Hak cipta dan penggunaan kembali:

Lisensi ini mengizinkan setiap orang untuk menggubah, memperbaiki, dan membuat ciptaan turunan bukan untuk kepentingan komersial, selama anda mencantumkan nama penulis dan melisensikan ciptaan turunan dengan syarat yang serupa dengan ciptaan asli.

Copyright and reuse:

This license lets you remix, tweak, and build upon work non-commercially, as long as you credit the origin creator and license it on your new creations under the identical terms.

Team project ©2017
Dony Pratidana S. Hum | Bima Agus Setyawan S. IIP

BAB II

TINJAUAN PUSTAKA

2.1 Plagiarisme

Plagiarisme atau plagiat adalah penjiplakan atau pengambilan karangan, pendapat, dan sebagainya dari orang lain dan menjadikannya seolah karang dan pendapat sendiri (Kamus Besar Bahasa Indonesia, 1997).

Menurut Felicia Utorodewo dkk (Utorodewo, 2007), hal-hal berikut dapat digolongkan sebagai tindakan plagiarisme :

1. Mengakui tulisan orang lain sebagai tulisan sendiri,
2. Mengakui gagasan orang lain sebagai pemikiran sendiri,
3. Mengakui temuan orang lain sebagai kepunyaan sendiri,
4. Mengakui karya kelompok sebagai kepunyaan atau hasil sendiri,
5. Menyajikan tulisan yang sama dalam kesempatan yang berbeda tanpa menyebutkan asal-usulnya,
6. Meringkas dan memparafrasekan (mengutip tak langsung) tanpa menyebutkan sumbernya, dan
7. Meringkas dan memparafrasekan dengan menyebut sumbernya, tetapi rangkaian kalimat dan pilihan katanya masih terlalu sama dengan sumbernya.

Sedangkan secara spesifik dalam konteks program, Goenawan, dkk (2005, p. 2) menggolongkan praktik plagiat menjadi dua yaitu Plagiarisme Leksikal, perubahan pada kode (*source code*) program dan Plagiarisme Struktural, perubahan struktur program.

Menurut Djuric dan Gasevic (Djuric & Gasevic, 2013) contoh dari plagiarisme leksikal adalah

1. Modifikasi format kode program,
2. Penambahan, modifikasi, atau penghapusan komentar,
3. Penerjemahan bahasa,
4. Modifikasi keluaran program,
5. Penggantian nama *identifier,*
6. Pemecahan atau penggabungan deklarasi variabel,
7. Penambahan, modifikasi, atau penghapusan *modifier,* dan
8. Modifikasi nilai konstan.

Dan contoh plagiarisme struktural adalah

1. Mengubah urutan variabel dalam *statement,*
2. Mengubah urutan *statement* dalam *code blocks,*
3. Menyusun ulang *code blocks,*
4. Penambahan yang berlebihan pada *statement* atau variabel,
5. Modifikasi struktur kontrol,
6. Mengubah tipe data dan modifikasi struktur data,
7. *Inlining* dan *refactoring method,*
8. *Redundancy*
9. Variabel dan subekspresi sementara, dan
10. Modifikasi *scope.*

2.1.1 Deteksi Plagiarisme

Pendeteksian plagiarisme biasanya merupakan perbandingan antara dua dokumen atau lebih untuk menentukan derajat persamaannya, yang diperlukan untk menetapkan sebuah nilai numerik sebagai skala persamaan masing-masing dokumen (Lukashenko, Graudina, & Grundspenkis, 2007).

Mozgovoy (2007, p. 17) mengelompokkan teknik pendeteksian plagiarisme menjadi tiga kelompok, yaitu.

1. *Fingerprint-based*

   Dalam pendekatan *fingerprint-based,* ide dasarnya adalah membuat sebuah "sidik jari" atau ciri khas dari setiap dokumen. Tiap sidik jari tersebut memuat beberapa atribut numerik yang merefleksikan struktur dokumen tersebut, misalnya jumlah baris, atau jumlah kata yang unik, dll. Jika terdapat terdapat dua sidik jari yang sama maka dapat dikatakan kedua dokumen tersebut serupa.

2. *String Matching-based*

   Metode *string matching* membandingkan dokumen sebagai *string.* Pendekatan ini biasanya tidak memperhitungkan struktur hirarki dari dokumen, dan menganggapnya sebagai data mentah.

3. *Parse Tree Mathcing-based*

   Metode ini mengurai isi dokumen berdasarkan bagian-bagian natural dari dokumen tersebut. Misalnya pada teks karangan, dibagi kedalam paragraf, kalimat, kata, dll. sedangkan kode program dibagi kedalam *class, function, control structures,* dll.

Dari berbagai teknik atau metode yang ada dalam mendeteksi plagiarisme, Kusmawan, dkk (2009, p. 2) menyebutkan tiga kebutuhan mendasar yang harus dipenuhi, yaitu.

1. *Whitespace Insensitivity* yang berarti dalam melakukaan pencocokan terhadap file teks seharusnya tidak terpengaruh oleh spasi, jenis huruf (kapital atau normal), tanda baca dan sebagainya.
2. *Noise Surpression* yang berarti menghindari penemuan kecocokan dengan panjang kata yang terlalu kecil atau kurang relevan. Setiap penemuan kata harus cukup besar untuk memberitahukan bahwa kata ini telah disalin dan bukan kata umum.
3. *Position Independence* yang berarti penemuan kecocokan atau kesamaan harus tidak bergantung pada posisi kata-kata Meskipun posisinya tidak sama, kecocokan harus dapat ditemukan.

2.2 Bahasa Pemrograman

Bahasa Pemrograman adalah cara penulisan untuk membuat program, yang secara spesifik adalah komputasi dan algoritma (Aaby, 1996). Bahasa pemrograman digunakan untuk mengimplementasikan algoritma menjadi sebuah program dengan intruksi-intruksi yang dapat dimengerti komputer untuk mengerjakan suatu fungsi tertentu. Dalam bahasa pemrograman dikenal juga pemrograman tersturktur dan pemrograman beroritentasi objek.

Menurut Bond (2000, pp. 1-3) Pemrograman terstruktur disebut juga *Top Down Design,* merupakan metode pemrograman yang memecah sebuah masalah menjadi beberapa masalah yang lebih kecil yang dapat diselesaikan. Contoh

bahasa pemrograman terstruktur adalah bahasa pemrograman C. Sedangkan pemrograman berorientasi objek merupakan pendekatan yang mengidentifikasi objek dan *message* dalam suatu masalah. Objek merupakan entitas mandiri yang memiliki *internal state* (data yang dimiliki) dan bisa merespond kepada *message*(sebuah pemanggilan ke fungsi yang dimiliki objek). Solusi yang dihasilkan adalah objek-objek yang masing-masing memiliki data dan fungsinya, dan tiap objek mengirim *message* untuk berinteraksi. Contoh bahasa pemrograman berorientasi objek adalah C++, Java, dan C#.

Dalam pembuatan aplikasi pendeteksi plagiarisme ini, aplikasi melakukan pendeteksian plagiarisme pada kode program berbahasa C, bahasa yang mudah dipahami sehingga biasa digunakan pembelajaran awal. Aplikasi ini sendiri diimplementasikan dengan bahasa permrograman C#.

2.3 Algoritma

Kamus Besar Bahasa Indonesia (KBBI) mendefinisikan algoritma sebagai urutan logis pengambilan keputusan untuk pemecahan masalah (Kamus Besar Bahasa Indonesia, 1997). Algoritma juga dapat dilihat sebagai alat untuk memecahkan masalah komputasi yang terperinci. Masalah dispesifikasikan secara umum sebagai hubungan antara masukan dan keluaran yang diinginkan, dan algortima mendeskripsikan prosedur komputasi yang spesifik untuk mencapai hubungan antara masukan dan keluaran tersebut (Cormen, Leiserson, & Rivest, 2001).

Dalam program yang melibatkan *text-editing,* termasuk aplikasi pendeteksi plagiarisme, masalah yang muncul adalah menemukan semua kemunculan sebuah

pola dalam sebuah teks (Cormen, Leiserson, & Rivest, 2001). Untuk memecahkan masalah tersebut diperlukan algoritma *string-matching* yang efisien, berikut dua algoritma *string-matching* yang digunakan dalam penelitian ini.

2.3.1 Algoritma Boyer Moore

Dalam mendeteksi plagiarisme leksikal, yaitu dengan pencocokan berdasarkan karakter, aplikasi pendeteksi plagiarisme ini menggunakan Algoritma Boyer Moore yang telah dibuktikan dalam penelitian sebelumnya. Algoritma Boyer-Moore termasuk algoritma yang cukup efisien dalam pencarian kata. Keefisienan yang algoritma ini punya berasal dari fakta bahwa setiap pencocokan yang gagal antara teks dan kata yang dicari, algoritma ini menggunakan informasi yang didapat dari proses awal untuk melewati karakter-karakter yang tidak cocok (Atmopawiro, 2006).

Algoritma Boyer-Moore dikenal sebagai algoritma yang paling efisien dalam berbagai aplikasi. Versi yang disederhanakan dari algoritma ini sering diimplementasikan dalam teks editor untuk perintah “*search*“ dan “*substitute*“ (Atmopawiro, 2006). Satu hal yang menarik pada algoritma Boyer-Moore ini adalah perbandingan karakter yang dicari terhadap string dilakukan dari belakang ke depan tetapi dengan pergeseran *window* tetap dari kiri ke kanan (Febriyanto, Zulfianto, & Wirotomo, 2006, p. 1). Penjelasan cara kerja algoritma Boyer-Moore dapat lebih mudah dijelaskan jika pengecekan menghasilkan ketidakcocokan.

Misalnya dengan pola string *T* dibawah ini kata yang dicari *P* adalah "MAKALAH", dengan tanda "|" adalah posisi pengecekan

```
    012345678901234567
T = ITUMERUPAKANALASAN
          |
P = MAKALAH
```

Pada pengecekan pertama, "U" pada teks tidak muncul sama sekali pada "MAKALAH", dan berarti tidak akan ada kecocokan pada pengecekan. Sehingga hanya dengan sekali pengecekan pada karakter ke-8, pengecekan karakter ke-1 sampai ke-7 tidak perlu dilakukan, dan dilakukan pergeseran sesuai dengan jamlah karakter *P*. Setelah dilakukan pergeseran sebanyak tujuh karakter, posisi nya menjadi

```
    012345678901234567
T = ITUMERUPAKANALASAN
                 |
P =        MAKALAH
```

Pengecekan kedua membandingkan "L" dengan "H" menghasilkan ketidakcocokan, tetapi "L" terdapat pada "MAKALAH" dengan jarak tiga karakter dari "H" sehingga *P* akan digeser sebanyak tiga karakter. Untuk mengetahui jumlah pergeseran yang dilakukan, dibuat tabel pada awal pengecekan, yang berisi banyaknya 'loncatan' karakter yang akan dilakukan setelah mendapati sebuah perbandingan yang tidak cocok. Sementara karakter yang tidak ditemukan pada *P* diisi dengan nol (Febriyanto, Zulfianto, & Wirotomo, 2006, p. 2). Untuk contoh diatas tabel yang dibuat dapat dilihat pada tabel 2.1.

Tabel 2.1 Tabel contoh pergeseran algoritma Boyer Moore
Sumber : (Febriyanto, Zulfianto, & Wirotomo, 2006, p. 2)

A	…	H	…	K	…	L	…	M
1	0	7	0	4	0	2	0	7

Namun dengan menggunakan tabel *bad character* tersebut saja tidak cukup, pada kasus berikut

```
    012345678901234567
T = ITUMERUPAKAKAHASAN
               |
P =       MAKALAH
```

Pergeseran yang dilakukan adalah dua karakter untuk mensejajarkan "K" pada *T* dan "K" pada *P*, padahal seharusnya dilakukan pergeseran penuh karena "AH" tidak muncul lagi di *P*. Sedangkan pada kasus berikut

```
    012345678901234567
T = ITUMERUPAKAHALASAN
           |
P =   MAKALAKA
```

Terdapat *substring* "AKA" pada P yang berulang pada index satu dan lima. Untuk mengoptimalkan pergesaran dengan pola yang memiliki *substring* berulang dibutuhkan perhitungan tabel *suffix* dan *good-suffix*.

Dalam algoritma Boyer Moore langkah pertama adalah menghitung Tabel pergeseran *bad character*. Tabel *bad character* menghitung pergeseran optimal pada pola pencarian yang tidak mengalami pengulangan pada *substring* nya. Cara penghitungan tabel ini dijelaskan pada *pseudocode* gambar 2.1.

```
function BadCharacter (input P:string, m:integer) →
integer[0..n-1]

/* Mengisi array of integer(tabel) dengan panjang
karakter s(m) bila tidak ada pada s, atau dengan posisi
karakter bila ada pada s  */

Deklarasi
  c : integer
  last : integer[0..n-1]

Algoritma:
  for c ← 0 to n-1
    last[c] ← 0
  endfor
  for c ← 0 to m-1
    last[P[c]] ← c
  endfor
  return last
```

Gambar 2.1 *Pseudocode* perhitungan tabel *bad character*
sumber : http://www.cs.cornell.edu/Courses/cs312/2002sp/lectures/lec25.htm

Selanjutnya dihitung Tabel *suffixes* yang digunakan untuk menentukan index dari pengulangan substring bila terdapat pengulangan susunan kata pada *P*.

```
function Suffixes (input P:string, m:integer) →
integer[0..m-1]
/* Mengisi array of integer(tabel) untuk mengetahui
adanya pengulangan suffix  */
Deklarasi
  j : integer
  i : integer
  prefix : integer[0..m-1]
Algoritma:
  prefix [1] ← 0
  j ← 0
  for i ← 1 to m - 1
  while j > 0 && P[j] != P[i]
    j = prefix[j]
  endwhile
  if P[j] = P[i]
    j = j + 1
  prefix[i+1] = j
  endfor
  return prefix
```

Gambar 2.2 *Pseudocode* perhitungan tabel *suffixes*
sumber : http://www.cs.cornell.edu/Courses/cs312/2002sp/lectures/lec25.htm

Gambar 2.2 menunjukkan perhiungan Tabel *suffixes*. Kemudian berdasarkan tabel *suffixes* tersebut dilakukan perhitungan tabel *good-suffix*. Tabel *good-suffix* digunakan untuk mengoptimalisasikan pergeseran bila terjadi

pengulangan *substring* pada *S.* Gambar 2.3 menunjukkan *pseudocode* perhitungan tabel *good-suffix.*

```
function good_suffix (input S:string, m:integer) →
integer[0..m-1]

/* Mengisi array of integer(tabel) untuk menentukan
pergeseran bila terdapat suffix  */

Deklarasi
  j : integer
  prefix : integer[0..m-1]
  good_suffix : integer[0..m-1]

Algoritma:
  prefix ← Suffixes (S, m)

  for j ← 0 to m - 1
        good_suffix [j] ← m - prefix[m]
  endfor
  P' = reverse(P)
  prefix' = Suffixes(P')
  for j ← 1 to m
    j = m - prefix'[j'] - 1
    good_suffix[j] = min(good_suffix, j' - prefix'[j']
  endfor
  return GS
```

Gambar 2.3 *Pseudocode* perhitungan tabel *good-suffix*
sumber : http://www.cs.cornell.edu/Courses/cs312/2002sp/lectures/lec25.htm

Ketiga tabel yang telah dihitung tersebut digunakan untuk menentukan pergeseran karakter secara optimal dalam algoritma Boyer Moore. Besar pergesran karakter yang dilakukan adalah nilai terbesar antara tabel *bad character* dan tabel *good-suffix.* Gambar 2.4 menunjukkan *pesudocode* dari keseluruhan algoritma Boyer Moore.

```
function BoyerMoore (input P:string, m:integer,
T:string, o:integer) → boolean

/* Mencari string S pada Teks T dan menghasilkan true
bila ketemu dan false bila tidak ditemukan*/

Deklarasi
  s : integer
  j : integer
  good_suffix : integer[0..m-1]
  last : integer[0..n-1]
  ketemu : boolean[0..n-1]

Algoritma:
  /* pengisian tabel */
  last ← BadCaharacter (S, m)
  good_suffix ← good_suffix (S, m)

  s ← 0
  while s <= n – m
       j ← m – 1
       while j >= 0 && P[j] == T[s+j]
           if i < 0
           ketemu ← true
       endwhile
       s = s + max(good_suffix[j], j – last[T[s+j]] + 1
  endwhile
  return ketemu
```

Gambar 2.4 *Pseudocode* algoritma Boyer Moore
sumber : http://www.cs.cornell.edu/Courses/cs312/2002sp/lectures/lec25.htm

2.3.2 Algoritma Smith-Waterman

Dan sebagai fokus penelitian ini, pendeteksian plagiarisme struktural mengimplementasikan Algoritma Smith-Waterman pada pencocokan berdasarkan baris. Algoritma ini pertama kali dicetuskan oleh Temple F. Smith dan Michael S. Waterman pada 1981 (Smith & Waterman, 1981). Algoritma Smith-Waterman adalah algoritma yang biasa digunakan dalam bioteknologi, yaitu untuk melakukan penyelarasan urutan lokal. Penyelarasan dilakukan untuk menentukan daerah yang sama antara dua sekuens nukleotida atau protein. Daripada melihat urutan total, algoritma Smith-Waterman membandingkan segmen dari semua panjang yang mungkin dan mengoptimalkan ukuran kemiripan. Berdasarkan fungsi proses penyejajaran sekuens tersebut, maka algoritma ini dapat

dikonversikan ke dalam pemrograman komputer untuk digunakan membantu proses pendeteksian dokumen teks yang dianggap cenderung plagiat dengan cara melihat kesamaan isi (*local similarities*) dari beberapa dokumen teks (Novanta, 2009). Penjelasan cara kerja algoritma Smith-Waterman dapat lebih mudah dijelaskan dengan penyejajaran dua buah sekuens protein.

Misalnya dengan dua sekuens T dan S dimana T sebagai Teks dan S sebagai *string* yang dicari

```
T = c g g g t a t c c a a
S = c c c t a g g t c c c a
```

Proses penyejajaran yang dilakukan akan menghasilkan dua string yang sudah disejajarkan yaitu

```
T = c g g g t a - - t - c c a a
S = c c c - t a g g t c c c - a
```

Jika dilihat berdasarkan T sebagai teks asli, perubahan yang terjadi pada S adalah penghilangan (*deletion*) ditandai dengan warna hijau dan penyisipan (*insertion*) ditandai dengan warna merah. Kesamaan kedua string tersebut dapat dinilai dengan menggunakan nilai positif untuk setiap kesamaan, *deletion,* dan *insertion* yang dideteksi. Namun selain dua *string* diatas terdapat juga kemungkinan seperti dua *string* berikut.

```
T = c - g g g t a - - t c c a a
S = c c - - c t a g g t c c c a
```

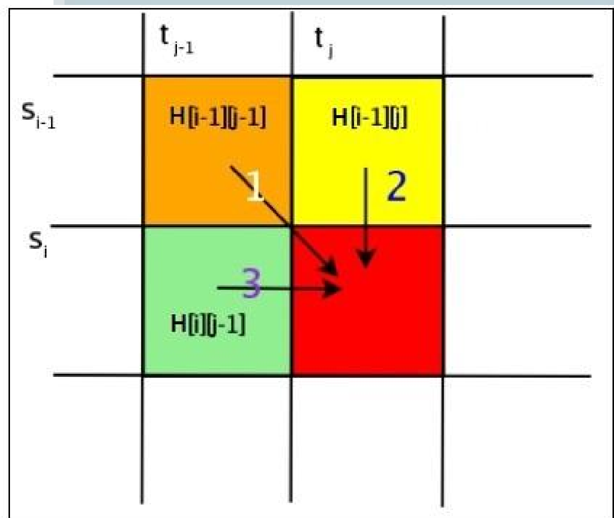
Kedua hasil tersebut merupakan hasil yang benar dalam penyejajaran T dan S. Untuk menentukan hasil mana yang optimal digunakanlah algoritma Smith-Waterman yang akan menghasilkan *scoring matrix* berdasarkan kesamaan,

deletion, dan *insertion* pada sekuens. Nilai tiap cell pada *scoring matrix* tersebut dihitung berdasarkan aturan pada gambar 2.5.

$$H(i,j) = \max \begin{Bmatrix} 0 & \\ H(i-1,j-1) + w(a_i,b_j) & \text{Match/Mismatch} \\ H(i-1,j) + w(a_i,-) & \text{Deletion} \\ H(i,j-1) + w(-,b_j) & \text{Insertion} \end{Bmatrix}, 1 \leq i \leq m, 1 \leq j \leq n$$

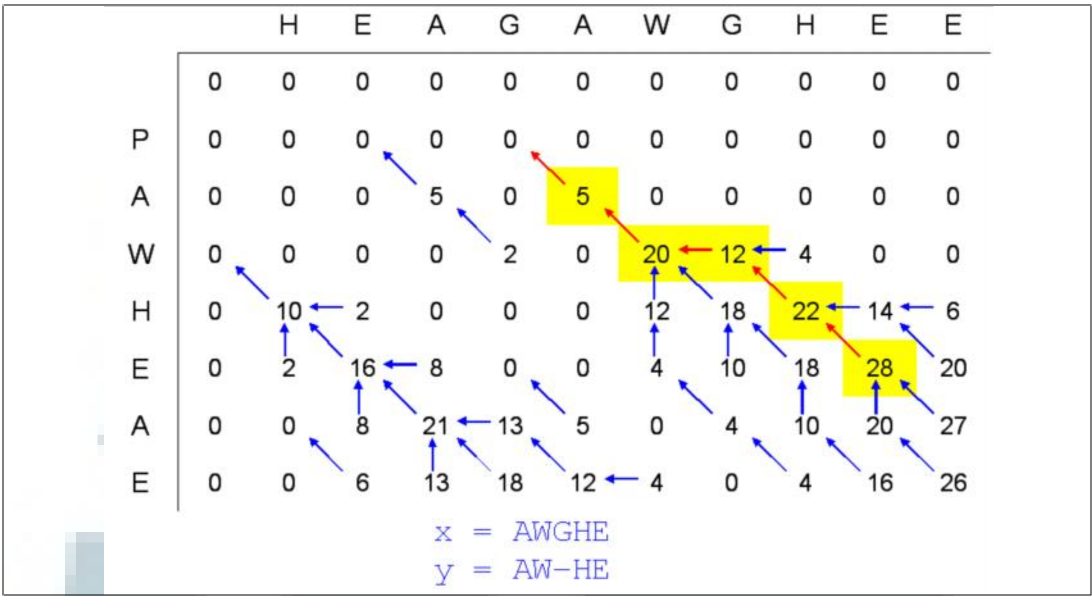
Gambar 2.5 *Matrix* Algoritma Smith-Waterman
Sumber : https://code.google.com/p/internal-external-embedded-dedicated-accelerator/wiki/SmithWaterman

Dengan $w(a_i, b_j)$ merupakan nilai positif bila terjadi kesamaan, dan negatif bila tidak cocok. Dan $w(a_{i,}-)$ dan $w(-, b_j)$ adalah nilai pinalti untuk *deletion* dan *insertion.* Dengan rumus tersebut setiap cell pada matrix membawa hasil *max* antara tiga cell sebelumnya, ditunjukkan pada gambar 2.5.



Gambar 2.6 Cell *scoring matrix*
Sumber : (Ayguade, Navarro, & Gonzalez, 2007, p. 15)

Kemudian, gambar 2.6 berikut merupakan contoh hasil pengisian sebuah *scoring matrix* dengan *string* “HEAGAWGHEE” dan “PAWHEAE”, dengan hasil



Gambar 2.7 *Scoring matrix* algoritma Smith-Waterman
Sumber : (Ayguade, Navarro, & Gonzalez, 2007, p. 18)

Setelah *scoring matrix* dibuat, penyejajaran yang paling optimal adalah sel dengan nilai terbesar. Setelah itu dilakukan *traceback* untuk merekonstruksi penyejajaran yang optimal.

Proses *Traceback* dilakukan dari nilai terbesar pada *matrix.* Berdasarkan nilai-nilai yang sudah dihitung sebelumnya, perpindahan diagonal berarti dilakukan penyejajaran (baik *match* ataupun *mismatch*) Perpindahan atas-bawah berarti terjadi *deletion,* dan kiri-kanan berarti terjadi *insertion.* Hasil rekonstruksi pada contoh di atas adalah

```
H E A G A W G H E E
- - - P A W - H E -
```

Dengan hasil penyejajarannya adalah empat *match,* lima *insertion* dan satu *mismatch*. *Pseudocode* algoritma Smith-Waterman ditunjukkan pada gambar 2.7.

```
function SmithWaterman (input X:string, n:integer,
Y:string, m:integer) → String[0..n-1]
/* Mencari penyejajaran optimal dari dua buah string */
Deklarasi
  i,j : integer
  F : integer[0..n-1][0..m-1]
  result1 : integer
  result2 : integer
  result3 : integer
  nilai : integer
  hasil : string[0..n-1]
  maxF, maxI, maxJ : integer
Algoritma:
  for i ← 0 to n
       F[i,0] = 0
  endfor
  for j ← 0 to m
       F[0,j] = 0
  endfor
  maxF ← 0
  for i ← 0 to n
    for j ← 0 to m
       if T[i] = S[j]
         nilai ← 1
       else
         nilai ← 0
       result1 ← F[i-1,j-1] + nilai
       result2 ← F[i-1,j]
       result3 ← F[i,j-1]
       F[i,j] ← MAX(0, result1, result2, result3)
       If maxF < F[i,j]
         maxF ← F[i,j]
         maxI ← i
         maxJ ← j
    endfor
  endfor
  i ← maxI
  j ← maxJ
  while F[i,j] > 0
    if F[i,j] = F[i-1,j-1]
       hasil[0] ← hasil[0] + T[i-1]
       hasil[1] ← hasil[1] + S[j-1]
       i ← i - 1
       j ← j - 1
    else if F[i,j] = F[i-1,j]
       hasil[0] ← hasil[0] + T[i-1]
       hasil[1] ← hasil[1] + "-"
       i ← i - 1
    else
       hasil[0] ← hasil[0] + "-"
       hasil[1] ← hasil[1] + S[j-1]
       j ← j - 1
  endwhile
  return ketemu
```

Gambar 2.8 *Pseudocode* algoritma Smith-Waterman
sumber : (Huson, 2008, p. 35)

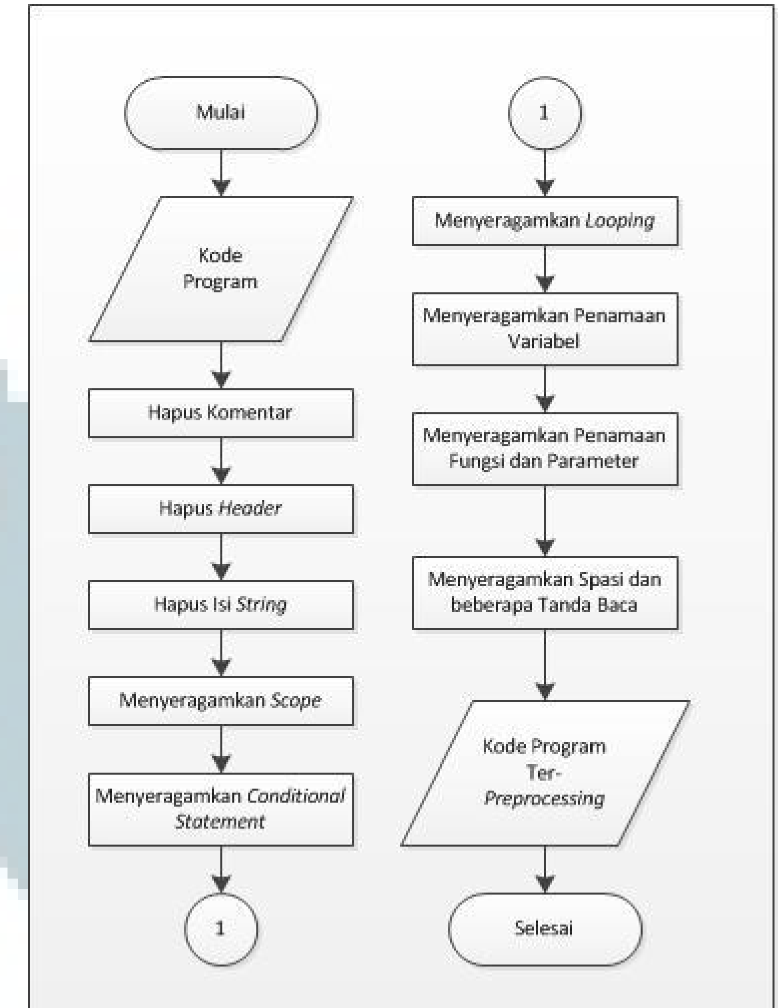
2.4 Preprocessing

Metode *preprocessing* digunakan karena algoritma standar yang digunakan dalam aplikasi pendeteksi plagiarisme hanya membandingkan secara eksplisit dua *string* tanpa mengetahui sifat-sifat yang membentuk kedua *string* tersebut. Oleh karena itu, dibutuhkan proses bantuan (*preprocessing*) dalam bentuk modul-modul tambahan, dimana modul-modul tersebut tersebar di dalam proses pembobotan sesuai fungsi masing-masing modul. (Novanta, 2009)

Metode *preprocessing* tentu akan menambah waktu proses sistem secara menyeluruh, tetapi dengan adanya pengurangan *noise* yang dilakukan proses ini, diharapkan dapat mengurangi kompleksitas pada saat perbandingan oleh algoritma pembanding yang digunakan untuk menghasilkan persentase kemiripan (Arifianto, 2011, p. 22).

Teknik *preprocessing* akan mengubah data mentah (*file* kode program awal) menjadi sebuah data yang siap dibandingkan sebagai *string* biasa. Teknik ini sangat berguna dalam pendeteksian plagiarisme kode program. Hal ini sebabkan karena dalam proses plagiarisme kode program ada beberapa bagian yang dapat diutbah tanpa mengubah makna program secara keseluruhan (Natalia, 2011).

Langkah-langkah *preprocessing* yang dilakukan di aplikasi ini dijelaskan dengan *flowchart* berikut.



Gambar 2.9 *Flowchart* Preprocessing
sumber : (Natalia, 2011, pp. 28-29)

Proses – proses yang digambarkan pada *flowchart* gambar 28 adalah sebagai berikut.

1. Menghapus seluruh komentar yang terdapat di dalam kode program, yaitu bagian yang diawali dengan tanda "//" maupun di antara tanda "/*" dan "*/",
2. Menghapus seluruh *header* dalam kode program, yaitu bagian yang diawali dengan tanda "#",

3. Menghapus seluruh isi *string*, yaitu teks yang berada diantara dua tanda petik dua ("…"),
4. Menyeragamkan *Scope* dengan menambahkan karakter '{' dan '}' pada setiap bentuk *if, while,* dan *do…while* satu baris,
5. Menyeragamkan/menukar posisi *conditional statement* lebih dari (>) dan lebih dari atau sama dengan (>=) menjadi bentuk kurang dari (<) dan kurang dari atau sama dengan (<=),
6. Menyeragamkan *Looping* dengan mengubah seluruh bentuk *for* menjadi bentuk *while,*
7. Menyeragamkan penamaan pada deklarasi variabel berdasarkan tipe data yang digunakan,
8. Menyeragamkan penamaan fungsi dan parameter fungsi berdasarakan tipe data parameter yang digunakan,
9. Menyeragamkan seluruh spasi, beberapa tanda baca dan simbol, serta memformat struktur kode program.

2.5 Metode Perancangan

Dalam perancangan sebuah aplikasi dikenal beberapa metode perancangan misalnya perancangan terstruktur dan perencangan berorientasi objek. Dalam aplikasi ini metode yang digunakan adalah perancangan terstruktur atau *Flowchart*, yaitu konsep perancangan dengan penekakan pada metode *Top Down Design* atau pemrograman terstruktur (Hafsah, Kaswidjanti, & Cili, 2010). Metode *Top Down Design* akan dibahas dalam subbab berikutnya. *Flowchart* atau diagram alir menggambarkan langkah-langkah sebuah aplikasi dalam sebuah

diagaram dengan simbol-simbol yang mempresentasikan proses-proses dalam aplikasi. Tabel 2.2 berisi simbol-simbol yang biasa digunakan dalam *Flowchart.*

Tabel 2.2 Simbol-Simbol *Flowchart*
Sumber : http://www.breezetree.com/articles/flow-chart-symbols.htm

	Terminasi	Menunjukkan awal atau akhir sebuah flowchart
	Proses	Menunjukkan kegiatan yang dilakukan
	Keputusan	Proses / Langkah dimana perlu adanya keputusan atau kondisi tertentu. Selalu ada dua keluaran untuk kondisi yang berbeda.
	Data	Menunjukkan *input* atau *output* suatu data
	Dokumen	Menunjukkan dokumen secara fisik atau *file* komputer
	Garus alir	Menunjukkan arah aliran proses