JUPITER
Journal of Computer & Information Technology
Vol. 2 No. 1 Feb 2021
Institut Bisnis Muhammadiyah Bekasi

PENDETEKSIAN *SOURCE CODE PLAGIARISM* DENGAN ALGORITMA *RABIN-KARP* PADA ONLINE JUDGE

YUNUS FADHILLAH
yunus@ibm.ac.id

Program Studi Teknik Informatika & Komputer Institut Bisnis Muhammadiyah Bekasi

ABSTRAK

Sulitnya melakukan pengujian kebenaran *source code* dan pendeteksian *source code plagiarism* merupakan masalah – masalah yang melatarbelakangi pada penelitian ini. Algoritma pencocokan *string* merupakan algoritma utama yang diperlukan dalam pendeteksian *source code plagiarism*. Saat ini terdapat banyak algoritma pencocokan *string* yang ada, maka dari itu perlu dilakukan perbandingan dan pemilihan algoritma pencocokan *string* yang sesuai dengan kebutuhan yang ada Sedangkan untuk sistemnya, peneliti melakukan studi pustaka dan observasi langsung untuk mencari sistem pengujian source code yang sesuai. Algoritma *Rabin-Karp* dipilih berdasarkan rata – rata kompleksitas waktu dan ruang yang baik. Dan juga dapat dilakukan modifikasi, sehingga algoritma *Rabin-Karp* ini dapat dibuat sedemikian rupa sehingga dapat memenuhi segala kebutuhan yang ada. Sedangkan untuk sistem pengujian *source code* peneliti memilih sistem *online judge* karena sistem tersebut dalam penggunaannya bersifat objektif.

Kata Kunci: Algoritma Pencocokan *String*, Algoritma *Rabin-Karp*

ABSTRACT

The difficulty of testing the truth of the source code and the detection of plagiarism source code are the underlying problems of this research. The string matching algorithm is the main algorithm required in the detection of source code plagiarism. Currently there are many existing string matching algorithms, therefore it is necessary to compare and select the appropriate string matching algorithm to the existing requirement. As for the system, the researcher conducts literature study and direct observation to find the appropriate source code testing system. The Rabin-Karp algorithm is chosen based on the average of good time and space complexity. And also can be modified, so that the Rabin-Karp algorithm can be made in such a way that can meet all the needs that exist. As for the system testing source code researchers choose the online judge system because the system in the test is objective.

Keywords: *String Matching Algorithm, Rabin-Karp Algorithm, Online Judge System.*

PENDAHULUAN

Teknik Informatika merupakan salah satu program studi yang populer saat ini. Hampir di setiap universitas maupun institusi di Indonesia membuka program studi tersebut. Mulai dari *programmer*, *system analyst*, *project manager*, *database administrator*, dan lain sebagainya, merupakan pilihan karir yang dapat diambil oleh lulusan dari program studi teknik informatika. Khususnya *programmer* sangat dibutuhkan saat ini (Sulistia Endah: 2017), dilihat dari kebutuhan masyarakat akan layanan yang dapat memudahkan mereka di berbagai bidang.

Di dalam perkuliahannya, program studi teknik informatika memiliki berbagai materi perkuliahan yang diberikan kepada mahasiswanya. Dari semua materi perkuliahan itu, hampir sebagian besar mata kuliahnya bersifat praktikum (belajar di lab. komputer). Itu bisa dilihat dari selalu adanya kelas lab. pada setiap semesternya. Seperti mata kuliah Algoritma dan Pemrograman contohnya. Mata kuliah ini diajarkan di lab. komputer dan bersifat praktik. Algoritma dan Pemrograman adalah mata kuliah dasar yang harus dikuasai oleh mahasiswa jurusan teknik informatika.

Dalam kesehariannya, proses perkuliahan pasti tidak lepas dengan yang namanya tugas harian. Dalam mata kuliah Algoritma dan Pemrograman ini, tugas harian biasa diberikan dalam bentuk soal algoritma. Mahasiswa diminta untuk memecahkan masalah yang ada di soal tersebut serta mengaplikasikannya ke dalam sebuah program, dan pada akhirnya yang dikumpulkan adalah berupa *source code* dari program tersebut. Setelah mahasiswa mengumpulkan *source code*, maka selanjutnya pengajar akan memeriksa dan menilai tugas harian tersebut.

Pada proses pemeriksaan dan penilaian tugas harian saat ini, pengajar masih menggunakan cara manual dalam memeriksa dan memberi nilai untuk tugas harian yang telah dikerjakan oleh mahasiswanya. Pengajar akan membuka satu per satu *source code* yang telah dikumpulkan mahasiswa, setelah itu *source code* tersebut akan di-*compile*. Setelah di-*compile*, hasil programnya akan dijalankan dan akan dimasukkan dengan *input-input* yang diminta. Setelah itu hasil output dari program tersebut akan dicocokan dengan *output* yang diharapkan. Selain menguji kebenarannya, pengajar akan membuka kembali *source code* untuk dilihat kriteria penilaian lain antara lain adalah penamaan variabel yang digunakan, seberapa efisien dan juga kerapian dari *source code* mahasiswa tersebut. Dan itu akan dilakukan berulang kali sebanyak jumlah *source code* yang dikumpulkan oleh mahasiswa. Hal ini tentu akan menjadi masalah ketika jumlah mahasiswa dalam suatu kelas yang mengambil mata kuliah Algoritma dan Pemrograman ini memiliki kuantitas yang sangat banyak. Belum lagi ditambah jika pengajar tersebut mengajar lebih dari satu kelas Algoritma dan Pemrograman. Tentu proses penilaian tersebut akan memakan waktu yang sangat lama.

Kendala lainnya adalah masih banyak ditemukan tindakan plagiat yang dilakukan oleh mahasiswa saat mengerjakan tugas hariannya. Dalam kasus ini yang menjadi bahan dari tindakan plagiat adalah *source code* dari hasil tugas harian mahasiswa yang dikumpulkan. Tindakan plagiat ini dapat dilihat mulai dari melakukan *copy-paste source code* keseluruhan atau sebagian; penambahan komentar, spasi, *enter*, atau *statement* yang tidak penting; mengganti nama variabel; atau memindahkan posisi dari *method-method* yang ada agar terlihat beda.

Dengan adanya tindakan plagiat sebagai kriteria penilaian diharapkan dapat memacu mahasiswa untuk mau belajar dan berusaha sendiri dalam mengerjakan tugas hariannya. Pada akhirnya, pemeriksaan tindakan plagiat ini akan menambah lagi waktu yang digunakan dalam pemeriksaan dan penilaian tugas harian yang dilakukan oleh pengajar secara manual.

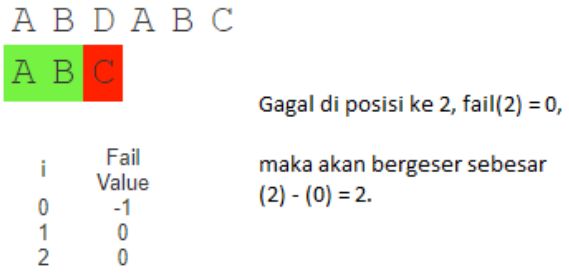
Oleh karena itu dibutuhkan suatu sistem yang dapat melakukan otomatisasi pemeriksaan dan penilaian *source code*, dan juga pendeteksian *source code plagiarism* yang dapat memangkas waktu dalam pemeriksaan dan penilaian tugas harian mahasiswa.

TINJAUAN PUSTAKA

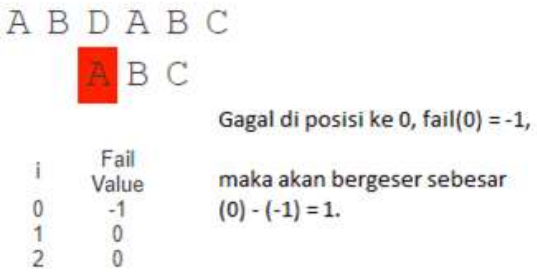
Beberapa algoritma pencocokan *string* yang akan dibandingkan oleh peneliti.

Algoritma *Knuth-Morris-Pratt* (KMP)

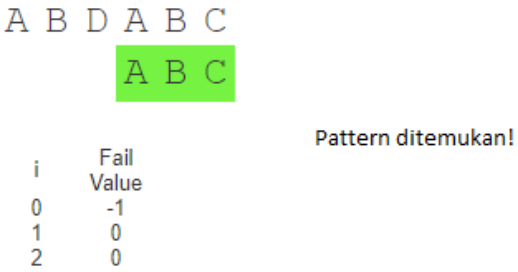
Algoritma *Knuth-Morris-Pratt* (KMP) merupakan salah satu algoritma pencocokan *string* untuk pencocokan 1 pola. Algoritma ini dibuat oleh D. E. Knuth, J. H. Morris, dan V. R. Pratt pada tahun 1974 tetapi baru dipublikasikan ke publik pada tahun 1977. Algoritma ini merupakan pengembangan dari algoritma yang sudah ada yaitu algoritma *Brute-Force* (Halimatus S:2017) (Maya Rosaria, Boko Susilo, Ernawati: 2015). Yang dikembangkan dalam algoritma *Knuth-Morris-Pratt* ini adalah saat melakukan pergeseran ketika ditemukan ketidakcocokkan. Jika algoritma *Brute-Force* akan melakukan pergeseran sebanyak 1 karakter ke kanan jika ditemukan ketidakcocokkan, maka algoritma *Knuth-Morris-Pratt* ini akan melakukan pergeseran sejauh mungkin yang bisa dilakukan dengan melihat tabel yang berisi nilai banyaknya karakter awal yang tidak cocok (Rivalri Kristianto Hondro et al: 2016).



Gambar 1. Ilustrasi Algoritma Knuth-Morris-Pratt (1)



Gambar 2. Ilustrasi Algoritma Knuth-Morris-Pratt (2)



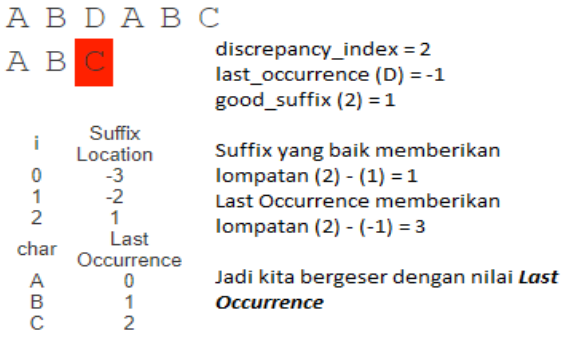
Gambar 3. Ilustrasi Algoritma Knuth-Morris-Pratt (3)

Jika ***n*** merupakan banyaknya karakter yang ada di dalam teks, dan ***m*** merupakan banyaknya karakter di dalam suatu pola yang akan dicari, maka algoritma ini memiliki kompleksitas waktu sebesar **O(*n*)** dalam proses pencocokan *string*, sedangkan **O(*m*)** dalam melakukan *preprocessing*. Dan algoritma ini memiliki kompleksitas ruang sebesar **O(*m*)** untuk 1 pola yang dicari (Cormen, Thomas H. et al: 2009) (Sedgewick, Robert dan Kevin Wayne: 2011).

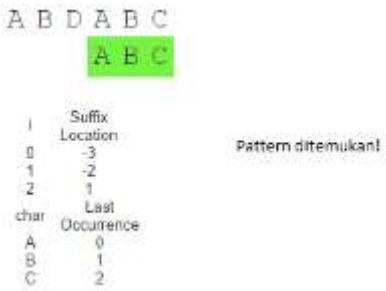
Algoritma *Boyer-Moore* (BM)

Algoritma *Boyer-Moore* (BM) merupakan salah satu algoritma pencocokan *string* untuk pencocokan 1 pola. Algoritma ini dibuat oleh Bob Boyer dan J. S. Moore pada tahun 1977. Algoritma ini merupakan pengembangan dari algoritma yang sudah ada yaitu algoritma *Brute-Force*.

Sama seperti algoritma *Knuth-Morris-Pratt*, dimana algoritma ini juga mengandalkan teknik pergeseran sejauh mungkin yang bisa dilakukan untuk menghindari perbandingan karakter pada *string* yang diperkirakan gagal, tetapi bedanya adalah algoritma ini ketika sedang melakukan pencocokan pola terhadap teks akan membandingkan mulai dari karakter paling kanan yang ada di pola terlebih dahulu, baru selanjutnya bergeser ke kiri (Eza Rahmanita: 2014).



Gambar 4. Ilustrasi Algoritma Boyer-Moore (1)



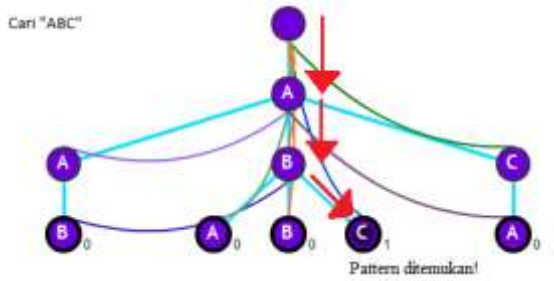
Gambar 5. Ilustrasi Algoritma Boyer-Moore (2)

Untuk teks yang memiliki panjang ***n*** dan pola yang dicari memiliki panjang ***m*** maka untuk kompleksitas algoritma ini adalah sebesar **O(*n/m*)** untuk waktu terbaik (*best case*), dan **O(*nm*)** untuk waktu terburuknya (*worst case*). Algoritma ini juga mempunyai waktu *preprocessing* sebesar **O(*m*+*k*)**, dimana ***k*** adalah besaran macam karakter yang digunakan (contoh: jika karakter yang digunakan adalah alfabet, maka besarnya adalah 26 macam karakter). Dan memiliki kompleksitas ruang sebesar **O(*k*)** (Sedgewick, Robert dan Kevin Wayne: 2011).

Algoritma *Aho-Corasick* (AC)

Algoritma Aho-Corasick (AC) merupakan salah satu algoritma pencocokan *string* yang dapat mengatasi pencocokan banyak pola. Algoritma ini dibuat oleh Alfred V. Aho dan Margaret J. Corasick pada tahun 1975.

Algoritma ini merupakan pengembangan dari algoritma sebelumnya yaitu algoritma *Knuth-Morris-Pratt*. Algoritma ini memakai struktur data *tries* yang dipergunakan untuk menyimpan set yang dinamis atau *array* assosiatif dimana kunci yang ada biasanya berupa *string*.

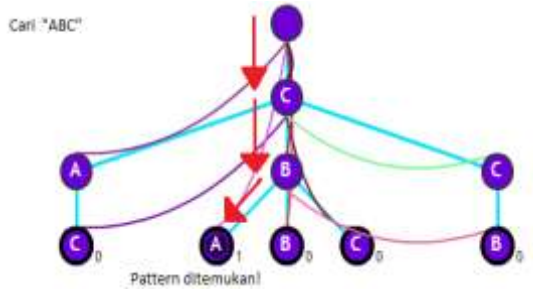


Gambar 6. Ilustrasi Algoritma Aho-Corasick

Algoritma ini memiliki kompleksitas waktu sebesar **O(*n*+*m*+*z*)**, dengan ***n*** merupakan banyak pola, ***m*** merupakan panjang dari pola yang digunakan dalam pencocokan, dan ***z*** merupakan jumlah output yang sesuai atau jumlah kemunculan pola (Achmad Nur Sholeh: 2017). Sedangkan untuk kompleksitas ruang yang digunakan adalah sebesar **O(*k***m***n*)** dimana ***k*** adalah besaran macam karakter yang digunakan (contoh: jika alfabet besarnya 26 macam karakter) (GeeksforGeeks: 2011).

Algoritma *Commentz-Walter* (CW)

Algoritma *Commentz-Walter* (CW) merupakan salah satu algoritma pencocokan *string* yang dapat mengatasi pencocokan banyak pola. Algoritma ini dibuat oleh Beate Commentz Walter pada tahun 1979. Algoritma ini merupakan pengembangan dari algoritma *Aho-Corasick* dan juga algoritma *Boyer-Moore*. Algoritma ini menggunakan struktur data *tries*. Algoritma ini bekerja layaknya algoritma *Aho-Corasick* tetapi yang membedakan adalah pola yang disimpan ke dalam *tries* dibalik terlebih dahulu (Joni, Arkanul Islam: 2014) (Yudi dan Tintin Chandra: 2013).



Gambar 7. Ilustrasi Algoritma Commentz-Walter

Algoritma *Rabin-Karp* (RK)

Algoritma *Rabin-Karp* (RK) merupakan salah satu algoritma pencocokan *string* yang dapat mengatasi pencocokan satu atau banyak pola. Algoritma ini dibuat oleh Richard M. Karp dan Michael O. Rabin pada tahun 1987. Algoritma ini tidak sama seperti 2 algoritma sejenis lainnya (seperti algoritma *Knuth-Morris-Pratt* dan algoritma *Boyer-Moore*) yang memakai teknik pergeseran sejauh mungkin untuk melangkahi penggalan teks (*substring*) yang sudah pasti bukan pola yang dicari, tetapi algoritma ini menggunakan fungsi *hash* untuk menemukan suatu pola yang terdapat pada teks.

Untuk teks yang memiliki panjang ***n*** dan memiliki pola sebanyak ***p*** yang merupakan kombinasi gabungan dari panjang ***m***, maka kompleksitas waktu rata-rata dan terbaiknya adalah sebesar **O(*n*+*m*)** dalam kompleksitas ruangnya sebesar **O(p)**, dan memiliki kompleksitas waktu terburuknya sebesar **O(*nm*)**. Dalam pengaplikasiannya, algoritma ini bisa digunakan untuk melakukan pendeteksian tindakan plagiat (Brinardi Leonardo dan Seng Hansung (2017), (Salmuasih dan Andi Sunyoto: 2013).

Rabin-Karp mempresentasikan setiap karakter ke dalam bentuk desimal digit (digit *radix*-d). ∑ = {0, 1, 2, 3, ..., d}, dimana d = |∑| (Thomas H. Cormen et al, 2009). Contoh misalkan dimasukkan karakter "ABC", maka karakter "A" akan dianggap nilainya sebagai 0, karakter "B" nilainya adalah 1, dan "C" nilainya adalah 2. Hal tersebut akan dipakai saat penghitungan *hash value* pada pola dan *substring* pada teks. Penghitungan *hash value* dilakukan dengan menggunakan rumus:

Hash value =
$(d^{m-1} . T[0]) + (d^{m-2} . T[1]) + ... + (d^{0} . T[m]) \bmod q$

Dimana:
d = basis {0,1,2,3, ..., dst}
T[] = *array* teks
m = panjang pola
q = nilai *modulo*

Contoh untuk menghitung *hash* value dengan pola "ABC" dengan basis 26 (A-Z).

Hash value ("ABC")
$= (26^2 . 0) + (26^1 . 1) + (26^0 . 2)$
$= 0 + 26 + 2 =$ **28**

Dalam melakukan pencocokan *hash value* pada *string* selanjutnya pada teks (bergeser 1 karakter ke kanan) maka algoritma *Rabin-Karp* tidak menghitung ulang kembali *hash value* satu per satu dari awal, tetapi melanjutkan perhitungan dengan *hash value* yang sudah ada (Salmuasih dan Andi Sunyoto, 2013; Pingky Alfa Ray Leo Lede et al, 2014). Menggunakan rumus:

Hash value selanjutnya:

$t_{s+1} = (d\,(t_s - T[s+1]\,h) + T[s+m+1] \bmod q$

Dimana:
t_s = *hash value* dengan panjang m dari *substring* T [s + 1 .. s + m], untuk s = 0, 1, ..., n – m
t_{s+1} = *hash value* selanjutnya yang dihitung dari t_s
d = *radix* desimal (bilangan basis 10)
h = d^{m-1}
n = panjang teks
m = panjang pola
q = nilai *modulo*

Contoh untuk menghitung *hash value* selanjutnya pada teks "ABCD".

Untuk mencari hash value selanjutnya ("BCD"):

Diketahui:
t_s = *Hash value* ("ABC") = 28
h = 26^2 = 676
T[s+1] = "A" = 0
T[s+m+1] = "D" = 3

$t_{s+1} = (d\,(t_s - T[s+1]\,h) + T[s+m+1]$
$t_{s+1} = (26\,(28 - (0 \cdot 676)) + 3$
$t_{s+1} = (26\,(28 - 0)) + 3 = 728 + 3 =$ **731**

Jika diperiksa dengan rumus *hash value* awal maka hasilnya sama
Hash value ("BCD") = $(26^2 \cdot 1) + (26^1 \cdot 2) + (26^0 \cdot 3) = 676 + 52 + 3 =$ **731**

Misal ada teks sebagai berikut "ABC" dan pola yang dicari adalah "ABC" maka langkah-langkah dalam Algoritma *Rabin-Karp* adalah sebagai berikut:

Langkah 1

Hash value pola ("ABC") = $(26^2 . 0) + (26^1 . 1) + (26^0 . 2) = 0 + 26 + 2 =$ **28**

Hash value teks [0..2] ("ACA")
$= (26^2 . 0) + (26^1 . 2) + (26^0 . 0)$
$= 0 + 52 + 0 =$ **52**

Hash Value akan dibandingkan (**28** = **52**), karena *hash value* **tidak sama**, maka algoritma tetap berlanjut.

Langkah 2

Hash value pola ("ABC") = $(26^2 . 0) + (26^1 . 1) + (26^0 . 2) = 0 + 26 + 2 =$ **28**

Hash value teks [1..3] ("CAB")
$= (26 (52 – (0 . 676)) + 1$
$= (26 (52 – 0)) + 2 = (26 . 52) + 1$
$=$ **1353**

Hash value akan dibandingkan (**28** = **1353**), karena *hash value* **tidak sama**, maka algoritma tetap berlanjut.

Langkah 3

Hash value pola ("ABC") = $(26^2 . 0) + (26^1 . 1) + (26^0 . 2) = 0 + 26 + 2 =$ **28**

Hash value teks [2..4] ("ABC")
= (26 (1353 – (2 . 676)) + 2
= (26 (1353 – 1352)) + 2 = (26 . 1) + 2
= **28**

Hash values akan dibandingkan (**28** = **28**), karena *hash value* **sama**,
maka akan dicek karakter per karakter, setelah dicek dan hasilnya **sama**,
maka pola **ditemukan**.

METODOLOGI PENELITIAN

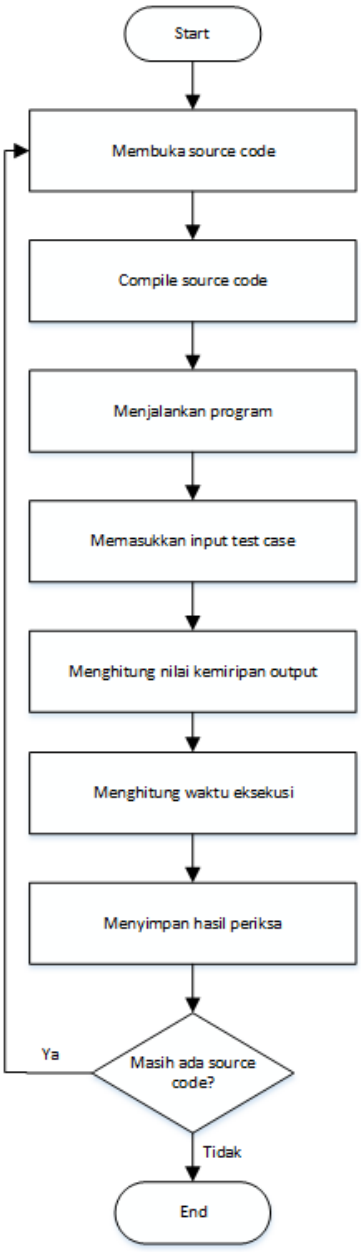
Pengujian *Source Code*

Untuk saat ini, pengujian yang dilakukan oleh pengajar dalam menguji kebenaran dari sebuah *source code* masih dengan menggunakan cara manual. Hal tersebut memiliki banyak kekurangan antara lain akan memakan waktu yang lama dalam melakukan pengujiannya, pengujian bisa jadi dipengaruhi oleh suasana hati dari penguji yang mana akan membuat pengujian menjadi subjektif bukan objektif, dan penilaian akan menjadi kurang konsisten.

Ada beberapa kriteria penilaian yang digunakan dalam melakukan pengujian *source code*, antara lain:

1. Kebenaran algoritma yang digunakan, sehingga dapat menerima *input* yang dijadikan sebagai *test case* dan dapat menghasilkan *output* sesuai dengan yang diharapkan.
2. Penamaan variabel, dimana nama-nama variabel yang digunakan harus sesuai dengan yang menjadi fungsinya (harus deskriptif), dimana nama variabel tersebut dapat menjelaskan fungsi dari variabel itu sendiri.
3. Kerapian *source code*, dilihat dari penggunaan spasi dan *enter* yang sesuai sehingga *source code* dapat mudah dibaca, penempatan-penempatan *method* yang terorganisir, dan juga dalam penempatan komentar yang baik (jika ada).
4. Keefektifan *source code*, dilihat dari seberapa ringkas *source code* yang dibuat tanpa mengurangi kebenaran dari algoritma suatu *source code*, dan tidak adanya pengulangan *statement* yang dirasa kurang perlu.

Berikut adalah *flowchart* yang menjelaskan proses dalam melakukan pengujian *source code* secara manual.



Gambar 9. Flowchart Pengujian Source Code Dengan Sistem Online Judge

Langkah-langkah dari pengujian *source code* secara manual adalah sebagai berikut:

1. Pertama-tama penguji akan membuka *source code* mahasiswa yang telah dikumpulkan satu per satu.
2. Selanjutnya, penguji akan melakukan *compile source code* dengan menggunakan *compiler* bahasa pemrograman yang digunakan.
3. Sesudah di-*compile*, maka penguji akan menjalankan program hasil dari yang sudah di-*compile* sebelumnya.
4. Setelah itu penguji akan memasukkan *input-input* sesuai dengan *input* yang dijadikan *test case* pada soal tersebut.
5. Selanjutnya, penguji akan memeriksa kebenaran *output* yang dikeluarkan oleh program tersebut, dan dicocokan dengan *output* yang diharapkan.
6. Selanjutnya, penguji akan melihat kembali *source code* untuk diperiksa dalam penamaan variabel yang digunakan.
7. Selanjutnya, penguji akan memeriksa kerapian *source code*.
8. Selanjutnya, penguji akan memeriksa keefektifan *source code*.
9. Setelah itu, penguji akan melakukan penilaian, dan nilai tersebut akan disimpan dalam suatu *file* excel.
10. Dan juga penguji akan menuliskan hasil periksa pada suatu *file* txt untuk diumumkan kepada mahasiswa.

Langkah-langkah ini akan berulang sampai tidak ada *source code* lagi yang harus diperiksa.

Pendeteksian *Source Code Plagiarism*

Pada dasarnya pendeteksian *source code plagiarism* dilakukan dengan cara membandingkan isi dari dokumen satu dengan dokumen yang lainnya. Setelah itu, proses yang dilakukan adalah melakukan perhitungan persentase kemiripan antara dokumen-dokumen tersebut. Dengan begitu, proses pendeteksian *source code plagiarism* menitikberatkan pada metode pencocokan *string* antara dua dokumen atau lebih. Oleh karena itu, algoritma yang akan dipakai adalah algoritma pencocokan *string*.

Saat ini, sudah banyak ditemukan algoritma dalam pencocokan *string*, contohnya algoritma *Knuth-Morris-Pratt*, algoritma *Boyer-Moore*, dan algoritma *Rabin-Karp*. Dari ketiga algoritma yang sudah disebutkan diatas, algoritma *Rabin-Karp* yang dipilih untuk penelitian kali ini memang bukanlah algoritma yang tercepat jika disejajarkan dengan algoritma lainnya. Itu semua bisa dilihat dari masing-masing kompleksitas waktunya.

Tetapi itu berlaku jika pencocokan *string* yang dilakukan hanya dengan 1 pola, sedangkan *Rabin-Karp* akan unggul dalam pencocokan *string* dengan banyak pola.

Itu dikarenakan dalam algoritma *Rabin-Karp* hanya memerlukan sekali *preprocessing* untuk setiap dokumen dimana *hash values* tersebut bisa dipakai berulang-ulang untuk dicocokan dengan dokumen lainnya. Dan jika kita memilih nilai *modulo* yang tepat, maka akan mempercepat waktu prosesnya (Pinky Alfa Ray Leo Lede, Adriana Fanggidae, dan Yulianto T. Polly: 2014). Untuk kasus ini, algoritma *Rabin-Karp* akan unggul dalam masalah kecepatan dibanding algoritma lainnya untuk mencocokan banyak pola yang dibutuhkan dalam pendeteksian *source code plagiarism*.

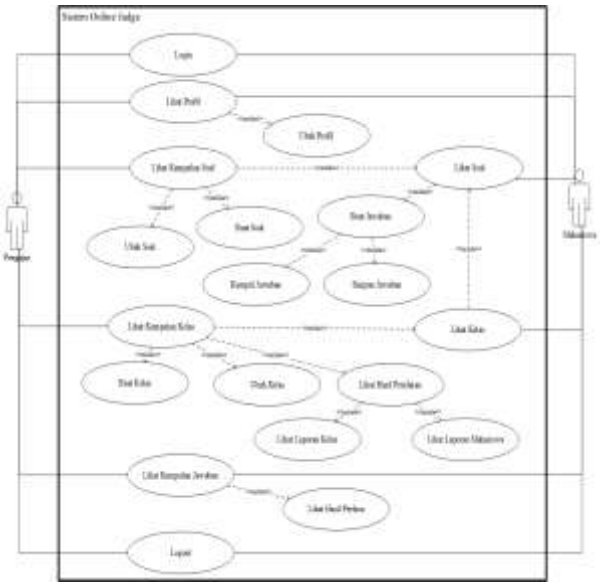
Tetapi saat ini, algoritma pencocokan *string* untuk banyak pola juga sudah banyak dikembangkan dari algoritma sebelumnya, yaitu algoritma *Aho-Corasick* dari algoritma *Knuth-Morris-Pratt*, dan algoritma *Commentz-Walter* yang dikembangkan berdasarkan algoritma *Boyer-Moore*. Kedua algoritma ini sudah pasti memiliki kompleksitas waktu yang lebih cepat dibandingkan algoritma *Rabin-Karp* karena kedua algoritma ini dikembangkan dari algoritma sebelumnya yang juga sudah diuji kompleksitas waktunya lebih cepat dibandingkan dengan algoritma *Rabin-Karp* ini. Dalam pencocokan banyak pola yang dilihat bukan hanya kompleksitas waktunya saja, tetapi juga kompleksitas ruang yang digunakan pada saat algoritma itu dijalankan (Salmela, Leena, Jorma Tarhio dan Jari Kyt Joki: 2006). Pada algoritma *Aho-Corasick* dan *Commentz-Walter*, kedua algoritma ini memakai struktur data *tries* dimana dalam pengaplikasiannya akan memakan memori yang cukup besar. Sedangkan, algoritma *Rabin-Karp* hanya menggunakan 1 variabel dengan tipe data bilangan bulat (*integer*) untuk menyimpan masing-masing *hash value* yang telah dihitung. Dimana itu akan memakan memori yang jauh lebih sedikit dibandingkan dengan penggunaan memori pada struktur data *tries*.

Dengan alasan diatas, maka peneliti memilih algoritma *Rabin-Karp* sebagai algoritma untuk pencocokan *string* yang akan dipakai dalam pendeteksian *source code plagiarism* pada penelitian kali ini.

PEMBAHASAN

Rancangan Use Case Diagram

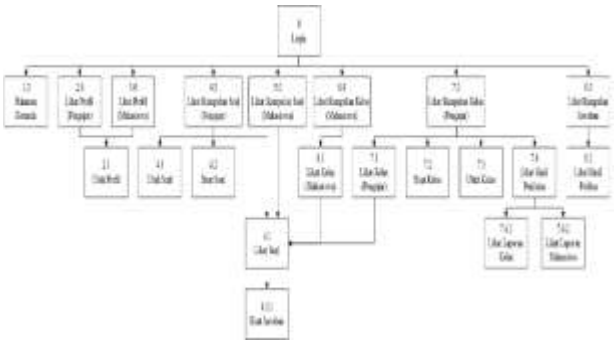
Gambar 10 Use Case Diagram Sistem On Line Judge



Gambar 10. Use Case Diagram Sistem Online Judge

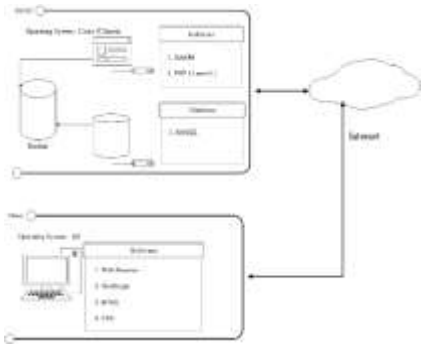
Rancangan Struktur Menu Sistem Online Judge

Gambar 11. Struktur Menu Sistem Online Judge



Rancangan Arsitektur Sistem

Gambar 12 Arsitektur Sistem Online Judge



HASIL EVALUASI SISTEM

Berikut merupakan tabel hasil uji untuk mengetahui seberapa besar perbandingan algoritma *Rabin-Karp* dengan algoritma *Rabin-Karp* yang telah dimodifikasi dengan menggunakan sampel 2 *source code* yang identik sama:

Tabel 1.
Hasil Uji Perbandingan Dengan Modifikasi Penghapusan Komentar, Spasi, dan Enter

Pengujian		Tingkat Kesulitan *Source Code*		
		Mudah (50 – 100 baris) =90an baris	Sedang (100 – 250 baris) =190an baris	Sulit (250 – 700 baris) =600an baris
Algoritma *Rabin-Karp*	Waktu	3,63762116 detik	9,82530785 detik	123,84402299 detik
	Persentase Kemiripan	100%	100%	100%
Algoritma *Rabin-Karp* yang dimodifikasi	Waktu	0,80536199 detik	2,99534011 detik	27,15321803 detik
	Persentase Kemiripan	100%	100%	100%

1. Untuk hasil uji perbandingan algoritma *Rabin-Karp* dengan algoritma *Rabin-Karp* yang dimodifikasi dalam penghapusan komentar, spasi, dan *enter* dapat dilihat pada Tabel 1. Pada tabel tersebut dapat dilihat bahwa dengan melakukan penghapusan komentar, spasi, dan *enter* dapat membuat waktu proses semakin cepat hingga 75,15% dari waktu awal tanpa mempengaruhi tingkat keakuratan dalam penghitungan persentase kemiripan.

Tabel 2.
Hasil Uji Perbandingan Dengan Modifikasi Proses Pencocokan String Yang Telah Disesuaikan Tanpa Modifikasi (1).

Pengujian		Tingkat Kesulitan *Source Code*		
		Mudah (50 – 100 baris) =90an baris	Sedang (100 – 250 baris) =190an baris	Sulit (250 – 700 baris) =600an baris
Algoritma *Rabin-Karp*	Waktu	3,63762116 detik	9,82530785 detik	123,84402299 detik
	Persentase Kemiripan	100%	100%	100%
Algoritma *Rabin-Karp* yang dimodifikasi	Waktu	0,00249100 detik	0,00675821 detik	0,01603889 detik
	Persentase Kemiripan	100%	100%	100%

Tabel 3.
Hasil Uji Perbandingan Dengan Modifikasi Proses Pencocokan String Yang Telah Disesuaikan Ditambah Dengan Modifikasi (2).

Pengujian		Tingkat Kesulitan *Source Code*		
		Mudah (50 – 100 baris) =90an baris	Sedang (100 – 250 baris) =190an baris	Sulit (250 – 700 baris) =600an baris
Algoritma *Rabin-Karp*	Waktu	0,80536199 detik	2,99534011 detik	27,15321803 detik
	Persentase Kemiripan	100%	100%	100%
Algoritma *Rabin-Karp* yang dimodifikasi	Waktu	0,00160599 detik	0,00253081 detik	0,00878906 detik
	Persentase Kemiripan	100%	100%	100%

2. Untuk hasil uji perbandingan algoritma *Rabin-Karp* dengan algoritma *Rabin-Karp* yang dimodifikasi dalam proses pencocokan *string*-nya dapat dilihat pada Tabel 2 dan 3. Pada tabel tersebut dapat dilihat bahwa dengan melakukan modifikasi pada proses pencocokan *string* dapat membuat waktu proses semakin cepat hingga 99,95% dari waktu awal tanpa mempengaruhi tingkat keakuratan dalam penghitungan persentase kemiripan.

3. Untuk hasil uji perbandingan algoritma *Rabin-Karp* dengan algoritma *Rabin-Karp* yang dimodifikasi dalam mengabaikan *statement* yang tidak penting dapat dilihat pada Tabel 4. Pada tabel tersebut dapat dilihat bahwa dengan semakin banyaknya penambahan *statement* yang tidak penting dan sedikitnya jumlah baris *source code* yang asli akan mempengaruhi tingkat persentase dan mengurangi tingkat keakuratan dalam penghitungan persentase kemiripan. Dengan melakukan modifikasi dalam mengabaikan *statement* yang tidak penting maka akan menghasilkan tingkat keakuratan yang maksimal (100%).

Tabel 4.
Hasil Uji Perbandingan Dengan Modifikasi Mengabaikan Statement Yang Tidak Penting Ditambah Dengan Modifikasi (1) dan (2)

Pengujian		Tingkat Kesulitan *Source Code*		
		Mudah (50 – 100 baris) ~90an baris	Sedang (100 – 250 baris) ~190an baris	Sulit (250 – 700 baris) ~600an baris
Algoritma *Rabin-Karp*	Penambahan 10 baris	91.28%	94.63%	98.37%
	Penambahan 50 baris	67.68%	77.91%	92.35%
	Penambahan 100 baris	51.15%	63.81%	85.79%
Algoritma *Rabin-Karp* yang dimodifikasi	Penambahan 10 baris	100%	100%	100%
	Penambahan 50 baris	100%	100%	100%
	Penambahan 100 baris	100%	100%	100%

4. Untuk hasil uji perbandingan algoritma *Rabin-Karp* dengan algoritma *Rabin-Karp* yang dimodifikasi dalam penamaan variabel dapat dilihat pada Tabel 5. Pada tabel tersebut dapat dilihat bahwa dengan melakukan modifikasi dalam penamaan variabel maka akan menghasilkan tingkat keakuratan yang maksimal (100%), walaupun dalam waktu proses mengalami penurunan sebesar 45,11%.

Tabel 5.
Hasil Uji Perbandingan Dengan Modifikasi Penamaan Variabel Ditambah Dengan Modifikasi (1), (2), dan (3)

Pengujian		Tingkat Kesulitan *Source Code*		
		Mudah (50 – 100 baris) ~90an baris	Sedang (100 – 250 baris) ~190an baris	Sulit (250 – 700 baris) ~600an baris
Algoritma *Rabin-Karp*	Waktu	0,0038421 detik	0,00262594 detik	0,00993609 detik
	Persentase Kemiripan	60.55%	78.36%	73.51%
Algoritma *Rabin-Karp* yang dimodifikasi	Waktu	0,00651598 detik	0,00738311 detik	0,02435994 detik
	Persentase Kemiripan	100%	100%	100%

Dengan menggunakan algoritma *Rabin-Karp* ini, pada akhirnya dapat memangkas waktu yang sangat besar dibandingkan dengan pemeriksaan secara manual. Berdasarkan hasil observasi langsung yang dilakukan peneliti, untuk melakukan pemeriksaan tindakan plagiat secara manual dapat memakan waktu rata – rata sebesar 30 – 45 menit untuk 30 – 40 *source code*. Sedangkan, sistem *online judge* ini hanya akan memakan waktu rata – rata sekitar 1 – 6 detik saja tergantung seberapa kompleks *source code* tersebut.

Tabel 6.
Perbandingan Waktu Pendeteksian Source Code Plagiarism pada Sistem Online Judge dengan Waktu Pendeteksian Source Code Plagiarism secara Manual

Pendeteksian *Source Code Plagiarism*	Waktu Terbaik	Waktu Terburuk	Waktu Rata-Rata
Sistem *Online Judge*	0 – 1 detik	5 – 10 detik	1 – 6 detik
Manual	15 menit	45 – 120 menit	30 – 45 menit

KESIMPULAN DAN SARAN

Berdasarkan hasil penelitian yang dilakukan oleh peneliti, maka peneliti dapat disimpulkan sebagai berikut:

1. Otomatisasi pengujian kebenaran *source code* dengan menggunakan Sistem *Online Judge* terbukti dapat memangkas waktu yang sangat besar, itu dapat dilihat pada hasil evaluasi sistem dimana dalam melakukan pengujian 1 *source code* hanya memakan waktu rata – rata 0 sampai dengan 3 detik, dibanding dengan pengujian kebenaran *source code* secara manual yang memakan waktu rata – rata 3 sampai dengan 5 menit.
2. Dengan melakukan modifikasi pada algoritma *Rabin-Karp* untuk pendeteksian *source code plagiarism* menghasilkan perbedaan waktu dan tingkat keakuratan yang sangat signifikan, itu dapat dilihat pada hasil evaluasi sistem dimana waktu proses pendeteksian dapat dipangkas sampai dengan 99.95%, dengan tingkat keakuratan yang mencapai 100% akurat.

Berikut merupakan saran dari peneliti untuk penelitian selanjutnya:
Dibutuhkannya penambahan *white-box testing* dalam pengujian *source code* untuk dapat diimplementasikan ke dalam sistem *online judge* yang menjadi kebutuhan berdasarkan dari hasil wawancara, khususnya dalam pembatasan *syntax* yang boleh digunakan dalam soal algoritma tertentu.

DAFTAR PUSTAKA

Achmad Nur Sholeh (2017), "Komparasi Algoritma String Matching Pada Pola Teks", Jurnal Mandiri, Vol. 1, No. 2.

Brinardi Leonardo dan Seng Hansung (2017), "*Text Documents Plagiarism Detection using Rabin-Karp and Jaro-Winkler Distance Algorithms*", Indonesian Journal of Electrical Engineering and Computer Science, Vol. 5, No. 2.

Cormen, Thomas H. et al (2009), *Introduction to Algorithms*, Edisi ke-3, London: MIT Press.

Eza Rahmanita (2014), "*Pencarian String Menggunakan Algoritma Boyer Moore Pada Dokumen*", Jurnal Ilmiah NERO, Vol. 1, No. 1.

GeeksforGeeks (2011), *Trie | (Insert and Search)*, sumber: https://www.geeksforgeeks.org/trie-insert-and-search/ (diakses 28 Mei 2018).

Halimatus S. (2017), "*Implementasi Algoritma Knuth-Morris-Pratt Pada Fungsi Pencarian Judul Tugas Akhir Repository*", Jurnal Komputasi, Vol. 14, No. 1.

Johnny, Arkanul Islam (2014), "*Analysis of Multiple String Pattern Matching Algorithms*", International Journal of Advanced Computer Science and Information Technology (IJACSIT), Vol. 3, No. 4.

Maya Rossaria, Boko Susilo, Ernawati (2015), "*Implementasi Algoritma Pencocokan String Knuth-Morris-Pratt Dalam Aplikasi Pencarian Dokumen Digital Berbasis Android*", Jurnal Rekursif, Vol. 3, No. 2.

Pingky Alfa Ray Leo Lede, Adriana Fanggidae, dan Yulianto T. Polly (2014), *"Implementasi Algoritma Rabin-Karp Untuk Mendeteksi Dugaan Plagiarisme Berdasarkan Tingkat Kemiripan Kata Pada Dokumen Teks"*, J-ICON, Vol. 2:50-64.

Rivalri Kristianto Hondro et al (2016), "*Implementasi Algoritma Knuth Morris Pratt Pada Aplikasi Penerjemah Bahasa Mandailing-Indonesia*", Jurnal Riset Komputer (JURIKOM), Vol. 3, No. 4.

Salmela, Leena, Jorma Tarhio dan Jari Kytöjoki (2006), "*Multi-Pattern String Matching with q-Grams*", ACM Journal of Experimental Algorithmics, Vol. 11.

Salmuasih dan Andi Sunyoto (2013), "*Implementasi Algoritma Rabin Karp untuk Pendeteksian Plagiat Dokumen Teks Menggunakan Konsep Similarity*", Seminar Nasional Aplikasi Teknologi Informasi (SNATI) 2013, Yogyakarta.

Sedgewick, Robert dan Kevin Wayne (2011), *Algorithms*, Edisi ke-4, Boston: Pearson Education.

Sulistia Endah (2017), *Indonesia Kekurangan Banyak Programmer Handal!*, sumber: https://telko.id/13687/indonesia-kekurangan-banyak-programer-handal/ (diakses 14 Juni 2018).

Yudi dan Tintin Chandra (2013), "*Simulasi Pencarian Pola Kata Pada File Teks Berdasarkan Multi Pattern Matching Dengan Metode Wu-Manber*", Jurnal Core IT, Vol. 1, No. 2.