (Testing).....	22
4.3 Analisis Hasil Pengujian.....	22
<b>BAB V.....</b>	<b>23</b>
5.1 Kesimpulan.....	23
5.2 Saran.....	23
5.3 Refleksi.....	24
<b>BAB VI.....</b>	<b>25</b>

6.1 Link Repository.....	25
6.2 Link Bonus Video (YouTube).....	25
6.3 Tabel Hasil.....	25

# BAB I

## DESKRIPSI TUGAS



**Gambar 1.1** CV ATS dalam Dunia Kerja

(Sumber: <https://www.antaranews.com/> )

Di era digital ini, keamanan data dan akses menjadi semakin penting. Perkembangan proses rekrutmen tenaga kerja telah mengalami perubahan signifikan dengan memanfaatkan teknologi untuk meningkatkan efisiensi dan akurasi. Salah satu inovasi yang menjadi solusi utama adalah Applicant Tracking System (ATS), yang dirancang untuk mempermudah perusahaan dalam menyaring dan mencocokkan informasi kandidat dari berkas lamaran, khususnya Curriculum Vitae (CV). ATS memungkinkan perusahaan untuk mengelola ribuan dokumen lamaran secara otomatis dan memastikan kandidat yang relevan dapat ditemukan dengan cepat.

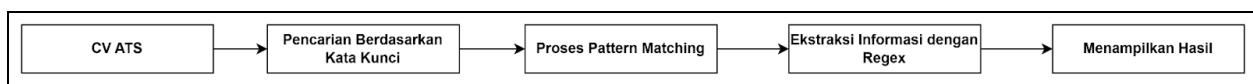
Meskipun demikian, salah satu tantangan besar dalam pengembangan sistem ATS adalah kemampuan untuk memproses dokumen CV dalam format PDF yang tidak selalu terstruktur. Dokumen seperti ini memerlukan metode canggih untuk mengekstrak informasi penting seperti identitas, pengalaman kerja, keahlian, dan riwayat pendidikan secara efisien. Pattern matching menjadi solusi ideal dalam menghadapi tantangan ini.

Pattern matching adalah teknik untuk menemukan dan mencocokkan pola tertentu dalam teks. Dalam konteks ini, algoritma Boyer-Moore dan Knuth-Morris-Pratt (KMP) sering digunakan karena keduanya menawarkan efisiensi tinggi untuk pencarian teks di dokumen besar. Algoritma ini memungkinkan sistem ATS untuk mengidentifikasi informasi penting dari CV pelamar dengan kecepatan dan akurasi yang optimal.

Di dalam Tugas Besar 3 ini, Anda diminta untuk mengimplementasikan sistem yang dapat melakukan deteksi informasi pelamar berbasis dokumen CV digital. Metode yang akan digunakan untuk melakukan deteksi pola dalam CV adalah Boyer-Moore dan Knuth-Morris-Pratt. Selain itu, sistem ini akan dihubungkan dengan identitas kandidat melalui basis data sehingga harapannya terbentuk sebuah sistem yang dapat mengenali profil pelamar secara lengkap hanya dengan menggunakan CV digital.

## Penjelasan Implementasi

Dalam tugas ini, Anda akan mengembangkan sebuah sistem ATS (Applicant Tracking System) berbasis CV Digital dengan memanfaatkan teknik Pattern Matching. Implementasi sistem ini akan menggunakan algoritma Boyer-Moore dan Knuth-Morris-Pratt (*Aho-Corasick* apabila mengerjakan bonus) untuk menganalisis dan mencocokkan pola dalam dokumen CV digital, sesuai dengan konsep yang telah dipelajari dalam materi dan slide perkuliahan.



Gambar 1.2 Skema Implementasi *Applicant Tracking System*

Sistem ini bertujuan untuk mencocokkan kata kunci dari user terhadap isi CV pelamar kerja dengan pendekatan pattern matching menggunakan algoritma KMP (Knuth-Morris-Pratt) atau BM (Boyer-Moore). Semua proses dilakukan secara in-memory, tanpa menyimpan hasil pencarian—hanya data mentah (raw) CV yang disimpan. Pengguna (HR atau rekruter) akan memberikan input berupa daftar kata kunci yang ingin dicari (misalnya: "python", "react", dan "sql") serta jumlah CV yang ingin ditampilkan (misalnya Top 10 matches). Setiap file CV dalam format PDF akan dikonversi menjadi satu string panjang yang memuat seluruh teks dari dokumen tersebut. Proses konversi ini bertujuan untuk mempermudah pencocokan pola menggunakan algoritma string matching, sehingga setiap keyword dapat dicari secara efisien dalam satu representasi data linear.

Untuk memberikan pemahaman yang lebih konkret, berikut disajikan contoh kasus penerapan sistem CV ATS beserta prosesnya dan contoh output yang dihasilkan. Dataset yang digunakan dalam contoh ini merupakan dataset CV ATS yang tercantum pada bagian referensi.

Tabel 1.1 Hasil ekstraksi teks dari CV ATS

CV ATS	Ekstraksi Text untuk Regex	Ekstraksi Text untuk <i>Pattern Matching</i> (KMP & BM)
10276858.pdf	<a href="#">Ekstraksi Text Regex.txt</a>	<a href="#">Ekstraksi Text Pattern Matching.txt</a>

Pada tahap implementasi ini, setiap CV yang telah dikonversi menjadi string panjang untuk mempermudah proses pencocokan. Representasi ini menjadi dasar dalam mencari CV yang paling relevan dengan kata kunci yang dimasukkan oleh pengguna. Proses pencarian dilakukan

dengan menggunakan algoritma pencocokan string Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM) untuk menemukan CV yang memiliki kemiripan tertinggi dengan kebutuhan yang ditentukan. Apabila tidak ditemukan satupun CV dalam basis data yang memiliki kecocokan kata kunci secara exact match menggunakan algoritma KMP maupun Boyer-Moore, maka sistem akan mencari CV yang paling mirip berdasarkan tingkat kemiripan di atas ambang batas tertentu (threshold). Hal ini mempertimbangkan kemungkinan adanya kesalahan pengetikan (typo) oleh pengguna atau HR saat memasukkan kata kunci. Anda diberikan **keleluasaan untuk menentukan nilai ambang batas persentase** kemiripan tersebut, dengan syarat dilakukan pengujian terlebih dahulu untuk menemukan nilai tuning yang optimal dan **dijelaskan secara rinci dalam laporan**. Metode perhitungan tingkat kemiripan harus diterapkan menggunakan algoritma **Levenshtein Distance**.

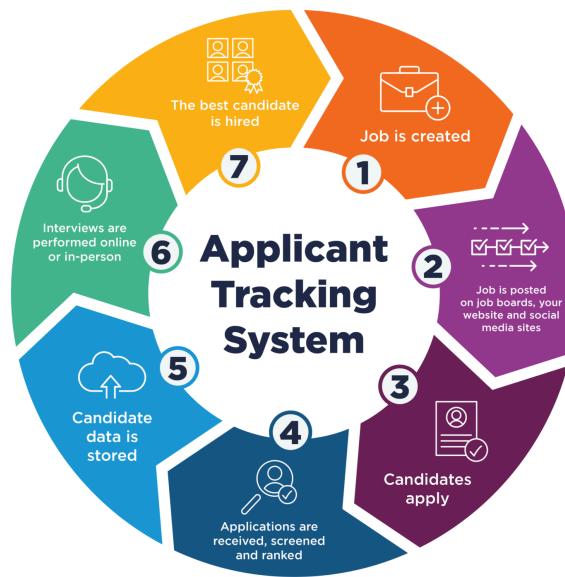
## BAB II

# LANDASAN TEORI

### 2.1 Konsep Sistem *Applicant Tracking System* (ATS)

#### 2.1.1 Pengertian ATS dan Peranannya dalam Rekrutmen

Applicant Tracking System (ATS) adalah sebuah sistem atau software yang dirancang untuk membantu perusahaan, terutama *recruiter* untuk mengotomatiskan dan mempermudah proses rekrutmen. Di era digital, di mana jumlah pelamar bisa mencapai ribuan, ATS berperan penting dalam menyaring berkas lamaran, terutama Curriculum Vitae (CV), untuk menemukan kandidat yang paling relevan dengan cepat dan efisien. Tantangan utama dalam pengembangan ATS adalah kemampuannya memproses CV dalam format PDF yang seringkali tidak terstruktur.



**Gambar 2.1.1** Alur kerja Applicant Tracking System dalam proses rekrutmen  
(Sumber: <https://www.bullhorn.com/uk/glossary/applicant-tracking-system/> )

#### 2.1.2 Pemanfaatan Pattern Matching dalam ATS

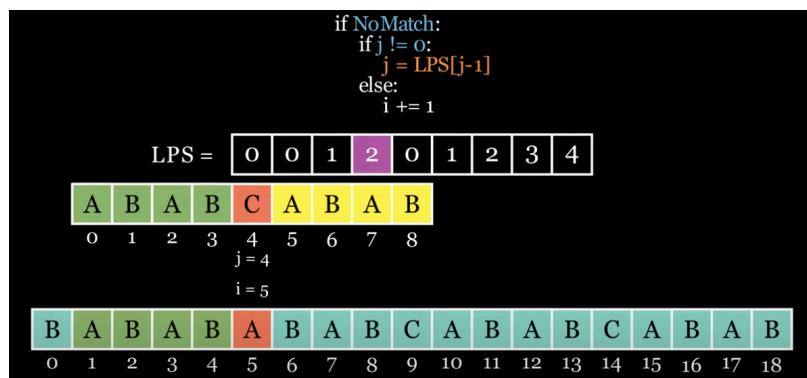
Di sinilah pemanfaatan pattern matching menjadi solusi ideal. Teknik ini memungkinkan sistem untuk menemukan dan mencocokkan pola-pola spesifik, seperti keahlian (*skills*), pengalaman kerja (*experience*), atau riwayat pendidikan (*education*) di dalam teks CV yang panjang. Dengan begitu, informasi penting dapat diekstrak secara akurat untuk dinilai lebih lanjut setelah melalui ATS.

## 2.2 Penjelasan Algoritma String-Matching

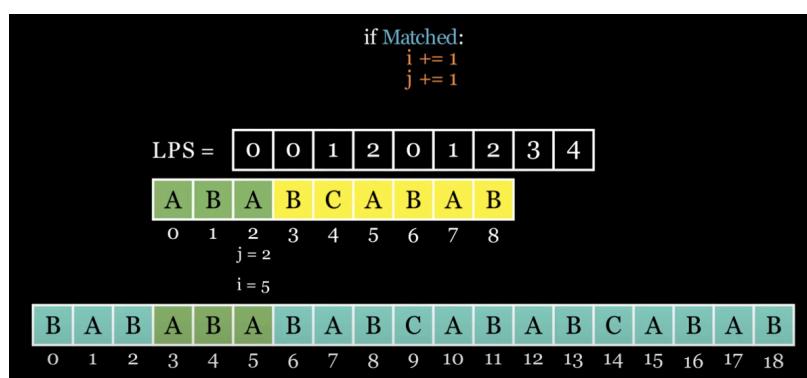
### 2.2.1 Algoritma Knuth-Morris-Pratt (KMP)

Algoritma Knuth-Morris-Pratt (KMP) adalah salah satu metode string matching yang sangat efisien untuk menemukan berapa kemunculan sebuah pola (*pattern*) di dalam sebuah teks. Algoritma ini ditemukan oleh Donald Knuth dan Vaughan Pratt bersama-sama, dan secara independen oleh James H. Morris pada tahun 1977.

Keunggulan KMP terletak pada kemampuannya untuk tidak mengulang perbandingan pada karakter teks yang sudah pernah cocok. Ketika terjadi ketidakcocokan, algoritma ini menggunakan sebuah tabel pra-pemrosesan (LPS array) untuk secara cerdas menggeser pola ke posisi berikutnya yang paling mungkin, sehingga membantu algoritma menghindari backtracking yang tidak perlu ketika ketidakcocokan terjadi selama proses pencocokan pola. Dalam proyek ini, KMP digunakan sebagai salah satu opsi exact matching untuk menemukan kata kunci yang dimasukkan pengguna di dalam data



**Gambar 2.2.1.1** Ilustrasi KMP algoritm dan tabel LPS pada kasus saat terjadi *mismatch*, pola tidak digeser satu per satu, melainkan melompat beberapa langkah  
(Sumber: [Knuth-Morris-Pratt Algorithm Visually Explained](#) )



**Gambar 2.2.1.2** Ilustrasi KMP algorithm, pola melompat ke indeks ke-2 (yaitu LPS indeks ke pointer j dikurang 1)

(Sumber: [Knuth-Morris-Pratt Algorithm Visually Explained](#) )

KMP merupakan algoritma yang baik untuk memproses file yang sangat besar yang dibaca dari perangkat eksternal atau melalui aliran jaringan. Namun, KMP juga memiliki kelemahan dan tidak bekerja dengan baik jika ukuran alfabet bertambah, sehingga lebih besar kemungkinan ketidakcocokan (lebih banyak kemungkinan ketidakcocokan), dan ketidakcocokan cenderung terjadi di awal pola, tetapi KMP lebih cepat.

Algoritma ini termasuk algoritma yang sangat cepat dan efisien dalam string matching dan sangat cepat dibanding *brute force*, dengan kompleksitas menghitung fungsi pinggiran  $O(m)$  dan kompleksitas pencarian string  $O(n)$ , sehingga kompleksitas waktu algoritma KMP keseluruhan:

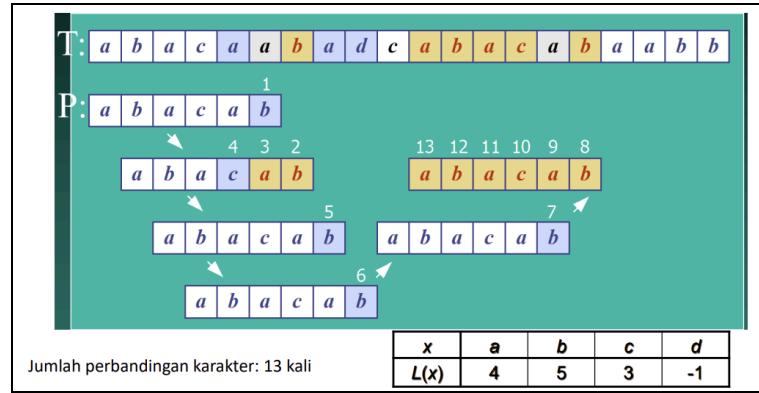
$$O(m + n)$$

Di mana:

- n: panjang teks
- m: panjang pola

## 2.2.2 Algoritma Boyer-Moore (BM)

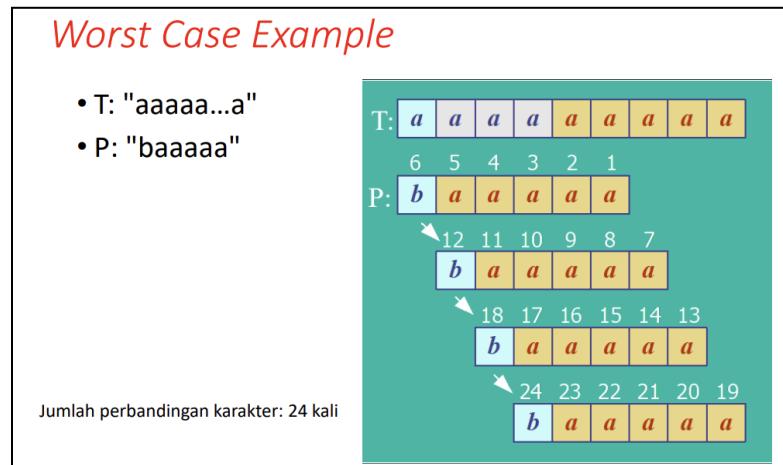
Algoritma Boyer-Moore (BM) adalah algoritma pencocokan string lain yang juga sangat efisien, seringkali lebih cepat dari KMP dalam praktik. Keunikan BM adalah pendekatannya yang membandingkan pola dari kanan ke kiri, bukan dari kiri ke kanan. Ketika terjadi ketidakcocokan, BM menggunakan dua aturan (heuristik) untuk melakukan pergeseran pola sejauh mungkin: bad character heuristic dan good suffix heuristic. Pendekatan ini memungkinkan algoritma untuk melewati sebagian besar teks tanpa perlu membandingkannya, menjadikannya sangat cepat untuk pencarian dalam dokumen besar seperti CV. Sama seperti KMP, BM diimplementasikan sebagai opsi untuk exact matching kata kunci.



**Gambar 2.2.2.1** Contoh ilustrasi algoritma Boyer-Moore untuk string matching

(Sumber: [Pencocokan String \(String/Pattern Matching - Rinaldi Munir\)](#))

Namun, Boyer-Moore memiliki beberapa kelemahan, terutama ketika pola yang dicari sangat pendek atau ketika teks dan pola memiliki banyak kemiripan karakter berulang. Dalam kasus tersebut (*worst case*), efektivitas heuristiknya menurun dan performanya mendekati *brute-force*, atau dengan kompleksitas waktu  $O(mn)$ . Selain itu, implementasi Boyer-Moore relatif lebih kompleks karena memerlukan dua tabel pra-proses: tabel karakter buruk dan tabel sufiks baik.



**Gambar 2.2.2.2** Ilustrasi *worst case* dari Boyer-Moore (ketika kemiripan karakter berulang), dengan performa sama dengan *brute force*

(Sumber: [Pencocokan String \(String/Pattern Matching - Rinaldi Munir\)](#))

Secara keseluruhan, Boyer-Moore termasuk algoritma yang sangat cepat dan efisien dalam praktik, terutama saat pola panjang dan alfabet besar, karena mampu "melompati" lebih banyak bagian teks. Kompleksitas waktu untuk tahap

pra proses (membangun tabel) adalah  $O(m + |\Sigma|)$  dan untuk pencarian string adalah  $O(n)$  dalam kasus terbaik, dengan kompleksitas waktu keseluruhan:

$$O(n/m)$$

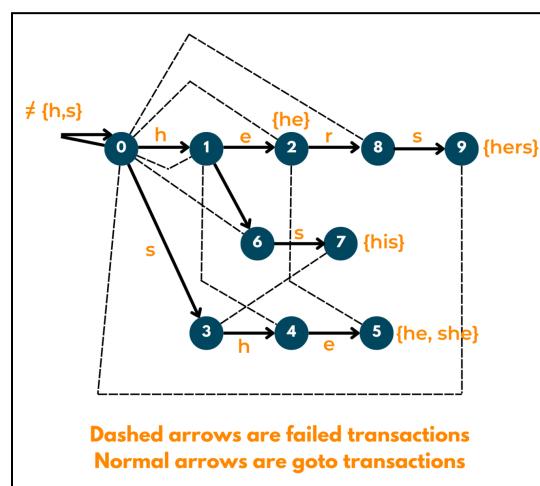
Di mana:

- n: panjang teks
- m: panjang pola

### 2.2.3 Algoritma Aho-Corasick (BONUS)

Untuk pencarian yang lebih optimal, terutama saat pengguna memasukkan banyak kata kunci sekaligus, diimplementasikan juga algoritma Aho-Corasick. Algoritma ini merupakan ekstensi dari KMP yang dirancang khusus untuk multi-pattern matching. Aho-Corasick membangun sebuah mesin status (finite state machine) dari semua kata kunci yang dicari, kemudian memproses seluruh teks hanya dalam satu kali lintasan (single pass) untuk menemukan semua kemunculan dari semua kata kunci. Ini jauh lebih efisien daripada menjalankan KMP atau BM berulang kali untuk setiap kata kunci. Algoritma pencocokan string Aho-Corasick menjadi dasar perintah Unix asli fgrep.

Untuk menangani mismatch saat pencocokan, Aho-Corasick menggunakan failure links, yaitu jalur fallback yang membawa mesin status kembali ke posisi sebelumnya yang masih memiliki kemungkinan kecocokan sebagian. Dengan memanfaatkan struktur ini, algoritma tetap dapat melanjutkan pencarian secara efisien tanpa harus memulai ulang dari awal teks. Mekanisme ini mirip dengan fungsi prefix pada KMP, namun diterapkan dalam konteks banyak pola sekaligus.



**Gambar 2.2.3** Ilustrasi algoritma Aho-Corasick  
(Sumber: <https://favtutor.com/blogs/aho-corasick-algorithm>)

Aho-Corasick termasuk algoritma yang paling cepat dalam pencarian string matching dengan kompleksitas waktu:

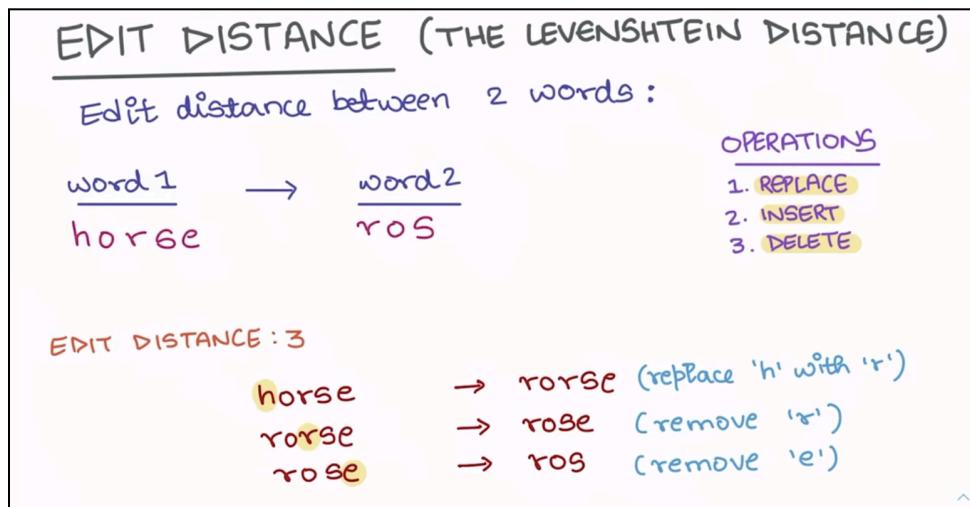
$$O(n + m + z)$$

Di mana:

- n: panjang teks
- m: panjang pola
- z: jumlah total kemunculan kata dalam teks

### 2.3 Levenshtein Distance

Terkadang, pengguna (perekut) bisa saja melakukan kesalahan ketik (typo) saat memasukkan kata kunci. Untuk mengatasi masalah ini dan tetap memberikan hasil yang relevan, sistem menerapkan pencarian fuzzy matching menggunakan Levenshtein Distance atau *Edit Distance*. Algoritma ini bekerja dengan menghitung "jarak" atau tingkat perbedaan antara dua string. Jarak ini didefinisikan sebagai jumlah minimum operasi penyisipan (*insertion*), penghapusan (*deletion*), atau penggantian (*replacement*) karakter yang diperlukan untuk mengubah satu string menjadi string lainnya. Jika jarak Levenshtein antara kata kunci yang dimasukkan dan kata dalam CV berada di bawah ambang batas tertentu, sistem akan menganggapnya sebagai kecocokan.



Gambar 2.3 Ilustrasi proses Levenshtein Distance atau Edit Distance dan operasi yang dipakai didalam proses penggantian string

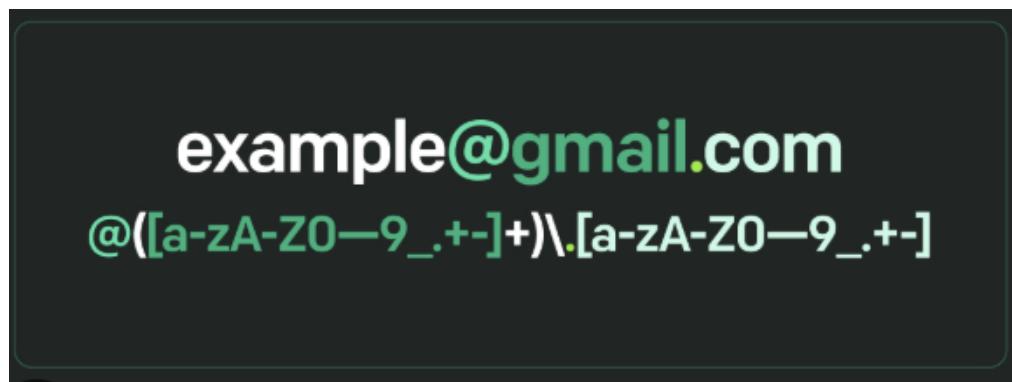
(Sumber: [Edit Distance between 2 Strings | The Levenshtein Distance Algorithm + Code](#))

Untuk menghitung jarak Levenshtein secara efisien, sistem biasanya menggunakan pendekatan Dynamic Programming. Algoritma ini membangun sebuah matriks dua dimensi yang merepresentasikan semua kemungkinan transformasi antara dua string.

Setiap sel dalam matriks menyimpan hasil subproblem, yaitu jumlah minimum operasi yang diperlukan hingga posisi karakter tertentu. Dengan cara ini, algoritma menghindari perhitungan ulang dan mencapai kompleksitas waktu  $O(m \times n)$ , di mana m dan n adalah panjang dua string yang dibandingkan.

## 2.4 Regular Expression (Regex)

Setelah CV dikonversi menjadi teks mentah, informasi di dalamnya masih belum terstruktur. Regular Expression (Regex) adalah sebuah sekuens karakter yang mendefinisikan sebuah pola pencarian. Dalam sistem ini, Regex digunakan untuk mengekstrak informasi penting dan terstruktur dari teks CV secara otomatis. Pola-pola Regex dirancang secara spesifik untuk mengenali dan mengambil data seperti nomor telepon, alamat email, riwayat pendidikan (misalnya, nama universitas dan tahun kelulusan), serta pengalaman kerja (misalnya, jabatan dan periode kerja). Penggunaan Regex menjadi fondasi yang kuat untuk membangun ringkasan profil pelamar secara otomatis dan cepat.



**Gambar 2.3** Ilustrasi pemakaian regex dalam formatting email  
(Sumber: <https://platform.text.com/resource-center/updates/regex>)

## BAB III

# ANALISIS PEMECAHAN MASALAH

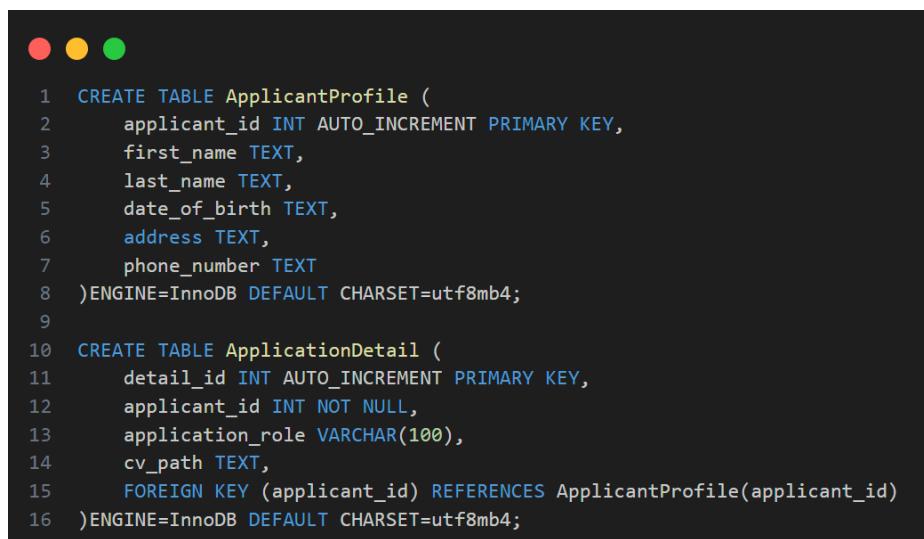
Bab ini membahas pendekatan strategis dan langkah-langkah yang diambil untuk mengembangkan sistem ATS berbasis CV digital ini. Fokus utama adalah bagaimana konsep dan algoritma yang telah dijelaskan dalam Landasan Teori diterapkan untuk menyelesaikan masalah pencarian dan ekstraksi informasi dari CV.

### 3.1 Langkah-langkah Pemecahan Masalah

Proses pemecahan masalah dalam sistem ATS ini mengikuti alur kerja yang dimulai dari inisialisasi basis data, pengisian data melalui seeding, proses enkripsi menggunakan RSA, hingga pemuatan data ke dalam memori aplikasi. Tahapan ini dirancang untuk memastikan integritas data, keamanan informasi pribadi, dan efisiensi pencarian.

#### 1. Inisialisasi dan Pemuatan Data Basis Data

Basis data MariaDB digunakan sebagai DBMS untuk menyimpan data profil pelamar *ApplicantProfile* dan detail aplikasi *ApplicationDetail*. Struktur tabel ini mencakup *applicant\_id*, *first\_name*, *last\_name*, *date\_of\_birth*, *address*, *phone\_number* untuk *ApplicantProfile*, serta *detail\_id*, *applicant\_id*, *application\_role*, dan *cv\_path* untuk *ApplicationDetail*.



```
CREATE TABLE ApplicantProfile (
    applicant_id INT AUTO_INCREMENT PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    date_of_birth TEXT,
    address TEXT,
    phone_number TEXT
)ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE ApplicationDetail (
    detail_id INT AUTO_INCREMENT PRIMARY KEY,
    applicant_id INT NOT NULL,
    application_role VARCHAR(100),
    cv_path TEXT,
    FOREIGN KEY (applicant_id) REFERENCES ApplicantProfile(applicant_id)
)ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

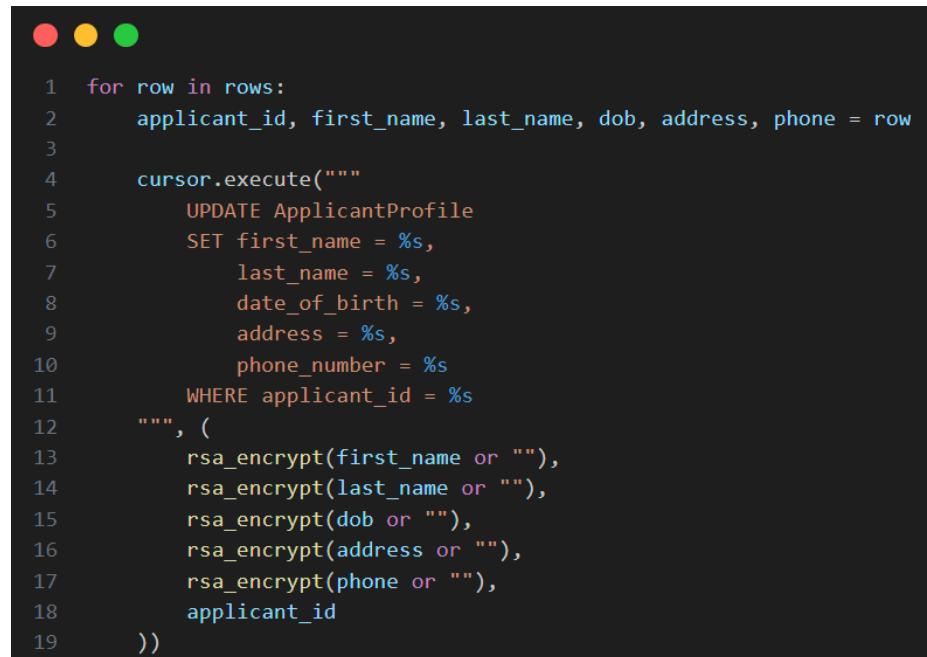
Gambar 3.1.1 Skema utama database

Data awal diisi melalui proses seeding dari file *tubes3\_seeding.sql*, yang memuat data dummy pelamar. Setelah data dimasukkan, kolom-kolom sensitif seperti

first\_name, last\_name, date\_of\_birth, address, dan phone\_number dienkripsi menggunakan RSA, yaitu algoritma kriptografi asimetris yang menggunakan sepasang kunci publik dan privat.

Tujuan enkripsi berupa:

- Melindungi data pribadi pelamar agar tidak dapat dibaca langsung di database.
- Enkripsi dilakukan di tahap awal untuk menjamin seluruh data yang disimpan sudah aman sejak awal proses.

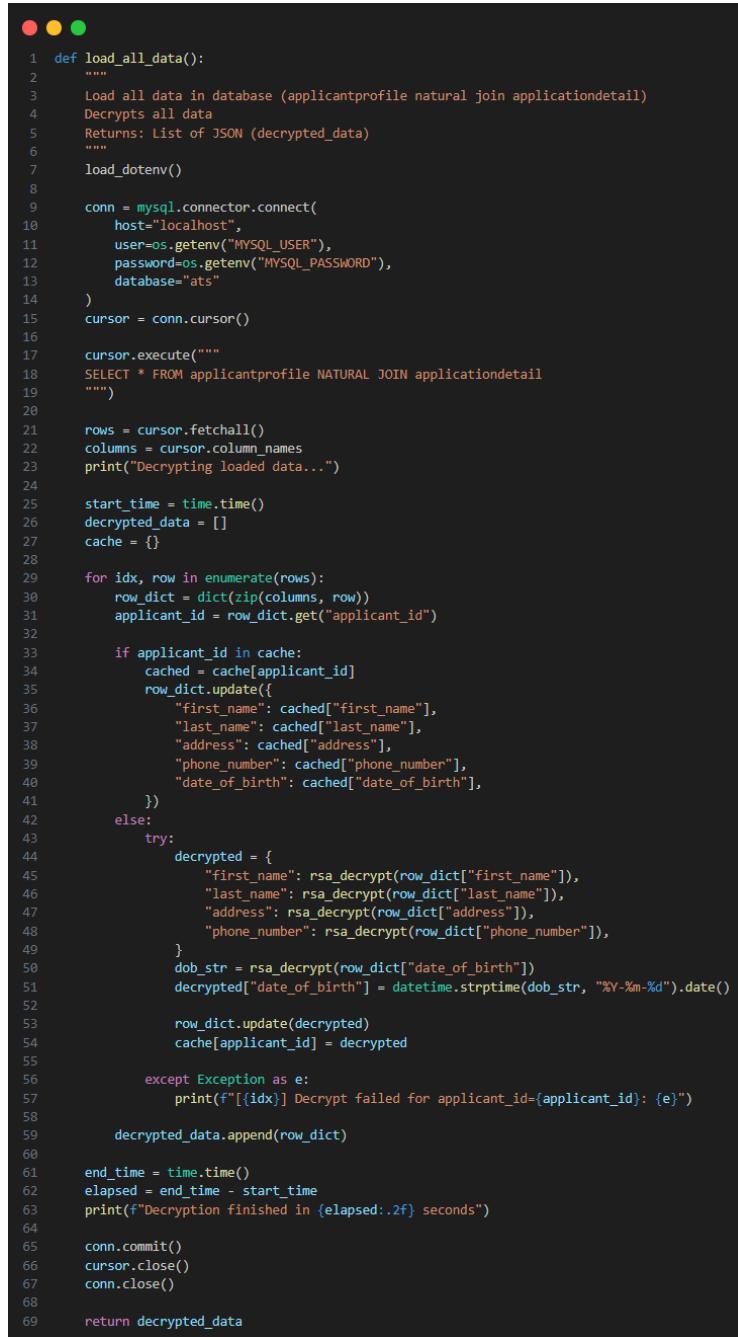


```
● ● ●
1 for row in rows:
2     applicant_id, first_name, last_name, dob, address, phone = row
3
4     cursor.execute("""
5         UPDATE ApplicantProfile
6             SET first_name = %s,
7                 last_name = %s,
8                     date_of_birth = %s,
9                         address = %s,
10                            phone_number = %s
11                            WHERE applicant_id = %s
12                            """, (
13        rsa_encrypt(first_name or ""),
14        rsa_encrypt(last_name or ""),
15        rsa_encrypt(dob or ""),
16        rsa_encrypt(address or ""),
17        rsa_encrypt(phone or ""),
18        applicant_id
19    ))
```

**Gambar 3.1.2** Proses enkripsi dalam fungsi seed\_data()

Seluruh data terenkripsi yang tersimpan di basis data (gabungan ApplicantProfile dan ApplicationDetail) dimuat ke dalam memori aplikasi saat inisialisasi menggunakan loader.py. Proses ini melibatkan eksekusi query SELECT \* FROM applicantprofile NATURAL JOIN applicationdetail. Kolom-kolom terenkripsi didekripsi kembali menggunakan fungsi rsa\_decrypt agar dapat dibaca oleh pengguna. Data yang sudah didekripsi ini kemudian menjadi cv\_dataset yang akan diproses lebih lanjut oleh algoritma pencarian. Agar mempercepat pemrosesan data, setiap entri pelamar yang telah didekripsi disimpan dalam cache sehingga tidak perlu didekripsi ulang. Hal ini penting karena proses dekripsi RSA bersifat mahal secara komputasi. Selain itu, penggunaan NATURAL JOIN menyebabkan satu pelamar dapat muncul dalam beberapa baris hasil query (karena memiliki lebih dari satu entri aplikasi), sehingga tanpa caching, proses

dekripsi akan dilakukan berulang kali untuk data yang sama, yang akan memperlambat inisialisasi sistem secara signifikan.



```
1 def load_all_data():
2     """
3         Load all data in database (applicantprofile natural join applicationdetail)
4         Decrypts all data
5         Returns: List of JSON (decrypted_data)
6     """
7     load_dotenv()
8
9     conn = mysql.connector.connect(
10         host="localhost",
11         user=os.getenv("MYSQL_USER"),
12         password=os.getenv("MYSQL_PASSWORD"),
13         database="ats"
14     )
15     cursor = conn.cursor()
16
17     cursor.execute("""
18         SELECT * FROM applicantprofile NATURAL JOIN applicationdetail
19     """)
20
21     rows = cursor.fetchall()
22     columns = cursor.column_names
23     print("Decrypting loaded data...")
24
25     start_time = time.time()
26     decrypted_data = []
27     cache = {}
28
29     for idx, row in enumerate(rows):
30         row_dict = dict(zip(columns, row))
31         applicant_id = row_dict.get("applicant_id")
32
33         if applicant_id in cache:
34             cached = cache[applicant_id]
35             row_dict.update({
36                 "first_name": cached["first_name"],
37                 "last_name": cached["last_name"],
38                 "address": cached["address"],
39                 "phone_number": cached["phone_number"],
40                 "date_of_birth": cached["date_of_birth"],
41             })
42         else:
43             try:
44                 decrypted = {
45                     "first_name": rsa_decrypt(row_dict["first_name"]),
46                     "last_name": rsa_decrypt(row_dict["last_name"]),
47                     "address": rsa_decrypt(row_dict["address"]),
48                     "phone_number": rsa_decrypt(row_dict["phone_number"]),
49                 }
50                 dob_str = rsa_decrypt(row_dict["date_of_birth"])
51                 decrypted["date_of_birth"] = datetime.strptime(dob_str, "%Y-%m-%d").date()
52
53                 row_dict.update(decrypted)
54                 cache[applicant_id] = decrypted
55
56             except Exception as e:
57                 print(f"[{idx}] Decrypt failed for applicant_id={applicant_id}: {e}")
58
59             decrypted_data.append(row_dict)
60
61     end_time = time.time()
62     elapsed = end_time - start_time
63     print(f"Decryption finished in {elapsed:.2f} seconds")
64
65     conn.commit()
66     cursor.close()
67     conn.close()
68
69     return decrypted_data
```

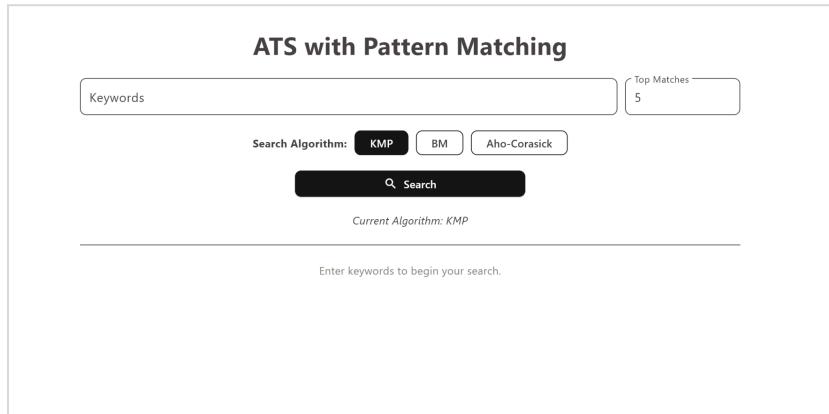
**Gambar 3.1.3** Fungsi load\_all\_data untuk memuat dan mendekripsi seluruh data dalam database ke memori aplikasi melalui loader.py.

## 2. Input Kata Kunci dan Top Matches

Pengguna (dalam kasus asli HR atau rekruter) dapat menginput satu atau lebih kata kunci (*keywords*) yang ingin dicari (dipisahkan koma) melalui kolom input di antarmuka. Selain itu, pengguna juga dapat menentukan jumlah CV teratas (Top Matches) yang ingin ditampilkan berdasarkan hasil pencocokan.

### 3. Pemilihan Algoritma String Matching

Pengguna memilih algoritma pencarian *exact matching* yang diinginkan: Knuth-Morris-Pratt (KMP) atau Boyer-Moore (BM), dan Aho-Corasick.



Gambar 3.1.4 Tampilan interface UI untuk penginputan

### 4. Ekstraksi Teks dan Pra-pemrosesan CV

Untuk setiap file CV yang disimpan dalam bentuk PDF (cv\_path di ApplicationDetail), sistem melakukan ekstraksi teks secara otomatis. Library PyPDF2 digunakan untuk membaca konten PDF dan mengubahnya menjadi satu string panjang yang memuat seluruh teks dari dokumen tersebut. Teks yang diekstrak kemudian diparse (misalnya, menghilangkan whitespace berlebih dan dikonversi ke lowercase) untuk standardisasi/normalisasi, yang kemudian siap untuk diproses dalam string matching dan ekstraksi informasi.

### 5. Proses Pencarian Kata Kunci (Pattern Matching):

Sistem menjalankan dua fase pencarian utama untuk setiap CV dalam cv\_dataset:

- **Fase 1: Exact Matching**

Algoritma yang dipilih oleh pengguna (KMP, BM, atau Aho-Corasick) digunakan untuk mencari kemunculan kata kunci secara persis (exact match) dalam string teks CV yang telah diproses.

KMP dan BM menganalisis teks secara efisien tanpa backtracking yang tidak perlu. Aho-Corasick mampu mencari banyak pola sekaligus dalam satu lintasan. Teori keseluruhan sudah lebih lengkap dijelaskan di Bab II Landasan Teori, dan fungsi serta struktur data spesifiknya akan dibahas di bab selanjutnya (Bab IV Implementasi dan Pengujian).

Waktu runtime yang dibutuhkan untuk proses exact matching dicatat dan akan ditampilkan sebagai metrik performa. Hasilnya mencakup jumlah kemunculan setiap kata kunci yang cocok secara exact.

- **Fase 2: Fuzzy Matching (Jika Exact Match Tidak Ditemukan)**

Apabila ada kata kunci yang tidak ditemukan satupun kemunculannya pada fase exact matching di suatu CV, sistem akan beralih ke fuzzy matching (*Levenshtein Distance*). Fuzzy matching merupakan teknik yang digunakan untuk mencari kesamaan antara data yang mungkin tidak sama persis, tetapi mewakili entitas yang sama dan memanfaatkan algoritma Levenshtein Distance (atau Edit Distance) untuk menghitung tingkat kemiripan antara kata kunci yang belum cocok dan keyword-keyword dalam teks CV.

Secara praktis, ini digunakan untuk mengatasi typo yang mungkin tidak terduga pada pencarian mau itu dari CV (biasanya) maupun dari input. Tingkat kemiripan itu akan dibandingkan dengan threshold yang sudah ditentukan. Jika tingkat kemiripan melebihi ambang batas tertentu (threshold dapat dikonfigurasi dan dituning sendiri), keyword tersebut dianggap sebagai fuzzy match. Waktu runtime yang dibutuhkan untuk proses fuzzy matching juga dicatat dan ditampilkan secara terpisah.

## 6. Ekstraksi Informasi Detail untuk Summary dengan Regular Expression (Regex)

Selain pencarian kata kunci, sistem juga menggunakan Regular Expression (Regex) untuk mengekstrak informasi penting dan terstruktur dari teks CV. Informasi yang diekstrak meliputi ringkasan pelamar (*summary*), keahlian, pengalaman kerja (tanggal dan jabatan), dan riwayat pendidikan (tanggal kelulusan, universitas, gelar). Informasi ini digunakan untuk membangun ringkasan profil pelamar yang akan ditampilkan di antarmuka pengguna.

**Tabel 3.1** Replacement character menjadi regex

Regex	Deskripsi
.	Semua karakter kecuali newline
^	Mencocokkan Awal String
\$	Mencocokkan Akhir String
\d,\w,\s	Digit,karakter [A-Za-z0-9],spasi
\D,\W,\S	Kecuali digit, karakter, dan spasi
[abc]	a atau b atau c
[a-z]	Dari karakter a sampai z
\,,\\,*	Karakteristik, backslash, dan bintang
aa   bb	aa atau bb
*	Pengulangan karakter sebelumnya sebanyak 0 atau lebih kali
+	Mencocokkan karakter sebelumnya setidaknya satu kali atau lebih.
?	Mencocokkan karakter sebelumnya setidaknya satu kali atau tidak sama sekali.

## 7. Tampilan Hasil (*Results*)

Setelah semua CV diproses, sistem menghitung total kecocokan (exact + fuzzy) untuk setiap CV. CV-CV ini kemudian diurutkan berdasarkan jumlah kecocokan kata kunci terbanyak. Hanya Top Matches yang telah ditentukan pengguna yang akan ditampilkan di antarmuka. Setiap hasil ditampilkan dalam bentuk card CV yang memuat nama kandidat, jumlah total kecocokan, dan ringkasan kata kunci yang cocok beserta frekuensinya, serta tombol untuk melihat CV atau melihat summary.

Gambar 3.1.5 Tampilan interface hasil pencarian berupa card hasil dan waktu runtime

### **3.2 Proses Pemetaan Masalah menjadi Elemen Algoritma KMP dan BM**

Implementasi sistem ATS berbasis CV digital memerlukan transformasi masalah pencarian informasi dalam dokumen tidak terstruktur menjadi permasalahan string matching yang dapat diselesaikan secara algoritmik. Bagian ini menjelaskan bagaimana karakteristik spesifik dari pencarian CV dipetakan ke dalam elemen-elemen kunci algoritma KMP dan Boyer-Moore.

#### **3.2.1 Abstraksi Masalah ATS**

Dalam konteks ATS, setiap CV yang telah diekstrak dari format PDF menjadi teks(T) dengan panjang n karakter, sementara kata kunci yang diinput pengguna menjadi pattern (P) dengan panjang m karakter. Proses pencarian CV yang relevan ditransformasi menjadi pencarian kemunculan pattern P dalam teks T, di mana setiap kemunculan yang ditemukan berkontribusi pada skor relevansi CV tersebut.

#### **3.2.2 Pemetaan Elemen Masalah ke Komponen Algoritma**

**Tabel 3.2.2** Pemetaan Elemen Masalah dalam Pemrosesan ATS

Aspect	Algoritma KMP	Algoritma Boyer-Moore
<b>Input Data</b>		
Representasi CV dalam format PDF untuk Pencarian	Teks T (target string)	Teks T (target string)
Kata kunci dari pengguna ( <i>Keywords</i> )	Pattern P (search string)	Pattern P (search string)
<b>Preprocessing</b>		
Persiapan algoritma	Pembangunan LPS array: O(m)	Bad Character & Good Suffix table: O(m +  Σ )
<b>Pencarian</b>		
Proses matching	Traversal kiri ke kanan	Traversal kiri ke kanan, pattern dibandingkan dari kanan ke kiri
Handling mismatch	Skip dan fallback berdasarkan LPS array	Skip dan fallback berdasarkan heuristic

		(cari last occurrence yang align jika ada)
<b>Output</b>		
Hasil pencarian	Hasil dan counter pattern yang match (exact & fuzzy)	Hasil dan counter pattern yang match (exact & fuzzy)
Runtime	Waktu dalam millisecond (ms)	Waktu dalam millisecond (ms)

### 3.3 Fitur Fungsional dan Arsitektur Aplikasi

#### 3.3.1 Basis Data

Dalam program ATS berbasis CV Digital ini, dimanfaatkan sistem manajemen basis data relasional (DBMS) MySQL/MariaDB untuk menyimpan data profil pelamar (*ApplicantProfile*) dan detail aplikasi mereka (*Application Detail*). Pemilihan ini didasarkan pada kompatibilitasnya yang tinggi dengan MySQL serta ketersediaan fitur yang memadai untuk kebutuhan aplikasi ini.

#### 3.3.2 Seeding

Seeding basis data adalah proses inisialisasi basis data dengan data awal atau contoh. Tujuan utama dari seeding adalah untuk mempersiapkan lingkungan pengembangan atau pengujian dengan data yang relevan, sehingga aplikasi dapat diuji fungsinya tanpa harus memasukkan data secara manual. Dalam konteks pengembangan sistem ATS ini, seeding data sangat penting untuk mengisi tabel *ApplicantProfile* dan *ApplicationDetail* dengan data CV digital pelamar. Pada proyek ini, proses seeding dapat dilakukan melalui dua pendekatan:

- **Seeding Resmi (`officialSeeder.py`)**

Metode seeding resmi melibatkan pemasukan data dari file SQL eksternal yang berisi data statis. Pendekatan ini digunakan untuk menyiapkan data dasar yang seragam, terutama menjelang pengumpulan tugas untuk demo.

- **Seeding Faker (`seeder.py`)**

Untuk seeding faker, pendekatannya menggunakan library Faker untuk men generate data palsu untuk berbagai keperluan, seperti pengujian, pembuatan prototipe, dan pengisian database.

### 3.3.3 Backend (Python)

Seluruh logika inti aplikasi dan backend dari aplikasi, termasuk pemrosesan data, implementasi algoritma string matching, regex, dan interaksi dengan database, diimplementasikan menggunakan bahasa pemrograman Python. Python dipilih karena ekosistemnya yang kaya akan library dan efisien untuk berbagai keperluan. Library yang digunakan untuk mengembangkan aplikasi ini sebagai berikut:

- mysql.connector: untuk berinteraksi dengan database MariaDB (yang *compatible* dengan MySQL). Dengan library ini, aplikasi dapat menjalankan berbagai operasi database seperti mengeksekusi query SQL (SELECT, INSERT, UPDATE, DELETE), mengambil data, dan mengelola transaksi secara langsung dari program Python.
- dotenv: untuk memuat variabel lingkungan (environment variables) sehingga informasi tersebut tidak perlu ditulis langsung di dalam kode program, sehingga membuat aplikasi lebih aman dan fleksibel.
- PyPDF2: untuk ekstraksi teks dari dokumen CV berformat PDF, mengubahnya menjadi satu string panjang yang siap diproses oleh algoritma string matching
- re (regex): untuk mengaktifkan *regular expressions* (regex) sebagai library pendukung pencocokan pola string.

### 3.3.4 Flet

Antarmuka pengguna aplikasi desktop ini dikembangkan menggunakan Flet, sebuah framework modern berbasis Python yang memungkinkan pembuatan aplikasi GUI (Graphical User Interface) secara intuitif, cepat, dan interaktif.

Flet dibangun di atas Flutter (framework UI buatan Google), namun dengan pendekatan yang tidak memerlukan penulisan kode Dart. Dengan Flet, developer cukup menggunakan Python, sehingga mempercepat proses pengembangan.

## 3.4 Contoh ilustrasi kasus

### 3.4.1 Pencarian Single Keyword dengan KMP (Exact Match)

- a. Aplikasi baru dijalankan

Pada saat aplikasi pertama kali dijalankan, dilakukan inisialisasi sistem yang mencakup inisialisasi GUI, ATSPProcessor, setup interface, dan load

database yang mencakup query database, deskripsi semua data yang terenkripsi, dan caching hasil deskripsi. Sehingga dihasilkan variabel cv\_dataset berupa list dictionary yang menyimpan data pelamar dan application mereka (Seperti yang telah dijelaskan pada bagian 3.1. Langkah-langkah Pemecahan Masalah).

b. Input User

- i. Keyword: “Python”
- ii. Top Matches: 5
- iii. Algoritma: Knuth-Morris-Pratt (KMP)

c. Proses

i. *Inisialisasi Data*

Keyword yang didapatkan diparsing menjadi list of keyword dan diubah menjadi lowercase: [“python”]

ii. *Preprocessing KMP*

Tiap CV dalam dataset diekstraksi (extract\_pdf\_match.py) menjadi long string untuk diproses. Kemudian, dilakukan perhitungan LPS Array (KMP Preprocessing), sehingga didapatkan: [0, 0, 0, 0, 0, 0] karena “python” tidak memiliki LPS.

iii. *Ekstraksi dan Pencarian*

Selanjutnya, dilakukan KMP *search* untuk setiap CV yang menghasilkan list posisi “python” ditemukan pada CV.

Misalnya pada pencarian salah satu CV ditemukan 4 kecocokan: found\_indexes = [52, 87, 125, 150].

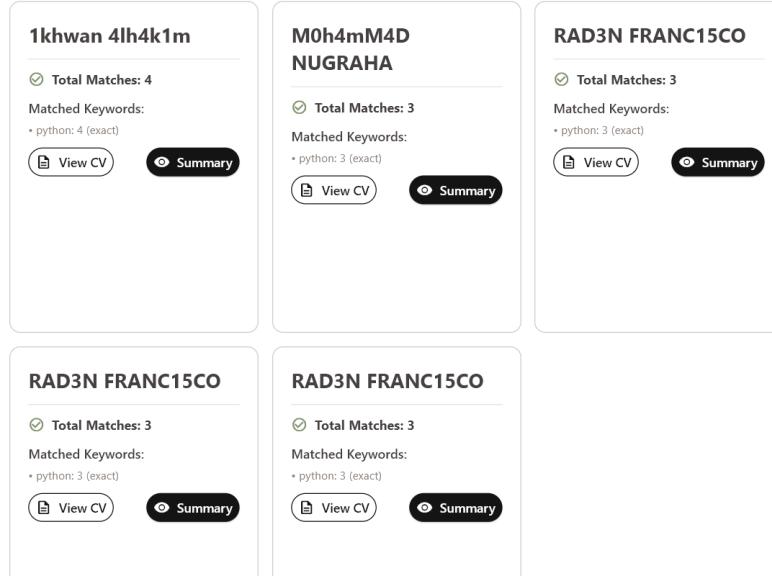
iv. *Hasil Pencarian*

Kemudian, hasil pencarian KMP tersebut diakumulasi untuk tiap CV berupa ‘match\_count’. Pada contoh CV sebelumnya, maka match\_count = 4. Lalu dari masing-masing CV, diurutkan berdasarkan ‘match\_count’ terbanyak, dan diambil sebanyak top\_matches, yaitu 5 teratas. Misalnya pada kasus ini didapatkan 100 kecocokan, maka 100 CV tersebut akan diurutkan secara descending dan diambil 5 CV dengan ‘match\_count’ terbanyak. Pada pemrosesan ini telah menemukan 5 top\_results, sehingga tidak dilakukan lagi pencarian secara fuzzy.

v. *Output Display*

Pada tahap sebelumnya, top\_results tidak hanya menyimpan data CV, tetapi juga menyimpan data pribadi *applicant*, *application\_detail*, nama *applicant*, match\_count, dan summary

tiap top CV. Sehingga sistem mengubah data top\_results tersebut menjadi card-card hasil pencarian.



Gambar 3.4.1. Ilustrasi Tampilan Output Pencarian Single Keyword dengan KMP (Exact Match).

### 3.4.2 Pencarian Multiple Keywords dengan Boyer-Moore (Exact & Fuzzy)

#### a. Aplikasi baru dijalankan

Sama seperti kasus sebelumnya, dilakukan inisialisasi sistem yang mencakup inisialisasi GUI, ATSPProcessor, setup interface, dan load database. Dihasilkan variabel cv\_dataset berupa list dictionary yang menyimpan data pelamar dan application mereka.

#### b. Input user

- Keyword: "Python, Machine Learning, Django"
- Top Matches: 5
- Algoritma: Boyer-Moore (BM)

#### c. Proses

- Inisialisasi data

Keywords yang didapatkan diparsing menjadi list of keywords dan diubah menjadi lowercase: ["python", "machine learning", "django"]

- Preprocessing Boyer-Moore

Tiap CV dalam dataset diekstraksi (extract\_pdf\_match.py) menjadi long string untuk diproses. Kemudian, untuk setiap keyword dilakukan preprocessing Boyer-Moore:

- Keyword "python":

Bad Character Table: {'p': 0, 'y': 1, 't': 2, 'h': 3, 'o': 4, 'n': 5}  
- Keyword "machine learning":

Bad Character Table: {'m': 0, 'a': 1, 'c': 2, 'h': 3, 'i': 4, 'n': 5, 'e': 6, ' ': 7, 'l': 8, 'e': 9, 'a': 10, 'r': 11, 'n': 12, 'i': 13, 'n': 14, 'g': 15}

- Keyword "django":

Bad Character Table: {'d': 0, 'j': 1, 'a': 2, 'n': 3, 'g': 4, 'o': 5}

### iii. Ekstraksi dan Pencarian

Selanjutnya, dilakukan Boyer-Moore search untuk setiap keyword di setiap CV. Boyer-Moore melakukan pencarian dari kiri ke kanan pada text, namun membandingkan pattern dari kanan ke kiri.

Misalnya untuk salah satu CV:

- "python": found\_indexes = [52, 87, 125] → 3 matches
- "machine learning": found\_indexes = [201, 289] → 2 matches
- "django": found\_indexes = [340] → 1 match

Total exact matches untuk CV ini = 3 + 2 + 1 = 6 matches.

### iv. Hasil Pencarian Exact

Hasil pencarian Boyer-Moore diakumulasi untuk tiap CV berupa 'match\_count'. Misalnya setelah memproses seluruh CV dataset (50 CV), didapatkan hasil seperti:

- CV Alice Wonderland: 6 matches (python: 3, machine learning: 3, django: 0)
- CV Bob Smith: 4 matches (python: 2, machine learning: 2, django: 0)
- CV Carol Davis: 2 matches (python: 1, machine learning: 1, django: 0)
- CV Mamang Racing: 1 match (python: 1, machine learning: 0, django: 0)
- 46 CV lainnya: 0 matches

Karena hanya 4 CV yang memiliki exact matches, sedangkan user meminta top 5, maka sistem akan melanjutkan ke fuzzy matching.

### v. Fuzzy Matching

Sistem melakukan fuzzy matching menggunakan Levenshtein Distance terhadap keyword yang tidak ditemukan dengan threshold 0.65:

Contoh fuzzy matching pada CV Emma Watson:

- "django" → tidak ditemukan exact, cek fuzzy:
  - "jango" (similarity: 0.83) → MATCH

Total fuzzy matches untuk CV Emma Watson = 1 match.

vi. Hasil Akhir

Setelah menggabungkan exact dan fuzzy results, diurutkan berdasarkan total matches:

1. CV Alice Wonderland: 6 matches (exact)
2. CV Bob Smith: 4 matches (exact)
3. CV Carol Davis: 2 matches (exact)
4. CV Mamang Racing: 1 match (exact)
5. CV Emma Watson: 1 match (fuzzy)

vii. Output Display

Sistem mengubah data top\_results menjadi card-card hasil pencarian yang menampilkan kombinasi exact dan fuzzy matches (2 cards contoh):

1. Card Alice Wonderland:  
Total Matches: 6
  - python: 3 (exact)
  - machine learning: 2 (exact)
  - django: 1 (exact)
2. Card Emma Watson:  
Total Matches: 1
  - 'jango': 1 (fuzzy for: django)

Status Display:

Found 5 relevant CVs.

Exact Match: 50 CVs scanned in 187ms.

Fuzzy Match: 50 CVs scanned in 543ms.

### 3.4.3 Pencarian dengan Typo menggunakan AHO-Corasick + Fuzzy

a. Aplikasi baru dijalankan

Sama seperti kasus sebelumnya, dilakukan inisialisasi sistem dengan load database dan setup interface.

b. Input user

- i. Keyword: "Javascript, Recat, Phyton" (mengandung typo)
- ii. Top Matches: 3
- iii. Algoritma: Aho-Corasick

c. Proses

- i. Inisialisasi data

Keywords yang didapatkan diparsing menjadi list of keywords dan diubah menjadi lowercase: ["javscript", "recat", "phyton"]

- ii. Preprocessing Aho-Corasick

Sistem membangun Finite State Automaton (FSA) untuk multiple pattern matching. Build Trie (Goto Function):

Sistem membangun trie dengan struktur berikut:

- Root (State 0) memiliki 3 cabang utama: j, r, dan p
- Cabang j: j-a-v-s-c-r-i-p-t (State 1-9) untuk "javascript"
- Cabang r: r-e-c-a-t (State 10-14) untuk "react"
- Cabang p: p-h-y-t-o-n (State 15-20) untuk "phyton"

Setiap akhir kata ditandai sebagai final state yang menunjukkan pattern lengkap ditemukan.

Build Failure Links:

Failure links mengarah kembali ke state yang sesuai ketika terjadi mismatch, memungkinkan pencarian multiple pattern dalam satu pass.

iii. Ekstraksi dan pencarian

Aho-Corasick melakukan pencarian untuk semua pattern sekaligus dalam satu traversal text. Misalnya pada salah satu CV:

Hasil Pencarian Exact (Aho-Corasick):

- "javascript": found\_indexes = [] → 0 matches (typo tidak ditemukan)
- "react": found\_indexes = [] → 0 matches (typo tidak ditemukan)
- "phyton": found\_indexes = [] → 0 matches (typo tidak ditemukan)

Total exact matches = 0 (karena semua keyword mengandung typo)

iv. Fuzzy matching untuk typo

Karena tidak ada exact matches ditemukan, sistem langsung melakukan fuzzy matching untuk semua keyword:

Contoh fuzzy matching pada CV John Developer:

Untuk "javscript" (typo dari "javascript"):

- Text n-grams: ["javascript", "java", "script", "js", ...]
- "javascript" vs "javscript" → similarity: 0.90 → MATCH
- "js" vs "javscript" → similarity: 0.22 → NO MATCH

Untuk "recat" (typo dari "react"):

- Text n-grams: ["react", "reactjs", "reactive", ...]
- "react" vs "recat" → similarity: 0.80 → MATCH
- "reactjs" vs "recat" → similarity: 0.57 → NO MATCH

Untuk "phyton" (typo dari "python"):

- Text n-grams: ["python", "pythonic", "py", ...]

- "python" vs "phyton" → similarity: 0.83 → MATCH
- "pythonic" vs "phyton" → similarity: 0.63 → NO MATCH

Total fuzzy matches untuk CV John Developer = 3 matches.

v. Hasil akhir

Setelah memproses seluruh dataset dengan fuzzy matching:

- CV John Developer: 3 matches (javascript, react, python via fuzzy)
- CV Saya Frontend: 2 matches (javascript, react via fuzzy)
- CV Dya Backend: 1 match (python via fuzzy)

vi. Output display

Sistem menampilkan hasil dengan keterangan fuzzy matches yang menunjukkan koreksi typo (contoh card John dan Saya):

1. Card John Developer:

Total Matches: 3

- 'javascript': 1 (fuzzy for: javascript)
- 'react': 1 (fuzzy for: recat)
- 'python': 1 (fuzzy for: phyton)

2. Card Saya Frontend:

Total Matches: 2

- 'javascript': 1 (fuzzy for: javscript)
- 'react': 1 (fuzzy for: recat)

Status Display:

Found 3 relevant CVs.

Exact Match: 50 CVs scanned in 98ms.

Fuzzy Match: 50 CVs scanned in 1247ms.

# BAB IV

## IMPLEMENTASI DAN PENGUJIAN

### 4.1 Spesifikasi Teknis Program

#### 4.1.1 Struktur Data, Fungsi, dan Prosedur

Pada implementasi, berbagai struktur data dibuat untuk mendukung pengembangan dan efisiensi dari aplikasi, termasuk kelas-kelas yang bertanggung jawab terhadap pengorganisasian logika, algoritma dan pengelolaan data:

##### 1. Database & Seeding



```
# Drop existing tables
cursor.execute("DROP TABLE IF EXISTS ApplicationDetail;")
cursor.execute("DROP TABLE IF EXISTS ApplicantProfile;")
print("Existing tables dropped.")

# Make tables
cursor.execute("""
CREATE TABLE IF NOT EXISTS ApplicantProfile (
    applicant_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    first_name TEXT DEFAULT NULL,
    last_name TEXT DEFAULT NULL,
    date_of_birth TEXT DEFAULT NULL,
    address TEXT DEFAULT NULL,
    phone_number TEXT DEFAULT NULL
);
""")

cursor.execute("""
CREATE TABLE IF NOT EXISTS ApplicationDetail (
    detail_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    applicant_id INT NOT NULL,
    application_role VARCHAR(100) DEFAULT NULL,
    cv_path TEXT,
    FOREIGN KEY (applicant_id) REFERENCES ApplicantProfile(applicant_id)
);
""")

print("Tables created successfully.")
cursor.close()
conn.close()

if __name__ == "__main__":
    create_tables()
```

Gambar 4.1.1.1. Fungsi create\_tables()

Code ini membuat database jika belum ada, membuat table database (tetapi drop table jika ada yang sama).

##### 2. Algoritma KMP (Knuth-Morris-Pratt)

Algoritma KMP menggunakan Longest Proper Prefix (LPS) array untuk menghindari pemeriksaan ulang karakter yang sudah cocok sebelumnya. Ketika terjadi mismatch, algoritma akan "melompat" ke posisi yang tepat berdasarkan informasi LPS.

```

def compute_lps(self, pattern) -> list:
    """
    Menghitung tabel LPS (Longest Proper Prefix)
    untuk pattern yang diberikan
    """
    m = len(pattern)
    lps = [0] * m
    length = 0 # panjang prefix terpanjang sebelumnya
    i = 1

    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1

    return lps

```

LPS array menyimpan panjang prefix terpanjang yang juga merupakan suffix. Contoh: untuk pattern "ABABCABAB", LPS = [0, 0, 1, 2, 0, 1, 2, 3, 4]. Digunakan untuk menentukan seberapa jauh kita bisa "melompat" saat terjadi mismatch.

```

● ● ●

def kmp_search(self, text, pattern) -> list:
    """
    Implementasi algoritma KMP untuk mencari semua kemunculan
    pattern dalam text
    """
    if not pattern:
        return []

    n = len(text)
    m = len(pattern)

    if m == 0:
        return []

    # Compute LPS array
    lps = self.compute_lps(pattern)

    found_indexes = []
    i = 0 # index text
    j = 0 # index pattern

    while i < n:
        if text[i] == pattern[j]:
            i += 1
            j += 1

            if j == m:
                # Pattern found
                start_index = i - j
                found_indexes.append(start_index)
                j = lps[j - 1]
        elif i < n and text[i] != pattern[j]:
            # Mismatch
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return found_indexes

```

Pada algoritma KMP ini, kompleksitas waktu adalah  $O(n + m)$  - linear time complexity. Tidak ada backtracking, sehingga setiap karakter di text hanya diperiksa sekali. Cocok untuk pemrosesan CV karena efisien untuk mencari multiple keywords dalam text panjang.

Implementasi:

- LPS Array: Menghitung tabel prefix terpanjang untuk menghindari pengecekan ulang
- Pencarian: Membandingkan karakter dari kiri ke kanan, menggunakan LPS untuk skip saat mismatch
- Hasil: Mengembalikan list indeks dimana pattern ditemukan

### 3. Algoritma BM(Boyer-Moore)

Kelas BM\_ATS mengimplementasikan algoritma pencarian string Boyer-Moore dengan heuristik bad character, yang sangat efisien dalam menemukan kemunculan pola (pattern) di dalam sebuah teks (text). Fungsi preprocess\_bad\_char(pattern) menyiapkan tabel bad character, yang menyimpan posisi terakhir kemunculan setiap karakter dalam pattern. Tabel ini digunakan untuk mempercepat pencarian dengan menentukan seberapa jauh pattern dapat digeser saat terjadi mismatch.

Fungsi ini menjalankan pencarian Boyer-Moore dan mengembalikan list indeks tempat pattern ditemukan dalam teks. Proses dimulai dengan pengecekan apakah pattern kosong atau lebih panjang dari teks; jika iya, maka langsung dikembalikan nilai 0 karena tidak mungkin ada kecocokan. Setelah itu, tabel bad character dibentuk dengan memanggil fungsi preprocess\_bad\_char, yang menyimpan posisi terakhir dari setiap karakter dalam pattern. Pencarian kemudian dilakukan dengan mencocokkan pattern terhadap text dari kanan ke kiri. Jika ditemukan karakter yang tidak cocok, algoritma akan mencari posisi terakhir karakter tersebut di dalam pattern, lalu menghitung jarak geser menggunakan rumus  $shift = j - last\_occurrence$ . Pattern kemudian digeser ke kanan sejauh  $\max(1, shift)$  untuk memastikan kemajuan dan mencegah infinite loop. Jika semua karakter cocok (artinya  $j < 0$ ), maka indeks saat itu dimasukkan ke hasil, dan pattern digeser sejauh panjangnya untuk mencari kemunculan selanjutnya.

```

gantim@Gantim-OptiPlex-5070:~/bmipy/bm_ATS$ python bm_search.py
  File "bm_search.py", line 1
    class BM_ATS:
      ^
SyntaxError: invalid syntax

```

```

class BM_ATS:
    def __init__(self):
        pass

    def preprocess_bad_char(self, pattern) -> dict:
        """
        Membuat tabel bad character yang berisi posisi terakhir
        dari setiap karakter dalam pattern
        """
        bad_char = {}
        for i in range(len(pattern)):
            bad_char[pattern[i]] = i
        return bad_char

    def bm_search(self, text, pattern) -> list:
        """
        Implementasi Boyer-Moore menggunakan bad character heuristic.
        Mengembalikan jumlah kemunculan pattern dalam text.
        """
        if not pattern:
            return 0

        n = len(text)
        m = len(pattern)

        if m == 0 or m > n:
            return 0

        bad_char = self.preprocess_bad_char(pattern)

        found_indexes = []
        i = 0

        while i <= n - m:
            j = m - 1

            while j >= 0 and pattern[j] == text[i + j]:
                j -= 1

            if j < 0:
                found_indexes.append(i)
                i += m
            else:
                mismatch_char = text[i + j]
                last_occurrence = bad_char.get(mismatch_char, -1)
                shift = j - last_occurrence
                i += max(1, shift)

        return found_indexes

```

## Algoritma Boyer Moore

### 4. Algoritma Aho-Corasick

Implementasi:

- Trie Construction: Membangun trie dari semua pattern
- Failure Links: Membuat link fallback untuk efisiensi
- Multi-Search: Mencari semua pattern dalam satu traversal

```

    ● ● ●

def __init__(self, words):
    # filter alphanumeric
    self.words = []
    for word in words:
        # Keep only alphanumeric characters and spaces, convert to lowercase
        cleaned_word = re.sub(r'[^\\w\\s]', '', word.lower()).strip()
        if cleaned_word: # Only add non-empty words
            self.words.append(cleaned_word)

    # Guess how many 'nodes' or 'states' we might need (sum of word lengths)
    self.max_states = sum([len(word) for word in words])
    self.max_characters = 26

    # Final state (if a state marks end of a complete word), use bitmask to store
    self.out = [0]*(self.max_states+1)

    # Failure link
    self.fail = [-1]*(self.max_states+1)

    # From a state, given a character, where do we go next?
    self.goto = [[-1]*self.max_characters for _ in range(self.max_states+1)]

def build_matching(self) -> int:
    # ===== 1. Build the basic Trie (main paths) =====
    k = len(self.words)
    states = 1 # root

    for i in range(k):
        word = self.words[i]
        current_state = 0

        for character in word:
            ch = self.char_to_index(character)

            # if no path for curr char -> create new state
            if self.goto[current_state][ch] == -1:
                self.goto[current_state][ch] = states
                states += 1
            current_state = self.goto[current_state][ch]

        # Flag state as end of word (word ke i ends here)
        self.out[current_state] |= (1<<i)

    # ===== 2. Build the failure link (fallback paths) =====
    queue = [] # BFS tiap state
    for ch in range(self.max_characters):
        # Liat semua yang bisa di reach dari root, set fallback ke root
        if self.goto[0][ch] != 0:
            self.fail[self.goto[0][ch]] = 0
            queue.append(self.goto[0][ch])

    while queue:
        state = queue.pop(0)
        for ch in range(self.max_characters):
            # For each state, cari failure link dari semua children-nya
            if self.goto[state][ch] != -1:
                failure = self.fail[state]

                while self.goto[failure][ch] == -1:
                    failure = self.fail[failure]

                failure = self.goto[failure][ch]
                self.fail[self.goto[state][ch]] = failure
                self.out[self.goto[state][ch]] |= self.out[failure]
                queue.append(self.goto[state][ch])

    return states

def search_words(self, text):
    """
    Search for matching words in text
    """
    if not self.words:
        return defaultdict(list)

    text = re.sub(r'[^\\w\\s]', ' ', text.lower())
    current_state = 0
    result = defaultdict(list)

    for i in range(len(text)):
        current_state = self.find_next_state(current_state, text[i])

        # If this state doesn't mark the end of any word, just keep going.
        if self.out[current_state] == 0:
            continue

        # Final state, check matching words
        for j in range(len(self.words)):
            # Check bitmask flag. Does the flag for the j-th word exist here?
            if (self.out[current_state] & (1<<j)) > 0:
                word = self.words[j]

                result[word].append(i-len(word)+1)
    return result

```

Algoritma AHO yang diimplementasikan:

- Membangun Trie untuk menyimpan semua patterns
- Failure links menghubungkan states untuk optimasi pencarian
- Menggunakan bitmask untuk menandai multiple patterns yang berakhir di state yang sama
- Dapat mencari semua patterns dalam satu kali traversal text

## 5. Fuzzy Matcher (Levenshtein)

Class FuzzyMatcher dipakai dalam program ini sebagai pendukung dan penanggung jawab logika dari *fuzzy matching* yang menerapkan perhitungan levenshtein distance atau *edit distance*.

Class ini memiliki atribut kelas:

- **threshold (float)**: Nilai ambang batas similaritas (default 0.65). Atribut ini digunakan untuk menentukan apakah dua string dianggap mirip atau tidak dalam proses fuzzy matching, di mana nilai yang lebih tinggi menandakan kriteria pencocokan yang lebih ketat. Nilai 0.65 dipilih sebagai threshold berdasarkan testing dan tuning sendiri dengan pertimbangan berapa batas typo atau levenshtein distance yang masih ditoleransi untuk kemiripan sebuah kata.

```
class FuzzyMatcher:  
    def __init__(self, threshold = 0.65):  
        self.threshold = threshold;
```

Fungsi init class FuzzyMatcher

Fungsi **levenshtein\_distance** merupakan fungsi utama yang dipakai dalam perhitungan logika fuzzy matching. Fungsi ini menerima parameter word1 dan word2 berupa string, sehingga fungsi ini menghitung jarak edit suatu kata dengan kata pembanding. Algoritma Dynamic Programming dipakai dalam perhitungan fungsi ini, dan menghitung jarak edit di mana jarak edit berupa jumlah minimum operasi dari insert (penambahan), delete (penghapusan), dan replace (penggantian) antara dua string. Fungsi ini

mengembalikan edit distance dalam bentuk integer yang kemudian akan dipakai pada fungsi calculate\_similarity.

```
● ○ ●
1 def levenshtein_distance(self, word1, word2):
2     ...
3     Menghitung levenshtein distance/ edit distance
4     ...
5     row, col = len(word1), len(word2)
6     cache = [[float("inf")] * (col + 1) for i in range(row + 1)]
7
8     for j in range (col + 1):
9         cache[row][j] = col - j
10    for i in range (row + 1):
11        cache[i][col] = row - i
12
13    for i in range(row - 1, -1, -1):
14        for j in range (col - 1, -1, -1):
15            if word1[i] == word2[j]:
16                cache[i][j] = cache[i+1][j+1]
17            else:
18                cache[i][j] = 1 + min(
19                    cache[i + 1][j], #delete
20                    cache[i][j+1], #insert
21                    cache[i+1][j+1] #replace
22                )
23
24    return cache[0][0]
```

Fungsi **calculate\_similarity** merupakan fungsi yang berfungsi untuk mengkonversi edit distance menjadi skor similaritas. Fungsi ini menerima parameter word1 dan word2 berupa string yang akan dibandingkan tingkat kemiripannya. Proses perhitungan dilakukan dengan memanggil fungsi levenshtein\_distance untuk mendapatkan edit distance, kemudian menghitung panjang maksimum dari kedua string. Skor similaritas dihitung menggunakan formula  $1 - (\text{distance} / \text{max\_length})$  sehingga menghasilkan nilai float antara 0.0 hingga 1.0, di mana nilai yang lebih tinggi menandakan tingkat kemiripan yang lebih tinggi.

```
● ● ●
1 def calculate_similarity(self, word1, word2):
2     ...
3     Menghitung score similaritynya berdasarkan levenshtein distance
4     ...
5     # cth
6     # word1 -> keyword
7     # word2 -> candidate n gram dari CV
8
9     distance = self.levenshtein_distance(word1, word2)
10    max_len = max(len(word1), len(word2))
11
12    if max_len == 0:
13        return 1.0
14
15    similarity = 1 - (distance / max_len)
16    return similarity
17
```

Fungsi `get_ngrams` dipakai sebagai method static dengan parameter `text(string)` dan `n(int)` lalu mengeluarkan result berupa ngram dalam bentuk list of string. Fungsi ini memiliki tujuan untuk menangani input keyword yang berupa banyak kata (dipisahkan whitespace). Ngrams merupakan struktur data berupa list yang menyimpan n-gram dari teks input. List ini berisi string-string yang merupakan kombinasi n kata berturut-turut yang digunakan untuk proses pencocokan fuzzy.

Sebagai contoh, keyword “React Native” berupa 2 kata namun tetap termasuk sebagai 1 keyword. Oleh karena itu, fungsi ini dipanggil oleh `fuzzy_search()` yang akan menerima keyword dan mencari panjang dari keyword (berapa kata), melalui: `n = max(1, len(keyword.split()))`. Keyword yang hanya berupa satu kata akan memiliki `n=1`, sedangkan yang banyak akan menyesuaikan `n` dengan panjang keywordnya.

Setelah itu, di fungsi `get_ngrams` dilakukan split tanpa menghilangkan white space sehingga array berupa words atau 1 string 1 kata. Kemudian, Result merupakan hasil array di mana kombinasi n kata berturut-turut dibuat. Contohnya, “React Native American” dengan `n=2` akan membandingkan text “React Native” dan “Native American” dalam CV dan mencari kecocokan tiap 2 kata.

```
1  @staticmethod
2  def get_ngrams(text, n):
3      ...
4      Mengambil n-gram dari teks (consecutive words), regex only alphanumeric
5      ...
6      # remove non-alphanumeric, space/tab/newline masih ada
7      text = re.sub(r'[^w\s]', '', text)
8
9      words = text.split()
10
11     result = [' '.join(words[i:i+n]) for i in range(len(words)-n+1)]
12     # n=1 -> "React Native American" -> ["react", "native", "american"]
13     # n=2 -> ["react native", "native american"]
14
15     return result
```

Fungsi **fuzzy\_search** merupakan fungsi utama untuk melakukan pencarian fuzzy matching dalam teks CV. Fungsi ini menerima parameter keyword berupa string yang dicari, cv\_text berupa string teks CV yang akan dicari, dan threshold berupa float optional untuk ambang similaritas. Proses dalam fungsi ini dimulai dengan menentukan ukuran n-gram berdasarkan jumlah kata dalam keyword, kemudian menggenerate n-gram dari cv\_text. Setiap n-gram dibandingkan dengan keyword menggunakan calculate\_similarity, dan hasil yang memenuhi threshold disimpan dalam list matches. Fungsi ini juga menangani pencocokan untuk keyword yang mengandung spasi dengan mencari individual words. Akhirnya, hasil diurutkan berdasarkan skor similaritas dan duplikasi dihapus, kemudian fungsi mengembalikan tuple berupa (count, matches).

```

1  def fuzzy_search(self, keyword, cv_text, threshold=None):
2      """
3          Find count & matching phrases,
4          Returns (count, matches) where matches is list of (similarity, phrase) tuples
5          ...
6      if threshold is None:
7          threshold = self.threshold
8
9      n = max(1, len(keyword.split()))
10
11     ngrams = self.get_ngrams(cv_text, n)
12
13
14     matches = []
15
16     for phrase in ngrams:
17         similarity = self.calculate_similarity(keyword, phrase)
18         if similarity >= threshold:
19             matches.append((similarity, phrase))
20
21     if ' ' in keyword:
22         clean_keyword = keyword.replace(' ', '')
23         single_words = self.get_ngrams(cv_text, 1) # individual words
24
25         for word in single_words:
26             similarity = self.calculate_similarity(clean_keyword, word)
27             if similarity >= threshold:
28
29                 # Check if this match is already covered to avoid duplicates
30                 is_duplicate = any(word in existing_match[1] for existing_match in matches)
31                 if not is_duplicate:
32                     matches.append((similarity, word))
33
34     matches.sort(reverse=True, key=lambda x: x[0])
35
36     unique_matches = []
37     seen_phrases = set()
38     for sim, phrase in matches:
39         if phrase not in seen_phrases:
40             unique_matches.append((sim, phrase))
41             seen_phrases.add(phrase)
42
43     return len(unique_matches), unique_matches

```

## 6. ATSProcessor

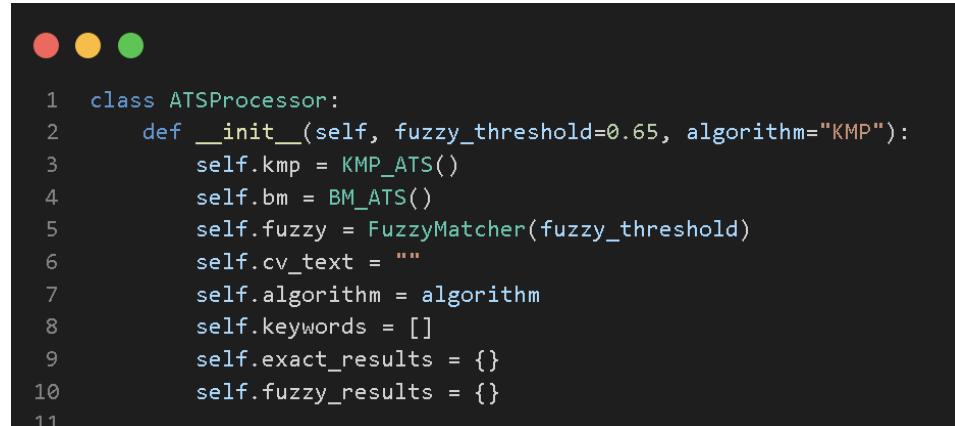
Class ini memiliki atribut kelas:

- **kmp**, **bm** merupakan instance algoritma KMP dan Boyer-Moore untuk exact matching.

- **fuzzy** merupakan instance FuzzyMatcher untuk fuzzy matching.
- **cv\_text** merupakan string teks CV yang sedang diproses.
- **algorithm** merupakan string nama algoritma yang dipilih ("KMP", "BM", "Aho-Corasick").
- **keywords** merupakan list keyword yang akan dicari.
- **exact\_results, fuzzy\_results** merupakan dictionary yang menyimpan hasil pencarian dengan struktur {keyword: {count, matches}}.

ATSProcessor menggunakan berbagai struktur data untuk mengelola proses pencarian dalam sistem ATS. Instance kmp dan bm menyimpan algoritma KMP dan Boyer-Moore untuk exact matching, sedangkan fuzzy menyimpan instance FuzzyMatcher untuk fuzzy matching. Cv\_text berupa string yang menyimpan teks CV yang sedang diproses, sementara algorithm menyimpan string nama algoritma yang dipilih seperti "KMP", "BM", atau "Aho-Corasick". Keywords berupa list yang berisi keyword yang akan dicari, dan exact\_results serta fuzzy\_results merupakan dictionary yang menyimpan hasil pencarian dengan struktur {keyword: {count, matches}}.

Proses ATS dimulai dengan fungsi init yang menginisialisasi ATSProcessor dengan parameter fuzzy\_threshold dan algorithm, sekaligus membuat instance semua algoritma pencarian yang diperlukan. Fungsi set\_algorithm kemudian dapat digunakan untuk mengubah algoritma exact matching dengan validasi input dan default ke "KMP" jika tidak valid.



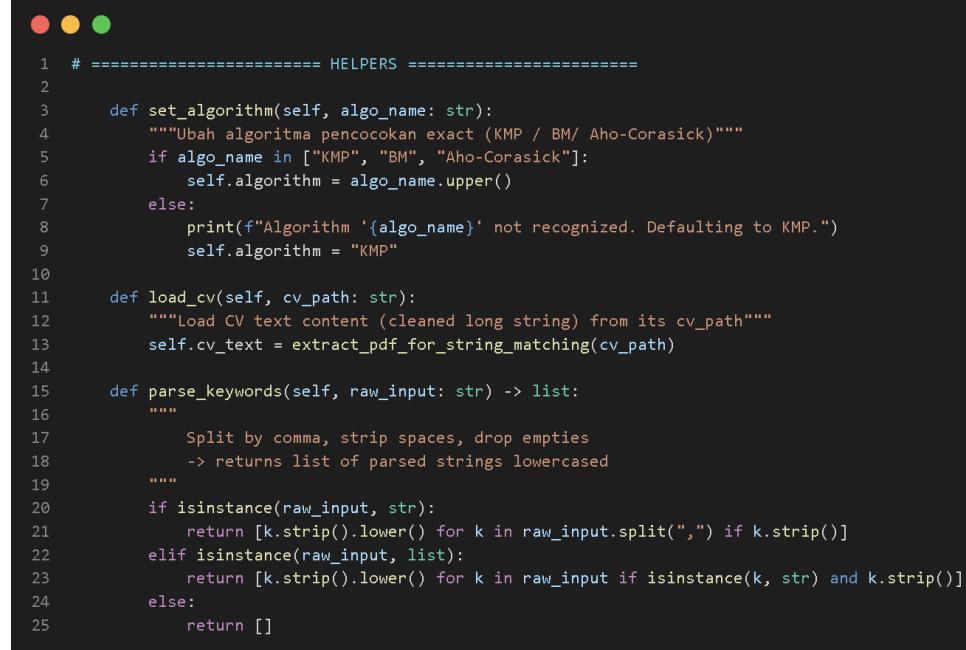
```

1  class ATSPProcessor:
2      def __init__(self, fuzzy_threshold=0.65, algorithm="KMP"):
3          self.kmp = KMP_ATS()
4          self.bm = BM_ATS()
5          self.fuzzy = FuzzyMatcher(fuzzy_threshold)
6          self.cv_text = ""
7          self.algorithm = algorithm
8          self.keywords = []
9          self.exact_results = {}
10         self.fuzzy_results = {}
11

```

Untuk memproses data, fungsi load\_cv memuat teks CV dari file PDF menggunakan extract\_pdf\_for\_string\_matching dan menyimpan dalam

cv\_text, sementara fungsi parse\_keywords memproses input keyword dengan memisahkan berdasarkan koma, membersihkan whitespace, dan mengembalikan list keyword lowercase. Fungsi print\_results berfungsi menampilkan hasil exact dan fuzzy matching dalam format terstruktur dengan total count dan detail setiap keyword.



```
1 # ===== HELPERS =====
2
3     def set_algorithm(self, algo_name: str):
4         """Ubah algoritma pencocokan exact (KMP / BM/ Aho-Corasick)"""
5         if algo_name in ["KMP", "BM", "Aho-Corasick"]:
6             self.algorithm = algo_name.upper()
7         else:
8             print(f"Algorithm '{algo_name}' not recognized. Defaulting to KMP.")
9             self.algorithm = "KMP"
10
11    def load_cv(self, cv_path: str):
12        """Load CV text content (cleaned long string) from its cv_path"""
13        self.cv_text = extract_pdf_for_string_matching(cv_path)
14
15    def parse_keywords(self, raw_input: str) -> list:
16        """
17            Split by comma, strip spaces, drop empties
18            -> returns list of parsed strings lowercased
19        """
20        if isinstance(raw_input, str):
21            return [k.strip().lower() for k in raw_input.split(",") if k.strip()]
22        elif isinstance(raw_input, list):
23            return [k.strip().lower() for k in raw_input if isinstance(k, str) and k.strip()]
24        else:
25            return []
```

Proses pencarian dilakukan dalam dua tahap utama melalui fungsi search\_exact yang melakukan pencarian exact matching menggunakan algoritma terpilih dan mengembalikan tuple (total\_exact, found\_exact\_keywords), dilanjutkan dengan fungsi search\_fuzzy yang melakukan fuzzy matching untuk keyword yang belum ditemukan exact match dan mengembalikan total fuzzy matches. Seluruh proses terintegrasi dalam fungsi get\_top\_search\_results yang mengintegrasikan proses pencarian dengan dua fase (exact lalu fuzzy), memproses seluruh CV dataset, dan mengembalikan tuple (top\_results, exact\_match\_time, fuzzy\_match\_time) berupa ranking CV terbaik yang sesuai dengan keyword yang dicari.

```

1 # ===== SEARCH =====
2
3     def search_exact(self) -> dict:
4         """
5             Algo untuk cari exact match
6
7             Returns:
8                 dict: (total_exact, found_exact_keywords)
9             """
10            # self.keywords = self.parse_keywords(keywords)
11            self.exact_results = {}
12
13            # AHO
14            if self.algorithm == "Aho-Corasick":
15                if not self.keywords:
16                    return (0, 0)
17
18                aho_instance = AHO_ATS(self.keywords)
19                found_matches = aho_instance.search_words(self.cv_text)
20                for keyword in self.keywords:
21                    indices = found_matches.get(keyword, [])
22                    if indices:
23                        self.exact_results[keyword] = {
24                            'count': len(indices),
25                            'matches': [keyword] * len(indices)
26                        }
27
28                found_exact_keywords = set(self.exact_results.keys())
29                total_exact = sum(res.get('count', 0) for res in self.exact_results.values())
30
31                print(f"Total exact matches: {total_exact}")
32
33            return (total_exact, found_exact_keywords)
34
35            # KMP/BM
36        else:
37            for keyword in self.keywords:
38                indexes = []
39
40                if self.algorithm == "BM":
41                    indexes = self.bm.bm_search(self.cv_text, keyword)
42                else:
43                    indexes = self.kmp.kmp_search(self.cv_text, keyword)
44
45                count = len(indexes)
46                if count > 0:
47                    self.exact_results[keyword] = {
48                        'count': count,
49                        'matches': [keyword] * count
50                    }
51
52            # simpen exact match
53            found_exact_keywords = self.exact_results.keys()
54            total_exact = sum(res.get('count', 0) for res in self.exact_results.values())
55
56            print(f"Total exact matches: {total_exact}")
57            return (total_exact, found_exact_keywords)

```

Gambar 4.2.2. Fungsi search\_exact untuk pencarian KMP/BM/AHO

```

def search_fuzzy(self, found_exact_keywords) -> dict:
    """Algo untuk cari fuzzy match"""
    # self.keywords = self.parse_keywords(keywords)
    self.fuzzy_results = {}

    for keyword in self.keywords:
        if keyword not in found_exact_keywords:
            fuzzy_count, fuzzy_matches = self.fuzzy.fuzzy_search(keyword, self.cv_text, self.fuzzy.threshold)
            if fuzzy_count > 0:
                self.fuzzy_results[keyword] = {
                    'count': fuzzy_count,
                    'matches': fuzzy_matches
                }

    total_fuzzy = sum(res.get('count', 0) for res in self.fuzzy_results.values())

    print(f"Total fuzzy matches: {total_fuzzy}")
    return (total_fuzzy)

```

Gambar 4.2.3. Fungsi search\_fuzzy untuk pencarian fuzzy matching

## 7. Enkripsi (RSA)

Dalam sistem ATS ini, RSA digunakan untuk mengenkripsi data sensitif seperti nama, alamat, tanggal lahir, dan nomor telepon pelamar sebelum disimpan ke dalam basis data. RSA (Rivest–Shamir–Adleman) adalah algoritma kriptografi kunci publik, yang berarti ia menggunakan pasangan kunci: satu kunci publik untuk mengenkripsi dan satu kunci privat untuk mendekripsi.

```

def generate_keypair(bits=512):
    """Generate RSA public and private keypair, print as .env entries"""
    p = generate_prime(bits)
    q = generate_prime(bits)
    while p == q:
        q = generate_prime(bits)

    n = p * q
    phi = (p - 1) * (q - 1)

    e = 65537
    if gcd(e, phi) != 1:
        e = 3
        while gcd(e, phi) != 1:
            e += 2

    d = pow(e, -1, phi)

    print("== COPY THE FOLLOWING INTO YOUR .env FILE ===\n")
    print(f"PUBLIC_N={n}")
    print(f"PUBLIC_E={e}")
    print()
    print(f"PRIVATE_N={n}")
    print(f"PRIVATE_D={d}")
    print("\n=====")

```

Gambar 4.2.4 Fungsi generate keypair

Fungsi rsa\_encrypt menerima plaintext dan mengenkripsi setiap karakter satu per satu menggunakan rumus: ''.join(str(pow(ord(char), PUBLIC\_E, PUBLIC\_N)) for char in plaintext). Setiap karakter dikonversi ke bilangan bulat dengan ord(), lalu dienkripsi, dan hasilnya disusun sebagai string angka-angka terenkripsi, dipisahkan spasi. Sebaliknya, fungsi rsa\_decrypt mengambil string terenkripsi, memisahkannya berdasarkan spasi, lalu menerapkan rumus: ''.join(chr(pow(int(chunk), PRIVATE\_D, PRIVATE\_N)) for chunk in ciphertext.split())).

```

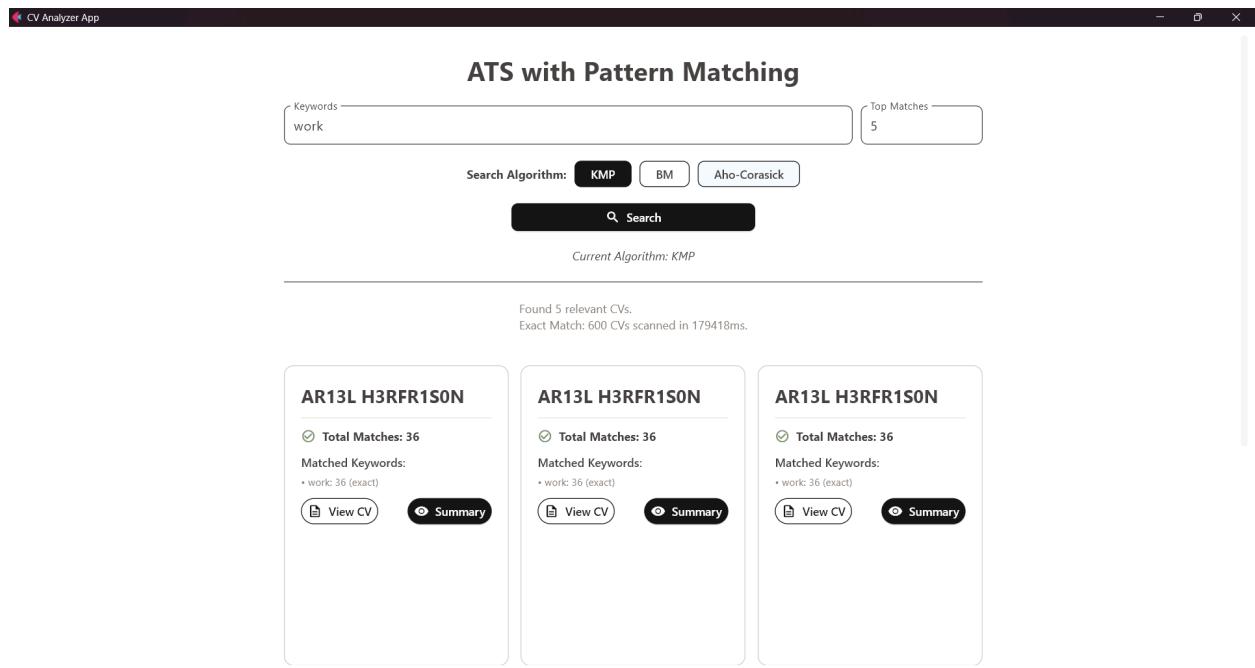
def rsa_encrypt(plaintext: str) -> str:
    """
    Encrypt plaintext string using RSA public key from .env
    Returns space-separated string of encrypted integers
    """
    return ' '.join(str(pow(ord(char), PUBLIC_E, PUBLIC_N)) for char in plaintext)

def rsa_decrypt(ciphertext: str) -> str:
    """
    Decrypt space-separated string of encrypted integers using RSA private key from .env
    Returns original plaintext string
    """
    return ''.join(chr(pow(int(chunk), PRIVATE_D, PRIVATE_N)) for chunk in ciphertext.split())

```

#### 4.1.2 Interface dan Fitur-fitur yang Disediakan Program

##### i. GUI (Main)



Gambar 4.1.2.1. GUI Main Page

##### ii. GUI (Summary Page)

The screenshot shows the 'CV Summary' page of the CV Analyzer App. At the top, there is a header with a back arrow and the title 'CV Summary'. Below the header, the profile ID 'AR13L H3RFR1SON' is displayed. Underneath the profile ID, there is a section for personal information: 'Birthdate: 03-08-2003', 'Address: Jl. Sawo No. 11, Depok', and 'Phone: 082154321789'. A 'Skills' section follows, featuring four circular buttons labeled 'Accounting', 'General Accounting', 'Accounts Payable', and 'Program Management'. The 'Job History' section is also visible.

Gambar 4.1.2.2. GUI Summary Page (Profil, Skills, Job History)

The screenshot shows the 'Education' section of the CV Analyzer App. It displays a single educational entry for 'USA Emphasis in Business' from 'College 1994 Associate' in 1994. The entry includes a brief description of the program and its requirements, such as Accounting City, State, USA GPA: GPA: 3.41 174 Hours, Quarter Attended Husson College, major Accounting 78 semester hours toward Bachelors degree, Professional Military Comptroller School, 6wk, 4-98; Managerial Accounting I, 09-98; Interest-Based Bargaining Training for Management, 24hrs, 09-01; Auditing Methods and Concepts 09-98; Organizational Leadership, 32hrs, 07-03; Management Development II, 32hrs, 07-03.

Gambar 4.1.2.3. GUI Summary Page (Education)

iii.

## 4.2 Hasil Pengujian (*Testing*)

### 4.2.1 Pengujian RSA

Pengujian ini dilakukan untuk menunjukkan hasil enkripsi dan lama dekripsi ketika fungsi `load_all_data()` dipanggil.

**Tabel 4.2.1.1** Pengujian hasil enkripsi

Tujuan	Menunjukkan lima baris dari tabel applicantprofile
Query	<code>SELECT * FROM applicantprofile LIMIT 5;</code>
Hasil	

**Tabel 4.2.1.2** Pengujian lama dekripsi

Tujuan	Menunjukkan lama dekripsi
Perintah	<code>\$ uv run src/main.py</code>
Hasil	<pre>\$ uv run src/main.py Decrypting loaded data... Decryption finished in 13.75 seconds</pre>

### 4.2.2 Pengujian Berdasarkan Tipe Algoritma

**Tabel 4.2.2.1** Nomor pengujian dengan Inputnya

ID	Key	Top Matches
1	react	1
2	ReAct	2

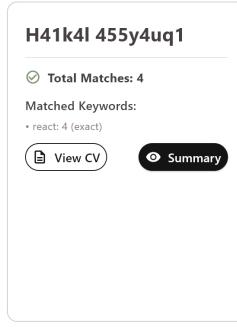
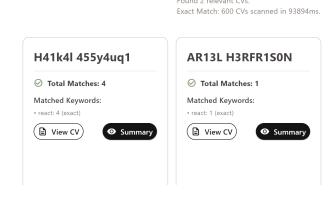
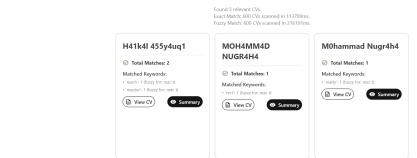
3	r e a c t	4
4	recat	1
5	rEaC t	3
6	react, python	5

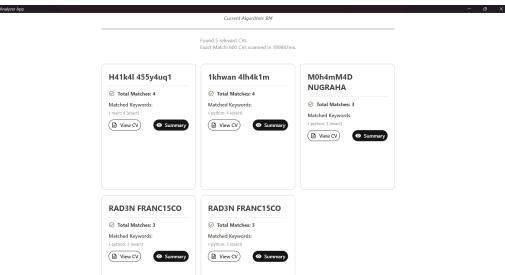
**Tabel 4.2.2.2 Pengujian KMP**

ID	Runtime (s)	Found CV	Tampilan
1	69.876	1	<p>Found 1 relevant CVs. Exact Match: 600 CVs scanned in 69.87633633613586ms.</p> <div style="border: 1px solid #ccc; padding: 10px; width: fit-content;"> <b>H41k4I 455y4uq1</b>  <input type="radio"/> Total Matches: 4            Matched Keywords:            • react: 4 (exact)  <a href="#">View CV</a> <a href="#">Summary</a> </div>
2	68.884	2	<p>Found 2 relevant CVs. Exact Match: 600 CVs scanned in 68.84431433677673ms.</p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid #ccc; padding: 10px; width: 45%;"> <b>H41k4I 455y4uq1</b>  <input type="radio"/> Total Matches: 4            Matched Keywords:            • react: 4 (exact)  <a href="#">View CV</a> <a href="#">Summary</a> </div> <div style="border: 1px solid #ccc; padding: 10px; width: 45%;"> <b>AR13L H3RFR1SON</b>  <input type="radio"/> Total Matches: 1            Matched Keywords:            • react: 1 (exact)  <a href="#">View CV</a> <a href="#">Summary</a> </div> </div>

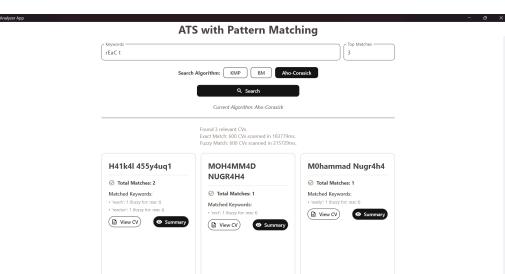
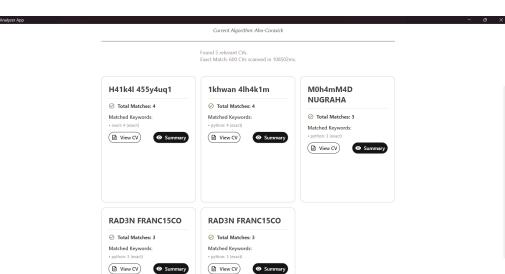
3	69.266 + 99.777	4	<p>Found 4 relevant CVs. Exact Match: 600 CVs scanned in 69.26681113243103ms. Fuzzy Match: 600 CVs scanned in 99.77702760696411ms.</p> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <b>H41k4l 455y4uq1</b>  <input type="radio"/> Total Matches: 2            Matched Keywords:            • 'reach': 1 (fuzzy for: r e a c t)            • 'reactor': 1 (fuzzy for: r e a c t)  <a href="#">View CV</a> <a href="#">Summary</a> </div> <div style="text-align: center;"> <b>MOH4MM4D NUGR4H4</b>  <input type="radio"/> Total Matches: 1            Matched Keywords:            • 'rect': 1 (fuzzy for: r e a c t)  <a href="#">View CV</a> <a href="#">Summary</a> </div> <div style="text-align: center;"> <b>M0hammad Nugr4h4</b>  <input type="radio"/> Total Matches: 1            Matched Keywords:            • 'reality': 1 (fuzzy for: r e a c t)  <a href="#">View CV</a> <a href="#">Summary</a> </div> </div> <div style="text-align: center;"> <b>Mohammad Nugr4h4</b>  <input type="radio"/> Total Matches: 1            Matched Keywords:            • 'reality': 1 (fuzzy for: r e a c t)  <a href="#">View CV</a> <a href="#">Summary</a> </div>
4	71.080 + 78.853	1	<p>Found 1 relevant CVs. Exact Match: 600 CVs scanned in 71.08024168014526ms. Fuzzy Match: 600 CVs scanned in 78.85399413108826ms.</p> <div style="text-align: center;"> <b>Moh4mm4d Nu9r4h4</b>  <input type="radio"/> Total Matches: 1            Matched Keywords:            • 'repeat': 1 (fuzzy for: recat)  <a href="#">View CV</a> <a href="#">Summary</a> </div>
5	69.843 + 84.510	3	<p>Found 3 relevant CVs. Exact Match: 600 CVs scanned in 69.84334397315979ms. Fuzzy Match: 600 CVs scanned in 84.51010274867085ms.</p> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <b>H41k4l 455y4uq1</b>  <input type="radio"/> Total Matches: 2            Matched Keywords:            • 'reach': 1 (fuzzy for: react)            • 'reactor': 1 (fuzzy for: react)  <a href="#">View CV</a> <a href="#">Summary</a> </div> <div style="text-align: center;"> <b>MOH4MM4D NUGR4H4</b>  <input type="radio"/> Total Matches: 1            Matched Keywords:            • 'rect': 1 (fuzzy for: react)  <a href="#">View CV</a> <a href="#">Summary</a> </div> <div style="text-align: center;"> <b>M0hammad Nugr4h4</b>  <input type="radio"/> Total Matches: 1            Matched Keywords:            • 'reality': 1 (fuzzy for: react)  <a href="#">View CV</a> <a href="#">Summary</a> </div> </div>
6	78.557	5	<p>Found 5 relevant CVs. Exact Match: 600 CVs scanned in 78.55797338485718ms.</p> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <b>H41k4l 455y4uq1</b>  <input type="radio"/> Total Matches: 4            Matched Keywords:            • 'reach': 4 (exact)  <a href="#">View CV</a> <a href="#">Summary</a> </div> <div style="text-align: center;"> <b>1khwan 4lh4k1m</b>  <input type="radio"/> Total Matches: 4            Matched Keywords:            • 'python': 4 (exact)  <a href="#">View CV</a> <a href="#">Summary</a> </div> <div style="text-align: center;"> <b>M0h4mM4D NUGRAHA</b>  <input type="radio"/> Total Matches: 3            Matched Keywords:            • 'python': 3 (exact)  <a href="#">View CV</a> <a href="#">Summary</a> </div> </div> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <b>RAD3N FRANC15CO</b>  <input type="radio"/> Total Matches: 3            Matched Keywords:            • 'python': 3 (exact)  <a href="#">View CV</a> <a href="#">Summary</a> </div> <div style="text-align: center;"> <b>RAD3N FRANC15CO</b>  <input type="radio"/> Total Matches: 3            Matched Keywords:            • 'python': 3 (exact)  <a href="#">View CV</a> <a href="#">Summary</a> </div> </div>

**Tabel 4.2.2.3 Pengujian BM**

ID	Runtime (ms)	Found CV	Tampilan
1	99450	1	<p>Found 1 relevant CVs. Exact Match: 600 CVs scanned in 99450ms.</p> 
2	93894	2	<p>ATS with Pattern Matching</p> <p>Keywords: ReAct   Top Matches: 2</p> <p>Search Algorithm: KMP   BM   Aho-Corasick   Current Algorithm: BM</p> <p>Found 2 relevant CVs. Exact Match: 600 CVs scanned in 93894ms.</p> 
3		4	
4		1	
5	113789 + 216191	3	<p>ATS with Pattern Matching</p> <p>Keywords: ReAct   Top Matches: 3</p> <p>Search Algorithm: KMP   BM   Aho-Corasick   Current Algorithm: BM</p> <p>Found 1 relevant CVs. Exact Match: 600 CVs scanned in 113789ms. Exact Match: 600 CVs scanned in 216191ms.</p> 

6	189847	5	
---	--------	---	--

**Tabel 4.2.2.4 Pengujian Aho-Corasick**

ID	Runtime (ms)	Found CV	Tampilan
1		1	
2		2	
3		4	
4		1	
5	183779 + 215729	3	
6	108502	5	

### 4.3 Analisis Hasil Pengujian

#### 4.1.1 Kompleksitas Waktu dan Ruang

Algoritma	Preprocessing	Searching	Space Complexity
KMP	$O(m)$	$O(n)$	$O(m)$
BM	$O(m + \sigma)$	$O(nm)$ worst, $O(n/m)$ best	$O(m + \sigma)$
Aho	$O(\Sigma m)$	$O(n + z)$	$O(\Sigma m)$

#### 4.2.2. Perbandingan Kinerja Algoritma KMP vs Boyer-Moore vs Aho-Corasick

Dalam implementasi sistem ATS, ketiga algoritma menunjukkan karakteristik yang berbeda. KMP menawarkan kompleksitas linear  $O(n+m)$  yang konsisten dengan menggunakan LPS array untuk menghindari backtracking, namun harus melakukan loop terpisah untuk setiap keyword. Boyer-Moore memberikan performa praktis terbaik untuk single pattern dengan kemampuan skip karakter menggunakan bad character heuristic, tetapi memiliki worst case  $O(nm)$  dan tidak efisien untuk multiple keywords. Aho-Corasick unggul dalam multi-pattern search dengan kompleksitas  $O(n+z)$  untuk semua keywords sekaligus, sangat cocok untuk ATS yang biasanya mencari 3-10 keywords bersamaan meskipun membutuhkan memory overhead lebih besar untuk trie structure.

Berdasarkan implementasi kode, Aho-Corasick optimal untuk sistem ATS karena dapat memproses semua keywords dalam satu traversal text menggunakan automaton yang dibangun sekali. Boyer-Moore efektif untuk pencarian tunggal pada CV dengan natural language, sementara KMP memberikan konsistensi yang reliable namun tidak memiliki keunggulan khusus dalam konteks ATS.

Fuzzy matching menggunakan Levenshtein distance dengan kompleksitas  $O(|text| \times |keyword|^2)$  per keyword, dijalankan secara conditional hanya untuk keywords yang tidak exact match. Threshold 0.65 yang digunakan memberikan balance optimal antara precision dan recall - cukup toleran untuk typo umum seperti "react" vs "recat" tetapi tidak menghasilkan terlalu banyak false positives.

Impact fuzzy matching sangat tergantung distribusi exact matches: jika banyak exact matches ditemukan, fuzzy hanya berkontribusi ~10% processing time, namun bisa mencapai 80% jika exact matches sedikit. Strategi conditional execution dalam implementasi current efektif mengoptimalkan performa dengan menjalankan fuzzy search hanya jika hasil exact matching belum mencukupi target top\_n.

Aho-Corasick sebagai primary algorithm optimal untuk ATS dengan kemampuan multi-pattern search. Threshold 0.65-0.75 direkomendasikan untuk balance

precision-recall yang baik. Implementasi hybrid approach current dengan exact matching first dan conditional fuzzy matching sudah optimal untuk production use, memberikan efisiensi maksimal dengan akurasi yang acceptable untuk HR review.

## **BAB V**

### **KESIMPULAN DAN SARAN**

#### **5.1 Kesimpulan**

Implementasi sistem Applicant Tracking System (ATS) berbasis CV digital ini diselesaikan sebagai Tugas Besar 3 IF2211 Strategi Algoritma. Sistem ini memanfaatkan algoritma pattern matching seperti Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM) untuk pencarian kata kunci, serta algoritma Levenshtein Distance untuk pencarian fuzzy matching guna mengatasi kesalahan pengetikan. Program berhasil memenuhi seluruh spesifikasi wajib, termasuk ekstraksi informasi penting dari CV PDF menggunakan Regular Expression (Regex), penyimpanan data pada basis data MySQL, serta tampilan antarmuka pengguna (GUI) yang intuitif untuk pencarian dan menampilkan ringkasan CV.

Dari hasil implementasi, dapat disimpulkan bahwa penggunaan algoritma pattern matching seperti KMP dan BM sangat efektif dalam menyaring ribuan dokumen CV secara otomatis dan cepat untuk menemukan kandidat yang relevan berdasarkan kata kunci. Algoritma Aho-Corasick juga dikembangkan sebagai metode multi-pattern matching yang lebih efisien dan cepat, memungkinkan pencarian seluruh daftar kata kunci sekaligus dalam satu proses. Fitur fuzzy matching dengan Levenshtein Distance terbukti krusial untuk meningkatkan relevansi hasil pencarian, bahkan ketika terdapat perbedaan minor atau kesalahan penulisan pada input pengguna (*typo*). Selain itu, kemampuan sistem untuk mengkonversi CV PDF menjadi teks panjang dan mengekstrak informasi penting secara otomatis menjadi fondasi yang kuat bagi sistem ATS modern, mempercepat identifikasi dan penilaian awal pelamar. Pengukuran waktu eksekusi untuk exact match dan fuzzy match juga memberikan gambaran yang jelas mengenai efisiensi algoritma yang digunakan.

#### **5.2 Saran**

Program ini dapat dikembangkan lebih lanjut untuk meningkatkan fungsionalitas dan pengalaman pengguna.. Untuk peningkatan keamanan data, fitur enkripsi pada data profil pelamar dalam basis data dapat diterapkan menggunakan skema enkripsi yang lebih kompleks dan aman, tanpa bergantung pada pustaka bawaan Python. Terakhir, pengembangan fitur visualisasi atau dashboard yang lebih interaktif untuk menampilkan statistik pencarian, tren keahlian, atau analisis data pelamar secara lebih mendalam akan sangat bermanfaat. Selain itu, User Interface/User Experience (UI/UX) pada tampilan ringkasan dan detail CV juga dapat diperindah sehingga memberikan pengalaman yang lebih baik bagi pengguna.

### **5.3 Refleksi**

Menurut penulis, Tugas Besar 3 Strategi Algoritma ini sangat menambah wawasan dan pengalaman, terutama dalam memahami cara kerja ATS karena benar-benar dipakai dalam kehidupan nyata di dunia pekerjaan. Selain itu, tugas besar ini juga menambah wawasan tentang algoritma string matching yang tersedia, maupun dari segi teknis seperti lebih mengerti database, regex, GUI, dan Python secara keseluruhan. Ketika dihadapkan pada berbagai pilihan algoritma seperti KMP, Boyer-Moore, dan Aho-Corasick, penulis belajar menimbang efisiensi waktu, kompleksitas implementasi, dan kesesuaian algoritma terhadap konteks penggunaan secara nyata. Ini tidak hanya memperluas wawasan tentang teori algoritma, tapi juga menekankan pentingnya pemilihan strategi yang tepat dalam pemrograman praktis.

Proyek ini mengajarkan bahwa membuat program itu ga cuma sekadar bisa jalan, tapi juga harus efisien, mudah dipakai, dan bisa dikembangkan nanti. Tantangan seperti ini persis seperti yang akan dihadapi di dunia kerja profesional, di mana kualitas sistem dan bagaimana pengguna merasakan aplikasi kita itu sama pentingnya dengan algoritma canggih di baliknya.

Selain itu, kerja sama tim dan bagaimana kita mengatur waktu jadi kunci sukses di proyek ini. Penulis jadi sadar kalau pembagian tugas yang jelas dan komunikasi yang lancar itu sangat membantu menyelesaikan tugas besar ini tepat waktu. Kalau dikerjakan sendiri, tentu kemampuan mengatur prioritas dan beban kerja jadi pelajaran berharga yang tidak kalah penting.

Singkatnya, penulis menganggap Tugas Besar ini bukan hanya soal menyelesaikan sesuatu untuk nilai, tapi lebih kepada pengalaman menyeluruh dalam membangun sistem yang “hidup” dan bisa benar-benar dipakai dalam kehidupan sehari-hari. Selain itu, yang tidak kalah penting juga belajar berpikir logis, belajar membangun sistem dari nol, dan belajar untuk terus memperbaiki solusi yang telah dibuat. Dan mungkin yang paling penting, tugas ini menanamkan rasa percaya diri bahwa dengan pendekatan yang tepat dan semangat eksploratif, tantangan keos apa pun bisa ditaklukkan.

## **BAB VI**

## **LAMPIRAN**

### **6.1 Link Repository**

[Github Repository Link](#)

### **6.2 Link Bonus Video (YouTube)**

<https://youtu.be/7wuYzfCoU-c>

### **6.3 Tabel Hasil**

No	Poin	Ya	Tidak
1	Aplikasi dapat dijalankan.	✓	
2	Aplikasi menggunakan basis data berbasis SQL dan berjalan dengan lancar.	✓	
3	Aplikasi dapat mengekstrak informasi penting menggunakan Regular Expression (Regex).	✓	
4	Algoritma <i>Knuth-Morris-Pratt (KMP)</i> dan <i>Boyer-Moore (BM)</i> dapat menemukan kata kunci dengan benar.	✓	
5	Algoritma Levenshtein Distance dapat mengukur kemiripan kata kunci dengan benar.	✓	
6	Aplikasi dapat menampilkan <i>summary CV applicant</i> .	✓	
7	Aplikasi dapat menampilkan <i>CV applicant</i> secara keseluruhan.	✓	
8	Membuat laporan sesuai dengan spesifikasi.	✓	
9	Membuat bonus enkripsi data profil <i>applicant</i> .	✓	
10	Membuat bonus algoritma Aho-Corasick.	✓	
11	Membuat bonus video dan diunggah pada Youtube.	✓	

