Pencarian Lagu dari Potongan Lagu Menggunakan Pendekatan Algoritma Pattern Matching

Rahmat Nur Ibrahim Santosa - 13516009[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]rahmatnsantosa@gmail.com

***Abstract*—Bahan kimia saat ini sudah banyak dimanfaatkan oleh manusia, mulai dari pemanfaatan di kehidupan sehari-hari, sampai dengan pemanfaatan dalam jumlah besar pada suatu industri kimia. Salah satu permasalahan dalam penggunaan bahan kimia adalah masalah pengaturan penyimpanannya. Hal ini dikarenakan bahan kimia merupakan bahan yang reaktif dan dapat bereaksi dengan zat lainnya. Pengaturan penyimpanan bahan kimia menjadi hal yang penting untuk mencegah terjadinya reaksi-reaksi membahayakan yang tidak diinginkan, seperti terbentuknya api atau ledakan. Salah satu metode yang dapat digunakan untuk membuat pengaturan penyimpanan bahan kimia adalah dengan penggunakan teori pewarnaan graf. Teori ini tepat digunakan sebagai penyelesaian permasalahan karena prinsipnya sama dengan permasalahan ini secara keseluruhan.**

***Keywords*—Bahan kimia, pengaturan penyimpanan, pewarnaan graf, Welch-Powell**

I. PENDAHULUAN

Musik merupakan sebuah representasi seni, yang merupakan sekumpulan melodi atau harmoni yang terstruktur dan memiliki ritme tertentu. Musik dibuat untuk mengekspresikan suatu emosi tertentu kepada pendengarnya. Hal ini sangat erat kaitannya dengan psikologi manusia, karena musik dipercaya dapat berpengaruh secara langsung kepada perasaan manusia. Dengan mendengarkan musik, manusia bisa merasakan perasaan senang, sedih, ataupun marah sekalipun.

Musik telah menjadi bagian dari kehidupan manusia sehari-hari. Tidak bisa dipungkiri, kita dapat menemukan musik dimana-mana. Kita dapat menemukan musik hampir di setiap tempat, dimulai dari telefon selular milik kita sendiri, di televisi, di radio mobil, ataupun di tempat makan sekalipun. Penikmatnya pun beragam, mulai dari yang masih berusia muda sampai yang sudah berusia tua. Oleh karena itu, musik telah menjadi bagian yang penting bagi manusia, karena keberadaannya dapat ditemukan dimana-mana, dan banyak manusia yang menyukainya.

Pada kehidupan sehari-hari, seringkali kita mendengarkan musik dan kemudian langsung menyukainya. Permasalahan yang sering terjadi adalah, kita tidak tahu apa judul lagu tersebut dan siapa penyanyinya. Bisa saja kita bertanya kepada teman kita yang mungkin tahu, tetapi kenyataannya seringkali teman kita juga tidak mengetahuinya, sehingga kita tidak tahu harus bertanya kemaa lagi. Permasalahan lain muncul ketika kita akan mencari tahu lagu tersebut di internet, tetapi kita kemudian bingung metode apa yang harus digunakan. seringkali kita tidak sempat untuk menghafalkan lirik lagu tersebut agar dapat dilakukan pencarian di *search engine*.

Salah satu solusi atas permasalahan ini adalah dengan melakukan pencarian dengan melakukan pendekatan algoritma pencocokan string, atau sering dikenal dengan *string matching.* Algoritma ini merupakan algoritma yang menerima sebuah pola atau *pattern,* dan kemudian melakukan pencocokan untuk menemukan apakah pattern tersebut berada dalam sebuah *string* atau tidak. Dengan menggunakan algoritma ini, kita akan merepresentasikan sebuah lagu sebagai suatu rentetan string. Potongan lagu yang akan dicari juga direpresentasikan sebagai sebuah string. Setelah keduanya telah menjadi bentuk string, maka langkah selanjutnya adalah menerapkan algoritma ini untuk menentukan apakah potongan lagu yang akan kita cari terdapat di dalam lagu tersebut atau tidak.

II. LANDASAN TEORI

2.1. Algoritma Pencarian String

Persoalan pencarian sebuah pattern atau pola dalam suatu string dikenal dengan algoritma *string matching* atau *pattern matching.* Algoritma ini sangat banyak digunakan dalam bidang informatika.

Pada dasarnya permasalahan yang akan diselesaikan dengan menggunakan algoritma pencarian string adalah sebagai berikut.

1. Terdapat suatu text (sebuah string panjang), dengan panjang n buah karakter
2. Terdapat suatu pattern (sebuah string pendek), dengan panjang m karakter yang akan dicari di dalam teks, dimana $m < n$.

Adapun permasalahannya adalah mencari lokasi pertama dimana *pattern* ditemukan dalam text. Sebagai contoh, permasalahan *pattern matching* adalah sebagai berikut.

Contoh 1:
- *pattern* : kota
  text : Saya akan pergi ke **kota** Bandung

Contoh 2:
- *pattern* : not
  text : nobody **not**iced him

Algoritma ini juga sekaligus dapat mengembalikan lokasi dimana *pattern* pertama kali ditemukan. Jika terdapat beberapa *pattern* dalam suatu teks sekaligus, maka posisi patern yang dikembalikan merupakan posisi pattern yang pertama kali ditemukan. Perhatikan contoh 3 di bawah ini.

Contoh 3:

- *pattern* : apa
  text : Ken**apa** dia tidak tahu apa-apa?

Perhatikan bahwa pada contoh tersebut, posisi yang dikembalikan merupakan posisi awal string *apa* ditemukan, yaitu pada kata "Kenapa", bukan pada kata "apa-apa"

Dalam menyelesaikan permasalahan pencocokan string, terdapat setidaknya tiga algoritma yang dapat digunakan. Algoritma tersebut adalah algoritma brute force, algoritma Knuth-Morris-Pratt, dan algoritma Boyer-Moore. Ketiga algoritma ini memiliki kelebihan dan kekurangannya masing-masing, yang akan dijelaskan pada bagian berikutnya.

2.2. Algoritma Brute Force

Algoritma brute force merupakan algoritma yang paling sederhana dibandingkan dengan algoritma yang lainnya. Hal ini dikarenakan algoritma *brute force* merupakan algoritma yang menggunakan pendekatan *straight forward*. Penyelesaiannya menggunakan algoritma yang sederhana, namun waktu yang dibutuhkan untuk menyelesaikannya tidak begitu mangkus.

Asumsikan bahwa text berada dalam sebuah array T[1..n] dan pattern berada dalam suatu array P[1..m]. Algoritma *brute force* untuk pencocokan string adalah sebagai berikut:

1. *Pattern* P akan dicocokkan dengan awal teks T.
2. Bandingkan setiap karakter dalam *pattern* P dari mulai karakter dengan indeks paling kiri ke kanan dengan karakter yang bersesuaian dalam teks T, sampai:
   a. semua karakter yang dibandingkan sama, yang artinya *pattern* ditemukan dalam text, atau
   b. terdapat sebuah ketidakcocokan karakter, yang artinya *pattern* belum ditemukan dalam text.
3. Jika *pattern* P belum ditemukan dalam teks T, maka geser pattern P satu karakter ke kanan, dan ulangi lagi langkah ke 2, sampai teks T habis atau pattern P ditemukan.

Visualisasi dari proses pencocokan string dengan menggunakan algoritma brute force dapat dilihat pada contoh 4 di bawah ini.

Contoh 4
Text : nobody noticed him
Pattern : body

```
        nobody noticed him
 s = 0  body
 s = 1   body
 s = 2    body
```

Dari contoh 4, dapat dilihat bahwa algoritma ini merupakan algoritma yang sederhana, dan terlihat mudah dan cepat. Hal ini dikarenakan *pattern* yang akan ditemukan pada contoh tersebut berada di bagian awal teks, sehingga proses pencocokkan pattern yang dilakukan hanya sedikit. Namun bayangkan apabila sebuah pattern yang akan dicari terletak di bagian akhir text, di mana ukuran teks tersebut cukup besar, di sertai pattern yang tidak pendek. Hal ini akan menghasilkan waktu yang cukup besar, yang menyebabkan algoritma pencarian menjadi tidak efektif.

Pseudo-code algoritmana antara lain adalah sebagai berikut:

```
procedure BruteForceSearch(input m, n:
   integer, input P : array[1..m] of char,
   input T : array[1..n] of char,  output idx
   : integer)
Deklarasi
   s, j : integer
   found : boolean
Algoritma:
   s←0
   found←false
   while (s ≤ n-m) and (not found) do
      j←1
      while (j ≤ m) and (P[j] = T[s+j]) do
         j←j+1
      endwhile
      {j>m or P[j] ≠ T[s+j]}

      if j = m then
         found←true
      else
         s←s+1
      endif
   endwhile

   { s > n – m or found }

   if found then
      idx←s+1
   else
      idx←-1
   endif
```

Tabel 1 Algoritma Brute Force
(Sumber: Diktat Strategi Algoritma, Rinaldi Munir, 2009)

Kasus terbaik yang dapat terjadi dalam penggunaan algoritma *brute force* adalah ketika karakter pertama pattern P tidak pernah sama dengan karakter teks T yang dicocokkan. Hal ini berarti proses pencocokan string tidak pernah terjadi, sampai ditemukan karakter yang huruf awalnya sama dengan pattern dan pencocokan langsung berhasil. Perhatikan contoh 5 di bawah ini.

Contoh 5
Text : Saya akan pergi ke kota Bandung
Pattern : Bandung

Perhatikan bahwa karakter "B" tidak terdapat dalam teks hingga ditemukan pada kata "Bandung". Oleh karena itu, untuk kasus ini, jumlah perbandingan yang dilakukan paling banyak sebanyak *n* kali. Maka kompleksitas waktu untuk kasus terbaik adalah O(*n*).

Adapun kasus terburuknya dari permasalahan ini adalah ketika pencocokan string dilakukan setiap saat dikarenakan huruf pertama teks selalu sama dengan huruf pattern yang akan

dicocokkan. Perhatikan contoh 6 di bawah ini.

Contoh 6
Text : aaaaaaaaaaaaaaac
Pattern : aaaac

Pada kasus tersebut, akan selalu dilakukan pencocokan string untuk tiap pergeseran sebanyak m kali, dimana pergeseran dilakukan sebanyak n-m+1 kali. Oleh karena itu, kompleksitas algoritma dari kasus worst case adalah O(mn).

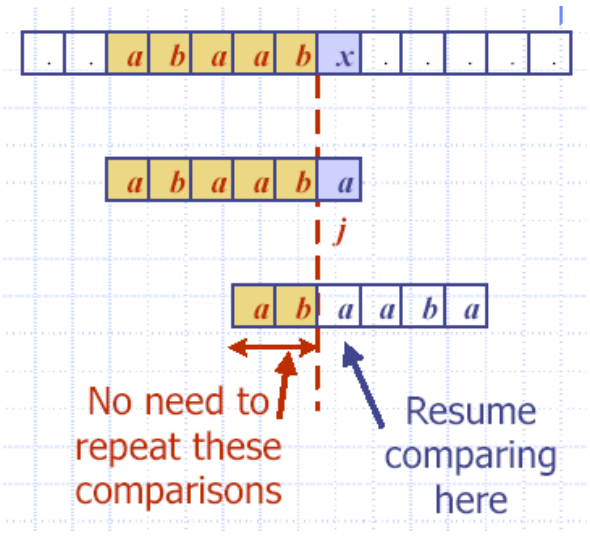
2.3 Algoritma Knuth-Morris-Pratt (KMP)

Algoritma Knuth-Morris-Pratt merupakah algoritma yang prosesnya mirip dengan algoritma bruteforce, hanya saja memiliki algoritma pergeseran yang berbeda sehingga dapat memangkas waktu pencarian. Algoritma ini pertama kali dikembangkan oleh D. E. Knuth, bersama dnegan J. H. Morris dan V. R. Pratt pada tahun 1977. Algoritma ini merupakan pengembangan dari algoritma brute force.

Pada algoritma *brute force,* proses pencocokan dilakukan dengan bergeser ke kanan sebanyak satu karakter. Namun, pada algoritma KMP, jumlah pergeseran bisa lebih dari satu karakter, tergantung bagaimana sifat dari pattern itu sendiri. Untuk menyimpan jumlah pergeseran yang harus dilakukan oleh algoritma, dibuatlah fungsi pinggiran (*border function*) yang dibuat saat awal proses terhadap *pattern* P.

Kegunaan dari fungsi pinggiran ini adalah untuk menghindari pencocokan yang tidak perlu dilakukan lagi. Misalkan dalam sebuah pattern, jika dalam pattern tersebut terdapat bagian yang sama dengan prefix dari pattern tersebut, maka tidak perlu dilakukan lagi pencocokan string pada bagian prefix tersebut dalam pergeseran selanjutnya, dikarenakan sudah pasti akan menghasilkan hasil sama. Hal ini digunakan untuk mereduksi pencocokan yang tidak diperlukan, karena sebelumnya sudah dilakukan pencocokan oleh algoritma.

Visualisasi dari proses pencocokan string dengan menggunakan algoritma Knuth-Morris-Pratt dapat dilihat pada gambar 1 di bawah ini.



Gambar 1 – Visualisasi Algoritma Knuth-Morris-Pratt
(sumber: Slide Pattern Matching, Rinaldi Munir, 2018)

Adapun fungsi pembatas dihitung dengan melihat ukuran prefix paling panjang dari pattern pada P[0..j], yang juga merupakan suffix pada P[1..j]. Indeks j merupakan nilai pergeseran yang akan diberikan ketika terjadi ketidakcocokan pada indeks j di pattern P. Beberapa literatur menyebut fungsi pembatas ini sebagai *failure function,* sehingga untuk persoalan ini dapat kita deklarasikan sebuah array *fail* yang merupakan fungsi pembatas dari algoritma KMP.

Untuk memperjelas penggunaan fungsi pinggiran dalam KMP, mari kita gunakan contoh *pattern* P = "abaaba". Untuk menghitung b[4], dimana b adalah sebuah array border function, maka kita akan menghitung ukuran prefix dari P[0..4] yang juga merupakan suffix dari P[1..4]. Dari penghitungan ini didapatkan bahwa substring yang memenuhi adalah "ab", sehingga nilai dari b[4] adalah 2. Perhitungan ini diterapkan pada seluruh isi pattern, sehingga hasil akhir fungsi pembatas untuk pattern P adalah sebagai berikut.

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

Tabel 2 Fungsi Pembatas dari Pattern "abaaba"

Pseudo-code dari algoritma penghitungan fungsi pinggiran adalah sebagai berikut.

```
procedure HitungPinggiran (input m: integer,
   P: array[1..m] of char, output b:
   array[1..m] of integer)

   {menghitung nilai b[1..m] untuk pattern
   P[1..m]}

Kamus Lokal
  k, q : integer

Algoritma
  b[1]←0
  q←2
  k←0
  for q←2 to m do
    while ((k > 0)) and (P[q] ≠ P[k+1])) do
       k←b[k]
    endwhile
    if P[q] = P[k+1] then
       k←k+1
    endif
    b[q] = k
  endfor
```

Tabel 2 Algoritma Perhitungan Fungsi Pinggiran
Sumber: Diktat Strategi Algoritma, Rinaldi Munir, 2009

Sedangkan pseudo-code dari algoritma Knuth-Morris-Pratt secara keseluruhan adalah sebagai berikut.

```
procedure KMPsearch (input m, n: integer,
   input P: array[1..m] of char, input T: array
   [1..n] of char, output idx : integer)

   {Masukan: pattern P yang panjangnya m dan
```

```
   teks T yang panjangnya n}

   {Keluaran: posisi awal kecocokan (idx).
   Jika P tidak ditemukan, idx = -1}

Kamus Lokal
  i, j : integer
  ketemu = boolean
  b : array[1..m] of integer

  procedure HitungPinggiran (input m: integer,
  P: array[1..m] of char, output b:
  array[1..m] of integer)
  {menghitung nilai b[1..m] untuk pattern
  P[1..m]}

Algoritma
  HitungPinggiran(m, P, b)
  j←0
  i←1
  ketemu ← false
  while (i ≤ n and not ketemu) do
      while ((j > 0) and (P[j+1] ≠ T[i])) do
         j←b[j]
      endwhile
      if P[j+1] = T[i] then
         j←j+1
      endif
      if j = m then
         ketemu ← true
      else
         i←i+1
      endif
  endwhile
  if ketemu then
      idx ← i-m+1
      { catatan : jika indeks array
        dimulai dari 0, smaka idx ← i-m }
  else
      idx←-1
  endif
```

Tabel 3 Algoritma KMP
Sumber: Diktat Strategi Algoritma, Rinaldi Munir, 2009

Kompleksitas waktu dari algoritma KMP ditentukan oleh dua proses. Proses yang pertama adalah waktu menghitung fungsi pinggiran, dimana kompleksitasnya adalah O(m). Proses yang kedua adalah proses pencarian string, dimana kompleksitasnya adalah O(n). Oleh karena itu, kompleksitas waktu algoritma untuk KMP adalah O(m+n). Hal ini menunjukkan bahwa algoritma KMP akan mengeksekusi program jauh lebih cepat dibandingkan algoritma *brute force*.

2.4 Algoritma Boyer-Moore

Algoritma Boyer-Moore merupakan algoritma pencocokan string yang melakukan perbandingan pattern P dari kanan ke kiri. Pada algoritma ini terdapat dua buah teknik dasar, yaitu:

1. Teknik *looking-glass*

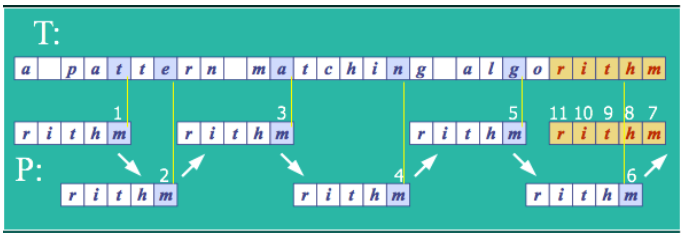
   Mencari pattern P pada text T dengan cara berjalan mundur melalui pattern P, dimulai dari akhir pattern.

2. Teknik *character-jump*

   Ketika suatu karakter ditemukan tidak cocok pada T[i] = x, yaitu saat karakter pada pattern P[j] tidak sama dengan karakter T[i]. Terdapat tiga kasus yang memungkinkan dari penerapan teknik ini. Asumsikan bahwa x adalah T[i], dimana ketidakcocokan terjadi. Ketiga kemungkinan tersebut adalah:

   1. Terdapat x dalam P, dan x berada di sebelah kiri P[j] dimana terjadi mismatch, maka P di geser ke kanan agar sejajar tempat kemunculan terakhir x di P dan di T[i].
   2. Terdapat x dalam P, dan x berada di sebelah kanan P[j] dimana terjadi mismatch, maka P digeser ke kanan sebanyak 1 karakter.
   3. Tidak terdapat x dalam P, maka mulai pencarian baru, yaitu dengan cara menggeser pattern ke kanan sehingga P[0] sejajar dengan T[i+1],

Visualisasi dari proses pencocokan string dengan menggunakan algoritma Knuth-Morris-Pratt dapat dilihat pada gambar 2 di bawah ini.



Gambar 2 – Visualisasi Algoritma Boyer-Moore
(sumber: Slide Pattern Matching, Rinaldi Munir, 2018)

Dalam penerapannya, algoritma Boyer-Moore juga menerapkan suatu fungsi pinggiran (*bounding function*). Fungsi ini dinamakan dengan *Last Occurrence Function*. Fungsi ini memetakan seluruh karakter yang digunakan dalam string menjadi sebuah nilai integer yang menyatakan indeks terakhir ditemukannya alfabet tersebut dalam pattern P. Secara definisi, L(x) adalah indeks terbesar i sehingga P[i] = x, atau nilainya -1 jika tidak terdapat x dalam P, dimana x adalah sebuah alfabet. Contoh tabel hasil pemetaan fungsi *last occurrence* dapat dilihat pada tabel di bawah ini, dimana pattern adalah "abacab".

P	a	b	a	c	a	b
i	0	1	2	3	4	5

Tabel 4 Pattern "abacab" dan indeksnya

Misalkan karakter alfabet yang digunakan untuk fungsi ini adalah A = {a, b, c, d}, maka fungsi last occurrence L(x) akan menghasilkan:

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

Tabel 5 Hasil Penghitungan Fungsi Last Occurence
Sumber: Diktat Strategi Algoritma, Rinaldi Munir, 2009

Pseudo-code dari algoritma penghitungan fungsi pinggiran last occurrence adalah sebagai berikut.

```
function buildLast (pattern: array[1..m] of
  char)→array of integer

  {mengembalikan array sorting index dari
  last occurrence dari setiap nilai ASCII
```

```
   dari karakter pada pattern}

   Kamus Lokal
      i, j : integer
      last : array[0..127] of integer {himpunan
      karakter ASCII}

   Algoritma
      for i←0 to 127 do
         last[i]←-1 {menginisialisasiarray}
      endfor
      for j←0 to m-1 do
         last[pattern[j]] ← j
      endfor
      → last
```

Tabel 6 Algoritma Fungsi Last Occurence
Sumber: Slide Strategi Algoritma, Rinaldi Munir, 2009

Sedangkan pseudo-code dari algoritma Booyer-More secara keseluruhan adalah sebagai berikut.

```
function bmMatch (text: array[1..a] of char,
   pattern: array [1..b]of char )→integer
   {mengembaikan indeks pertama ditemukannya
   pattern pada teks, mengembalikan -1 jika
   tidak ketemu}

Kamus Lokal
   i, j, n, m, lo, idx : integer
   last : array of integer

Algoritma
   last ← buildLast(pattern)
   n←a
   m←b
   if i > n-1 then
   {tidak cocok jika pattern lebih panjang
dari teks}
        idx←-1
   else
      j←m-1
      while i ≤ n-1
         if pattern[j] = text[i] then
            if j=0 then
               idx←i {match}
            else {teknik looking glass}
               i←i-1
               j←j-1
            endif
         else {teknik character jump}
            lo    ←    last[text[i]]    {last
occurence}
            i ← i +m-min(j, 1+lo)
            j←m-1
            idx←-1
         endif
      endwhile
   endif
   → idx
   {mengembalikan indeks pertama ditemukannya
   pattern pada teks, -1 jika tidak ketemu}
```

Tabel 7 Algoritma Boyer-Moore
Sumber: Slide Strategi Algoritma, Rinaldi Munir, 2009

Kompleksitas waktu dari algoritma Boyer-Moore ditentukan oleh jumlah alfabet yang terdapat dalam pattern. Algoritma Boyer-Moore akan sangat bagus jika alfabet yang digunakan banyak, dan akan menjadi lambat jika alfabet yang digunakan sedikit. Untuk kasus terburuk, kompleksitas waktu eksekusi Boyer-Moore adalah O(nm+A). Hal ini membuat algoritma ini lebih cocok digunakan untuk teks berbahasa, dibandingkan untuk teks binary. Walaupun begitu, kecepatannya tetap lebih cepat dibandingkan dengan algoritma *brute force*.

III. PEMBAHASAN

Sebuah lagu pada dasarnya adalah bentuk dari representasi musik yang pada awalnya berbentuk perpaduan dari beberapa gelombang suara. Pada dunia digital, lagu ini kemudian diubah dari sinyal analog (gelombang suara pada umumnya) menjadi digital, yang merupakan rangkaian dari kode biner. Kemudian sinyal audio ini akan diubah ke suatu format file musik tertentu sehingga dapat diputar oleh media pemutarnya.

Proses pencarian lagu dari potongan lagu membutuhkan sebuah potongan lagu yang berisi pattern dari lagu tersebut. Langkah-langkah dasar pemrosesan masalah ini antara lain adalah sebagai berikut:

1. Terima suatu potongan lagu dalam bentuk file audio (.mp3, .wav, dsb)
2. Baca potongan lagu tersebut dalam bentuk biner atau hexadesimal, untuk memudahkan pencarian lagu.
3. Cari potongan lagu tersebut dalam suatu database lagu yang diproses dalam bentuk hexadesimal.
4. Algoritma akan mengembalikan posisi lagu tersebut dalam file biner atau hexadesimal, atau mengembalikan -1 jika tidak terdapat potongan lagu tersebut di dalam lagu.

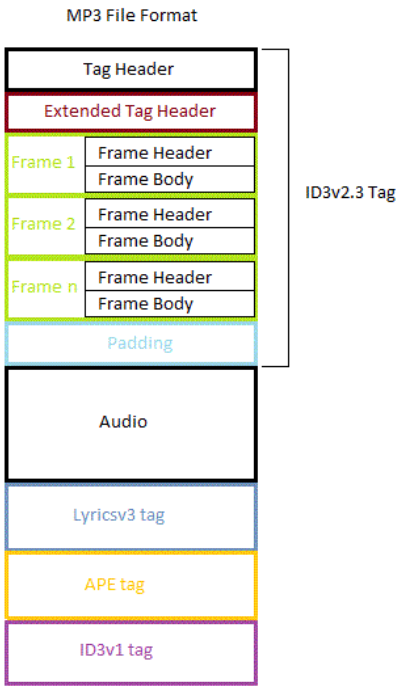
Pada permasalahan ini, akan digunakan pencarian dengan menggunakan file yang telah di-decode menjadi hexadesimal. Saat ini banyak tools yang dapat digunakan di internet menjadi file hexadesimal, di antaranya dapat ditemukan di website tomeko.net. Web ini dapat mengubah suatu file ke dalam bentuk hexadesimalnya. Di bawah ini digunakan contoh file mp3 yang telah di-decode menjadi sebuah file hexadesimal.

```
A3C9D2BF48400017191A4FE98600008806777977D5A68
A03CB01305F7FBDBC27962E4940282828639067136 40A
0A0A19FFE9372F7FBDA5FA257EF74F08908F68882870
1C041C5E910020EFE0FBEB7CA7BE18F587FC400FF39
FA040EFF77F7BFD40EEF0ECEF0BEE711244CDCA858
BE1500214563794E3B79631E0E03386393D8CB6804422
3FFFFFE13DF0EC87FBD8C8FFF888CD310139C28030F
9C53A273241A0FA41074307F661850603138B7F103FD4
73FD1F2FB9D039FFFEDF3EC4215ABEABCAAE99AB8
89A7FA6D24B2A893271540AA0868055A3826A8729900
0AF8C9038CAC1DCC616CD76023B31628656D821F709
448425A94B4C40C6B148CDD3AD8499F0B5C9EAEBB9
96BCAD6DA08736B28C811DDDCADB153495AC53C86
3075A6CF1BBA95A01DF393DDDE7AEE7CEC460B7FE
```

Tabel 8 Contoh File Audio dalam Bentuk Hexadesimal
Sumber: Dokumentasi Pribadi

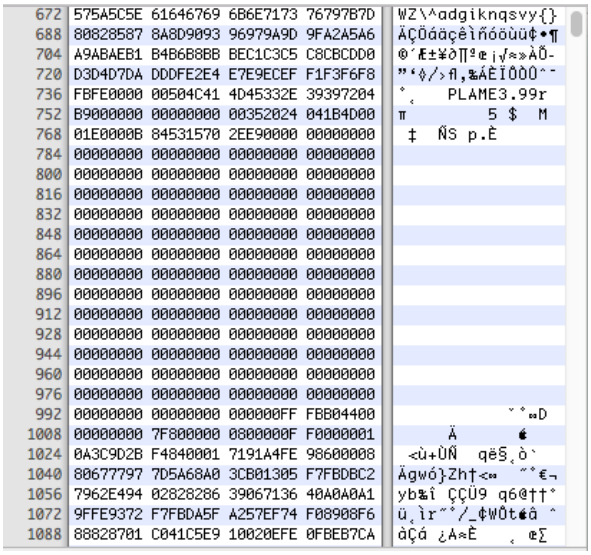
Setelah lagu selesai dikonversi menjadi file hexadesimal, kita perlu mengetahui struktur data dari file audio tersebut agar dapat melakukan analisis lebih lanjut. Adapun file yang digunakan

pada percobaan ini adalah file dengan ekstensi .mp3. Struktur datanya dapat dilihat pada gambar di bawah ini.



Gambar 3 Struktur data file MP3
Sumber: http://www.beaglebuddy.com/content/pages/

Dari struktur data tersebut, dapat dilihat bahwa pada sebuah file mp3, bagian pertama merupakan bagian header yang berikan format ID3v2.3 tag. Bagian setelahnya adalah bagian audio. Untuk dapat menganalisis bagian audio, kita akan mengidentifikasi bagian audio tersebut dalam file hexadesimal. Pada file hexadesimal yang telah dibuat, terdapat suatu bagian yang berisikan rangkaian bilangan 0 yang sangat panjang. Hal ini mengartikan bahwa bagian tersebut adalah hasil pemisahan dari alokasi kontigu antara bagian header dan audio. Pemisah tersebut dapat dilihat pada gambar di bawah ini.



Gambar 4 Pemisah antara Header dan Body
Sumber: Dokumentasi Pribadi

Setelah didapatkan bagian yang merupakan bagian audio, dapat dilakukan analisis lebih lanjut untuk proses pencarian potongan lagu dalam sebuah lagu. Untuk melakukan sebuah uji coba, digunakan potongan lagu yang kemudian di-decode ke dalam sebuah file hexadesimal juga. Misalkan potongan lagu tersebut adalah sebagaimana berikut ini:

A0FA41074

Potongan lagu tersebut akan dilakukan pencarian dengan menggunakan algoritma Brute Force, algoritma KMP, serta algoritma Boyer-Moore. Dari ketiga algoritma tersebut, kita dapat melihat manakah algoritma yang lebih mangkus dan sangkil dalam melakukan *pattern matching*. Dengan menggunakan algoritma yang dibuat dari *pseudo-code* pada bagian II, serta dengan menggunakan test case potongan lagu pada tabel 8, dihasilkan hasil sebagai berikut:

```
BRUTE FORCE:
Pattern ditemukan pada indeks 318
Waktu : 76101
BM:
Pattern ditemukan pada indeks 318
Waktu : 66573
KMP:
Pattern ditemukan pada indeks 318
Waktu : 51189
```

Gambar 5 Perbandingan waktu eksekusi
Sumber: Dokumentasi Pribadi

Untuk uji coba kasus yang kedua, digunakan sebuah pattern yang tidak terdapat dalam text. Untuk itu, kita coba menggunakan pattern dengan ukuran 5, yaitu:

BE125

Hasil pencarian dengan menggunakan pattern tersebut, dihasilkan hasil pencarian sebagai berikut.

```
Masukkan Pattern : BE125
BRUTE FORCE:
Pattern tidak ditemukan
Waktu : 115287
BM:
Pattern tidak ditemukan
Waktu : 103583
KMP:
Pattern tidak ditemukan
Waktu : 99334
```

Gambar 6 Perbandingan waktu eksekusi
Sumber: Dokumentasi Pribadi

IV. KESIMPULAN

Metode pencarian potongan lagu dapat dilakukan dengan menggunakan algoritma *pattern matching*. Hal ini dibuktikan pada gambar 5, dimana hasil pengujian dengan menggunakan test case tersebut mengembalikan indeks dimana potongan lagu tersebut dikembalikan dalam suatu file hexadesimal. Akan tetapi, metode pencarian ini hanya dapat dilakukan untuk memproses lagu yang memiliki komposisi file audio yang sama persis dengan lagu yang akan dicari. Hal ini menyebabkan

pencarian lagu tidak bisa dilakukan dengan menggunakan rekaman dari nyanyian suara, atau dari versi lagu yang berbeda.

Dari ketiga algoritma yang dicoba untuk melakukan pencocokan string terhadap file hexadesimal dari lagu, terdapat beberapa perbedaan kecepatan. Algoritma yang paling cepat adalah algoritma Knuth-Morris-Pratt, kemudian selanjutnya adalah Boyer-Moore dan yang terakhir adalah algoritma brute force. Algoritma Boyer-Moore dapat melakukan proses lebih cepat dari algoritma Knuth-Morris-Pratt dikarenakan alfabet yang digunakan dalam teks dan pattern terhitung sedikit, yaitu hanya terdiri dari 16 jenis karakter saja. Sedangkan Algoritma Boyer-Moore akan jauh lebih efektif jika alfabet yang digunakannya sangat banyak.

V. UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada Tuhan Yang Maha Esa karena atas berkat dan rahmat-Nya lah penulis berhasil menyelesaikan makalah ini. Terima kasih penulis sampaikan juga kepada kedua orang tua saya dan saudara-saudara saya yang selalu memberikan dukungannya kepada saya dalam proses penyelesaian masalah. Penulis juga menyampaikan terima kasih kepada Dr. Ir. Rinaldi Munir, MT., Dr. Masayu Leylia Khodra, ST., MT, serta Dr. Nur Ulfa Maulidevi ST, M.Sc. yang telah memberikan ilmu pengetahuan tentang mata kuliah Strategi Algoritma yang saya gunakan untuk menyelesaikan makalah ini.

REFERENCES

[1] http://www.beaglebuddy.com/content/pages/javadocs/index.html. Diakses 13 Mei 2018
[2] http://tomeko.net/online_tools/file_to_hex.php?lang=en. Diakses 13 Mei 2018.
[3] Slide Pencocokan String, Kuliah Strategi Algoritma, 2018
[4] Munir, Rinaldi. Diktat Kuliah IF2211 Strategi Algoritma Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2009.
[5] http://ridiculousfish.com/hexfiend/. Diakses 14 Mei 2018.

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, 14 Mei 2018

Rahmat Nur Ibrahim Santosa
13516009