# Penerapan Algoritma String Matching untuk Mencari Kemiripan Audio

Indraswara Galih Jayanegara - 13522119
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522119@std.stei.itb.ac.id, indrswaragalihjayanegara@gmail.com

***Abstrak*—Audio merupakan sebuah suara atau kebisingan yang bisa didengar oleh telinga manusia. Sinyal audio diukur dalam hertz dan di dalam komputer dihasilkan *sound card* dan bisa didengar melalui *speakers* atau *headphones*. Data pada komputer biasa direpresentasikan dengan *binary* yaitu, angka 1 dan 0 tidak terkecuali audio sehingga dari sebuah audio bisa diketahui representasi *binary*. Dalam Makalah ini, penulis akan membandingkan audio menggunakan algoritma *String Matching* yang membandingkan ASCII dari sebuah audio yang diambil dari representasi *binary*. Algoritma yang dipakai sendiri ada Boyer Moore dan–Morris–Pratt.**

***Kata Kunci—Audio; String; Boyer Moore; KMP; String Matching; Brute Force; Algoritma***

## I. PENDAHULUAN

Audio adalah sebuah hal yang sangat lumrah dan tak terpisahkan dari kehidupan kita pada zaman ini. Kegunaan dari audio sangatlah banyak dan bervariasi, sehingga tanpa audio, banyak dari apa yang kita gunakan dan nikmati saat ini akan terasa hambar dan kurang bermakna. Sebagai contoh, musik: tanpa adanya audio, musik tidak akan ada dan tidak bisa dinikmati oleh kita. Platform seperti YouTube juga sangat bergantung pada audio; tanpa audio, kita hanya akan melihat gambar yang bergerak tanpa ada konteks suara yang menyertainya, sehingga pengalaman menonton menjadi kurang lengkap. Begitu juga dengan komunikasi melalui smartphone atau telepon: tanpa adanya audio, kita tidak mungkin bisa melakukan komunikasi verbal, karena suara adalah elemen kunci dalam percakapan dan penyampaian pesan. Oleh karena itu, audio memainkan peran yang sangat penting dalam berbagai aspek kehidupan modern, mulai dari hiburan hingga komunikasi, dan tanpanya, banyak hal akan kehilangan esensinya.

## II. TEORI DASAR

### A. Algoritma String Matching

*String Matching* adalah algoritma yang digunakan untuk melakukan pencarian kecocokan dari suatu *pattern* dengan *teks* lain. Pencarian pola ini sangat penting dalam berbagai aplikasi mulai dari pencarian teks dalam dokumen, Forensik Digital, Pemerisa Ejaan, Filter Spam, Mesin Pencari, Sistem Deteksi Intrusi, dan juga Bioinformatika untuk pencarian kecocokan DNA. Terdapat beberapa algoritma *string matching* yang paling terkenal yaitu algoritma *Boyer Moore*, *Brute Force*, dan *Knuth–Morris–Pratt* untuk selanjutnya akan disingkat menjadi KMP. Algoritma *Boyer Moore* dan *KMP* hanya melakukan *Brute Force*, tetapi dengan cara yang cerdas, tidak seperti *Brute Force* yang hanya membandingkan tiap kararakter yang ada tanpa melakukan konsiderasi apapun. Pada makalah kali ini hanya akan digunakant tiga algoritma yaitu *Boyer Moore*m *Knuth–Morris–Pratt* (KMP), dan *Brute Force*.

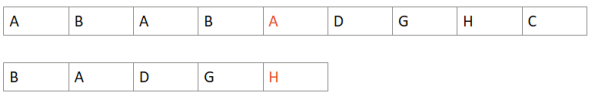
### B. Boyer Moore

Algoritma *Boyer-Moore* adalah salah satu algoritma *String Matching* yang paling efisien dalam pencarian kecocokan *String*. Algoritma *Boyer-Moore* dipublikasikan oleh Robert S. Boyer, dan J. Strother Moore. Tidak seperti algoritma KMP dan *Brute Force* perbandingan pada *Boyer Moore* dilakukan dari belakang ke depan atau pencarian secara mundur. Terdapat dua teknik utama untuk mempercepat proses pencarian

1. *Bad Character Heuristic*

   Ide dari heuristik karakter buruk itu cukup sederhana. Karakter dari teks yang dibandingkan saat ini jjika tidak sesuai dengan karakter pada *pattern* disebut dengan *Bad Character*. Jika, tidak cocok akan terjadi penggeseran pola hingga

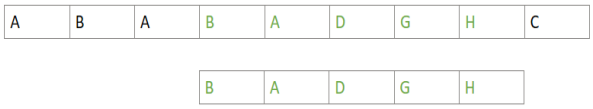
   a. Ketidakcocokan menjadi sebuah kecocokan




**Gambar 1. Terjadi ketidakcocokan pada saat pencarian**
Sumber: dokumen pribadi

Pada gambar satu yang bisa dilihat perbandingannya dimulai dari kanan ke kiri. Pada bagian karakter A pada *text* dan H pada *pattern*. Dikarenakan terjadi ketidakcocokan

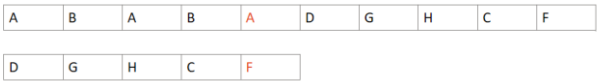
maka akan dicari karakter yang cocok pada *text* yang terdapat pada *pattern*




**Gambar 2. Melakukan pergeseran sampai cocok**
Sumber: dokumen pribadi

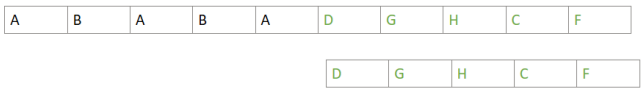
Pada gambar dua dilakukan pergeseran sampai karakter pada *text* yaitu 'A' disandingkan dengan karakter 'A' yang ada pada *pattern* lalu dilakukan kembali pencocokan dari belakang.

b. *Pattern* bergerak melewati karakter yang tidak cocok




**Gambar 3. Ketidakcocokan terjadi dan tidak ada karakter yang pada pattern yang sesuai pada text**
Sumber: dokumen pribadi

Pada gambar di atas terlihat bahwa pada *text* karakter 'A' tidak cocok dengan karakter 'H' pada *pattern* dan pada *pattern* tidak terdapat karakter 'A' sehingga yang terjadi adalah menggeser seluruh *pattern* untuk melewati karakter 'A' pada *text*.



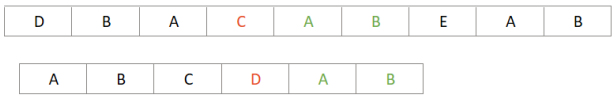

**Gambar 4. Pergeseran terjadi setelah ketidakcocokan**
Sumber: dokumen pribadi

Pada gambar di atas pergeseran terjadi karena karakter 'A' pada *text* tidak memiliki pasangan pada karakter-karakter yang ada pada *pattern*. Sehingga pergeseran yang terjadi melewati karakter tersebut.

2. *Good Suffix Heuristic*

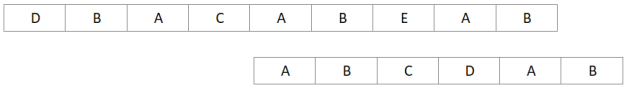
*Good Suffix Heuristic* atau *Match Heuristic* melakukan pergeseran berdasarkan posisi ketidakcocokan karakter yang terjadi. Maksudnya untuk membuat *heuristic* ini perlu diketahui posisi mana terjadi sebuah ketidakcocokan. Posisi ketidakcocokan itulah yang akan menentukan besar pergeseran. Ada dua kasus yang akan muncul pada *Good Suffix Heuristic*.

1. Terdapat *substring* u dari *Pattern* yang mana u = t (*substring Text)* dan u tidak didahului oleh karakter yang mendahului t
2. Tidak ada *substring* u dari *Pattern* sehingga u = t dan u didahului oleh karakter selain karakter yang mendahului t



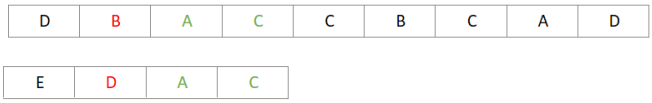

**Gambar 5. Terjadi ketidakcocokan setelah suffix 'AB'**
Sumber: dokumen pribadi

Pada gambar di atas terdapat ketidakcocokan terjadi setelah suffix 'AB', tetapi setelahnya tidak terdapat suffix/prefix lain selain pada *pattern[0]* dan *pattern[1]* sehingga pergeseran dilakukan sejauh itu.



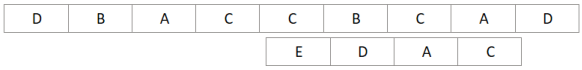

**Gambar 6 Pergeseran untuk menyamakan suffix/prefix**
Sumber: dokumen pribadi

3. Tidak ada *substring* u dari *Pattern* sehingga u = t dan u didahului oleh karakter selain yang mendahului t dan tidak ada awalan *Pattern* yang juga merupakan akhiran dari t.




**Gambar 7 Terjadi Ketidakcocokan dan tidak terdapat suffix yang cocok**
Sumber: dokumen pribadi

Pada gambar di atas terdapat ketidakcocokan setelah suffix 'AC', tetapi tidak ada yang bisa menggantikan lagi sehingga digeser untuk melewati suffix 'AC'.




**Gambar 8. Pergeseran melewati suffix 'AC'**
Sumber: dokumen pribadi

Pada gambar enam pergeseran dilakukan untuk melewati suffix 'AC' dan memulai pencocokan dari kanan ke kiri lagi dari awal.

### *C. Knuth–Morris–Pratt*

Algoritma Knuth-Morris-Pratt (KMP) adalah salah satu algoritma pencarian string yang dikembangkan secara independen oleh tiga tokoh besar dalam bidang *computer science*, yaitu Donald E. Knuth, James H. Morris, dan Vaughan R. Pratt. Algoritma ini bekerja dengan cara mencocokkan pola (pattern) dari kiri ke kanan, mirip dengan algoritma *Brute Force*. Namun, KMP lebih efisien dan cerdas dibandingkan dengan Brute Force karena mampu menggeser pola sesuai dengan tabel LPS (*Longest Prefix Suffix*) yang telah dihitung sebelumnya.

Untuk mencapai efisiensi ini, KMP melakukan pre-*processing* pada pola yang akan dicari, menghasilkan tabel LPS yang berfungsi sebagai panduan untuk menentukan

seberapa jauh pola dapat digeser setelah ketidakcocokan terjadi. Dengan adanya tabel LPS, KMP menghindari pengulangan perbandingan karakter yang tidak perlu, sehingga mengurangi jumlah perbandingan yang harus dilakukan. Oleh karena itu, meskipun prinsip dasarnya serupa dengan *Brute Force*, algoritma KMP memberikan peningkatan yang signifikan dalam kecepatan dan efisiensi pencarian *string*.

Langkah-langkah yang dilakukan untuk melakukan *String Matching* dengan algoritma KMP:

1. Melakukan *Pre-processing* pada *pattern* untuk mencari *Bounding function* atau LPS (*Longest Proper Suffix*).

   *Pattern:* abaaba

| j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| P[j] | a | b | a | a | b | a |

Tabel 1 Pattern dari ABAABA
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

*Bounding Function*

| k | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| B[k] | 0 | 0 | 1 | 1 | 2 |

Tabel 2 Bounding Function dari tabel ABAABA
sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

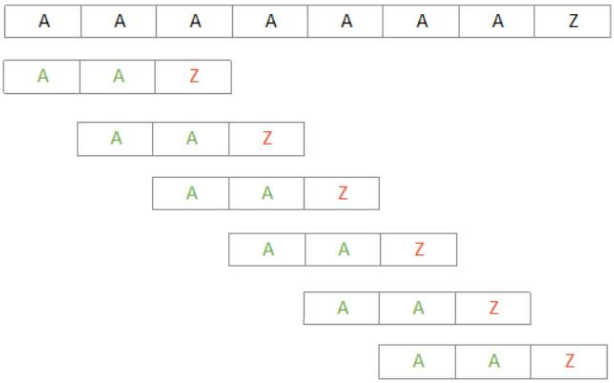
Memulai pencarian dengan membandingkan *Pattern* dan *Text*.

**Gambar 9. Pencocokan dengan Algoritma KMP**
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

Jika terjadi ketidakcocokan pada *Pattern* dan *Text Pattern* akan digeser sebanyak jumlah prefix suffix yang bersesuaian pada *Bounding Function*. Ini dilakukan agar pencarian lebih efisien. Setelah dilakukan pergeseran, ada bagian yang tidak perlu dicek lagi karena pasti sudah benar sesuai dengan *bounding function*.

### D. *Brute Force*

Algoritma *Brute Force* adalah salah satu algoritma yang paling sering kita jumpai karena *Brute Force* sendiri adalah metode yang *straightforward* tidak butuh pemikiran yang memusingkan seperti halnya algoritma Boyer-Moore dan KMP yang membutuhkan *pre-processing* terlebih dahulu.




**Gambar 10. Ilustrasi String Matching dengan Bruteforce**
sumber: dokumen pribadi

Pseudocode untuk algoritma *Brute Force*

```
Function BruteForceSearch(pattern, text):
    Set patternSize = length of pattern
    Set textSize = length of text

    For i from 0 to textSize - patternSize:
        Set j = 0
        While j < patternSize and text[i + j] == pattern[j]:
            Increment j by 1

        If j == patternSize:
            Return True

    Return False
```

## III. IMPLEMENTASI

### A. *Proses Mengubah Audio Menjadi Binary Text*

Proses pertama yang perlu dilakukan adalah mengubah Audio menjadi representasi *Binary Text* supaya bisa dilakukan Pencarian kecocokan menggunakan *String Matching*. Akan, tetapi melakukan pencarian dengan bentuk *Binary* adalah kelemahan dari Algoritma *String Matching Boyer Moore* sehingga perlu dilakukan *convert* dari data *binary* menjadi *ASCII* sehingga perbandingannya seimbang.

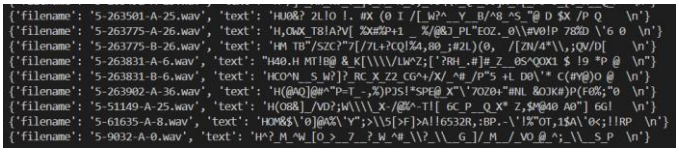
### B. *Mengubah Audio Menjadi ASCII*

Proses selanjutnya yang perlu dilakukan adalah mengubah *Binary Text* yang sudah didapat menjadi *ASCII*. Berikut adalah cuplikan dari kode untuk *converter* dari Audio menjadi *Binary* dan dari *Binary* menjadi *ASCII*.

```
import wave
import binascii

class converter:
    def __init__(self, path: str):
        self.path = path
        self.binary_data = self.convert_to_binary()
        self.ascii_data = self.binary_to_ascii()

    def convert_to_binary(self) -> bytes:
        w = wave.open(self.path, "rb")
        binary_data = w.readframes(20)
        w.close()
        return binary_data

    def binary_to_ascii(self) -> str:
        self.ascii_data = binascii.b2a_uu(self.binary_data)
        return self.ascii_data.decode('ascii')
```

**Gambar 11. cuplikan kode *converter* Audio menjadi Binary dan ASCII**
Sumber: dokumen pribadi

Dari cuplikan kode di atas dapat dilihat bahwa *converter* data audio dengan ekstensi .wav menjadi *binary* denganmenggunakan library yang telah disediakan oleh bahasa Python yaitu *wave*.



```
{'filename': '5-263501-A-25.wav', 'text': 'HUØ? 2L!O !. #X (0 I /[_W?^ _Y _B/^8_^S_"@ D $X /P Q   \n'}
{'filename': '5-263775-A-26.wav', 'text': 'H,OWX_T8!A?V[ %X#%P+1 _ %/@&J_PL"EOZ._0\\#V0!P 78%D \'6 0  \n'}
{'filename': '5-263775-B-26.wav', 'text': 'HM TB"/SZC?"?/[/7L+?CQ!%4,80_;#2L)(0,  /[ZN/4*\\,;QV/D[    \n'}
{'filename': '5-263831-A-6.wav', 'text': "H40.H MT!B@ &_K[\\\\/LW^Z;['?RH_.#]#_Z__0S^QOX1 $ !9 *P @  \n"}
{'filename': '5-263831-B-6.wav', 'text': 'HCO^N__S_W?]?_RC_X_Z2_CG^+/X/_^#_/P"5 +L D0\'* C(#Y@)O @  \n'}
{'filename': '5-263902-A-36.wav', 'text': 'H(@AQ]@#^"P=T_-,%)PJS!*SPE@_X"\'7OZ0+"#NL &OJK#)P(F0%;"0  \n'}
{'filename': '5-51149-A-25.wav', 'text': 'H(O8&]_/VD?;W\\\\_X-/@%^-T![ 6C_P__Q_X* Z,$M@40 A0"] 6G!   \n'}
{'filename': '5-61635-A-8.wav', 'text': 'HOM%$\'0]@A%\'Y";>\\5[>F]>A!!6532R,:BP.-\'!%"OT,1$A\'0<;!!RP  \n'}
{'filename': '5-9032-A-0.wav', 'text': 'HA?_M_^W_[O_>__7__?_W_^#_\\?_\\__G_]/_M__/_VO_@_^;_\\__S_P  \n'}
```

**Gambar 12. Hasil *convert* dari audio menjadi ASCII**
sumber: dokumen pribadi

### *C. Pre-Processing Pattern*

Kedua algoritma ini *Boyer Moore* dan KMP memerlukan *Pre-Processing* terlebih dahulu sebelum melakukan pemrosesan. Algoritma Boyer Moore menjadi *Bad Character* dan KMP mencari *LPS* (*Longest Proper Suffix*).

1. *Boyer Moore*
   Pada gambar di bawah adalah fungsi pre-processing untuk Algoritma *Boyer-Moore*. Fungsi Pre-Processingnya sendiri adalah lastOccurences yang digunakan sebagai referensi perpindahan jika terjadi ketidakcocokan pada *text* dan *pattern*.

```
function lastOccurrences():
    occurences = empty dictionary
    for each unique letter in pattern:
        last_index = find the last occurrence index of the letter in pattern
        occurences[letter] = last_index
    return occurences
```

2. *Knuth–Morris–Pratt*
   Pada bagian ini akan dicari LPS (*Longest Proper Suffix*) yang digunakan dalam pencarian menggunakan Algoritma KMP.

```
function GenerateLPS():
    size = length of pattern
    lps = list of size elements initialized to 0
    i = 0
    for j from 1 to size - 1:
        while i > 0 and lps[i] != lps[j]:
            i = lps[i - 1]
        if pattern[i] == pattern[j]:
            i += 1
        lps[j] = i
    return lps
```

### *D. Program Algoritma KMP dan Boyer-Moore*

1. KMP
   Pada kelas KMP terdapat 3 *method* yang digunakan yaitu, konstruktor, KMPSearch, dan GenerateLPS.
   a. KMPSearch
      Adalah *Method* yang digunakan untuk melakukan mencari kecocokan string dari *pattern* dan *text*.
   b. GenerateLPS
      Adalah *method* yang digunakan untuk membuat *bounding function* untuk algoritma KMP atau LPS (*Longest Proper Suffix*).

```
class listKMP:
    def __init__(self, list_text: list[dict[str, str]], pattern: str):
        self.list_text: list[dict[str, str]] = list_text
        self.pattern: str = pattern
        self.results: list[(bool, int)] = self.search_in_list()

    def search_in_list(self) -> list[(bool, int)]: #(found_status, comparison_count)
        self.results: list[(bool, int)] = [(False, 0)] * len(self.list_text)
        idx: int = 0
        for entry in self.list_text:
            text = entry['text']
            kmp = KMP(text, self.pattern)
            result: (bool, int) = [kmp.KMPSearch(), kmp.comparison_count]
            self.results[idx] = result
            idx += 1
        return self.results

    def result(self) -> (dict[str, str], int):
        idx_ans: int = -1
        for (idx, item) in enumerate(self.results):
            if(item[0] == True):
                idx_ans = idx
                break
        if(idx_ans != -1):
            return self.list_text[idx_ans], self.results[idx_ans][1]
```

2. BM
   Pada Kelas BM terdapat 3 Method yang digunakan untuk memproses keberlangsungan program. konstruktor, BMSearch, dan lastOccurences.
   a. BMSearch

Method yang digunakan untuk memproses pencarian dari *text* dan *pattern* menggunakan algoritma *Boyer Moore*.

b. lastOccurences
Method yang digunakan untuk melakukan pre-processing sebelum melakukan pencarian dengan method BMSearch.

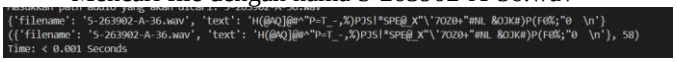
```
class listBM:
    def __init__(self, list_text: list[dict[str, str]], pattern: str):
        self.list_text: list[dict[str, str]] = list_text
        self.pattern: str = pattern
        self.results: list[(int, int)] = self.search_in_list() #[index_found, comparison]

    def search_in_list(self) -> list[(int, int)]:
        self.results: list[(int, int)] = [(-1, 0)] * len(self.list_text)
        idx: int = 0
        for entry in self.list_text:
            text = entry['text']
            bm = BM(text, self.pattern)
            result: (int, int) = [bm.BMSearch(), bm.comparison_count]
            self.results[idx] = result
            idx += 1
        return self.results

    def result(self) -> (dict[str, str], int):
        idx_ans: int = -1
        for idx, item in enumerate(self.results):
            # print(item)
            if(item[0] != -1):
                idx_ans = idx
                break

        if(idx_ans != -1):
            return (self.list_text[idx_ans], self.results[idx_ans][1])
```

3. *Brute Force*
Pada kelas BruteForce terdapat dua *method* yang digunakan untuk memproses pencarian *string*. Tidak seperti Algoritma lainnya yang memerlukan *pre-processing*. Algoritma *Brute Force* bisa langsung memulai pencarian. Bisa dilihat dari cuplikan kode tidak terdapat *method* yang diperuntukkan untuk *pre-processing*.

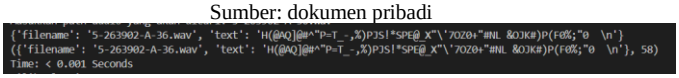
```
class Bruteforce:
    def __init__(self, pattern: str, text: str):
        self.pattern = pattern
        self.text = text
    def BruteforceSearch(self) -> (bool, int):
        textLength: int = len(self.text)
        patternLength: int = len(self.pattern)
        comparison_count: int = 0
        for i in range(textLength - patternLength + 1):
            j = 0
            while j < patternLength and self.text[i + j] == self.pattern[j]:
                j += 1
                comparison_count += 1
            if j == patternLength:
                return (True, comparison_count)

        return (False, comparison_count)
```

*E. Perbandingan Algoritma Bruteforce, Boyer-Moore, dan KMP.*

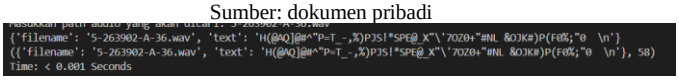
1. Test Case 1
Mencari file dengan nama 5-263902-A-36.wav




**Gambar 13. Pencarian dengan Boyer-Moore**
Sumber: dokumen pribadi

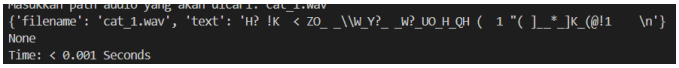



**Gambar 14. Pencarian dengan KMP**
Sumber: dokumen pribadi

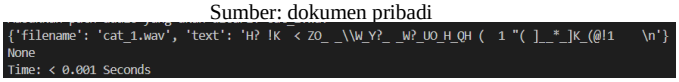



**Gambar 15. Pencarian dengan Bruteforce**
Sumber: dokumen pribadi

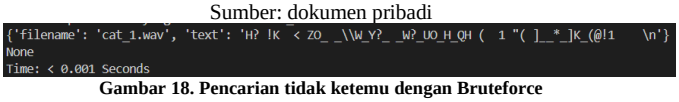
2. Test Case 2




**Gambar 16. Pencarian tidak ketemu dengan Boyer-Moore**
Sumber: dokumen pribadi




**Gambar 17. Pencarian tidak ketemu dengan KMP**
Sumber: dokumen pribadi




**Gambar 18. Pencarian tidak ketemu dengan Bruteforce**
Sumber: dokumen pribadi

| file | algoritma | Waktu (detik) | Jumlah perbandingan |
|---|---|---|---|
| Cat_1.wav | BM | <0.001 | - |
| | KMP | <0.001 | - |
| | Bruteforce | <0.001 | - |
| 5-263902-A-36.wav | BM | <0.001 | 58 |
| | KMP | <0.001 | 58 |
| | Bruteforce | <0.001 | 58 |

**Tabel 3. Perbandingan Algoritma String Matching**
Sumber: dokumen pribadi

## IV. KESIMPULAN

Pada percobaan kali ini penulis melakukan pencarian kemiripan Audio menggunakan *String Matching* menggunakan

algoritma *Bruteforce*, Boyer-Moore, dan KMP. Selain itu, penulis juga membandingkan ketiga algoritma tersebut untuk mencari kemiripan audio. Akan, tetapi penulis menggunakan sample audio terbatas hanya pada audio yang berekstensi .wav dan juga penulis menggunakan audio yang panjangnya sekitar 4-5 detik saja.

Hasil yang didapat dari percobaan ini adalah selalu bisa menemukan audio yang dicari meskipun nama filenya berbeda, tetapi memiliki representasi ASCII yang sama. Bisa dilihat pada tabel 3 terlihat bahwa waktu dan jumlah perbandingan karakter dari setiap algoritma sama antara *Bruteforce*, KMP, dan Boyer-Moore. Hal ini terjadi karena penulis hanya menggunakan data audio sebesar 4-5 detik saja. Sehingga representasi ASCII-nya bisa lebih pendek.

Pada percobaan ini hampir tidak terlihat perbedaan yang signifikan antara *Bruteforce*, KMP, dan Boyer-Moore karena beberapa faktor yang diantaranya sudah penulis sebut sebelumnya yaitu, data yang dipakai hanya data berekstensi .wav dengan panjang 4-5 detik saja, panjang *pattern* dan *text* selalu sama, sehingga tidak terjadinya pergeseran pada saat pencarian kecocokan *string*.

Dalam kondisi di mana audio yang digunakan memiliki durasi yang singkat dan hanya terbatas pada ekstensi .wav, perbedaan antara ketiga algoritma *string matching* tersebut tidak terlalu mencolok. Meskipun tidak ada perbedaan yang signifikan terlihat dalam percobaan ini, penting untuk dicatat bahwa penggunaan algoritma yang tepat tetaplah penting tergantung pada konteks dan karakteristik dari data yang diproses.

Selain itu, tujuan awal dari makalah ini dibuat tercapai yaitu, mencari kemiripan dari audio menggunakan *string matching* dan hasilnya cukup memuaskan bagi penulis meskipun audio yang digunakan hanyalah audio yang berekstensi .wav dan berdurasi 4-5 detik. Namun, hal tersebut sudah cukup untuk memperlihatkan bahwa mencari kemiripan audio menggunakan *string matching* adalah mungkin.

## V. PENUTUP

Dengan penuh rasa syukur, penulis ingin mengucapkan puji dan syukur kepada Allah SWT atas berkat, rahmat, dan hidayat-Nya yang telah membimbing penulis dalam menyelesaikan tugas makalah ini dengan baik. Pada kesempatan yang mulia ini, penulis juga ingin menyampaikan ungkapan terima kasih yang tulus kepada Orang Tua penulis atas doa, dukungan, dan kasih sayang yang tak pernah berhenti mengalir. Tidak lupa, penulis ingin mengucapkan rasa hormat dan terima kasih yang setinggi-tingginya kepada Dr. Rinaldi Munir, Dr. Nur Ulfa Maulidevi, dan Dr. Rila Mandala sebagai dosen yang telah memberikan ilmu, panduan, dan bimbingan yang berharga dalam mata kuliah Strategi Algoritma. Dan tak kalah pentingnya, penulis ingin mengungkapkan rasa terima kasih kepada seluruh teman-teman penulis yang turut serta membantu dalam proses penulisan makalah ini. Semoga kerja keras dan bantuan dari semua pihak menjadi amal ibadah yang diterima di sisi Allah SWT.

## LAMPIRAN

Tautan repository: https://github.com/Indraswara/Audio-Matching

## REFERENSI

[1] Munir, Rinaldi. "Pencocokan string (String matching/pattern matching)," URL:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf diakses pada: 10 Juni 2024

[2] TopCoder, "Boyer-Moore Algorithm with Bad Character Heuristic," Top Coder Thrive. [Online]. URL: https://www.topcoder.com/thrive/articles/boyer-moore-algorithm-with-bad-character-heuristic diakses pada: 11 Juni 2024

[3] Madhu, Neetha. "Good Suffix Rule in Boyer-Moore Algorithm Explained Simply," Medium. [Online].

URL:https://medium.com/@neethamadhu.ma/good-suffix-rule-in-boyer-moore-algorithm-explained-simply-9d9b6d20a773: diakses pada 11 Juni 2024

[4] GeeksforGeeks. "Boyer-Moore Algorithm for Pattern Searching," GeeksforGeeks. [Online]. URL: https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/ diakses pada: 11 Juni 2024

[5] GeeksforGeeks. "KMP Algorithm for Pattern Searching," GeeksforGeeks. [Online]. URL: https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/: diakses pada 11 Juni 2024

[6] PrepBytes. "String Matching Algorithm," PrepBytes Blog. [Online]. URL: https://www.prepbytes.com/blog/strings/string-matching-algorithm/ diakses pada: 12 Juni 2024

## PERNYATAAN

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

Bandung, 12 Juni 2024

Indraswara Galih Jayanegara
13522119