BAB II

LANDASAN TEORI

2.1 Plagiarisme

Plagiarisme merupakan praktik penyalahgunaan hak kekayaan intelektual milik orang lain dan karya tersebut diakui secara tidak sah sebagai hasil karya pribadi (Sulianta, 2007). Neville (2010), *plagiarisme* merupakan tindakan atau praktek yang dianggap oleh universitas merupakan suatu kecurangan dengan cara mengambil ide atau tulisan orang lain tanpa menyebutkan rujukan dan diklaim sebagai miliknya.

Menurut Sastroasmoro (2007) klasifikasi *plagiarisme* berdasarkan proporsi atau persentasi kata, kalimat, *paragraph* yang dibajak, terbagi menjadi tiga yaitu *plagiarisme* ringan: <30%, *plagiarisme* sedang: 30% - 70%, dan *plagiarisme* berat atau total: >70%. Ada banyak faktor penyebab terjadinya tindakan *plagiarisme*. Studi empiris oleh Hutton and French (2006) dalam Hartanto (2012) mengemukakan faktor penyebab tindakan *plagiarisme* yaitu adanya kemalasan pada diri sendiri, karena merasa stress, memiliki keyakinan bahwa perilaku tidak akan diketahui dan perilaku tersebut bukan merupakan hal yang salah dan merugikan.

Adapun tipe-tipe *plagiarisme* menurut Iyer dan Sing (2005), yaitu:

1. *Word-forword plagiarisme* adalah menyalin setiap kata secara langsung tanpa diubah sedikitpun.
2. *Plagiarisme of authorship*, adalah mengakui hasil karya orang lain sebagai hasil karya sendiri.
3. *Plagiarisme of ideas*, adalah mengakui hasil pemikiran atau ide orang lain.

4. *Plagiarisme of sources*, jika seseorang penulis mengggunakan kutipan dari penulis lain tanpa mencantumkan sumbernya.

2.2 Algoritma

Algoritma berasal dari nama seorang ahli Matematika bangsa Arab yaitu Abu Ja'far Muhammad ibnu Musa al-Khuwarizmi. Al-Khuwarizmi dibaca oleh orang Barat menjadi *Algorism* (Knuth, 1973). Perubahan kata *algorism* menjadi *algorithm* karena kata *algorism* sering dikelirukan dengan *arithmetic*, sehingga akhiran –sm berubah menjadi –thm. Lambat laun kata *algorithm* dipakai sebagai metode perhitungan (komputasi) secara umum, sehingga kehilangan makna aslinya (Parsons, 1995). Dalam Bahasa Indonesia kata *algorithm* diserap menjadi algoritma. Definisi algoritma adalah susunan langkah-langkah sistematis dan logis dalam pemecahan suatu masalah (Saniman dan Fathoni, 2008).

Menurut Sjukani (2005), algoritma adalah alur pemikiran dalam menyelesaikan suatu pekerjaan yang dituangkan secara tertulis. Yang ditekankan pertama adalah alur pikiran, sehingga algoritma seseorang dapat berbeda dari algoritma orang lain. Sedangkan penekanan kedua adalah tertulis, yang artinya dapat berupa kalimat, gambar, atau tabel tertentu. Jadi dapat disimpulkan bahwa algoritma lebih merupakan alur pemikiran untuk menyelesaikan suatu pekerjaan atau suatu masalah daripada pembuatan program komputer. Dengan adanya algoritma maka suatu permasalahan dapat diselesaikan berdasarkan urutan langkah yang logis. Urutan langkah logis, berarti algoritma harus mengikuti suatu urutan tertentu, tidak boleh melompat-lompat dan disusun secara sistematis. Sedangkan yang dimaksud dengan langkah-langkah

logis adalah suatu tahapan proses yang dapat mengetahui dengan pasti berdasarkan setiap langkah yang telah dibuat (Microsoft Press Computer and Internet Dictionary, 1998).

Menurut Knuth (1973) sebuah langkah dapat dikategorikan algoritma yang baik jika memiliki syarat-syarat berikut:

1. *Finiteness*: Algoritma harus berakhir setelah melakukan sejumlah langkah proses
2. *Definiteness*: Setiap langkah algoritma harus didefinisikan dengan tepat dan tidak menimbulkan makna ganda
3. *Input*: Sebuah algoritma memiliki nol atau lebih masukan (input) yang diberikan kepada algoritma sebelum dijalankan
4. *Output*: Setiap algoritma memberikan satu atau beberapa hasil keluaran
5. *Effectiveness*: Langkah-langkah algoritma dikerjakan dalam waktu yang "wajar"

2.2.1 Kompleksitas Algoritma

Suatu masalah dapat mempunyai banyak algoritma penyelesaian. Algoritma yang digunakan tidak saja harus benar, namun juga harus efisien. Efisiensi suatu algoritma dapat diukur dari waktu eksekusi algoritma dan kebutuhan ruang memori. Besaran yang digunakan untuk menjelaskan model pengukuran waktu dan ruang ini adalah kompleksitas algoritma (Azizah, 2013). Menurut Goldreich (2008) kompleksitas algoritma dapat diukur berdasarkan kinerjanya dengan menghitung waktu eksekusi suatu algoritma. Waktu eksekusi tersebut dapat diklasifikasikan menjadi tiga kelompok besar, yaitu *best-case* (kasus terbaik), *average-case* (kasus rerata) dan *worst-case* (kasus terjelek).

Kompleksitas algoritma terdiri dari dua jenis yakni kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu disimbolkan dengan T(n) dan kompleksitas ruang S(n). Kompleksitas waktu, T(n), diukur dari jumlah tahapan komputasi yang dibutuhkan untuk menjalankan algoritma sebagai fungsi dari ukuran masukan n. Sedangkan komplesitas ruang, S(n), diukur dari memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari ukuran masukan n. Dengan menggunakan besaran kompleksitas waktu atau ruang algoritma, dapat menentukan laju peningkatan waktu, dalam hal ini ruang, yang diperlukan algoritma dengan meningkatnya ukuran masukan n (Saputro, 2010).

Dalam struktur data terkadang tidak diperlukan kompleksitas waktu yang detail dari sebuah algoritma, namun yang diperlukan adalah besaran kompleksitas waktu yang menghampiri kompleksitas waktu yang sebenarnya. Kompleksitas waktu yang demikian disebut kompleksitas waktu asimptotik yang dinotasikan dengan "O" (Big O) (Cormane, dkk., 2001).

Tabel 2.1 Pengelompokan algoritma berdasarkan notasi Big-O (Cormen, dkk., 2001)

No	Kelompok	Nama
1.	O(1)	Konstan
2.	O (log n)	Logaritmik
3.	O (n)	Lanjar
4.	O (n log n)	n log n
5.	O (n^2)	Kuadratik
6.	O (n^3)	Kubik
7.	O (2^n)	Eksponensial
8.	O (*n!*)	Faktorial

2.2.2 Performa Algoritma

Menurut Russell dan Norvig (1995), performa dari suatu algoritma dapat dilihat dari empat kriteria berikut ini.

1. *Completeness* adalah apakah suatu algoritma tersebut menjamin ditemukannya solusi jika solusi tersebut ada?
2. *Time Complexity* adalah berapa lama waktu yang dibutuhkan untuk menemukan solusi tersebut?
3. *Space Complexity* adalah berapa banyak memori yang dibutuhkan untuk menemukan solusi?
4. *Optimality* adalah apakah algoritma tersebut menjamin menemukan solusi yang terbaik jika terdapat beberapa solusi yang lain?

2.3 String Matching

String matching atau pencocokan *string* adalah suatu metode yang digunakan untuk menemukan suatu keakuratan atau hasil dari satu atau beberapa pola teks yang diberikan. *String matching* merupakan pokok bahasan yang penting dalam ilmu komputer karena teks merupakan bentuk utama dari pertukaran informasi antar manusia, misalnya pada literatur, karya ilmiah, halaman *web* (Hulberg dan Helger, 2007).

String matching atau pencocokan *string* merupakan algoritma untuk melakukan pencarian semua kemunculan *string* dengan mencari kesamaannya. Prinsip dari *string matching* adalah mencari semua kemunculan *string* pendek yang disebut pola atau *pattern* P[0……n-1] di dalam string yang lebih panjang yang disebut *teks* T[0…m-1],

dengan m dan n adalah panjang *string*. Kedua *string* dibentuk dari kumpulan karakter terbatas yang disebut alfabet, dinotasikan Σ dengan ukuran σ (Breslauer, 1992).

Algoritma *string matching* dapat diklasifikasikan menjadi tiga bagian menurut arah pencariannya, yakni (Charras, 1997:12).

1. *From left to right*

   Dari arah yang paling alami, dari kiri ke kanan, yang merupakan arah untuk membaca. Algoritma yang termasuk kategori ini adalah algoritma *Brute Force,* algoritma *Knuth Moris Pratt.*

2. *From right to left*

   Dari arah kanan ke kiri, arah yang bisanya menghasilkan hasil terbaik secara *praktikal*. Algoritma yang termasuk kategori ini adalah algoritma *Boyer-Moore.*

3. *In a specific order*

   Dari arah yang ditentukan secara spesifik oleh algoritma tersebut, arah ini menghasilkan hasil terbaik secara teoritis. Algoritma yang termasuk kategori ini adalah algoritma *Colossi* dan algoritma *Crochemore-Perrin.*

Ada bermacam-macam tipe pencarian *string*, antara lain:

1. *Brute Force Algorithm*
2. *Colussi Algorithm*
3. *Boyer-Moore Algorithm*
4. *Rabin-Karp Algorithm*
5. *Crochemore-Perrin Algorithm*
6. *Jaro-Winkler Distance Algorithm*

2.4 Algoritma Rabin-Karp

Algoritma Rabin-Karp diciptakan oleh Michael O. Rabin dan Richard M. Karp pada tahun 1987 yang menggunakan fungsi *hashing* untuk menemukan *pattern* di dalam *string* teks. Menurut Abdeen dan Rawan (2011), pada prinsipnya Rabin-Karp algorithm menghitung sebuah fungsi *hash* untuk mencari pola di dalam sebuah teks yang diberikan.

Setiap karakter M *subsequence* dari teks akan dikomparasi. Jika nilai *hash* tidak sama, algoritma akan menghitung nilai *hash* untuk karakter M *subsequence* berikutnya, dan jika nilai *hash* sama maka algoritma akan melakukan perbandingan secara *brute-force* antara pola dan karakter M *subsequence*. Dengan cara ini hanya akan ada satu perbandingan per teks *subsequence* dan *brute-force* hanya dibutuhkan jika nilai *hash* cocok atau sama (Jain, dkk., 2012).

Secara umum karakteristik algoritma Rabin-Karp sebagai berikut (Fernando, 2009):

1. Menggunakan sebuah fungsi *hashing*.
2. Fase preprocessing menggunakan kompleksitas waktu O(m).
3. Untuk fase pencarian kompleksitasnya O(mn).
4. Waktu yang diperlukan O(n+m).

Fungsi *hash* yang diterapkan pada algoritma ini setidaknya harus menyediakan empat properti yaitu (Novian, dkk., 2012):

1. Mampu melakukan komputasi secara efisien.
2. Diskriminasi *string* yang tinggi.
3. Fungsi *hash* (y[j+1 ... j+m]) harus mudah mengkomputasi dari :

hash (y[j...j+m-1])

hash (y[j+m])

2.4.1 Konsep Algoritma Rabin-Karp

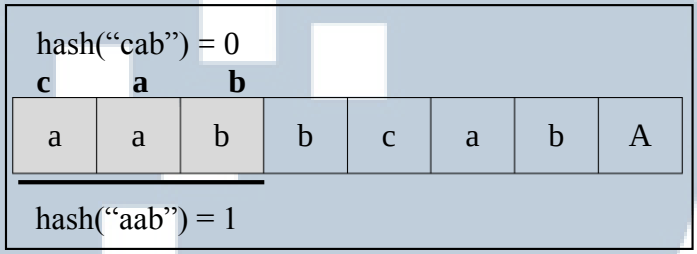
Algoritma Rabin-Karp adalah algoritma pencocokan *string* yang menggunakan *fungsi hash* untuk membandingkan antara *string* yang dicari (m) dengan *substring* pada teks (n). Apabila *hash value* keduanya sama, maka akan dilakukan perbandingan sekali lagi terhadap karakter-karakternya. Apabila hasil keduanya tidak sama, maka *substring* akan bergeser ke kanan. Pergeseran dilakukan sebanyak (n-m) kali. Perhitungan nilai *hash* yang efisien pada saat pergeseran akan mempengaruhi performa dari algoritma ini (Firdaus, 2008).

```
function RabinKarp (input s:
string[1..m], teks: string[1..n])
boolean
{ Melakukan pencarian string s pada
 string teks dengan algoritma Rabin-K}
Deklarasi
i : integer
ketemu = boolean

Algoritma:
ketemu ←false
hs ←hash(s[1..m])
for i ← 0 to n-m do
      hsub ←hash(teks[1..i+m-1])
      if hsub = hs then
         if teks[i..i+m-1] = s then
         ketemu ←true
      else
      hsub ←hash(teks[i+1..i+m])
endfor
return ketemu
```

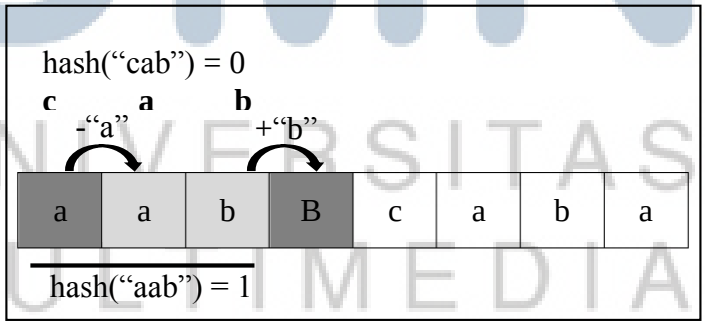
Gambar 2.1 Algoritma Rabin-Karp (Firdaus, 2008)

Berikut adalah ilustrasi cara kerja algoritma Rabin-Karp (Nugroho, 2011): Diberikan masukan “cab” dan teks “aabbcaba”. Fungsi *hash* yang dipakai misalnya akan menambahkan nilai keterurutan setiap huruf dalam *alphabet* (a = 1, b =2, dst.) dan melakukan modulus dengan 3. Didapatkan nilai *hash* dari “cab” adalah 0 dan tiga karakter pertama pada teks yaitu “aab” adalah 1.

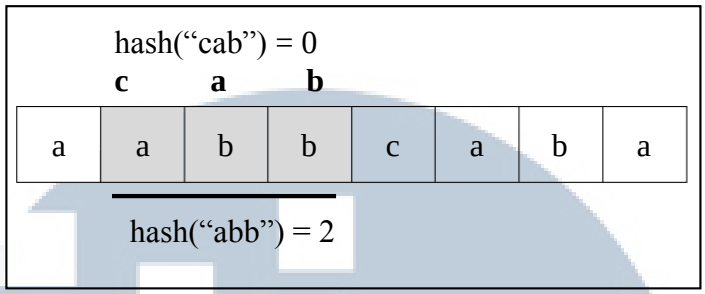


Gambar 2.2 Pengecekan tiga karakter pertama (Nugroho, 2011)

Hasil perbandingan ternyata tidak sama, maka *substring* pada teks akan bergeser satu karakter ke kanan. Algoritma tidak menghitung kembali nilai *hash substring*. Disinilah dilakukan apa yang disebut *rolling hash* yaitu mengurangi nilai karakter yang keluar dan menambahkan nilai karakter yang masuk sehingga didapatkan kompleksitas waktu yang relatif konstan pada setiap kali pergeseran. Setelah pergeseran, didapatkan *nilai hash* dari *fingerprint* “abb”(abb = abb - a + b) menjadi dua (2 = 1 - 1 + 2).

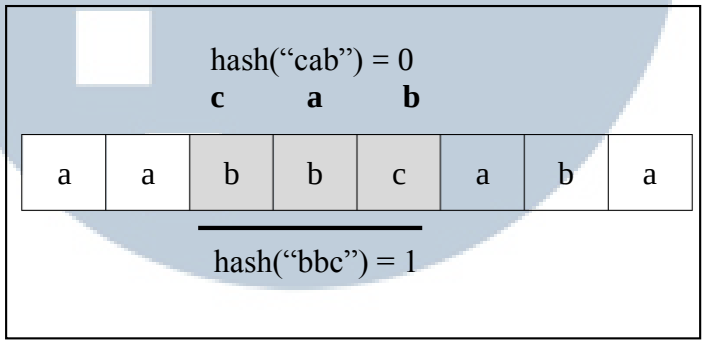


Gambar 2.3 Pengecekan terhadap *substring* berikutnya (Nugroho, 2011)



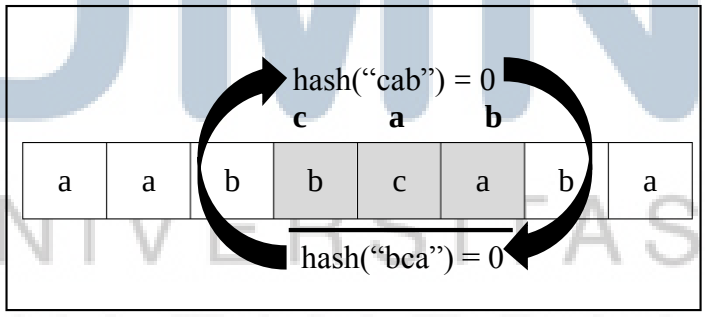
Gambar 2.4 Pengecekan pattern “c a b” dengan substring “a b b” (Nugroho, 2011)

Hasil perbandingan juga tidak sama, maka dilakukan pergeseran. Begitu pula dengan perbandingan ketiga. Pada perbandingan keempat, didapatkan nilai *hash* yang sama.



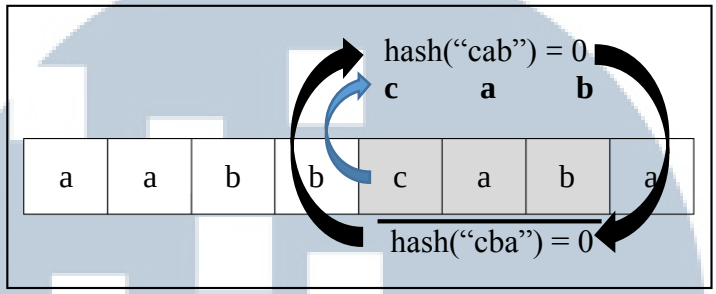
Gambar 2.5 Pembandingan pattern dengan substring (Nugroho, 2011)

Karena nilai *hash* sama, maka dilakukan perbandingan *string* karakter per karakter antara “bca” dan “cab”. Didapatkan hasil kedua *string* tidak sama.



Gambar 2.6 Pembandingan pattern yang mempunyai nilai hash sama dengan substring (Nugroho, 2011)

Maka, kembali *substring* bergeser ke kanan. Pada perbandingan yang kelima, kedua nilai *hash* dan karakter pembentuk *string* sesuai, sehingga solusi ditemukan.



Gambar 2.7 Hasil pencarian *pattern* ditemukan (Nugroho, 2011)

2.5 Algoritma Jaro Winkler Distance

Algoritma Jaro Winkler Distance adalah algoritma untuk mengukur kesamaan antara dua *string* dan sebagian besar algoritma ini digunakan dalam bidang pendeteksian duplikasi (Kornain, dkk., 2014). Algoritma ini bermula dari algoritma Jaro Distance yang ditemukan oleh Matthew A. Jaro yang kemudiaan dikembangkan oleh William E. Winkler dan Thibaudeau dengan memodifikasi Jaro Distance untuk memberikan bobot yang lebih tinggi untuk *prefix* kemiripan. Semakin tinggi Jaro Winkler Distance untuk dua *string* maka semakin mirip dengan string tersebut. Nilai normalnya ialah 0 menandakan tidak ada kesamaan dan 1 yang menandakan adanya kesamaan (Kurniawati, dkk., 2010).

Dasar dari algoritma ini memiliki tiga bagian yaitu (Kurniawati, dkk., 2010):

1. Menghitung panjang *string*,
2. Menemukan jumlah karakter yang sama di dalam dua *string*, dan
3. Menemukan jumlah transposisi.

Secara umum karakteristik Jaro Winkler Distance Algorithm sebagai berikut (Wirawan, 2004):

1. Fase preprocessing menggunakan kompleksitas waktu O(m)
2. Fase pencarian kuadratik O (n^2)
3. Kompleksitas waktu yang dibutuhkan adalah O ($m+n^2$)

2.5.1 Konsep Algoritma Jaro Winkler Distance

Pada algoritma Jaro Winkler Distance digunakan rumus untuk menghitung jarak (d_j) antara dua *string* yaitu S_1 dan S_2 adalah

$$d_j = \frac{1}{3} \times \left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) \quad \ldots (2.1)$$

dimana:

m = jumlah karakter yang sama persis

$|s_1|$ = Panjang *String* 1

$|s_2|$ = Panjang *String* 2

t = Jumlah Transposisi

Pada algoritma Jaro-Winkler (d_w) menggunakan skala *prefix* (p) yang memberikan awalan pada *set string*. Dengan rumus sebagai berikut:

$$d_w = d_j + \left(lp(1 - d_j)\right) \quad \ldots (2.2)$$

dimana:

d_j adalah hasil perhitungan kemiripan *string* S_1dan S_2.

l adalah panjang karakter atau *prefik* yang sama pada awalan *string* sebelum ditemukan adanya ketidaksamaan dengan batas maksimum sampai empat karakter.

p adalah *konstanta scaling factor*. Nilai standar untuk konstanta menurut Winkler adalah p = 0,1.

Nilai perhitungan yang didapat dari 0 hingga 1. Dengan nilai 0 setara dengan tidak ada kemiripan dan 1 adalah sama persis.

Berikut ini adalah contoh perhitungan dengan Jaro Winkler Distance. Jika string S_1 "MARTHA" dan S_2 "MARHTA" maka (Kurniawati, dkk., 2010):

m = 6

S_1 = 6

S_2 = 6

t = 1, hal ini dikarenakan ada karakter yang sama tapi tertukar urutannya. Karakter tersebut yaitu T dan H.

Maka nilai *Jaro distancenya* adalah :

$$d_j = {}^1/_3 \text{ x } ({}^6/_6 + {}^6/_6 + {}^{6\text{-}1}/_6) = 0.944$$

Kemudian bila diperhatikan sususan S_1 dan S_2 dapat diketahui nilai l = 3, dengan nilai konstan p = 0.1. Maka nilai Jaro Winkler Distance adalah

$$d_w = 0.944 + (3 \text{ x } 0.1 \text{ (1-0.944)}) = 0.961$$

2.6 Analisis Perbandingan Rabin-Karp Algorithm dan Jaro Winkler Distance Algorithm

Terdapat berbagai macam algoritma untuk melakukan proses *string matching pattern*. Masing-masing algoritma mempunyai kelebihan dan kekurangannya dalam proses *string matching pattern*. Menurut Wicaksono, dkk. (2012) untuk membuat suatu sistem pendeteksi plagiat diperlukan algoritma yang baik untuk jenis *multiple string matching pattern*. Salah satu algoritma yang cocok untuk permasalahan *multiple string matching pattern* adalah *Rabin-Karp algorithm*.

Algoritma Rabin-Karp menggunakan *hashing* untuk menemukan sebuah *substring* dalam sebuah teks. Algoritma ini tidak bertujuan menemukan *string* yang cocok dengan *string* masukan, melainkan menemukan pola (*pattern*) yang sekiranya sesuai dengan teks masukan. Algoritma ini menghasilkan efisiensi waktu yang baik dalam mendeteksi *string* yang memiliki lebih dari satu pola (Firdaus, 2008). Hal ini juga didukung oleh Andreas, dkk. (2006) yang menyatakan bahwa selain menggunakan fungsi *hashing*, algoritma ini memiliki fase *preprocessing* dengan kompleksitas waktu O(m), sehingga baik dalam menghemat waktu pencarian *hash key* ataupun dalam penggunaan memori.

Menurut Fernando (2009), algoritma Rabin-Karp ini sering digunakan dalam pendeteksian pencontekan atau kecurangan. Bila diberikan *source material* atau dokumen, algoritma ini dapat dengan cepat mencari seluruh kesamaan dari setiap kalimat, dengan mengabaikan *lowercase* atau *uppercase*, tanda titik, tanda seru,tanda tanya serta tanda baca lainnya. Kelebihan dari algoritma Rabin-Karp dibandingkan algoritma pencocokan *string* lainnya adalah kemampuan dalam mencari banyak pola *string* (Cormen, dkk., 2003).

Seperti umumnya sebuah algoritma, algoritma Rabin-Karp juga memiliki kekurangan. Menurut Atmopawiro (2006), kekurangan dari algoritma ini adalah membutuhkan waktu yang lama dalam membandingkan kata yang ditunjukkan dengan fase pencarian yang memiliki kompleksitas O(mn). Hal ini praktis sangat lambat dan mengambil ruang ekstra (Sunita, dkk., 2014). Masalah tersebut ditimbulkan oleh penggunaan metode *hashing*, dimana terdapat banyak sekali kata dalam kalimat yang berbeda. Untuk menjaga agar nilai *hash* kecil harus diberikan beberapa kata dengan

nilai *hash* yang sama. Yang berarti jika suatu nilai *hash* sama maka belum tentu sebuah kata itu juga sama dan sebaliknya, sehingga diperlukan pengecekan kembali yang membutuhkan waktu yang lebih lama untuk panjang berapapun (Atmopawiro, 2006).

Algoritma Jaro Winkler Distance adalah algoritma yang menggunakan pendekatan *string metric*, yaitu melakukan perbandingan *string* dengan memasukkannya ke dalam fungsi matematis tertentu. Beberapa algoritma yang berdasarkan kepada *string metric* diantaranya adalah *Levenshtein distance*, *TF/IDF*, *Needleman-Wunsch distance*, *Jaro-Winkler distance*, dan sebagainya. Dari algoritma yang telah disebutkan Jaro Winkler distance memiliki ketepatan yang baik di dalam pencocokan *string* yang relatif pendek. Metode ini dipilih dikarenakan setelah dilakukannya proses *tokenizing*, algoritma ini dapat secara akurat memeriksa salinan antar dokumen (Kurniawati, dkk., 2010).

Algoritma Jaro Winkler Distance memiliki kelebihan dari segi waktu. Menurut Kurniawati, dkk. (2010), algoritma ini memiliki *quadratic runtime complexity* yang sangat efektif pada *string* pendek dan dapat bekerja lebih cepat. Analisa menggunakan *string metrik* telah banyak digunakan untuk deteksi kecurangan, *fingerprint*, deteksi plagiat, dan sebagainya. Dalam penelitian yang dilakukan oleh Faranika, dkk. (2013) didapat bahwa kompleksitas waktu dari algoritma ini adalah $O(n^2)$ atau kuadratik sehingga algoritma ini lebih baik daripad algoritma sejenisnya dan sangat baik untuk diterapkan dalam pendeteksian plagiat.

Berdasarkan kesimpulan dari penelitian yang dilakukan oleh Kurniawati, dkk. (2010) yang menyatakan bahwa dalam ujicobanya aplikasi dapat berjalan dengan baik untuk memeriksa kemiripan dokumen yang identik atau sama seratus persen. Hal ini dikarenakan urutan kata-kata yang dibandingkan sangat sesuai. Akan tetapi, saat

memeriksa kemiripan dokumen dengan urutan yang berbeda, aplikasi ini tidak mampu mendeteksi kemiripannya. Hal tersebut juga didukung oleh latar belakang dari penelitian Santoso, dkk. (2014) yang menyatakan bahwa algoritma Jaro Winkler Distance masih memperhatikan urutan kata, sehingga dokumen yang memiliki urutan kata berbeda walaupun isi tekstualnya sama akan sulit untuk dideteksi kemiripannya.