BAB II
KAJIAN PUSTAKA DAN DASAR TEORI

2.1. Kajian Pustaka

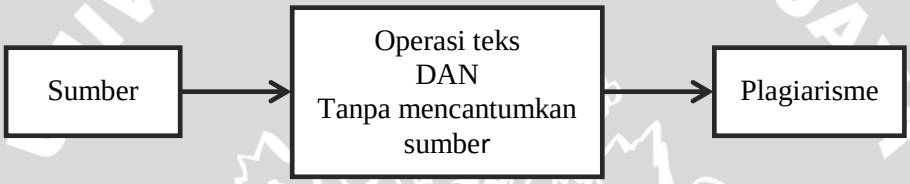
Rajender Singh Chillar dan Barjesh Kochar dalam penelitiannya yang berjudul "*Rabin-Karp-Matcher: String Matching Technique*" menyajikan algoritma pencocokan *string* varian baru (Chillar, 2008). Algoritma varian Chillar-Kochar ini memodifikasi algoritma pencocokan *string* Rabin-Karp yang memakai fungsi *hash*. Metode Rabin-Karp yang konvensional membandingkan nilai *hash* yang merupakan produk modulo, dan mencocokkan karakter *substring* antar dokumen bila ditemui nilai *hash* yang sama. Karena ada beberapa kemungkinan ditemui *substring* yang memiliki nilai *hash* yang sama meskipun karakternya berbeda, ini akan menimbulkan kondisi *spurious hit* (sisa modulo substring sama tapi karakter berbeda). Kondisi *spurious hit* tentu menjadi beban komputasi karena harus melakukan pencocokan karakter yang tidak perlu.Tetapi dalam kasus varian Rabin-Karp yang telah dimodifikasi, Chillar-Kochar selain mencocokkan sisa hasil bagi dengan modulo (*remain*) juga mencocokkan hasil bagi dengan modulo (*quotient*). Hal ini tentu mengurangi bahkan meniadakan *spurious hit* sehingga menghindari pencocokan karakter yang tidak perlu. Metode ini menghasilkan kompleksitas waktu $O(n-m+1)$, lebih baik dari algoritma Rabin-Karp konvensional yang kompleksitas waktunya $O((n-m+1)*m)$ (Chillar, 2008).

Untuk mengatasi adanya penggantian kata dengan synonym Mudafiq dkk. dalam "*Aplikasi Pendeteksi Duplikasi Dokumen Teks Bahasa Indonesia Menggunakan Algoritma Winnowing Dengan Metode K-Gram Dan Synonym Recognition"* menggunakan pendekatan *synonym recognition* (Mudafiq, 2012). *Synonym recognition* adalah metode yang dipakai untuk mendeteksi kata-kata yang mengadung *synonym*. Pendekatan ini dilakukan untuk membantu meningkatkan tingkat kesamaan yang ditemukan dalam mendeteksi plagiarisme. Penggunaan *synonym recognition* terbukti mampu meningkatkan deteksi kesamaan kata dengan perbedaan mencapai ± 0.82 % lebih besar daripada tanpa menggunakan *synonym recognition* (Mudafiq, 2012).

2.2. Plagiarisme

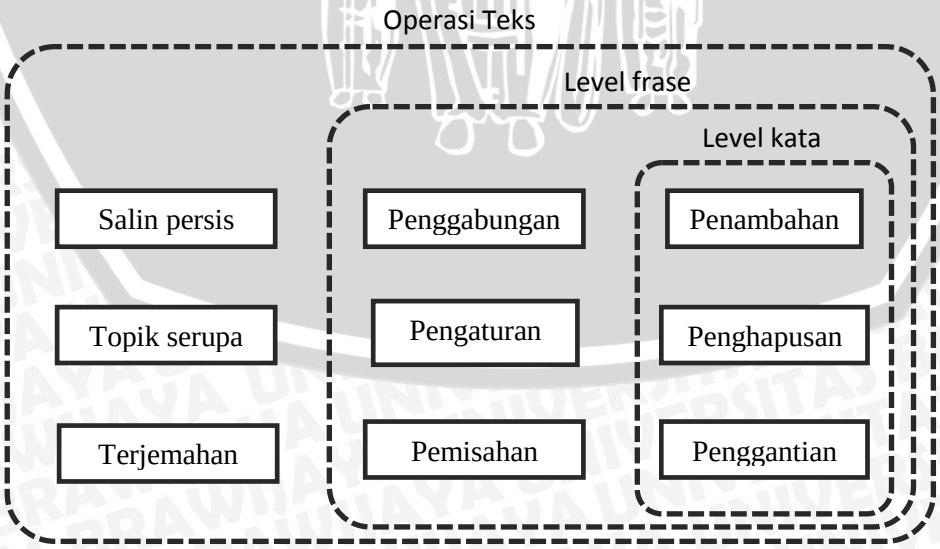
2.2.1. Definisi

Plagiarisme atau plagiat adalah tindakan mengambil alih ide, metoda, kata-kata hasil karya orang lain tanpa menyebutkan sumber asli dengan jelas dan menjadikannya seolah karya sendiri. Sebuah pencurian literatur. Berasal dari kata Latin *plagiarus* yang berarti penculik, pembajak; plagiarisme sudah menjadi ancaman serius selama berabad-abad bagi dunia literatur dan terutama insitusi akademik. Plagiarisme termasuk dalam pelanggaran hak cipta dan pelanggaran etika. Pelaku plagiat dikenal dengan sebutan plagiator. Penulis mendeskripsikan plagiarisme seperti pada Gambar2.1(Almquist, 2010).



Gambar 2.1. Definisi plagiarisme.
(Sumber: Per Almquist, 2010)

Sebuah teks memiliki sumber berupa nama penulis, tempat, dan kapan teks tersebut dibuat. Tidak menutup kemungkinan teks tersebut diakses dan dilakukan beberapa operasi teks baik parsial maupun keseluruhan seperti pada Gambar 2.2.



Gambar 2.2. Operasi-operasi teks yang bisa mengarah pada plagiarisme.
(Sumber: Per Almquist, 2010)

Operasi teks terbagi ke dalam beberapa tingkat tergantung seberapa luas akses atau modifikasi yang dilakukan terhadap teks. Operasi pada level kata merupakan yang paling spesifik, sehingga tindakan level kata juga beroperasi dalam level frase, dan begitu seterusnya. Sebuah dokumen teks yang telah dimanipulasi haruslah mencantumkan asal sumbernya karena jika tidak teks tersebut dikategorikan plagiat terhadap teks sumber.

Ada beberapa kategori yang termasuk dalam plagiarisme, antara lain (Maurer, 2006):

- Salin – tempel. Penyalinan kata per kata secara langsung tanpa menyertakan sumber sama sekali.Menyalin terlalu banyak kata & ide juga termasuk plagiarisme meskipun mengutip sumber lengkap sekalipun.
- Parafrase atau menggubah kalimat dengan kata-kata berbeda bermakna sama (meringkas) tapi tidak mencantumkan sumber dengan jelas.
- Menerjemahkan naskah asing tanpa referensi sumber asli.
- Plagiarisme artistik. Menyajikan ide atau karya orang lain dalam bentuk lain misal tabel, Gambar, dan sebagainya tanpa kutipan.
- Plagiarisme ide. Serupa dengan plagiarisme kepengarangan (*authorship*) dimana ide orang lain atau kelompok diakui sebagai karya/hasil pemikiran pribadi tanpa menyertakan peran/kontribusi orang lain.
- Plagiarisme *source code*. Menggunakan kode program orang lain tanpa ijin atau rujukan.
- Penulisan tanda kutip yang tidak tepat. Atau kesalahan pada prosedur penulisan rujukan sehingga memberikan informasi yang salah tentang sumber referensi.

2.2.2. Deteksi Plagiarisme

Deteksi plagiarisme bisa dilakukan dengan membandingkan dua atau lebih dokumen teks. Satu dokumen yang adalah sumber dan dokumen yang lain yang diduga hasil jiplakan. Dan untuk mengetahui adanya plagiarisme ditinjau dari derajat kesamaan kedua teks.Proses pendeteksian plagiarisme bisa dikerjakan dengan cara investigasi manual ataupun dengan bantuan komputer.

Ada beberapa teknik analisa plagiarisme yang bisa digunakan untuk mendeteksi adanya plagiarisme(Stein, 2006).

1. Pencocokan *Substring* (*Substring Matching*). Pendekatan ini mengidentifikasi kecocokan tiap *string* dengan membandingkan keseluruhan isi dokumen. Pencocokan *substring* bisa diaplikasikan pada dokumen yang berukuran besar, meskipun memakan waktu yang lama. Meskipun demikian cara ini terbilang cukup efektif karena kecocokan *string* bisa langsung dijadikan indikator penjiplakan. Satu kekurangannya lagi yaitu dokumen yang dibandingkan haruslah berada pada penyimpanan lokal. Dokumen-dokumen yang tersebar luas di internet tidak bisa dilakukan metode ini.
2. Kesamaan Kata Kunci (*Keyword Similarity*). Metode ini mengasumsikan bahwa plagiarisme biasanya terjadi pada dokumen yang memiliki kesamaan kata kunci. Gagasan dari metode ini adalah mengekstrak kata kunci dari sebuah dokumen dan membandingkannya dengan kata kunci dokumen yang lain. Jika jumlah kesamaan melebihi ambang batas, dokumen yang dibandingkan dibagi menjadi bagian-bagian yang lebih kecil. Yang kemudian pecahan tersebut dibandingkan lagi secara terus menerus.
3. Dokumen *Fingerprinting* (*Fingerprint Analysis*). Merupakan pendekatan paling populer untuk mendeteksi dokumen dengan rangkaian teks yang saling tumpang tindih. Dengan menggunakan teknik *hashing* metode ini sanggup menemukan kecocokan secara akurat baik dalam keseluruhan teks atau sebagian. *Hashing* adalah teknik mengubah teks (*string*) ke dalam bilangan integer yang unik. Dan *hash* yang sama menandakan adanya kesamaan pada *string* juga. Algoritma yang menggunakan metode ini adalah Winnowing, Manber dan juga Rabin-Karp.

Ada tiga sifat yang menjadi persyaratan khusus algoritma deteksi plagiarisme, yaitu (Schleimer, 2003):

1. *White insensitivity*. Dalam proses pencocokan teks seharusnya hanya memperhatikan abjad dan angka, tanpa terpengaruh spasi, tanda baca, huruf kapital/tidak, dan sebagainya. Meskipun ada juga beberapa aplikasi yang menginginkan penococokan tidak terpengaruh nama-nama variable.

2. *Noise Suppression.* Mengurangi pengecekan pada kata hubung, kata sambung atau *stopword* lainnya yang terlalu umum digunakan seperti "yang, dan, kalau" dan sebagainya. Kata-kata yang memiliki kecocokan haruslah cukup panjang dan bukan kata umum atau idiom untuk mengimplikasikan bahwa materi tersebut telah dijiplak.
3. *Position Independence.* Posisi kata dan urutan paragraf yang acak harusnya tidak berdampak pada penemuan kecocokan. Kecocokan yang ditemukan tidak bergantung pada posisi kata. Penambahan dan pengurangan bagian dari dokumen pun juga harus tidak mempengaruhi kecocokan pada dokumen yang baru.

2.3. Pencocokan *String* (*String Matching*)

2.3.1. Definisi

Menurut kamus algoritma danstruktur data, *National Institute of Standards and Technologies (NIST)*, *string* didefinisikan sebagai susunan karakter baik itu alfabet, angka, atau karakter yang lain sehingga membentuk kata, frase atau kalimat (Black, 1998). *String* umumnya direpresentasikan sebagai data *array.*

Pencocokan *string* (*string matching*) didefinisikan sebagai sebuah permasalahan untuk menemukan pola kecocokan susunan karakter *string* di dalam *string* lain atau bagian dari teks. Dibanding semua permasalahan *string* lainnya *string matching* merupakan yang paling sederhana.*String matching* memegang peranan yang sangat penting dalam domain pemrosesan teks, kompresi teks, analisis leksikal, dan temu balik informasi. Dan teknik untuk menyelesaikan permasalahan *string* biasanya akan berimplikasi langsung ke aplikasi *string* lainnya. Karena itu pencocokan string juga banyak diimplementasikan pada *spell-checker, search engine*, bahkan sampai bidang bionformatika seperti pencocokan rangkaian DNA.

Prinsip dari *string matching* adalah mencari semua kemunculan *string* pendek yang disebut pola (*pattern*) $P[0...m-1]$ di dalam *string* yang lebih panjang yang disebut teks $T[0...n-1]$. Dengan m dan n yang adalah panjang *string*. Kedua *string* dibentuk dari kumpulan karakter terbatas yang disebut alfabet, dinotasikan Σ dengan ukuran σ.

Secara garis besar *string matching* bisa dikelompokkan menjadi dua kategori, yaitu:

1. *Exact string matching*. Merupakan pencocokan *string* yang memperhatikan ketepatan *string* dimana jumlah dan urutan karakter *string* yang dicocokkan harus tepat sama. Contoh: kata "*sport*" hanya akan cocok dengan kata "*sport*".
2. *Inexact string matching*. Disebut juga *fuzzy string matching* karena mencocokkan *string* secara samar-samar. *String* yang dicocokkan di sini tidak harus memiliki kesamaan karakter, karena bisa saja berbeda dalam jumlah atau urutan susunan karakter. *String* yang dicocokkan dalam metode ini memiliki kemiripan, baik itu dari segi penulisan (*approximate string matching*) atau kemiripan dari segi pengucapan (*phonetic string matching*). Contoh: kata "frase" dengan kata "frasa", atau "system" dengan "sistem".

2.3.2. Algoritma Pencocokan *String* (*String Matching Algorithm*)

Algoritma pencocokan *strin g*merupakan komponen dasar yang dipakai dalam implementasi perangkat lunak praktis yang bekerja di bawah sistem operasi kebanyakan. Algoritma pencocokan *string* adalah algoritma yang dikembangkan untuk mempermudah proses pencocokan *string*.

Berdasarkan cara membaca teks algoritma pencocokan *string* dibagi menjadi dua, yaitu:

1. Dari kiri ke kanan. Merupakan jenis algoritma yang paling umum dan paling banyak ditemui. Bahkan sebagian besar algoritma pencocokan *string* mengadopsi cara ini, membaca teks dari kiri kanan. Contohnya adalah algoritma *brute force*, Rabin-Karp, Shift-Or, dan Knuth-Morris-Pratt.
2. Dari kanan ke kiri. Cara pengecekan teks yang tidak lazim ini menjadi ciri khas algoritma Boyer-Moore. Algoritma ini juga dikenal sebagai algoritma pencocokan paling efisien pada penggunaan biasa dan menjadi standar algoritma pencarian *string*.

Total ada sekitar 35 algoritma pencocokan *string* yang banyak digunakan. Dengan segala kelebihan dan kekurangannya algoritma-algoritma ini menjadi alternatif untuk proses pencocokan *string*. Umumnya algoritma pencocokan *string* dipilih berdasarkan tingkat akurasinya dan waktu kompleksitasnya. Berikut perbandingan kompleksitas waktu algoritma pencocokan *string* yang populer dan sudah banyak dipakai (Wirawan, 2003).

Tabel 2.1.Perbandingan kompleksitas waktu algoritma pencocokan *string*.

Algoritma	Fase *Pre-processing*	Fase Pencarian
Brute force	0 (tidak ada)	$O(mn)$
Rabin-Karp	$O(m)$	$O(mn)$
Shift-Or	$\boldsymbol{O}(m + \sigma)$	$O(n)$
Knuth-Morris-Pratt	$O(m)$	$O(m+n)$
Boyer-Moore	$\boldsymbol{O}(m + \sigma)$	$O(m/n)$, $O(n)$

2.4. Algoritma Rabin-Karp

Algoritma yang ditemukanoleh Michael O. Rabin dan Richard M. Karp tahun 1987 ini merupakan versi penyempurnaan dari *naive string matching* pada umumnya. Metode pencocokan *string* naif seperti *brute force* bekerja dengan menggeser *substring* selangkah demi selangkah untuk menemukan setiap kemungkinan kecocokan*string*. Untuk setiap pergeseran, setiap karakter dari pola (inputan)dicocokkan dengan karakter dalam teks sampai ujung teks. Tapi tidak seperti algoritma naif, algoritma Rabin-Karp menggunakan fungsi *hash*. Algoritma ini membandingkan nilai *hash* pola dengan nilai *hash* teks. Dan bila ditemui kecocokan nilai *hash* baru akan dicocokkan *string* keduanya. Jadi algoritma Rabin-Karp lebih cepat dan optimal karena tidak harus mencocokkan tiap karakter dalam semua pergeseran (Geeksforgeeks, 2011).

Algoritma Rabin-Karp memerlukan *pre-processing* untuk menghitung nilai *hash* dari pola dan semua *substring* dalam teks dengan panjang *m*. Untuk didapatkan nilai *hash* yang unik dan waktu *pre-processing* yang cepat dibutuhkan fungsi *hash* yang efisien. Selain menghasilkan nilai *hash* dengan perbedaan cukup tinggi antar *string*, fungsi *hash* yang baik harus mampu menghitung *hash* pada pergeseran berikutnya dari nilai *hash* sebelumnya.

2.4.1. *Hashing*

Hashing adalah proses mengubah karakter dari bentuk *string* ke dalam bilangan integer dengan panjang tertentu yang disebut nilai *hash* (*hash value*). Bentuk baru ini bersifat unik dan dijadikan sebagai fingerprint (penanda). Hal ini tentu memudahkan dalam proses pencocokan karena memakai beberapa digit angka dengan 10 kemungkinan dibandingkan *string* dengan panjang bervariasi dan 26 kemungkinan karakter. Nilai *hash* diperoleh dari teknik yang disebut fungsi *hash*. Fungsi *hash*dan pemilihan *hash* value inilah yang menjadi kunci kecepatan waktu komputasi algoritma Rabin-Karp.

Dalam mencari nilai *hash* Rabin-Karp memakai aturan Horner dan *operand denumerator* modulo. Modulo digunakan untuk menghasilkan nilai *hash* dengan mencari sisa hasil bagi dari output aturan Horner.Untuk sebuah kata (*string*) *w* dengan panjang *m*,rumus *hash*(*w*) dituliskan seperti pada persamaan (2-1) berikut (Lecroq, 1997):

$$Hash(w[0...m\text{-}1]) = (w[0]*b^{m-1} + w[1]*b^{m-2} + ... + w[m\text{-}1]*b^{0}) \bmod q \qquad (2\text{-}1)$$

Dengan *hash* adalah nilai *hash*,*w*[*i*] adalah nilai ASCII karakter ke-*i*,*b* adalah basis, *m* adalah banyaknya karakter dalam pola, *q* adalah *operand denumerator* modulo.Contoh, jika terdapat *substring* "hi" dan dipilih basis $b = 10$ dan $q = 101$, maka $hash(\text{‘hi’}) = (\text{‘h’}*b^1 + \text{‘i’}*b^0) \bmod q = (104*10+105*1) \bmod 101 = 34$.

Umumnya basis dipilih 10 untuk merepresentasikan sepuluh kemungkinan angka (‘0’-‘9’). Fungsi dari modulo adalah untuk memperkecil memori yang dipakai karena nilai variable membesar secara eksponensial berbanding lurus dengan panjang pola. Dan modulo biasanya memakai bilangan prima yang cukup besar untuk memperlebar varian output sehingga mengurangi kemungkinan dua *corresponding number value* yang sama. Dari Persamaan (2-1) di atas terlihat kompleksitas waktu yang dibutuhan algoritma Rabin-Karp dalam *pre-processing* (penghitungan *hash* dengan panjang *m*) adalah *O*(*m*).

2.4.2. *RollingHash*

Salah satu keunggulan Rabin-Karp yang lain adalah kemampuannya untuk meminimalisi perhitungan dalam algoritma yaitu dengan *rolling hash.* Alih-alih menghitung *hash* semua *substring* di setiap pergeseran, metode ini memodifikasi nilai *hash* dari *substring* yang sebelumya. Hal ini tentu menghemat waktu pencarian nilai *hash* dan penggunaan memori.

Dengan *shift* (pergeseran) *s, hash* (*T*[*s*+1 .. *s*+*m*]) bisa dihitung secara efisien dari *hash* (*T*[*s* .. *s*+*m*-1]) dan *T*[*s*+*m*].Dengan kata lain *hash* (*T*[*s*+1 .. *s*+*m*])= *rehash* (*T*[*s*], *T*[*s*+*m*], *hash*(*T*[*s* .. *s*+*m*-1])) dan *rehash* membutuhkan waktu konstan *O*(1) per operasi.

Hash(*T*[*s* .. *s*+*m*-1]) adalah nilai *hash* pada pergeseran *s.*

Hash(*T*[*s*+1 .. *s*+*m*]) adalah nilai *hash substring* berikutnya (posisi pergeseran *s*+1).

$$Rehash\ (x,y,hash) = ((hash - x * b^{m-1}) * b + y) \bmod q \qquad (2\text{-}2)$$

Hash adalah *hash* sebelumnya,*x* adalah nilai ASCII karakter awal *substringT*[*s* .. *s*+*m*-1] yang akan dihapus (*old-hi-order-digit*),*y* adalah nilai ASCII karakter terakhir *substring T*[*s*+1 .. *s*+*m*] (*new-hi-order-digit*). Dengan *b* adalah basis dan *q* adalah modulo.

Berikutnya proses pencarian *string* hanya mencocokkan *hash*(*P*) dengan *hash* (*T*[*s* .. *s*+*m*-1]) dengan $0 \leq j < n-m$. Dan jika ditemui kecocokan *hash* akan diperiksa karakter-karakter untuk menemukan *P*= *T*[*s* .. *s*+*m*-1]. Dengan waktu konstan *O*(*m*) selama *hashing* dan *O*(*n*) selama pembandingan *hash* maka total waktu komputasi untuk melakukan semua operasi adalah *O*(*m*+*n*) (linear). Kemungkinan terburuk *O*(*mn*) sangat jarang terjadi. Kasus ini terjadi jika mencari pola AAAA di dalam AAAAAAA, sehingga kecocokan terjadi di setiap posisi dan akan selalu membandingkan karakter di setiap pergeseran (PEGWiki, 2011).

2.4.3. Prinsip Kerja Algoritma Rabin-Karp

Secara prinsip cara kerja algoritma Rabin-Karp dijabarkan sebagai berikut:

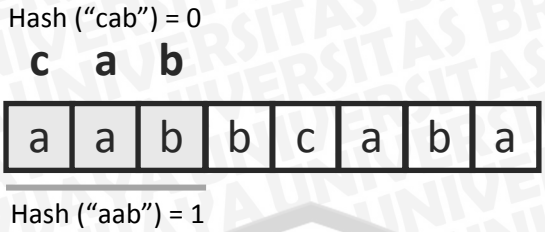
1. Diberikan inputan pola dengan panjang *m*, dilakukan *hashing*.
2. Dilakukan *hashing* terhadap *m* karakter pertama dari *string* teks (*substring*).
3. Membandingkan nilai *hash* pola dengan *substring* teks. Jika sama, dilakukan pengecekan terhadap karakter-karakter dalam pola dengan *substring*.
4. *Substring* teks bergeser ke kanan. Dilakukan *hashing* lagi dan mengulangi proses seperti pada langkah 3 sampai akhir teks (Sparknote, 2013).

Algoritma Rabin-Karp berdasar padafakta jika dua *string* sama maka nilai *hash*-nya juga sama. Tapi *string* yang berbeda bisa juga memiliki nilai *hash* yang sama, karena itu diperlukan pengecekan ulang terhadap karakter-karakter *string*. Meskipun kasus *hash collisions* ini jarang terjadi jika *q* yang dipilih cukup besar. *Hash* yang tidak sama dipastikan *string* keduanya juga tidak mungkin sama. Jadi tidak perlucek ulang *string*.Jika disimpulkan ada tiga kasus yang berbeda untuk output algoritma Rabin-Karp (Chillar, 2008):

- Kasus 1: *Successful hit*. Jika nilai *hash* pola = nilai *hash substring* teks dan karakter dalam pola cocok dengan karakter dari *substring*.
- Kasus 2:*Spurious hit*. Jika nilai *hash* pola = nilai *hash substring* teks tapi karakter dalam pola tidak cocok dengan karakter *substring*.
- Kasus 3:Jika nilai *hash* pola tidak sama dengan *hash substring*, tidak perlu membandingkan karakter.

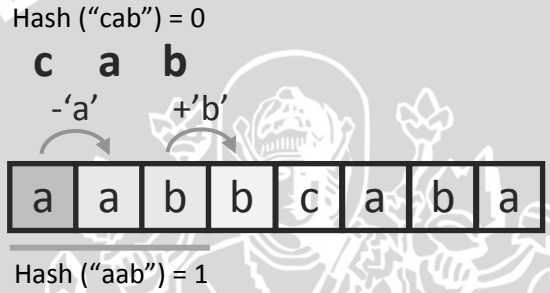
Berikut ini adalah contoh sederhana proses pencocokkan *string* dengan algoritma Rabin-Karp (Sparknote, 2013):

Misal terdapat pola “cab” yang akan dicari kecocokannya di dalam teks *string* “aabbcaba”. Agar memudahkan, nilai ASCII dari *string* diganti dengan angka 0 sampai 26 untuk mewakili nilai tiap karakter dari ‘a’ sampai ‘z’ (a = 1, b = 2, c = 3, dan seterusnya). Setelah itu hasil penambahannya dilakukan modulo dengan 3. Dari perhitungan *hashing* diketahui nilai *hash* pola“cab” = 0 ((3+2+1) mod 3 = 0), dan nilai *hash* dari *substring* pertama teks “aab” = 1. Proses lengkapnya bisa dilihat dalam penjelasan di bawah.



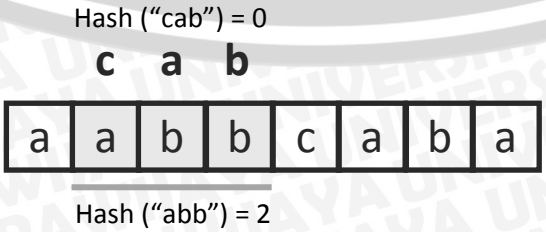
Gambar 2.3. Pembandingan pertama.

Gambar 2.3 menunjukkan pembandingan pertama. Setelah dicocokkan *hash* “cab” dan “aab” tidak sama, jadi *substring* berlanjut bergeser satu karakter ke kanan. *Substring* baru menjadi “abb”.



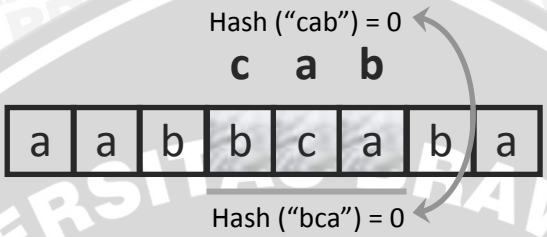
Gambar2.4. Pergeseran dan *updatesubstring*.

Di sinilah salah satu keunggulan algoritma Rabin-Karp. Menggunakan teknik *rolling hash*, nilai *hash* yang sebelumnya cukup dimodifikasi dengan mengurangi karakter pertama dari “aab” yaitu “a” dan menambahkan karakter baru yaitu “b”. Sehingga dari “aab” didapatkan “abb”=”aab”-‘a’+’b’. Begitu juga dengan nilai *hash*-nya menjadi (2=1-1+2) mod 3 = 2. Menggunakan cara ini tentu menghemat waktu komputasi karena waktu komputasi yang dibutuhkan konstan, *O*(1) di setiap pergeseran. Proses pergeseran dan *rolling hash* bisa dilihat pada Gambar 2.4 di atas.



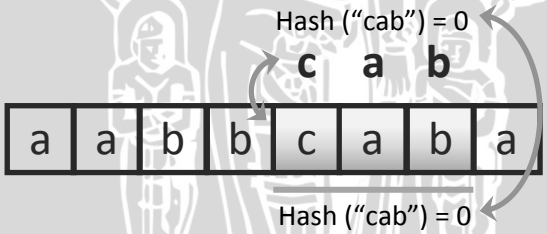
Gambar 2.5. Pembandingan kedua

Pada Gambar 2.5 setelah dilakukan pergeseran dan di-*update*, nilai *hash substring* "abb"=2 dibandingkan dengan nilai *hash* pola "cab"=0. Karena nilai *hash* tidak sama, lanjut bergeser ke arah kanan. Hal yang serupa terjadi pada perbandingan ketiga. Sampai pada pergeseran keempat ditemukan nilai *hash* "cab" sama dengan nilai *hash substring* "bca" yaitu 0.



Gambar 2.6. Pembandingan keempat. *Spurious hit.*

Karena pada pembandingan keempat (Gambar 2.6) ditemukan nilai *hash* sama, dilakukan pencocokan karakter antara *string* "bca" dengan "cab". Setelah diperiksa ternyata karakter-karakter kedua *string* tidak sama, jadi dilanjutkan lagi pergeseran substring ke kanan.



Gambar 2.7. Pembandingan kelima. *Successful hit.*

Pada Gambar 2.7 lagi-lagi ditemukan nilai *hash* sama. Dan setelah dilakukan pengecekan karakter *string* "cab" dan "cab" ditemukan kecocokan antar *string*. Masalah pun terpecahkan, proses berhenti. Waktu komputasi yang dibutuhkan adalah $O(m+n)$. $O(m)$ untuk *pre-processing* (proses *hashing*) dan $O(n)$ untuk mencari kesamaan *hash* sepanjang teks. Sangat jauh dari kasus terburuk yang mencapai $O(mn)$.

Secara garis besar proses dalam algoritma Rabin-Karp bisa digambarkan dalam pseudocode berikut (Firdaus, 2008):

```
function RabinKarp
(input p: string[1..m], sub: string[1..n])→boolean

//Deklarasi
    i: integer
    ketemu = boolean

//Algoritma
    ketemu ←false
    hp ← hash(p[1..m])                          //hashing pola
    for i ← 1 to n-m+1 do
        hsub ← hash(sub[1..i+m-1])              //hashing substring
        if hsub = hp then                       //perbandingan hash
            if sub[i..i+m-1] = p then           //perbandingan karakter
                ketemu ←true
        else
            hsub ← hash(sub[i+1..i+m])          //menggeser substring ke kanan
    endfor
    return ketemu
```

Gambar 2.8. Algoritma Rabin-Karp.

Seperti terlihat *spurious hit* bisa menjadi beban jika terjadi cukup sering. Hal ini seiring meningkatnya waktu komputasi karena harus mencocokkan teks dengan pola meskipun itu bukan *string* yang cocok. Kendati jumlah *hash* yang sama penyebab *spurious hit* bisa diminimalisasi dengan modulo yang besar, ada cara lain untuk menghindari *spurious hit* sehingga tidak harus melakukan pencocokan yang tidak perlu. Selain membandingkan sisa modulus *hash*, sebaiknya juga membandingkan hasil bagi setelah perhitungan aturan horner dengan *q* (Chillar, 2008).

Remain atau sisa ($h1/q$) = sisa ($h2/q$) (2-3)

Quotient atau hasil bagi ($h1/q$) = hasil bagi ($h2/q$) (2-4)

Persamaan (2-3) adalah hasil modulo nilai *hash* hasil aturan Horner dengan *q* (sisa). Dan persamaan (2-4) adalah hasil bagi nilai *hash* hasil aturan Horner dengan *q*. Disebutkan h1 adalah nilai *hash* dokumen asli, h2 adalah nilai *hash* dokumen yang ingin diuji, dan *q* adalah *operand denumerator modulus*. Dengan memenuhi kedua syarat perlu tersebut, bisa diketahui apakah *hash* tidak hanya sama setelah dimodulus dengan *q*, tapi juga hasil bagi aturan horner dengan *q*. Dengan begitu bisa dipastikan itu adalah *successful hit*. Jadi bila kedua-nya atau salah satu tidak sama tidak perlu dilakukan pengecekan ulang terhadap karakter-karakternya. Hal ini berarti tidak ada komputasi tambahan dari *spurious hit* jika hasilnya memenuhi syarat Persamaan (2-3) dan (2-4) di atas (Chillar, 2008).

2.4.4. Pencocokan *String* Berpola Banyak (*Multiple Pattern Search*)

Dalam hal pencarian *string* tunggal Algoritma Rabin-Karp memang masih kalah dengan algoritma-algoritma *string matching* lainnya. Algoritma Rabin-Karp kurang optimal karena skenario terburuk waktu komputasinya yang mencapai $O(mn)$. Sedangkan algoritma lain seperti algoritma Knuth-Morris-Pratt memiliki waktu komputasi $O(m+n)$, dan algoritma Boyer-Moore $O(m/n)$, bahkan $O(n)$. Meskipun begitu algoritma Rabin-Karp menjadi pilihan jika menghadapi kasus pencarian *string* dengan pola banyak.

Jika ada suatu kasus dimana ingin menemukan sejumlah pola dengan panjang tertentu di dalam sebuah teks, misalkan *k* pola, bisa digunakan varian algoritma Rabin-Karp. Algoritma pencarian *string* lain mencari sejumlah *k* pola dengan mengulangi pencarian pola tunggal selama $O(n)$, sehingga total memakan waktu $O(nk)$. Sebaliknya varian algoritma Rabin-Karp pada Gambar 2.8 mampu menemukan semua *k* pola dalam waktu $O(n+k)$, karena pembandingan dalam *hash* table antara *hash substring* dengan *hash* pola hanya memakan waktu $O(1)$. Itulah kenapa algoritma Rabin-Karp cukup optimal dan mangkus jika digunakan dalam pencocokan string berpola banyak. Dan keunggulan ini banyak diaplikasikan untuk memecahkan kasus plagiarisme. Bila digambarkan *pseudocode* algoritma Rabin-Karp untuk pencocokan *string* berpola banyak adalah seperti pada Gambar 2.8 berikut (Firdaus, 2008):

```
function RabinKarpSet
(input s: set of string[1..m], sub: string[1..n], m: integer)→integer

//Deklarasi
    i: integer
    str: string
    ketemu = integer

//Algoritma
    ketemu ←0
    set hs ← (set kosong)
    for each str in s do
        masukkan hash(s[1..m]) ke dalam hs
    for i ← 1 to n-m+1 do
        hsub ← hash(sub[i..i+m-1])
        if hsub ∈ hsthen
            if sub[i..i+m-1] = sebuah substring dengan hash hsubthen
                ketemu ←ketemu + 1
        else
            hsub ← hash(sub[i+1..i+m])          //menggeser substring ke kanan
    endfor
    return ketemu
```

Gambar 2.9. Algoritma Rabin-Karp untuk pencarian pola banyak.

2.4.5. Penilaian Kecocokan

Untuk mengukur tingkat kesamaan *string* digunakan pendekatan *k-gram*. *K-gram* adalah rentetan *substring* bersebelahan dengan panjang *k*. Sebuah dokumen teks dibagi menjadi *k-gram* dimana *k* adalah parameter yang ditentukan oleh user. Misal terdapat kalimat "A do run run run, a do run run". Untuk memperoleh rangkaian 5-gram dari kalimat teks tersebut melalui proses:

1. Dilakukan eliminasi tanda baca dan spasi. Huruf besar diganti huruf kecil, sehingga menjadi: adorunrunrunadorunrun
2. Memangkas teks menjadi rangkaian *substring* dengan banyak karakter 5 karakter tiap *substring*. Dengan pergeseran pemangkasan tiap satu karakter. Rangkaian 5-*grams* yang dihasilkan dari teks adalah:
   adoru dorun orunr runru unrun nrunr runru unrun nruna runad unado nador adoru dorun orunr runru unrun(Schleimer, 2003).

Ide dasar dari pendekatan *k-gram*s dibagi menjadi dua langkah sederhana. Pertama, membagi kata ke dalam *k-gram* (himpunan *substring-substring* yang saling berdekatan dengan panjang k). Kedua, mengelompokkan kata-kata yang sama, dalam hal ini adalah *string-string* yang memiliki struktur *k-gram* yang sama. Dari kata-kata yang sudah terkumpul digunakan koefisien kesamaan Dice (*Dice's similarity coefficient*) untuk tiap pasangan kata dalam rangka menilai kesamaan kata. Nilainya diperoleh dari persamaan:

$$S = \frac{2C}{A + B} \tag{2-5}$$

Dimana S adalah nilai kesamaan, A dan B jumlah *k-gram* unik masing-masing dalam *string A* dan *string B*, dan C adalah jumlah *k-gram* unik yang memiliki kesamaan dari kedua *string* yang dibandingkan.

Contoh, misalkan terdapat tiga kata: *photography*, *photographic*, dan *phonetic*. Jumlah *k* yang dipakai adalah 2 (bigram). Hasil perhitungan nilai kesamaannya adalah (Kosinov, 2002):

Tabel 2.2. Kesamaan antar dua *string* dari kata *photography*, *photographic*, dan *phonetic.*

String yang dibandingkan*	K-gram unik yang sama:	Nilai kesamaan:
Photography (9) dan *Photographic* (10)	Ph ho ot to og gr ra ap 8	2*8/(9+10) = 0,84
Photography (9) dan *Phonetic* (7)	Ph ho 2	2*2/(9+7) = 0,25
Photographic (10) dan *Phonetic* (7)	Ph ho ic 3	2*3/(10/7) = 0,35

*jumlah *k-gram* unik dari tiap *string* ditulis di dalam tanda kurung.

Dari nilai kesamaan bisa dikonversi ke dalam presentase kesamaan yang akan dijadikan indikator tingkat plagiarisme berdasarkan ambang batas. Presentase ini menjadi pertimbangan kadar plagiarisme (penjiplakan). Adapun ambang batas penjiplakan berdasarkan tingkat kesamaan adalah sebagai berikut:

- 0%. Tingkat kesamaan nihil. Berarti dokumen sama sekali berbeda dan bebas plagiarisme.
- < 15%. Tingkat kesamaan kecil. Sangat kecil kemungkinan plagiarisme.
- 15-50%. Tingkat kesamaan kecil sampai mendekati setengah dari frase atau kalimat dalam dokumen. Kemungkinan plagiarisme kecil sampai menengah.
- >50%. Tingkat kesamaan besar. Sudah bisa dikatakan plagiarisme meskipun dokumen tidak sama 100%.
- 100%. Tingkat kesamaan menyeluruh. Meski tanpa memperhatikan *stopword* kategori ini secara pasti sudah termasuk plagiarisme total.

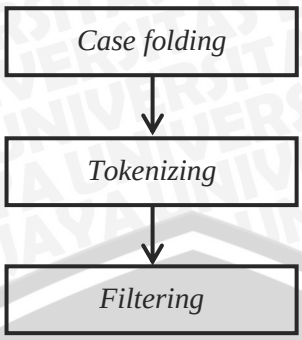
Dengan asumsi bahwa sebuah dokumen dikatakan plagiarisme (menjiplak) jika tingkat kesamaannya di atas 50%.

2.5. *TextPre-processing*

Pre-processing pada teks adalah proses menyiapkan data yang berupa dokumen teks (yang tidak terstruktur) ke dalam bentuk data yang terstruktur lebih baik. Karena struktur data yang baik lebih memudahkan bila akan dilakukan proses komputerisasi secara otomatis. Oleh karena itu dari dokumen teks yang merupakan data tidak terstruktur perlu dilakukan proses *filtering* dan pengubahan struktur data. Salah satu operasi dalam *pre-processing* teks adalah pembersihan teks, seperti pada proses ekstraksi dokumen.

2.5.1. Ekstraksi dokumen

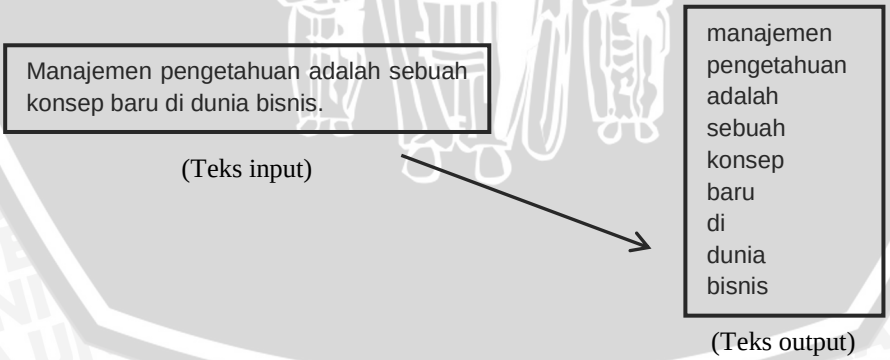
Dokumen teks yang akan dibandingkan umumnya berdimensi besar, memiliki *noise* di dalamnya dan tidak terstruktur. Agar data-data yang dipakai sesuai dan mangkus perlu ditentukan terlebih dulu fitur-fitur dari dokumen teks. Fitur-fitur ini mewakili kata-kata untuk setiap fitur dalam teks keseluruhan. Dan untuk mendapatkan fitur-fitur yang mewakili dilakukan *pre-processing* teks seperti *case folding, tokenizing, filtering* dan *stemming*. Alur proses ekstraksi dokumen bisa dilihat pada Gambar2.9 di bawah (Triawati, 2009).



Gambar 2.10. *Pre-processing*.
(Sumber: Triawati, 2009)

2.5.1.1.*Case Folding*dan *Tokenizing*

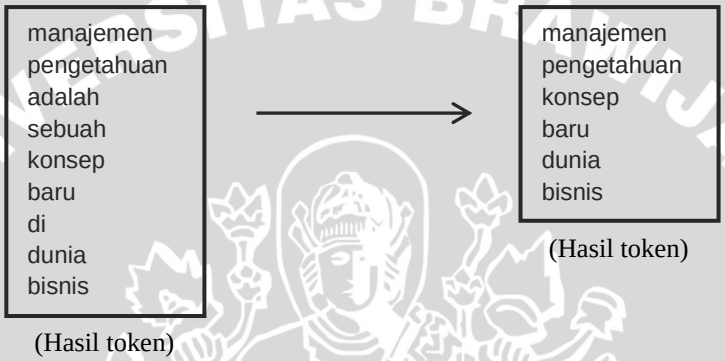
Case folding adalah proses menyamakan *case* huruf menjadi sama rata. Umumnya *case folding* mengubah menjadi huruf kecil. Selain itu biasanya pada proses ini juga menghilangkan semua tanda baca. Hanya karakter huruf (‘a’ sampai ‘z’) dan angka (‘0’ sampai ‘9’) saja yang diterima, karakter lain dianggap *delimiter* (dihilangkan). *Tokenizing* adalah proses memecah teks menjadi token kata-kata tunggal. Dalam hal ini dokumen teks dipotong menjadi *string* terpisah berdasarkan tiap kata penyusun-nya. Berikut adalah contoh hasil *case folding* dan *tokenizing* sebuah teks(Triawati, 2009):



Gambar 2.11. *Case folding* dan *tokenizing*.
(Sumber: Triawati, 2009)

2.5.1.2.*Filtering*

Filtering (*stopword removal*) adalah proses meng-eliminasi kata-kata tidak penting dalam teks. Dengan membuang kata-kata tidak penting dan mengambil kata-kata pentingnya saja akan memperkecil memori dan waktu komputasi. *Stopword* adalah daftar kata tak bermakna yang sebagian besar kata sambung seperti: "yang", "dan", "atau", "meskipun", dan lain-lain. Biasanya sudah ada file teks yang memuat *stopword* dan tersedia dalam bahasa Indonesia, sehingga tidak perlu mencatat sendiri dari kamus. Contoh filtering (Triawati, 2009):



Gambar 2.12. *Filtering*.
(Sumber: Triawati, 2009)

2.6.*Synonym Recognition*

Synonym recognition atau pengenalan sinonim adalah salah satu pendekatan yang digunakan untuk membantu proses pendeteksian plagiarisme. *Synonym recognition* merupakan pendekatan kata lain terhadap dokumen teks. Pendekatan ini memanfaatkan kesamaan makna dalam kata yang kemungkinan banyak terjadi. Dengan mendeteksi kata-kata yang sama (*synonym*) antara dokumen teks satu dengan yang lain pendekatan ini menambah tingkat keakuratan proses deteksi plagiarisme. Pada proses ini *string* hasil pembandingan Rabin-Karp yang tidak sama akan dicocokkan dengan kamus *synonym*. Jika kata yang dicocokkan memiliki *synonym* (kata yang sama), kata lain tersebut akan dimasukkan ke dalam proses Rabin-Karp dan dicocokkan kembali. Dan apabila *string* teks tidak memiliki *synonym*, proses *synonym recognition* tidak akan dilakukan. Langkah ini akan terus berulang sampai seluruh proses pencocokan *string* selesai.