Penerapan Regex dan Algoritma *String Matching* pada Pendeteksian *Tweet Spam* Pada Sosial Media X (Twitter)

Dewantoro Triatmojo - 13522011
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 113522011@std.stei.itb.ac.id

***Abstract*—X atau Twitter adalah salah satu platform media sosial terbesar di dunia, yang memungkinkan pengguna untuk berbagi informasi dalam bentuk *tweet*. Namun, dengan popularitasnya, Twitter juga menjadi target utama bagi spammer yang menyebarkan konten tidak diinginkan. Penelitian ini mengkaji penerapan ekspresi reguler (*regex*) dan algoritma *string matching* dalam mendeteksi *tweet spam*. Algoritma yang digunakan meliputi *Brute Force*, Knuth-Morris-Pratt (KMP), dan Boyer-Moore untuk mencocokkan pola teks dengan data *tweet*. Hasil percobaan menunjukkan bahwa Algoritma Boyer-Moore (BM) memberikan efisiensi paling tinggi. Dengan penerapan strategi ini, diharapkan dapat membantu mengurangi penyebaran *spam* di *platform* Twitter dan meningkatkan pengalaman pengguna secara keseluruhan.**

Keywords—Twitter, Pendeteksi Spam, Regex, Brute Force, Knuth-Morris-Pratt, Boyer-Moore.

I. Pendahuluan

X atau Twitter adalah salah satu *platform* media sosial terbesar dan paling populer di dunia, dengan jutaan pengguna aktif setiap harinya. Pengguna dapat berbagi informasi, berita, dan opini melalui sebuah *tweet*, yang masing-masing dibatasi panjangnya hingga 280 karakter. Popularitas Twitter sebagai media komunikasi yang cepat dan luas juga menjadikannya sasaran empuk bagi para *spammer*. *Spam* di Twitter dapat berupa berbagai jenis konten seperti judi *online*, konten tidak senonoh, promosi penjualan produk, dan berbagai jenis promosi lainnya yang mengganggu kenyamanan para pengguna twitter.

Masalah *spam* di Twitter tidak hanya merugikan pengguna dengan menurunkan kualitas pengalaman mereka, tetapi juga dapat mengancam keamanan dan privasi. *Spam* dapat digunakan untuk menyebarkan *malware*, *phishing*, atau penipuan lainnya. Oleh karena itu, deteksi dan pencegahan spam menjadi salah satu fokus utama dalam pengelolaan dan pengembangan *platform* media sosial ini.

Dalam usaha untuk mendeteksi *spam*, berbagai teknik pemrosesan teks dapat diterapkan. Salah satu teknik yang efektif adalah penggunaan ekspresi reguler (*regex*) untuk mengidentifikasi pola-pola teks tertentu yang sering muncul dalam konten *spam*. Regex memungkinkan pencarian dan pengenalan pola teks yang kompleks dengan efisiensi tinggi, menjadikannya alat yang sangat berguna dalam analisis teks otomatis.

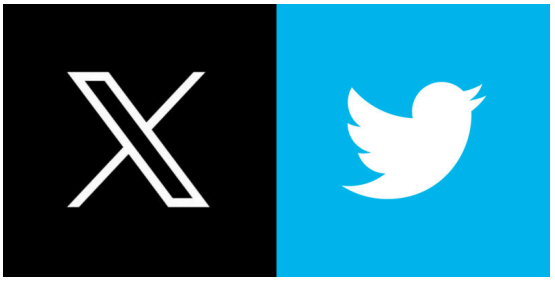
Selain itu, algoritma string matching seperti *Brute Force*, Knuth-Morris-Pratt (KMP), dan Boyer-Moore dapat digunakan untuk mendeteksi *spam* dengan mencocokkan pola teks yang dicurigai dengan kata kunci *spam* yang ada. Algoritma-algoritma ini memiliki keunggulan masing-masing dalam hal efisiensi dan kecepatan pencarian, yang dapat dioptimalkan sesuai dengan kebutuhan spesifik dalam deteksi *spam*.

Penelitian ini bertujuan untuk mencari pendekatan yang paling efisien untuk mendeteksi *tweet spam* di Twitter serta membandingkan algoritma *string matching* paling baik untuk kasus ini. Dengan penelitian ini, diharapkan dapat dicapai peningkatan yang signifikan dalam akurasi dan kecepatan deteksi spam, sehingga dapat memberikan solusi yang lebih efektif dalam menjaga keamanan *platform* Twitter.

II. Landasan Teori

A. *Twitter*

Twitter adalah *platform* media sosial dan layanan jejaring sosial yang memungkinkan pengguna untuk mengirim dan membaca pesan yang dikenal sebagai *tweet* [1]. Pengguna terdaftar dapat memposting, menyukai, dan me-*retweet tweet*, sementara pengguna yang tidak terdaftar hanya dapat membaca *tweet* yang bersifat publik [2]. Twitter dikenal sebagai salah satu *platform* komunikasi yang paling cepat dan efisien, sering digunakan untuk berbagi berita terbaru, tren, dan opini publik.



Gambar 1. Logo Twitter (X) Baru (Kiri) dan Logo Twitter

Lama (Kanan)
Sumber:
https://www.zilliondesigns.com/blog/wp-content/uploads/Twitter-New-Logo-X-2023-.jpg

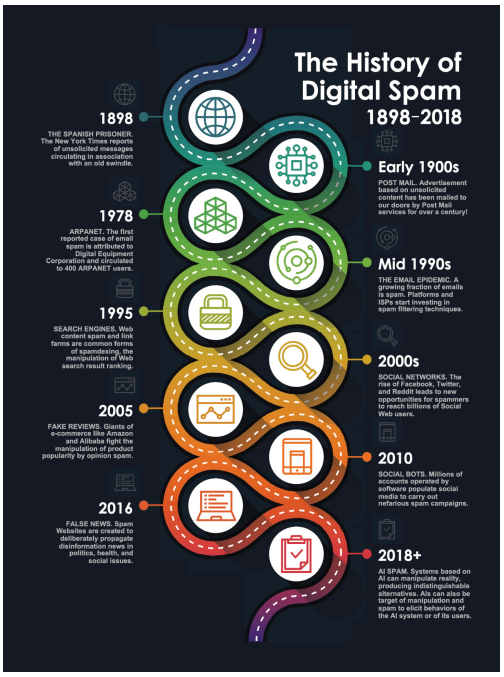
Twitter didirikan pada bulan Maret 2006 oleh Jack Dorsey, Noah Glass, Biz Stone, dan Evan Williams [3]. Ide awal Twitter berasal dari Jack Dorsey yang menginginkan *platform* di mana pengguna bisa berbagi status secara cepat dan singkat. Pada bulan Juli 2006, layanan ini diluncurkan untuk umum dan dengan cepat mendapatkan popularitas karena formatnya yang sederhana dan mudah digunakan.

Saat ini, Elon Musk adalah pemilik dan pemimpin utama Twitter [3]. Elon Musk, yang juga dikenal sebagai CEO Tesla dan SpaceX, mengakuisisi Twitter pada tahun 2022. Sejak akuisisinya, dia telah membuat berbagai perubahan signifikan pada struktur dan operasional perusahaan, yang mempengaruhi cara *platform* ini digunakan dan dikelola.

Pada tahun 2017, Twitter mengklaim memiliki 330 juta pengguna aktif, namun angka tersebut terbukti tidak akurat [3]. Sebuah studi pada tahun 2020 mengungkapkan bahwa hingga 15% dari pengguna saat ini bukanlah manusia, melainkan *bot* yang terlibat dalam *phishing*, keterlibatan palsu, dan aktivitas tidak jujur lainnya [3]. Meskipun demikian, jumlah pengguna Twitter terus meningkat, baik manusia maupun *bot*.

B. *Spam*

Spam adalah istilah yang digunakan untuk menggambarkan pesan yang tidak diinginkan, biasanya dalam jumlah besar, yang dikirimkan melalui berbagai *platform* komunikasi seperti *email*, media sosial, dan pesan instan [4]. *Spam* sering kali berisi iklan, penipuan, atau konten yang tidak relevan bagi penerima. Dalam konteks media sosial Twitter, *spam* bisa berupa *tweet*, komentar, atau pesan langsung yang mengandung promosi produk, tautan *phishing*, atau konten yang mengganggu [4].



Gambar 2. Sejarah spam digital
Sumber:
https://dl.acm.org/cms/attachment/08d17abf-ba4b-414a-9233-86159b5467f3/f1.jpg

Spam memiliki dampak signifikan terhadap keamanan dan privasi pengguna di *platform* media sosial. Pesan *spam* sering kali mengandung tautan yang mengarahkan pengguna ke situs web berbahaya yang dapat mencuri informasi pribadi atau menginfeksi perangkat dengan *malware* [4]. Selain itu, *spam* juga dapat digunakan untuk melakukan *phishing*, di mana penyerang mencoba mendapatkan informasi sensitif seperti kata sandi dan nomor kartu kredit dengan menyamar sebagai entitas terpercaya.

Selain masalah keamanan dan privasi, spam juga secara signifikan mengurangi kenyamanan pengguna. Pengguna media sosial yang menerima banyak pesan *spam* mungkin merasa terganggu dan kesulitan menemukan konten yang relevan di antara tumpukan pesan tidak diinginkannya itu. Hal ini dapat menurunkan kualitas pengalaman pengguna dan membuat mereka enggan menggunakan *platform* tersebut.

C. *Regular Expression (Regex)*

Regular expression, atau yang lebih dikenal sebagai *regex*, adalah serangkaian karakter yang membentuk pola pencarian [5]. *Regex* digunakan untuk mencocokkan teks dalam berbagai aplikasi pemrosesan teks, termasuk pencarian, pemfilteran, dan manipulasi teks. Dengan menggunakan pola yang ditentukan, *Regex* dapat menemukan dan mengekstrak substring tertentu dari teks, mengganti substring tersebut, atau melakukan validasi terhadap teks yang sesuai dengan pola tertentu.

Special Characters in Regular Expressions & their meanings

Character	Meaning	Example
*	Match **zero, one or more** of the previous	`Ah*` matches "Ahhhhh" or "A"
?	Match **zero or one** of the previous	`Ah?` matches "Ai" or "Ah"
+	Match **one or more** of the previous	`Ah+` matches "Ah" or "Ahhh" but not "A"
\	Used to **escape** a special character	`Hungry\?` matches "Hungry?"
.	Wildcard character, matches **any** character	`do.*` matches "dog", "door", "dot", etc.
( )	**Group** characters	See example for |
[ ]	Matches a **range** of characters	`[cbf]ar` matches "car", "bar", or "far" `[0-9]+` matches any positive integer `[a-zA-Z]` matches ascii letters a-z (uppercase and lower case) `[^0-9]` matches any character not 0-9.
|	Matche previous **OR** next character/group	`(Mon)|(Tues)day` matches "Monday" or "Tuesday"
{ }	Matches a specified **number of occurrences** of the previous	`[0-9]{3}` matches "315" but not "31" `[0-9]{2,4}` matches "12", "123", and "1234" `[0-9]{2,}` matches "1234567..."
^	**Beginning** of a string. Or within a character range [] negation.	`^http` matches strings that begin with http, such as a url. `[^0-9]` matches any character not 0-9.
$	**End** of a string.	`ing$` matches "exciting" but not "ingenious"

Gambar 3. Aturan Umum Pada *Regex*
Sumber:
https://computersciencewiki.org/index.php?title=File:Regex_Cheat_Sheet.png

Regex bekerja dengan cara mendefinisikan pola yang menggambarkan urutan karakter dalam string teks [5]. Pola ini terdiri dari kombinasi karakter biasa dan karakter khusus (metakarakter) yang memiliki makna tertentu [5]. Misalnya, dalam *Regex*, karakter titik (.) mewakili sembarang karakter tunggal, sementara tanda bintang (*) menunjukkan bahwa karakter sebelumnya dapat muncul nol atau lebih kali [5].

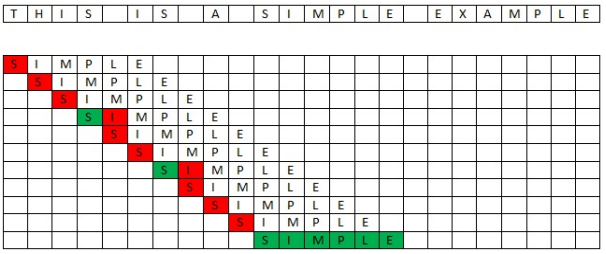
Ketika *Regex* diterapkan pada teks, mesin *Regex* akan memindai teks tersebut dan mencoba mencocokkan pola yang ditentukan dengan *substring* dalam teks [5]. Hasil dari pencocokan ini dapat berupa posisi substring yang cocok atau indikasi ditemukan atau tidaknya substring tersebut.

D. *Brute Force*

Brute force adalah metode pemecahan masalah yang menggunakan pendekatan pencarian semua kemungkinan solusi (*straight-forward*) secara sistematis hingga menemukan solusi yang tepat [6]. Dalam konteks pencocokan *string*, algoritma *brute force* mencoba mencocokkan pola dengan setiap posisi yang mungkin dalam teks, satu per satu, tanpa menggunakan heuristik atau optimasi lainnya [7]. Meskipun sederhana, metode ini sering kali tidak efisien untuk pencarian dalam teks yang panjang atau pola yang kompleks.

Algoritma *string matching* dengan *brute force* bekerja dengan cara sebagai berikut [7]:

1. Misalkan kita memiliki teks `T` dengan panjang `n` dan pola `P` dengan panjang `m`.
2. Algoritma memulai dengan memeriksa apakah pola `P` cocok dengan substring teks `T` yang dimulai dari posisi pertama.
3. Jika tidak cocok, algoritma akan bergeser satu posisi ke kanan dan memeriksa lagi.
4. Proses ini diulangi sampai pola `P` cocok dengan substring dari teks `T` atau sampai semua kemungkinan posisi telah diperiksa.



Gambar 4. Visualisasi Pencocokan *String* dengan Algoritma *Brute Force*
Sumber:
https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEiqB6K6BPKG_DKx6J2MTKpP0hXpQYYA8UrsU1q4rldbJ9mNYly-HWdOKYR0FwXwnkDzAV-gl6ZbpR1McsCfRJnhsa4XKA2o3ABWq6Ft5xM6zfO3vC_lj_Wx5y_UCjQStBByGXJDDqmXH4w/s1600/Brute+Force.jpg

Berikut adalah *pseudocode* untuk algoritma *string matching* dengan *brute force* [7]:

```
// Pseudocode Algoritma String Matching Brute Force

n = length of T

m = length of P

for i from 0 to n - m do

   j = 0

   while j < m and T[i + j] == P[j] do

        j = j + 1

   if j == m then

        return i  // Found

return -1  // Not found
```

Kompleksitas waktu dari algoritma *brute force* untuk *string matching* adalah `O(nm)` [7]. Ini karena dalam kasus terburuk, algoritma harus memeriksa setiap posisi dalam teks `T` dan membandingkan setiap karakter dalam pola `P` dengan substring dari teks.

E. *Knuth-Morris-Pratt (KMP)*

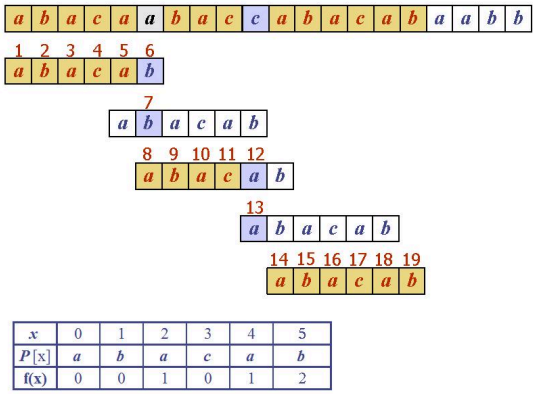
Algoritma Knuth-Morris-Pratt (KMP) adalah salah satu algoritma pencocokan *string* yang efisien, yang dikembangkan oleh Donald Knuth, Vaughan Pratt, dan James H. Morris pada tahun 1977 [7]. Algoritma ini dirancang untuk menemukan kemunculan pertama dari pola `P` dalam teks `T` dengan memanfaatkan informasi dari pencocokan sebelumnya untuk menghindari perbandingan yang tidak perlu.

Algoritma KMP bekerja dengan dua tahap utama: *preprocessing* dan pencocokan. Tahap *preprocessing* bertujuan untuk membangun tabel *border function* yang digunakan untuk menghindari perbandingan yang berulang. Tahap

pencocokan menggunakan tabel ini untuk membantu untuk melompat ke posisi yang tepat dalam pola ketika terjadi ketidakcocokan.

Algoritma *string matching* dengan KMP bekerja dengan cara sebagai berikut [7]:

1. Bangun tabel *border function* `b[k]` di mana `k` adalah indeks karater sebelum *mismatch*. `b[k]` menunjukkan panjang prefiks yang cocok yang juga merupakan sufiks untuk pola hingga posisi `i`.
2. Algoritma memindai teks `T` dari kiri ke kanan dan menggunakan tabel `b[k]` untuk menentukan pergeseran yang tepat ketika terjadi ketidakcocokan.
3. Dengan menggunakan informasi dari tabel `b[k]`, algoritma dapat melompat beberapa posisi ke depan dalam teks, sehingga mengurangi jumlah perbandingan.



x	0	1	2	3	4	5
P[x]	a	b	a	c	a	b
f(x)	0	0	1	0	1	2

Gambar 5. Visualisasi Pencocokan String dengan Algoritma KMP
Sumber: https://koding4fun.wordpress.com/wp-content/uploads/2010/05/kmpexample.jpg

Berikut adalah *pseudocode* untuk tahap *preprocessing* dan pencocokan dalam algoritma KMP [7]:

```
// Pseudocode border function
function computeBorder(pattern):
    m = length(pattern)
    b = array of length m
    b[0] = 0
    j = 0
    i = 1
    while i < m:
        if pattern[j] == pattern[i]:
                j = j + 1
                b[i] = j
                i = i + 1
        else if j > 0:
                j = b[j - 1]
        else:
                b[i] = 0
                i = i + 1
    return b

// Pseudocode KMP Searching
function kmpSearch(text, pattern):
    n = length(text)
    m = length(pattern)
    b = computeBorder(pattern)
    i = 0
    j = 0
    while i < n:
        if pattern[j] == text[i]:
                if j == m - 1:
                return i - m + 1  // Match found
                i = i + 1
                j = j + 1
        else if j > 0:
                j = b[j - 1] // Use border func
        else:
                i = i + 1
    return -1  // No match found
```

Kompleksitas waktu dari algoritma KMP adalah `O(n+m)` [7]. Tahap *preprocessing* memiliki kompleksitas `O(m)` karena setiap karakter dalam pola diperiksa sekali. Tahap pencocokan memiliki kompleksitas `O(n)` karena setiap karakter dalam teks diperiksa sekali dengan memanfaatkan tabel *border function* untuk melompat ke posisi yang tepat dalam pola.

F. Boyer-Moore

Algoritma Boyer-Moore (BM) adalah salah satu algoritma pencocokan *string* yang paling efisien, terutama ketika pola yang dicari lebih panjang. Algoritma ini dikembangkan oleh Robert S. Boyer dan J Strother Moore pada tahun 1977. Boyer-Moore bekerja dengan melakukan pencarian dari kanan ke kiri dalam pola dan menggunakan dua heuristik utama, yaitu *looking-glass* rule dan *character-jump* rule, untuk mempercepat proses pencarian dengan melompat beberapa posisi ketika terjadi ketidakcocokan.

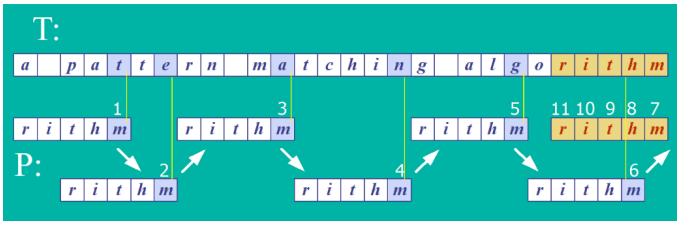
Algoritma *string matching* dengan BM bekerja dengan cara sebagai berikut [7]:

1. Bangun tabel *last occurence* dengan mengisi *array last occurrence* dengan posisi terakhir kemunculan setiap karakter dalam pola.
2. Jika panjang pola lebih besar dari panjang teks, maka pencarian dihentikan dan mengembalikan false.

3. Selama `i` lebih kecil dari panjang teks `n`, (`i` iterator pada teks dan `j` iterator pada pattern)
   Jika karakter dalam pattern sama dengan karakter dalam teks,
   a. Jika sudah mencapai awal pola, artinya pola ditemukan dalam teks, kembalikan true.
   b. Jika belum mencapai awal pola, geser satu posisi ke kiri pada teks dan pola.

   Jika karakter dalam pattern tidak sama dengan karakter dalam teks,

   a. Temukan posisi terakhir karakter teks yang tidak cocok dalam pola menggunakan array *last occurrence*.
   b. Geser pola ke kanan dengan jumlah yang ditentukan oleh aturan Boyer-Moore, yaitu `m - Math.Min(j, 1 + k)`.
   c. Kembalikan `j` ke posisi akhir pola.



Gambar 6. Visualisasi Pencocokan String dengan Algoritma KMP
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

Berikut adalah *pseudocode* untuk tahap *preprocessing* dan pencocokan dalam algoritma BM [7]:

```
// Pseudocode last occurrence
function buildLast(pattern):
   last = array of length 128 // ASCII character set
   for i from 0 to 127 do
       last[i] = -1 // Initialize array
   for i from 0 to length(pattern) - 1 do
       last[pattern[i]] = i
   return last


// Pseudocode BMSearch
function bmMatch(text, pattern):
   last = buildLast(pattern)
   n = length(text)
   m = length(pattern)
   i = m - 1
   if i > n - 1:
       return -1 // No match
   j = m - 1
   do
       if pattern[j] == text[i]:
           if j == 0:
               return i // Match found
           else:
               i = i - 1
               j = j - 1
       else:
           lo = last[text[i]] // Last occur
           i = i + m - Min(j, 1 + lo)
           j = m - 1
   while i <= n - 1
   return -1 // No match found
```

Kompleksitas waktu dari algoritma Boyer-Moore dalam kasus rata-rata adalah `O(n/m)`. Hal ini karena algoritma Boyer-Moore melakukan pergeseran yang cukup besar ketika terjadi ketidakcocokan. Dalam kasus terburuk, kompleksitasnya adalah `O(nm)` [7], namun ini jarang terjadi dalam praktik.

III. IMPLEMENTASI

Program pendeteksian *spam tweet* pada sosial media X atau Twitter akan dibuat menggunakan bahasa Typescript (Node.js). Struktur folder program terdiri dari folder `src` untuk menyimpan source code program, `test` untuk menyimpan test-case, dan `doc` untuk menyimpan pdf makalah.

Berikut merupakan struktur data yang digunakan untuk menyimpan solusi untuk setiap kata kunci atau *pattern* yang ditemukan *match*. Setiap solusi disimpan indeks mulainya dan juga indeks akhirnya pada teks.

```
export type Solution = {
  start: number; // Start of the solution
  end: number; // End of the solution (inclusive)
};
```

Gambar 7. Implementasi Struktur Data Penyimpanan Solusi
Sumber: Milik Pribadi

Implementasi *regex* menangani berbagai kata dari *spam word list*, *URL*, dan karakter-karakter yang berulang. Berikut adalah source code program untuk algoritma *string matching* menggunakan *regex*:

```ts
import { type Solution } from "./types";
import { spamWordsList } from "./spam-words-list";

export const regexMatch = (text: string): Solution[] => {
  // Convert the spam words list to a regex pattern
  const spamPattern = new RegExp(
    spamWordsList.map((word) => `\\b${word}\\b`).join("|"),
    "gi"
  );

  // Regex spam
  const spamPatterns = [
    spamPattern, // Keywords
    /(?:http:\/\/|https:\/\/|www\.)\S+/gi, // URLs
    /(.)\1{3,}/gi, // Repeated characters
  ];

  // Initialize the result
  const result: Solution[] = [];

  // Iterate through the spam patterns
  for (const spamPattern of spamPatterns) {
    // Find all the matches
    const matches = text.matchAll(spamPattern);

    // Iterate through the matches
    for (const match of matches) {
      // Add the match to the result
      result.push({
        start: match.index,
        end: match.index + match[0].length - 1,
      });
    }
  }

  // Return the result
  return result;
};
```

Gambar 8. Implementasi Algoritma *String Matching* dengan *Regex*
Sumber: Milik Pribadi

Berikut adalah *source code* program untuk algoritma *string matching* dengan *brute force*:

```ts
export const Bf = (text: string, pattern: string): number => {
  // Get the length of the text and the pattern
  const n = text.length;
  const m = pattern.length;

  // Perform the Brute Force search
  for (let i = 0; i <= n - m; i++) {
    // Check for a match at position i
    let j = 0;
    while (j < m && text[i + j] === pattern[j]) {
      j++;
    }

    // If the pattern is found, return the starting index
    if (j === m) {
      return i;
    }
  }

  // If the pattern is not found, return -1
  return -1;
};
```

Gambar 9. Implementasi Algoritma *String Matching* dengan *Brute Force*
Sumber: Milik Pribadi

Berikut adalah *source code* program untuk algoritma *string matching* KMP:

```ts
export const Kmp = (text: string, pattern: string): number => {
  // Get the length of the text and the pattern
  const n = text.length;
  const m = pattern.length;

  // Initialize the Longest prefix suffix (LPS) array
  const lps = new Array(m).fill(0);

  // Calculate the LPS array
  let i = 1;
  let j = 0;
  while (i < m) {
    if (pattern[i] === pattern[j]) {
      // If the characters match, increment both i and j
      lps[i] = j + 1;
      j++;
      i++;
    } else if (j > 0) {
      // If the characters do not match and j > 0, update j
      j = lps[j - 1];
    } else {
      // If the characters do not match and j = 0, increment i
      lps[i] = 0;
      i++;
    }
  }

  // Perform the KMP search
  i = 0;
  j = 0;
  while (i < n) {
    if (pattern[j] === text[i]) {
      // If the characters match
      if (j === m - 1) {
        // If the pattern is found, return the starting index
        return i - j;
      } else {
        // Increment both i and j
        i++;
        j++;
      }
    } else if (j > 0) {
      // If the characters do not match and j > 0, update j
      j = lps[j - 1];
    } else {
      // If the characters do not match and j = 0, increment i
      i++;
    }
  }

  // If the pattern is not found, return -1
  return -1;
};
```

Gambar 10. Implementasi Algoritma *String Matching* dengan KMP.
Sumber: Milik Pribadi

Berikut adalah *source code* program untuk algoritma *string matching* dengan BM:

```ts
export const Bm = (text: string, pattern: string): number => {
  // Get the length of the text and the pattern
  const n = text.length;
  const m = pattern.length;

  // Initialize the Bad Character Shift table
  const last = new Array(256).fill(-1);

  // Fill last occurrence of each character in pattern
  for (let i = 0; i < m; i++) {
    last[pattern.charCodeAt(i)] = i;
  }

  // Initialize the index
  let i = m - 1;
  // Not found if pattern is longer than text
  if (i > n - 1) {
    return -1;
  }

  let j = m - 1;
  // Search for the pattern in the text
  while (i <= n - 1) {
    if (pattern[j] === text[i]) {
      if (j === 0) {
        // Pattern found
        return i;
      }
      // Move to the left
      i--;
      j--;
    } else {
      // Shift the pattern to the right
      i = i + m - Math.min(j, 1 + last[text.charCodeAt(i)]);
      j = m - 1;
    }
  }

  // Pattern not found
  return -1;
};
```

Gambar 11. Implementasi Algoritma *String Matching* dengan BM
Sumber: Milik Pribadi

Agar solusi program lebih mudah dicetak, solusi perlu direduksi agar setiap solusi tidak beririsan. Berikut merupakan *source code* untuk mereduksi solusi dan mencetak text yang sudah disorot (*highlight*) *pattern*-nya.

```ts
import chalk from "chalk";
import type { Solution } from "./types";

export const getNoSubsetSolution = (
  initialSolution: Solution[]
): Solution[] => {
  // No solution
  if (initialSolution.length === 0) {
    return [];
  }

  // [0, 3] [4, 5] [1, 2] [1,4] returns [0, 5]

  // Sort the solutions by start index
  const solutions = initialSolution.sort((a, b) => a.start - b.start);

  // Initialize the result
  const result: Solution[] = [solutions[0]];

  // Iterate through the solutions
  for (let i = 1; i < solutions.length; i++) {
    const currentSolution = solutions[i];
    const lastSolution = result[result.length - 1];

    // Check if the current solution is a subset of the last solution
    if (currentSolution.start <= lastSolution.end) {
      // Update the last solution
      result[result.length - 1] = {
        start: lastSolution.start,
        end: Math.max(lastSolution.end, currentSolution.end),
      };
    } else {
      // Add the current solution to the result
      result.push(currentSolution);
    }
  }

  return result;
};

export const printHighlitedText = (
  text: string,
  solutions: Solution[]
): void => {
  const noSubsetSolution = getNoSubsetSolution(solutions);

  // For each solution, highlight the text
  let highlightedText = "";

  // Initialize the start index
  let i = 0;

  for (const solution of noSubsetSolution) {
    // Add the text before the solution
    highlightedText += text.substring(i, solution.start);

    // Highlight the solution
    highlightedText += chalk.red(
      text.substring(solution.start, solution.end + 1)
    );

    // Update the start index
    i = solution.end + 1;
  }

  // Add the remaining text
  highlightedText += text.substring(i);

  // Print the highlighted text
  console.log(highlightedText);
};
```

Gambar 12. Implementasi Pencetakan Hasil Solusi dengan Penyorotan pada *Pattern* yang Ditemukan
Sumber: Milik Pribadi

Terakhir, berikut merupakan *source code* program utama yang menggunakan seluruh modul-modul yang telah disebutkan sebelumnya.

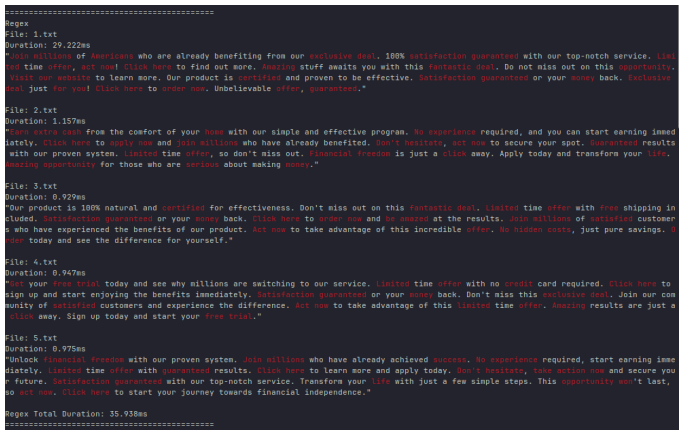
```
import { Bf } from "./bf";
import { Kmp } from "./kmp";
import { Bm } from "./bm";
import { regexMatch } from "./regex";
import chalk from "chalk";
import { printHighlitedText } from "./print-highlight";
import { spamWordsList } from "./spam-words-list";
import { readFileSync } from "fs";
import { readdirSync } from "fs";
import type { Solution } from "./types";

const main = () => {
  // Read test folder
  const testFolder = "./test";
  const files = readdirSync(testFolder).sort(
    (f1, f2) => parseInt(f1.split(".")[0]) - parseInt(f2.split(".")[0])
  );

  // Regex
  console.log("============================================");
  console.log("Regex");
  console.time("Regex Total Duration");
  files.forEach((file) => {
    console.log("File:", file);
    console.time("Duration");

    // Read the content of the file
    const text = readFileSync(`${testFolder}/${file}`, "utf8");

    // Search for spam words using Regex
    const solutions = regexMatch(text);

    // Stop the timer
    console.timeEnd("Duration");

    // Print
    printHighlitedText(text, solutions);

    // If no spam words are found
    if (!solutions.length) {
      console.log(chalk.red("No spam words found"));
    }
    console.log();
  });
  console.timeEnd("Regex Total Duration");
  console.log("============================================");

  // For each file in the test folder, read the content and search for the first spam word
  console.log("============================================");
  console.log("Brute Force");
  console.time("Brute Force Total Duration");
  files.forEach((file) => {
    console.log("File:", file);
    console.time("Duration");

    // Read the content of the file
    const text = readFileSync(`${testFolder}/${file}`, "utf8").toLowerCase();

    // Search for spam words using Brute Force
    const solutions: Solution[] = [];
    for (const word of spamWordsList) {
      const res = Bf(text.toLowerCase(), word.toLowerCase()); // Case insensitive
      if (res !== -1) {
        solutions.push({ start: res, end: res + word.length - 1 });
      }
    }

    // Stop the timer
    console.timeEnd("Duration");

    // Print
    printHighlitedText(text, solutions);

    // If no spam words are found
    if (!solutions.length) {
      console.log(chalk.red("No spam words found"));
    }
    console.log();
  });
  console.timeEnd("Brute Force Total Duration");
  console.log("============================================");
```

Gambar 13. Implementasi Program Utama (Bagian 1)
Sumber: Milik Pribadi

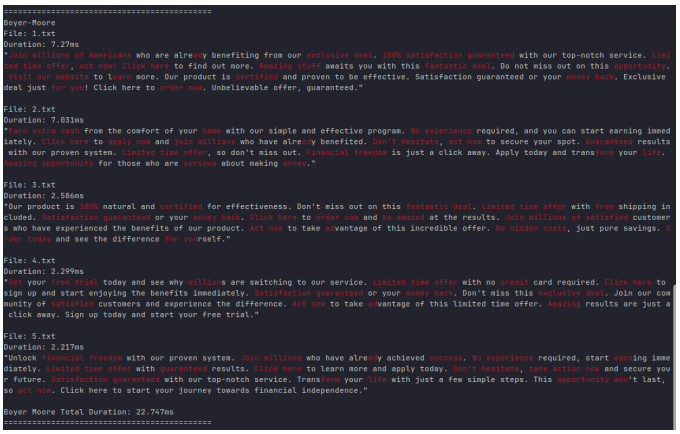
```
  console.log("============================================");
  console.log("Knuth-Morris-Pratt");
  console.time("Knuth Morris Pratt Total Duration");
  files.forEach((file) => {
    console.log("File:", file);
    console.time("Duration");

    // Read the content of the file
    const text = readFileSync(`${testFolder}/${file}`, "utf8");

    // Search for spam words using Knuth-Morris-Pratt
    const solutions: Solution[] = [];
    for (const word of spamWordsList) {
      const res = Kmp(text.toLowerCase(), word.toLowerCase()); // Case insensitive
      if (res !== -1) {
        solutions.push({ start: res, end: res + word.length - 1 });
      }
    }

    // Stop the timer
    console.timeEnd("Duration");

    // Print
    printHighlitedText(text, solutions);

    // If no spam words are found
    if (!solutions.length) {
      console.log(chalk.red("No spam words found"));
    }
    console.log();
  });
  console.timeEnd("Knuth Morris Pratt Total Duration");
  console.log("============================================");

  // Boyer-Moore
  console.log("============================================");
  console.log("Boyer-Moore");
  console.time("Boyer Moore Total Duration");
  files.forEach((file) => {
    console.log("File:", file);
    console.time("Duration");

    // Read the content of the file
    const text = readFileSync(`${testFolder}/${file}`, "utf8");

    // Search for spam words using Boyer-Moore
    const solutions: Solution[] = [];
    for (const word of spamWordsList) {
      const res = Bm(text.toLowerCase(), word.toLowerCase()); // Case insensitive
      if (res !== -1) {
        solutions.push({ start: res, end: res + word.length - 1 });
      }
    }

    // Stop the timer
    console.timeEnd("Duration");

    // Print the highlighted text
    printHighlitedText(text, solutions);

    // If no spam words are found
    if (!solutions.length) {
      console.log(chalk.red("No spam words found"));
    }
    console.log();
  });
  console.timeEnd("Boyer Moore Total Duration");
  console.log("============================================");
};

main();
```

Gambar 14. Implementasi Program Utama (Bagian 2)
Sumber: Milik Pribadi

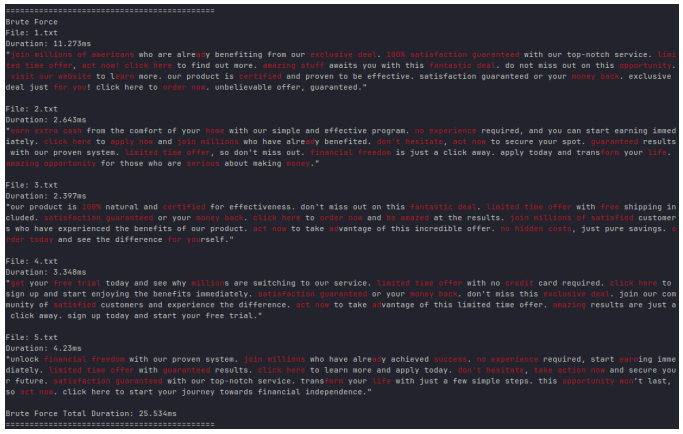
Contoh keluaran dari program ketika dijalankan seperti berikut:



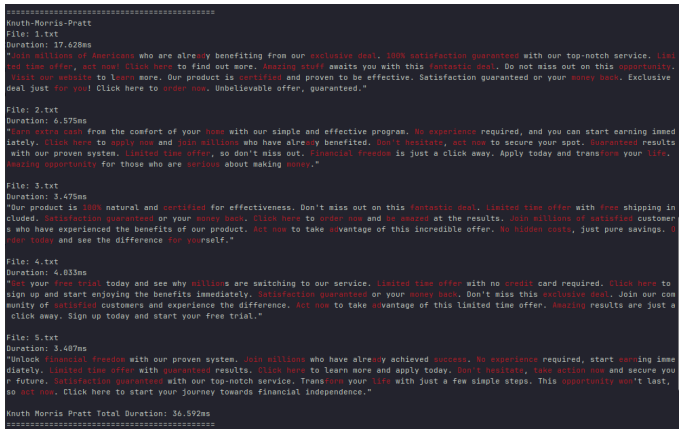
Gambar 15. Hasil Eksekusi Algoritma *String Matching* dengan *Regex*
Sumber: Milik Pribadi



Gambar 16. Hasil Eksekusi Algoritma *String Matching* dengan BM
Sumber: Milik Pribadi



Gambar 16. Hasil Eksekusi Algoritma *String Matching* dengan *Brute Force*
Sumber: Milik Pribadi



Gambar 17. Hasil Eksekusi Algoritma *String Matching* dengan KMP
Sumber: Milik Pribadi

IV. Analisis

Dalam 10 kali percobaan menjalankan program diatas, diperoleh beberapa data berikut:

TABLE I. Hasil Percobaan Program Utama

Percobaan	Regex (ms)	Brute Force (ms)	KMP (ms)	BM (ms)
1	35.95	25.54	36.59	22.75
2	35.44	28.75	33.85	25.36
3	78.05	36.19	48.76	36.09
4	46.92	29.21	37.02	26.89
5	43.95	39.80	54.57	27.73
6	39.47	26.95	22.76	22.15
7	37.04	26.24	32.73	36.38
8	40.10	26.52	35.61	22.43
9	46.30	28.55	32.68	24.97
10	32.72	29.02	33.44	25.64
Avg	43.59	29.68	36.80	27.04
Std Dev	13.00	4.64	8.89	5.19
N (1 s)	22.94	33.70	27.17	36.98

Dari hasil eksperimen tersebut dapat dilihat bahwa rata-rata pemrosesan paling cepat dilakukan oleh algoritma BM, kemudian diikuti oleh algoritma *brute force*, kemudian algoritma KMP dan terakhir *regex*. *N* adalah jumlah rata-rata *tweet* yang bisa diproses dalam satu detik (1000 ms) menggunakan metode tertentu ($N = 1000/t_{avg}$). Artinya, algoritma yang paling efisien untuk memproses pendeteksian *tweet spam* pada twitter adalah algoritma BM. Hal ini disebabkan kompleksitas algoritma BM pada kasus terbaik lebih baik dibandingkan algoritma-algoritma lain-nya.

V. Kesimpulan

Regex dan algoritma string matching *brute force*, KMP, dan BM dapat digunakan sebagai pendeteksi *tweet spam* pada sosial media X (Twitter). Dari percobaan sebelumnya, dapat disimpulkan bahwa algoritma *string matching* yang paling efisien untuk pendeteksian *tweet spam* pada twitter adalah algoritma BM.

VI. Ucapan Terima Kasih

Penulis berterima kasih kepada Tuhan yang Maha Esa karena telah memberikan kemudahan dalam penulisan makalah ini. Penulis juga ingin berterima kasih kepada Dr. Ir. Rinaldi, M.T. selaku dosen dan pembimbing penulis dalam mata kuliah IF2211 Strategi Algoritma. Selain itu, penulis juga ingin berterima kasih kepada keluarga serta teman-teman yang sudah mendukung penulis dalam menyelesaikan penulisan makalah ini. Terakhir, penulis juga ingin berterima kasih kepada *band rock* amerika, The Eagles yang sudah menghibur penulis dengan lagu-lagunya dalam menyelesaikan penulisan makalah ini.

Video Link at Youtube

https://youtu.be/mCPlcZ7K4Ns?si=T7CMowlUH3teIhO8

Repository Link at Github

https://github.com/dewodt/makalah-stima

References

[1] Gil, Paul. 2023. "What Is X (Formerly Twitter)?". https://www.lifewire.com/what-exactly-is-twitter-2483331 (Diakses pada 12 Juni 2024).

[2] Boyd, Danah, Scott Golder, Gilad Lotan. 2010. "Tweet, Tweet, Retweet: Conversational Aspects of Retweeting on Twitter". https://ieeexplore.ieee.org/document/5428313 (Diakses pada 12 Juni 2024).

[3] MacArthur, Amanda. 2024. "The Real History of X (Formerly Twitter), in Brief". https://www.lifewire.com/history-of-twitter-3288854 (Diakses pada 12 Juni 2024).

[4] Ferrara, Emilio. 2019. "The History of Digital Spam". https://cacm.acm.org/research/the-history-of-digital-spam/ (Diakses pada 12 Juni 2024).

[5] Wibisono, Yudi, Masayu Leylia Khodra. 2020. "Modul Praktikum Kuliah Pengantar Regular Expression". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2019-2020/Modul-Praktikum-NLP-Regex.pdf (Diakses pada 12 Juni 2024).

[6] Munir, Rinaldi. 2023. "Algoritma Brute Force Bagian 1". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf (Diakses pada 12 Juni 2024).

[7] Munir, Rinaldi. 2023. "Pencocokan String". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf (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

Dewantoro Triatmojo 13522011