BAB II

LANDASAN TEORI

2.1 Plagiarisme

Dalam Kamus Besar Bahasa Indonesia, plagiarisme atau sering disebut plagiat didefinisikan sebagai pengambilan karangan, pendapat, dan sebagainya milik orang lain dan menjadikannya seolah-olah karangan atau pendapat sendiri (Tim Penyusun Kamus Pusat Bahasa, 2005).

Lebih lanjut lagi, dalam Peraturan Menteri Pendidikan Nasional Nomor 17 Tahun 2010 mendefinisikan plagiarisme dalam tingkat Perguruan Tinggi sebagai "Perbuatan secara sengaja atau tidak sengaja dalam memperoleh atau mencoba memperoleh kredit atau nilai untuk suatu karya ilmiah, dengan mengutip sebagian atau seluruh karya dan/atau karya ilmiah pihak lain yang diakui sebagai karya ilmiahnya, tanpa menyatakan sumber secara tepat dan memadahi" (Departemen Pendidikan Nasional, 2010).

Berdasarkan naskah yang sama, tindakan yang dapat dikategorikan sebagai plagiarisme meliputi, tetapi tidak terbatas pada:

1. Mengacu dan/atau mengutip, secara acak maupun tidak, kata-kata dan/atau kalimat, data dan/atau informasi dari suatu sumber tanpa menyatakan sumber secara memadahi.
2. Menggunakan sumber gagasan, pendapat, pandangan, atau teori tanpa menyatakan sumber secara memadahi.
3. Merumuskan dengan kata-kata dan/atau kalimat sendiri dari sumber kata-kata dan/atau kalimat, gagasan, pendapat, pandangan, atau teori tanpa menyatakan sumber secara memadahi.

4. Menyerahkan suatu karya ilmiah yang dihasilkan dan/atau telah dipublikasikan oleh pihak lain sebagai karya ilmiahnya tanpa menyatakan sumber secara memadahi.

Menurut Goenawan, dkk (2005: 1-2), beberapa teknik plagiat yang dikenal selama ini meliputi:

1. *Word-to-word plagiarism*: menyalin setiap kata secara langsung tanpa diubah sedikitpun.
2. *Plagiarism of the form of a source*: menyalin dan/atau menulis ulang kode-kode program tanpa mengubah struktur dan jalannya program.
3. *Plagiarism of the authorship*: mengakui hasil karya orang lain sebagai karya sendiri dengan cara mencantumkan nama sendiri menggantikan nama pengarang sebenarnya.

Sedangkan praktik plagiarisme pada sebuah kode program dapat dikelompokkan menjadi 2, yaitu:

1. Leksikan: perubahan pada kode program, misalnya:
   a. Komentar diubah (ditambah, dikurangi, atau diganti).
   b. Format penulisan diubah.
   c. Nama variabel diubah.
2. Struktural: perubahan struktur program, misalnya:
   a. Perubahan urutan algoritma yang tidak mengubah jalannya program.
   b. Prosedur diubah menjadi fungsi atau sebaliknya.
   c. Pemanggilan prosedur diubah menjadi isi prosedur itu sendiri.

Pendeteksian plagiarisme pada kode program dapat dicapai menggunakan berbagai cara. Roy dan James (2007: 44-59) mengungkapkan terdapat 6 teknik dasar dalam melakukan pendeteksian plagiarisme dalam kode program, yaitu:

1. *Text-based Techniques*: Dokumen kode program dianggap sebagai sederetan *lines* atau *string* dan dinormalisasikan sebelum dibandingkan secara langsung.
2. *Token-based Techniques*: Dokumen kode diubah menjadi serangkaian *token* yang kemudian akan ditelusuri untuk mencari adanya duplikasi dari *token* tersebut.
3. *Tree-based Techniques*: Dokumen kode diubah ke dalam bentuk *parse tree* atau *Abstract Syntax Tree*, dan dibandingkan berdasarkan setiap *node* yang terdapat pada *tree* tersebut.
4. *PDG-based Techniques*: Dokumen kode diubah ke dalam bentuk *Program Dependency Graph* untuk mendapatkan aliran data dan kontrol pada program, dan kemudian dibandingkan.
5. *Metrics-based Techniques*: Membandingkan data statistik dari sebuah kode program dengan data statistik kode program lainnya, seperti banyaknya variabel yang digunakan, banyaknya, pengulangan, dan sebagainya.
6. *Hybrid Approaches*: Menggabungkan 2 atau lebih teknik-teknik yang sudah ada untuk mencapai performa yang lebih baik.

Untuk mendeteksi plagiarisme, sudah terdapat beberapa contoh aplikasi pendeteksi plagiarisme yang dibuat sebagai sistem yang *structure oriented* yaitu SIM, MOSS, Plague, YAP, dan JPlag. SIM mampu mendeteksi program yang ditulis dalam bahasa Java, C, Pascal, Modula-2, List, dan Miranda meski dapat digunakan juga untuk memeriksa kesamaan dari dokumen tekstual biasa. MOSS

mendukung pendeteksian plagiarisme pada kode program Ada, Java, C, C++, Pascal, maupun *plain text*. Plague hanya mendukung pendeteksian kode program berbahasa C, dan kemudian dikembangkan menjadi YAP1, YAP2, dan YAP3. JPlag mendukung pendeteksian plagiarisme pada kode C, C++, Scheme, dan Java dan memberikan hasil dalam format HTML. (Ahmad Gull Liaqat, Aijaz Ahmad, 2011).

2.2 Algoritma Boyer Moore

Ide dasar dari algoritma ini adalah mendapatkan lebih banyak informasi dari pencocokan pola *pattern* sebelah kanan. Pada tahap awal, *pattern* yang akan dicari disejajarkan dengan teks *string* yang akan diperiksa, lalu karakter terakhir pada *pattern* akan dicocokkan dengan karakter *char* pada *string*. Bila *char* bukan merupakan karakter yang berada pada *pattern*, maka geser *pattern* ke kanan sebanyak panjang *pattern*. Alasannya adalah bila *pattern* digeser kurang dari panjang *pattern*, maka karakter yang tidak mungkin ada pada pattern tersebut akan sejajar dengan *pattern*. Bila karakter yang dicocokkan tidak sama, tetapi terdapat dalam *pattern*, maka geser *pattern* sebanyak jarak lokasi *char* pertama dari kanan pada *pattern* dengan karakter terakhir dari *pattern* (lebih lanjut disebut sebagai *Delta[char]*). Bila karakter yang dicocokkan sama, maka periksa ke karakter sebelumnya sampai seluruh karakter selesai diperiksa, atau ditemukan karakter yang tidak sama. Bila seluruh karakter sama, maka pencarian berhasil. Ketika ditemukan karakter yang berbeda pada indeks *i* setelah berhasil mencocokkan *m* karakter pada *pattern*, sebut saja sebagai *subpattern*, maka dilakukan pengecekkan karakter indeks *i string*, sebut saja sebagai *terminal character*, pada *pattern*. Bila tidak ada *terminal character* pada sisi kiri

subpattern, maka geser *pattern* sebanyak [panjang *pattern* – *m*]. Bila terdapat *terminal character* pada sisi kiri *subpattern*, maka geser pattern sebanyak jarak *i* dengan lokasi *subpattern* kedua dari kanan pada *pattern* dimana karakter yang mendahului *subpattern* kedua ini tidak sama dengan karakter pada indeks *i* *pattern* (lebih lanjut disebut sebagai *Delta[j]*) atau menggeser *pattern* sebanyak *Delta[char]*. Pergeseran sebanyak *Delta[char]* atau *Delta[j]* pada kasus ini ditentukan oleh nilai *Delta* mana yang dapat menggeser *pattern* lebih jauh. Setelah dilakukan pergeseran *pattern*, cocokkan kembali *pattern* dan *string* mulai dari karakter terakhir *pattern*. Proses ini akan terus berlangsung sampai seluruh karakter berhasil dicocokkan atau ujung kanan *pattern* sudah melebihi ujung kanan *string* (Robert S. Boyer, J Strother Moore, 1977).

Saat ini aturan untuk menggeser *pattern* sebanyak *Delta[char]* disebut sebagai *Bad Character Rule*, sedangkan aturan untuk menggeser *pattern* sebanyak *Delta[j]* disebut sebagai *Good Suffix Rule*.

Berikut adalah contoh penggunaan algoritma *Boyer Moore* dalam pencarian *pattern* “AT-THAT” pada *string* “WHICH-FINALLY-HALTS.—AT-THAT-POINT” yang diambil dari jurnal Robert S. Boyer, J Strother Moore (1977).

Pertama sejajarkan *pattern* dengan *string*.

Pattern : AT- THAT

String : WHI CH- FI NALLY- HALTS. - - AT- THAT- POI NT

Bandingkan karakter terakhir pada *pattern* dan karakter terakhir pada *string*. Karena ‘T’ ≠ ‘F’ dan karakter ‘F’ tidak terdapat pada *pattern*, maka geser *pattern* sebanyak panjang *pattern*, yaitu 7 karakter.

Pattern : AT- THAT

String : WHI CH- F I NALL Y- HALT S . - - AT- THAT- P OI NT

Karakter terakhir pada *pattern* dan *string* tidak sama, namun karakter '-' terdapat pada *pattern*. Pada kasus ini, terapkan *Bad Character Rule* dan geser sebesar jarak karakter terkanan '-' pada *pattern* dengan karakter paling terakhir, yaitu 4 karakter.

Pattern : AT- THAT

String : WHI CH- F I NALL Y- HALT S . - - AT- THAT- P OI NT

Karakter pertama dari kanan pada *pattern* dan *string* sama, tetapi terjadi ketidaksamaan pada karakter kedua dimana 'A' ≠ 'L'. Karena sudah ada karakter yang sama, maka dilakukan pengecekan lebih lanjut pada *pattern* dengan 'L' sebagai *terminal character* dan 'T' sebagai *subpattern*. Pada sebelah kiri *subpattern* tidak terdapat *terminal character* 'L', maka geser *pattern* sebanyak [panjang *pattern* – *m*] = 6 karakter, dimana *m* adalah panjang *subpattern*, yaitu 1.

Pattern : AT- THAT

String : WHI CH- F I NALL Y- HALT S . - - AT- THAT- P OI NT

Karakter terakhir pada *pattern* dan *string* tidak sama, namun karakter 'A' terdapat pada *pattern*. Terapkan *Bad Character Rule* dan geser sebanyak 1 karakter.

Pattern : AT- THAT

String : WHI CH- F I NALL Y- HALT S . - - AT- THAT- P OI NT

Dua karakter paling belakang pada *pattern* dan *string* sama, tetapi terjadi ketidaksamaan pada karakter ketiga dimana 'H' ≠ '-'. Lakukan pengecekan lebih lanjut pada *pattern* dengan '-' sebagai *terminal character* dan 'AT' sebagai *subpattern*. Pada sebelah kiri *subpattern* terdapat *terminal character* '-', sehingga

dapat diterapkan *Bad Character Rule* dengan *Delta[char]* = 2. Selain itu terdapat pula pengulangan *subpattern* pada *pattern* yang tidak didahului oleh karakter 'H', sehingga dapat juga diterapkan *Good Suffix Rule* dengan *Delta[j]* = 5.

Ketika *Bad Character Rule* dan *Good Suffix Rule* dapat diterapkan pada saat yang bersamaan, pergeseran dilakukan sesuai dengan *Rule* mana yang dapat melakukan pergeseran lebih banyak. Dalam hal ini, *Good Suffix Rule* diterapkan karena dapat memberikan pergeseran lebih besar dari *Bad Character Rule* dengan pergeseran sebanyak 5 karakter.

Pattern : AT- THAT

String : WHI CH- FI NALLY- HALTS. - - AT- THAT- POI NT

Setiap karakter pada *pattern* sama dengan setiap karakter pada *string* yang dibandingkan, sehingga dapat disimpulkan bahwa *pattern* ditemukan dalam *string*.

Gambar 2.1, gambar 2.2, gambar 2.3, dan gambar 2.4 adalah *pseudocode* untuk menerapkan algoritma *Boyer Moore* yang diambil dari laporan penelitian milik Natalia (2011).

Gambar 2.1 adalah *pseudocode* untuk menyusun tabel *bad character*.

```
procedure preBmBc (
    input P : array[0..n-1] of char,
    input n : integer,
    input/output bmBc : array[0..n-1] of integer
)
Deklarasi :
    i: integer
Algoritma :
    for (i := 0 to ASIZE-1)
        bmBc[i] := m;
    endfor
    for (i := 0 to m - 2)
        bmBc[P[i]] := m - i - 1;
    endfor
```

Gambar 2.1 *Pseudocode* penghitungan tabel *bad character*

Gambar 2.2 adalah algoritma untuk menyusun table *suffixes* yang akan digunakan untuk menyusun table *Good Suffix*.

```
procedure preSuffixes (
    input P : array[0..n-1] of char,
    input n : integer,
    input/output suff : array[0..n-1] of integer
)
Deklarasi :
    f, g, i: integer
Algoritma :
    suff[n-1] := n;
    g := n-1;
    for (i:=n - 2 downto 0)
        if (i > g and (suff[i + n - 1 - f] < i - g))
            suff [i] := suff [i + n - 1 - f];
        else
            if (i < g)
                g := i;
            endif
            f := i;
            while (g >= 0 and P[g] = P[g + n - 1 - f])
                --g;
            endwhile
            suff[i] = f - g;
        endif
    endfor
```

Gambar 2.2 *Pseudocode* penghitungan tabel *suffix*

Gambar 2.3 adalah algoritma untuk menyusun table *Good Suffix*.

```
procedure preBmGs (
    input P : array[0..n-1] of char,
    input n : integer,
    input/output bmBc : array[0..n-1] of integer
)
Deklarasi:
    i, j: integer
    suff: array [0..RuangAlphabet] of integer

    preSuffixes(x, n, suff);

    for (i := 0 to n - 1)
        bmGs[i] := n
    endfor
    j := 0
    for (i := n - 1 downto 0)
        if (suff[i] = i + 1)
            for (j := j to n - 2 - i)
                if (bmGs[j] = n)
                    bmGs[j] := n - 1 - i;
                endif
            endfor
        endif
    endfor
    for (i = 0 to n - 2)
        bmGs[n - 1 - suff[i]] := n - 1 - i;
    endfor
```

Gambar 2.3 *Pseudocode* penghitungan tabel *good suffix*

Gambar 2.4 adalah algoritma utama untuk melakukan pencarian *string* pada *string* T sepanjang m karakter.

```
procedure BoyerMooreSearch (
    input m, n : integer,
    input P : array[0..n-1] of char,
    input T : array[0..m-1] of char,
    output ketemu : array[0..m-1] of boolean
)
Deklarasi:
    i, j, shift, bmBcShift, bmGsShift: integer
    bmBc : array[0..255] of integer
    bmGs : array[0..n-1] of integer
Algoritma:
    preBmBc (n, P, BmBc)
    preBmGs (n, P, BmGs)
    i := 0
    while (i <= m-n) do
        j := n - 1
        while (j >= 0 and T[i+j] = P[j]) do
            j := j - 1
        endwhile
        if (j < 0) then
            ketemu[i] := true;
        endif
        bmBcShift:= BmBc[chartoint(T[i+j])] - n + j + 1;
        bmGsShift:= BmGs[j];
        shift := max (bmBcShift, bmGsShift);
        i := i + shift;
    endwhile
```

Gambar 2.4 *Pseudocode* algoritma pencarian *Boyer Moore*

Hasil dari algoritma ini adalah sebuah *array of Boolean* sepanjang m, dimana *array* ini akan bernilai *true* pada setiap *index* dimana *pattern* P ditemukan pada *string* T.

2.3 Algoritma Horspool

Merupakan penyederhanaan dari algoritma *Boyer Moore*. Algoritma ini menghapus penggunaan *Good Suffix Rule* dan memodifikasi *Bad Character Rule* sedemikian rupa sehingga ketika karakter terakhir pada teks yang dibandingkan sejajar dengan lokasi karakter tersebut selanjutnya pada *pattern* bila karakter

tersebut ada pada *pattern* atau geser sebanyak panjang *pattern* bila tidak ada (Domenico Cantone, Simone Faro, 2003a).

Algoritma *Horspool* menggunakan tabel *bad character* yang sama dengan algoritma *Boyer-Moore*, hanya saja memiliki sedikit perbedaan pada algoritma pencarian utamanya.

Gambar 2.5 *pseudocode* untuk pencarian utama algoritma *Horspool* yang diambil dari jurnal milik R. Nigel Horspool (1980).

```
Delta12[*] ← patlen;
for j ← 1 to patlen – 1 do
    delta12[pat[j]] ← patlen – j;
lastch ← pat[patlen];
i ← patlen;
while i ≤ stringlen do
    begin
        ch ← string[i];
        if ch = lastch then
            if string[i-patlen + 1 ... j] = pat then
                return i – patlen + 1;
        i ← i + delta12[ch];
    end;
return 0;
```

Gambar 2.5 *Pseudocode* algoritma pencarian *Horspool*

2.4 Algoritma Fast Search

Dikembangkan oleh Domenico Cantone dan Simone Faro pada tahun 2003, algoritma ini merupakan pengembangan dari algoritma *Boyer Moore* dengan menerapkan *Bad Character Rule* algoritma *Horspool*.

Penggunaan *Bad Character Rule* algoritma *Horspool* didasari oleh pembuktian bahwa aturan tersebut dapat memberikan pergeseran yang lebih besar jika dan hanya jika ketidakcocokan langsung terjadi, maksudnya karakter terakhir pada pola tidak sama dengan karakter pada teks yang dicocokkan. Bila ketidakcocokan

tidak langsung terjadi, gunakan *Good Suffix Rule* sesuai dengan algoritma asli *Boyer Moore* (Domenico Cantone, Simone Faro, 2003a).

Algoritma *Fast Search* menggunakan tabel *bad character* dan *good suffix* yang sama dengan algoritma *Boyer Moore*. Hanya saja pada algoritma pencarian utamanya, pergeseran yang menggunakan tabel *bad character* digunakan seperti pada algoritma *Horspool*.

Gambar 2.6 adalah *pseudocode* untuk pencarian utama algoritma *Fast Search* seperti yang disertakan pada jurnal milik Domenico Cantoned dan Simone Faro (2003a).

```
Fast-Search (P, T)
    n = length (T)
    m = length (P)
    T' = T.P
    bcp = precompute-bad-character (P)
    gsp = precompute-good-suffix (P)
    s = 0
    while bcp(T'[s + m – 1]) > 0 do
        s = s + bcp(T'[s + m - 1])
    while s ≤ n – m do
        j = m – 2
        while j ≥ 0 and P[j] = T'[s + j] do
            j = j – 1
        if j < 0 then
            print(s)
        s = s + gsp (j + 1)
        while bcp (T'[s + m – 1] )> 0 do
            s = s + bcp (T'[s + m - 1])
```

Gambar 2.6 *Pseudocode* algoritma *Fast Search*

2.5 Algoritma Forward Fast Search

Merupakan perkembangan lebih lanjut dari algoritma *Fast Search* oleh Domenico Coantone dan Simone Faro pada tahun 2003. Algoritma ini menerapkan *Forward Good Suffix Rule* sebagai pengganti *Good Suffix Rule* di

atas penerapan *Bad Character Rule* algoritma *Horspool* pada algoritma *Boyer Moore*.

Sunday, pada jurnal milik Cantone, dkk (2003b: 15), menjelaskan bahwa ketika ketidakcocokan terjadi pada indeks *i* setelah mencocokkan sebuah *subpattern* sepanjang *m*, karakter setelah *subpattern* pada teks akan selalu dibandingkan dengan *pattern*. Sehingga untuk memperbesar pergeseran *pattern* terhadap teks, karakter *forward* ini ikut dicari bersama dengan *subpattern* pada *pattern*.

Forward Good Suffix Rule mengikuti aturan pada *Good Suffix Rule*, hanya saja bila pada *Good Suffix Rule* pergeseran *Delta[j]* dihitung berdasarkan jarak *i* dengan lokasi *subpattern* selanjutnya dari kanan pada *pattern* dimana karakter yang mendahului *subpattern* kedua ini tidak sama dengan karakter pada indeks *i pattern*, pada *Forward Good Suffix Rule* pergeseran *Delta[j]* dihitung berdasarkan jarak *i* dengan lokasi *substring* selanjutnya dari kanan pada *pattern* dimana karakter yang mendahului *substring* kedua ini tidak sama dengan karakter pada indeks *i pattern*. Pada kasus ini, *substring* yang dibandingkan adalah *subpattern* yang dilanjutkan dengan *forward* character (Domenico Cantone, Simone Faro, 2003b).

Gambar 2.7 adalah *pseudocode* untuk menyusun tabel *forward good suffix*, dan gambar 2.8 adalah *pseudocode* untuk pencarian utama algoritma *Forward Fast Search* seperti yang dilampirkan pada jurnal Domenico Cantoned dan Simone Faro (2003b).

```
precompute-forward-good-suffix (P)
Initialization:
    m = length (P)
    for i = 0 to m do
        for c ∈ Σ do
            gs→ [i, c] = m + 1
    for i = 0 to m – 1 do
        next[i] = i – 1
Computation:
    for slen = 0 to m – 1 do
        last = m – 1
        i = next [last]
        while i ≥ 0 do
            if gs→ [m – slen, P[i + 1]] > m – 1 – i then
                if (i – slen < 0 or (i – slen ≥ 0 and P [i – slen] ≠ P [m – 1 – slen])) then
                    gs→ [m – slen, P [i + 1]] = m – 1 – i
            if (i – slen ≥ 0 and P [i – slen] = P [last – slen]) or (i – slen < 0) then
                next [last] = i
                last = i
            i = next [i]
        if gs→ [m – slen, P[0]] < m then
            gs→[m – slen, P[0]] = m
        next [last] = -1
    return gs→
```

Gambar 2.7 *Pseudocode* penghitungan tabel *forward good suffix*

```
Forward-Fast-Search (P, T)
    n = length (T)
    m = length (P)
    T' = T . P [m - 1]^(m+1)
    bc = precompute-bad-character (P)
    gs→ = precompute-forward-good-suffix (P)
    s = 0
    while bc[T'[s + m – 1]] > 0 do
        s = s + bc[T'[s + m - 1]]
    while s ≤ n – m do
        j = m – 2
        while j ≥ 0 and P[j] = T'[s + j] do
            j = j – 1
        if j < 0 then
            print(s)
        s = s + gs→[j + 1, T[s + m]]
        while bcp [T'[s + m – 1]] > 0 do
            s = s + bc [T'[s + m - 1]]
```

Gambar 2.8 *Pseudocode* algoritma *Forward Fast Search*

2.6 Preprocessing

Metode *preprocessing* sering digunakan dalam pendeteksian plagiarisme, baik pada teks maupun kode program. Metode ini bertujuan untuk menghilangkan bagian yang tidak penting, seperti komentar pada source code, dan menyeragamkan bentuk dari teks atau kode agar lebih mudah dibandingkan.

Menurut Novanta sebagaimana dilampirkan pada laporan penelitian milik Natalia (2011: 22) metode *preprocessing* perlu diterapkan dalam aplikasi pendeteksi plagiarism karena "algoritma standar yang digunakan dalam aplikasi pendeteksi plagiarisme hanya membandingkan secara eksplisit dua *string* tanpa mengetahui sifat-sifat yang membentuk kedua *string* tersebut. Metode *preprocessing* tentu akan menambah waktu proses sistem secara menyeluruh, tapi dengan adanya pereduksian *noise* yang dilakukan oleh proses bantuan ini, diharapkan dapat mengurangi kompleksitas pada saat pembandingan oleh algoritma dasar yang digunakan untuk menghasilkan bobot persentase kecenderungan plagiarisme".

Maxim Mozgovoy (2007: 26-28) mengusulkan 2 metode untuk digunakan pada tahap *preprocessing* kode program, yaitu dengan proses *Tokenization* dan *Parameterized Matching*. *Tokenization* adalah proses mengubah sebuah kode program menjadi serangkaian *token* yang mewakili elemen-elemen dalam kode seperti variabel, fungsi, struktur kontrol, dan sebagainya. *Token* sendiri dapat diartikan sebagai satuan minimal yang dapat digunakan untuk merepresentasikan sebuah bahasa. Sebagai contoh sebuah kode [a = b + 45] dapat diwakilkan dengan *token* [<IDENTIFIER> = <IDENTIFIER> + <VALUE>]. Secara umum komentar juga dihilangkan pada proses *tokenization*. Kelemahan dari metode ini

adalah hilangnya struktur relasi antar variabel-variabel pada kode, sehingga dapat mengeliminasi kemungkinan adanya perbedaan diantara kedua kode yang dibandingkan.

Parameterized Matching membandingkan relasi dari 2 kode yang dibandingkan. Dua buah potongan kode dapat dianggap sebagai sama bila salah satu dari kedua kode dapat ditemukan pada kode lainnya melalui sejumlah subsitusi *identifier* pada kode tersebut dengan asumsi algoritma dapat membedakan *identifier* dengan elemen yang bukan merupakan *identifier*. Ide dasar dari *Parameterized Matching* adalah menghapus informasi mengenai nama variabel namun tetap mempertahankan relasi antar variabel tersebut. Metode *Tokenization* dan *Parameterized Matching* dapat digunakan bersamaan untuk mendapatkan hasil yang lebih akurat.

Pada penelitian yang sama, Maxim Mozgovoy berpendapat bahwa proses *preprocessing* pada teks dengan *natural language* lebih rumit dari proses *preprocessing* pada kode program dikarenakan oleh tidak adanya struktur formal pada teks dengan *natural language*.

2.7 Persentase Kecenderungan Plagiarisme

Persentase kecenderungan plagiarisme merupakan nilai yang menunjukkan seberapa besar kemungkinan plagiarisme dapat terjadi. Berdasarkan Warsito pada laporan penelitian milik Regina Natalia (2011), persentase kecenderungan plagiarisme dapat diperhitungkan dengan teknik statistika sederhana, yaitu:

$$\text{Persentase Berdasarkan Karakter} = \frac{\text{Jumlah Pattern yang sama} * 100\%}{\text{nPattern}} \quad \text{..... Rumus 2.1}$$

**nPattern = Jumlah string yang dibandingkan pada file pattern*

2.8 Sistem Operasi

Definisi dari sistem operasi menurut Creech pada jurnal ilmiah milik Brian Randell (1971) adalah bagian dari sistem komputer yang mengupayakan untuk mengalokasikan dan mengkoordinasikan *resource* dari sistem untuk mencapai performa yang optimum dari sistem tersebut. Lebih lanjut lagi, Creech menjelaskan bahwa *resource* yang terkait dengan sistem operasi meliputi prosesor, perangkat *I/O*, fasilitas sistem operasi, *memory*, dan waktu. Selanjutnya Creech juga menyatakan bahwa pekerjaan dari sistem operasi dipersulit dengan kenyataan bahwa sistem operasi itu sendiri juga menggunakan *resource*.

Saat ini sudah banyak sistem operasi yang beredar. Diantaranya adalah sistem operasi UNIX atau sistem operasi menyerupai UNIX seperti GNU/Linux dan OS X, sistem operasi berbasis Windows, Mac OS, dan lain sebagainya. Berdasarkan data yang sudah terverifikasi pada netmarketshare sampai dengan Desember 2013, sistem operasi berbasis Windows menempati posisi teratas dengan Windows 7 pada posisi teratas dan Windows XP pada posisi kedua.

Selain sistem operasi berbasis Windows, banyak pula dikenal sistem operasi berbasis Linux. Sistem operasi Linux merupakan sistem operasi yang bersifat *open-source* dan memiliki banyak variasi, diantaranya adalah Debian, Ubuntu, dan Android. Berdasarkan *voting* yang dilakukan Lifehacker sebagaimana dicantumkan pada artikel yang ditulis Katherine Noyes untuk PCWorld (2012), Ubuntu merupakan sistem operasi berbasis Linux terpopuler diikuti oleh Linux Mint, Arch Linux, Debian, dan kemudian Fedora.