BAB 2
TINJAUAN PUSTAKA

2.1 Rumah Baca Buku Sunda jeung Sajabana

Rumah Baca Buku Sunda Jeung Sajabana adalah perpustakaan yang pada dasarnya berisi buku-buku Sunda. Perpustakaan ini didirikan pada tahun 2004, dan berada di salah satu perumahan kota Bandung, yaitu di Perumahan Margawangi JL. Margawangi VII No.5 Margacinta Bandung.

Perpustakaan ini terbuka untuk orang-orang yang berminat, tanpa dipungut biaya sedikit pun. Namun demikian, pengunjung hanya boleh membaca di tempat, tanpa bisa dibawa pulang. Dengan kata lain, semua bacaan yang ada di tempat ini tidak untuk dipinjamkan.

Setelah memasuki tempat ini, pengunjung akan menemukan banyak bacaan di luar Bahasa Sunda. Oleh karena itu, perpustakaan ini diberi nama Rumah Baca Buku Sunda Jeung Sajabana, "*Jeung Sajabana*" memiliki arti "dan sebagainya" yang merujuk pada bacaan-bacaan di luar Bahasa Sunda.

Bacaan yang ada di tempat ini, tidak sebatas buku terbitan tertentu, tetapi juga kumpulan cerita, serta artikel bahasa Sunda yang menarik dalam bentuk kliping.

2.2 *String*

Dalam ilmu komputer, *string* dapat diartikan sebagai sekuens dari karakter. Selain sebagai data abstrak yang menyimpan sekuens nilai data, *string* juga dapat berupa *bytes* yang merupakan elemen pembentuk karakter, sesuai dengan *encoding* karakter yang disepakati ASCII ataupun EBCDIC [9].

2.3 *String Matching*

Dalam Bahasa Indonesia *string matching* dikenal dengan istilah 'pencocokan *string'* atau 'pencocokan kata'. Berikut adalah perumusan kasus pencocokan kata (*String Matching*).

Diasumsikan teks adalah sebuah susunan *T[1..n]* dengan panjang *n* dan memiliki susunan *P[1..m]* dengan panjang *m*. Lalu, elemen P dan T merupakan karakter-karakter yang digambarkan dari ∑ abjad terbatas. Contoh, diketahui ∑ = {0,1} atau ∑ = {a,b,…,z}. Sedangkan karakter susunan P and T dapat didefinisikan sebagai kata (*string*) pada banyak karakter [9].

Pola yang dimaksud adalah dimana *P* muncul dengan *shift s* dalam teks *T* (atau , ekuivalen , pola *P* muncul pada permulaan di posisi *s + 1* pada teks *T*) Jika $0 \leq s \leq n - m$ dan $T[s+1..s+m] = P[1..m]$ (itu adalah, jika $T[s + j] = P[j], for\ 1 \leq j \leq m$). Jika *P* muncul dengan *shift s* di *T*, maka disebut *s* sebuah *shift* yang sah. Kasus pencocokan kata (*String Macthing*) merupakan masalah dari menemukan semua *shift* yang sah dengan pola pemberian *P* yang muncul di teks pemberian *T*. Kecocokan adalah apabila karakter pada teks *T* dan karakter pada pola *P* yang dibandingkan adalah sama. Mengacu pada gambar di bawah, bahwa pola *P* melakukan pergeseran dalam teks *T*. Masalah pencocokan *string* adalah masalah menemukan semua pergeseran yang valid dengan diberikan pola *P* yang terjadi pada teks *T*.



Gambar 2.1 Kasus *String Matching*

2.4 Algoritma

Algoritma adalah urutan langkah-langkah untuk memecahkan suatu masalah[1]. Terdapat beragam definisi dari para ahli mengenai algoritma, tetapi pada intinya prinsip algoritma adalah proses pemecahan suatu masalah. Berikut adalah definisi algoritma yang dikutip dari salah satu literatur.

Menurut Goodman Hadet Niemi, algoritma adalah urutan-urutan terbatas dari operasi-operasi yang terdefinisi dengan baik, yang masing-masing membutuhkan memori dan waktu yang terbatas untuk menyelesaikan masalah[1]. Langkah-langkah dalam algoritma harus logis dan harus dapat ditentukan bernilai

salah atau benar. Selain itu, dalam beberapa konteks, algoritma dapat diartikan sebagai spesifikasi urutan langkah untuk melakukan pekerjaan tertentu.

Terdapat beberapa pertimbangan dalam pemilihan algoritma[1]. Pertama, algoritma itu harus benar, artinya algoritma akan memberikan keluaran yang dikehendaki dari sejumlah masukan yang diberikan. Pertimbangan kedua adalah efektif, yaitu harus mengetahui seberapa baik hasil yang dicapai oleh algoritma tersebut. Hal tersebut dianggap sangat penting terutama pada algoritma untuk menyelesaikan masalah yang memerlukan aproksimasi hasil (hasil yang hanya berupa pendekatan). Pertimbangan ketiga yaitu efisiensi algoritma. Untuk menilai efisiensi algoritma dapat ditinjau dari 2 hal, yaitu efisiensi waktu dan memori. Meskipun algoritma memberikan keluaran yang benar atau paling mendekati, tetapi jika harus menunggu berjam-jam untuk mendapatkan keluaran dari algoritma tersebut, biasanya algoritma seperti itu tidak akan dipakai karena setiap orang menginginkan keluaran yang cepat. Sama halnya dengan memori, semakin besar memori yang terpakai maka semakin buruklah algoritma tersebut.

2.4.1 Ciri dan Sifat Algoritma

Setiap penggunaan algoritma memiliki ciri dan sifat. Menurut Donald E. Knuth dalam bukunya *The Art of Computer Programming*, algoritma harus mempunya 5 ciri penting [9], yaitu sebagai berikut:

1. Algoritma mempunyai awal dan akhir, artinya suatu algoritma harus berhenti setelah mengerjakan serangkaian tugas. Dengan kata lain, suatu algortima memiliki langkah yang terbatas.
2. Setiap langkah harus didefinisikan dengan tepat sehingga tidak memiliki arti ganda (*not ambiguous*).
3. Memiliki masukan (*input*) atau kondisi awal.
4. Memiliki keluaran (*output*) atau kondisi akhir.
5. Algoritma harus efektif atau benar-benar menyelesaikan persoalan.

Berdasarkan ciri algoritma yang dipaparkan oleh Donuld E. Knuth, maka dapat disimpulkan sifat utama suatu algoritma[9], yaitu :

1. *Input*

   Suatu algoritma memliki *input* atau kondisi awal sebelum algoritma dilaksanakan. *Input* tersebut dapat berupa nilai-nilai pengubah yang diambil dari himpunan khusus.

2. *Output*

   Suatu algortima akan menghasilkan *output* setelah dilaksanakan. Nilai *output* diperoleh dari nilai *input* yang telah diproses melalui algoritma.

3. *Definiteness*

   Langkah-langkah yang dituliskan dalam algoritma terdefinisi dengan jelas sehingga mudah dilaksanakan oleh sistem.

4. *Finiteness*

   Suatu algoritma harus memberikan kondisi nilai akhir atau *output* setelah melakukan sejumlah langkah yang terbatas jumlahnya untuk setiap kondisi awal atau *input* yang diberikan.

5. *Effectiveness*

   Setiap langkah dalam algoritma bisa dilaksanakan dalam selang waktu tertentu sehingga pada akhirnya memberi solusi sesuai yang diharapkan.

6. *Generality*

   Langkah-langkah algoritma berlaku untuk setiap himpunan *input* yang sesuai dengan persoalan yang akan diberikan, tidak hanya untuk himpunan tertentu.

2.5 Algoritma *String Matching*

Menurut Brassard dan Bratley, algoritma *string matching* adalah sebuah algoritma yang digunakan dalam pencocokkan suatu pola kata tertentu terhadap suatu kalimat atau teks [9]. Menurut arah pencarianya, algoritma pencocokan *string* dapat diklasifikasikan menjadi tiga bagian[9], yaitu sebagai berikut:

1. Dari arah yang paling alami, yaitu dari kiri ke kanan yang merupakan arah membaca, algoritma yang termasuk kategori ini adalah :
   1) Algoritma *Brute Force*
   2) Algoritma *Knuth-Morris-Pratt*
2. Dari kanan ke kiri, arah yang biasanya menghasilkan hasil terbaik secara praktikal, contohnya adalah Algoritma *Boyer-Moore* yang kemudian banyak dikembangkan menjadi algoritma turbo *Boyer-Moore,* algoritma tuned *Boyer-Moore* dan algoritma *Zhu-Takaoka.*
3. Dari arah yang ditentukan secara spesifik oleh algoritma tersebut (arah ini menghasilkan hasil terbaik secara teoretis), seperti:
   1) Algoritma *Colussi*
   2) Algoritma *Chrochemore-Perrin*

2.5.1 Algoritma *Zhu-Takaoka*

Algoritma *Zhu-Takaoka* merupakan salah satu algoritma pencocokan *string* (*String Matching*) yang merupakan pengembangan dari algoritma *Boyer-Moore*. Algoritma ini dipublikasikan oleh Zhu Rui Feng dan Tadao Takaoka pada tahun 1986 [10]. Dalam penelitiannya, Zhu dan Takaoka menyebut algoritma pencocokan *string* ini sebagai BM" Algorithm (*Boyer-Moore* Algorithm) karena merupakan modifikasi dari algoritma pencocokan *string Boyer-Moore*, yaitu algoritma yang dibuat oleh Boyer R.S. dan Moore J.S. Algoritma *Zhu-Takaoka* dan algoritma *Boyer-Moore* memiliki ciri yang sama dalam proses pencarian *string*, yaitu terdapat tahap *Preprocessing, Right-to-left scan, Bad character rule* dan *Good-suffix rule*[10]. Sementara itu, perbedaan dari kedua algoritma tersebut terletak pada tahap penentuan *Bad character rule.* Dalam *Boyer-Moore, bad character* hanya terdiri dari *array* satu dimensi, sedangkan dalam *Zhu-Takaoka array* dimodifikasi menjadi dua dimensi. Berikut ini adalah penjelasan mengenai karakter algoritma *Zhu-Takaoka.*

1. *Preprocessing*

Preprocessing adalah proses yang dilakukan sebelum proses inti, yaitu proses pencarian *pattern* dalam suatu teks. Dalam algoritma *Zhu-Takaoka*, *preprocessing* meliputi proses pencarian nilai pergeseran karakter *(good-suffix shift)* yang nilai nya ditentukan dalam *good suffix preprocessing*, dan pergeseran karakter jika ditemukan ketidakcocokan karakter *(bad-character shift)* yang nilai nya ditentukan dalam *bad character preprocessing*.

2. Right-to-Left Scan Rule

Sebagaimana disebutkan sebelumnya, bahwa *Zhu-Takaoka* merupakan salah satu algoritma yang melakukan pencocokan *string* dari kanan ke kiri, maka proses pencarian algoritma ini dilakukan dengan teknik *right-to-left scan rule*. Teknik ini dilakukan dengan membandingkan *pattern* dengan sumber teks. Perbandingan dilakukan dari karakter paling kanan pada *pattern* dengan karakter paling kanan pada sumber teks. Apabila dalam proses tersebut ditemukan kecocokan, maka perbandingan akan dilanjutkan dengan bergerak ke kiri sampai pada karakter pertama dari *pattern*. Tetapi apabila ditemukan ketidakcocokan, maka akan dilakukan pergeseran yang jumlah pergeserannya ditentukan oleh aturan *bad-character shift* atau *good-suffix shift*.

3. Bad-Charcter Shift Rule

Ketika ditemukan ketidakcocokan dalam proses perbandingan, aturan *bad-character shift* dibutuhkan untuk menghindari pengulangan perbandingan. Dalam aturan *bad-character shift*, jumlah pergeseran yang dilakukan disimpan dalam bentuk tabel *array* dua dimensi. Tabel tersebut terdiri dari beberapa kolom, yaitu kolom karakter dan kolom *shift* yang menunjukan jumlah pergeseran.

4. Good Suffix Shift Rule

Aturan *good-suffix shift* dibuat untuk menangani kasus dimana terdapat pengulangan karakter pada *pattern*.

5. Langkah – langkah pencarian

Proses inti pencarian Algoritma *Zhu-Takaoka* yaitu dilakukan dengan teknik *Right-to-left scan rule*. Teknik ini membandingkan *pattern* yang dicari dengan sumber teks dimulai dari kanan ke kiri. Berikut contoh pencocokan *string* dengan algoritma *Zhu-Takaoka* :

Pattern yang dicari : DONE

Sumber *string* (Teks) : INDONESIAINDONESI

Dari hasil preproccessing maka dihasil kan tabel ztBc :

Tabel 2.1 Tabel ztBc

	D	E	N	O
D	3	4	4	2
E	3	4	4	4
N	3	4	4	4
O	3	4	1	4

Tabel 2.2 bmGs:

i	0	1	2	3
X[i]	D	O	N	E
bmGs	4	4	4	1

Percobaan ke 1:

I	N	D	O	N	E	S	I	A	I	N	D	O	N	E	S	I
D	O	N	E													

Dari hasil percobaan ke-1, dapat dilihat bahwa karakter akhir *pattern* yang dicari yaitu karakter **e** sejajar dengan karakter **o** pada sumber *string* artinya pada percobaan pertama terjadi ketidakcocokan, maka dilakukan pergeseran sejauh dua karakter. Nilai pergeseran dua karakter ini diperoleh dari tabel ztBc dimana dicocokannya dua karakter akhir pada sumber *string* yang sejajar dengan karakter akhir *pattern*. Pada contoh ini diisi dengan karakter **d** dan **o**. maka cek pada tabel ztBc baris **d** dan kolom **o** maka bernilai dua.

Percobaan ke 2 :

I	N	D	O	N	E	S	I	A	I	N	D	O	N	E	S	I
		D	O	N	E											

Terlihat dari hasil percobaan ke-2, *pattern* yang dicari kemudian ditemukan pada sumber *string* maka akan diberi tanda dan dilanjutkan pada pencocokan selanjutnya dengan pergeseran berdasarkan nilai dari tabel bmGs. Dari tabel bmGs karakter **d** berada pada *array* ke nol dan memiliki nilai pergeseran empat, maka *pattern* digeser sejauh 4 karakter.

Percobaan ke 3 :

I	N	D	O	N	E	S	I	A	I	N	D	O	N	E	S	I
						D	O	N	E							

Dari hasil percobaan ke-3, dapat dilihat bahwa karakter akhir *pattern* yang dicari yaitu karakter **e** sejajar dengan karakter **i** pada sumber *string* artinya pada percobaan ketiga terjadi ketidakcocokan. Sama dengan sebelumnya nilai pergeseran diambil dari dua karakter akhir pada sumber *string* yang sejajar dengan karakter akhir *pattern* yang kemudian dicocokan dengan tabel ztBc. Untuk percobaan ke-3 diisi dengan karakter **a** dan **i** dan setelah dicocokan baris **a** dan kolom **i** pada tabel ztBc menghasilkan nilai pergeseran empat.

Percobaan ke 4 :

I	N	D	O	N	E	S	I	A	I	N	D	O	N	E	S	I
										D	O	N	E			

Pada hasil percobaan ke-4 terlihat bahwa karakter akhir *pattern* yang dicari yaitu karakter **e** sejajar dengan karakter **n** pada sumber *string* artinya pada percobaan keeempat pun terjadi ketidakcocokan. Dengan dua karakter akhir yang diambil yaitu **o** dan **n** maka nilai pergeseran dari ketidakcocokan adalah satu.

Percobaan ke 5 :

I	N	D	O	N	E	S	I	A	I	N	D	O	N	E	S	I
											D	O	N	E		

Hasil percobaan ke-5 menunjukan terjadinya kecocokan, maka pergeseran didasarkan pada tabel bmGs. Dari tabel bmGs, karakter **d** berada pada *array* ke nol

dan memiliki nilai pergeseran empat, maka *pattern* digeser sejauh 4 karakter. Karena sepanjang sumber *string* sudah habis maka pencocokanpun dihentikan. Dari contoh diatas dapat ditarik kesimpulan dengan panjang sumber *string* 18 karakter dan panjang *pattern* yang dicari empat dilakukan lima kali percobaan dan menghasilkan dua pola yang cocok.

I	N	D	O	N	E	S	I	A	I	N	D	O	N	E	S	I

2.5.2 Algoritma *Reverse Colussi*

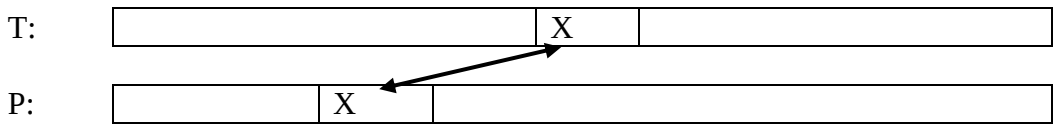
Algoritma *Reverse Colussi* adalah salah satu algoritma pencocokan *string* yang merupakan pengembangan dari algoritma *Boyer-Moore*. Algoritma ini terdiri dari dua fase, yaitu fase pemrosesan awal dan fase pencarian [5]. Fase pemrosesan awal dilakukan untuk menentukan pergeseran *pattern* dengan melakukan pencarian sepasang karakter, *special position* dan *non-special position*. Sementara itu, pada fase pencarian dilakukan proses pencocokan *pattern* terhadap sumber teks.

2.5.2.1 Fase Pemrosesan Awal

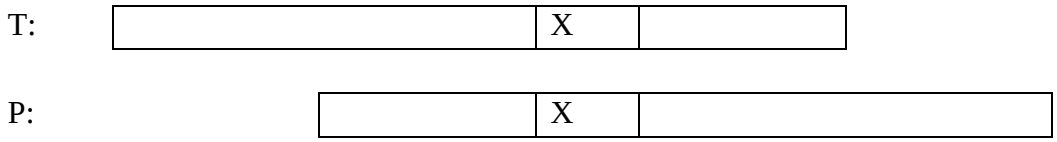
Pada algoritma *Reverse Colussi*, harus ditentukan beberapa *point* yang dianggap spesial dan beberapa *point* yang dianggap tidak spesial. Spesial *point* memungkinkan angka pergeseran yang lebih kecil dari pada ‘bukan’ spesial *point*. Oleh karena itu, dalam algoritma *Reverse Colussi* harus ditentukan *special position* terlebih dahulu.

Misal, Ti merupakan karakter ke-i pada T($1 \leq i \leq n$), dan Pj adalah karakter ke-j pada P($1 \leq j \leq m$). Berikut gambaran mengenai yang dimaksud sepasang karakter.

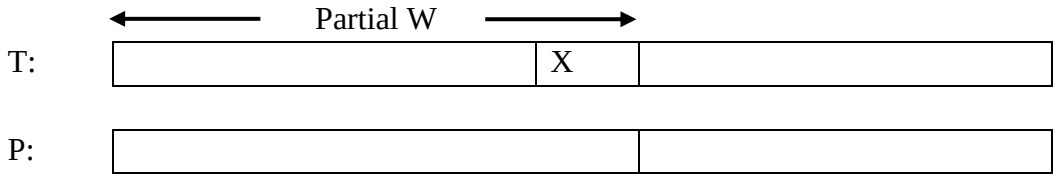
Untuk setiap karakter x pada T, cari karakter x terdekat pada P yang terletak pada sisi kiri dari karakter x pada T.



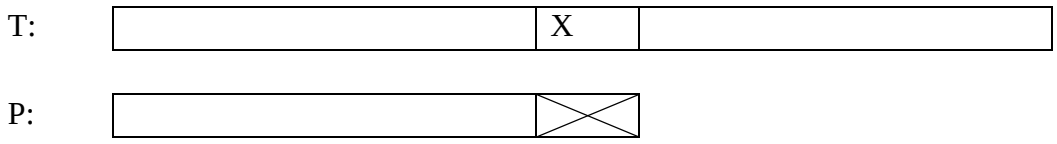
Jika terdapat karakter x pada P yang terletak disebelah kiri dari karakter x pada T, maka geser P sampai kedua karakter x tersebut sejajar.



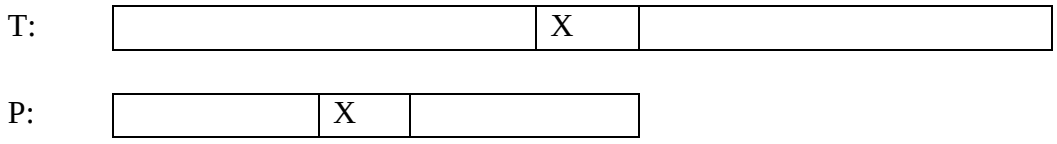
Jika tidak ada, maka buat partial window berdasarkan posisi karakter x pada T dan *string* yang ada di sebelah kirinya. Setelah ini, proses pencarian hanya sebatas ukuran window saja.



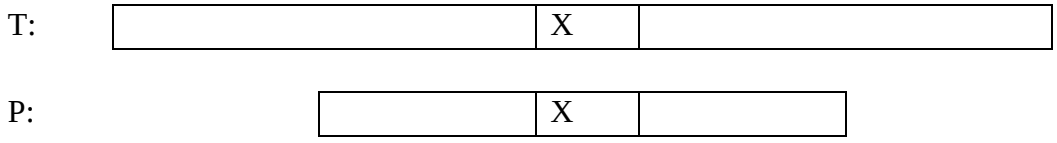
Misalkan karakter x yang terakhir pada window T tidak cocok dengan karakter terakhir pada P.



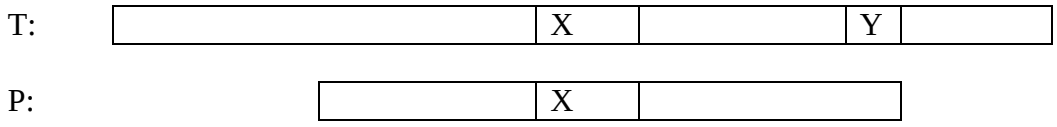
Misalkan terdapat karakter x pada P:



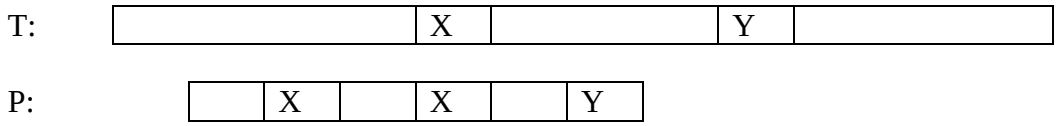
Maka, sejajarkan posisi x pada P dengan posisi x pada T



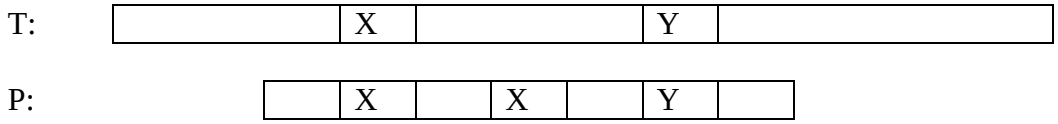
Misalkan karakter terakhir y dari window pada T tidak cocok dengan karakter terakhir dari P:



Jika tidak cocok, kemudian cari pasangan x dan y pada P. Misal posisi x dan y sebagai berikut:



Maka geser sampai posisi pasangan x dan y pada P sejajar dengan posisi x dan y pada T, sepasang karakter sudah diperoleh.



2.5.2.1.1 Pencarian sepasang karakter

Dalam algoritma *Reverse Colussi*, pencarian pasangan karakter dapat dilakukan dengan menggunakan bantuan tabel rcBc[5]. Misalkan:

Y = adalah karakter terakhir dari window pada T

s = adalah panjang pergeseran yang akan digunakan pada langkah terakhir

k = bilangan integer

m = panjang *pattern*

Kondisi 1:

Jika ditemukan Pm-k-1=Y and Pm-k-s-1=Pm-s-1,

masukkan k terkecil ke tabel rcBc[Y,s]

Kondisi 2:

Jika ditemukan Pm-k-1=Y and k>m-s-1,

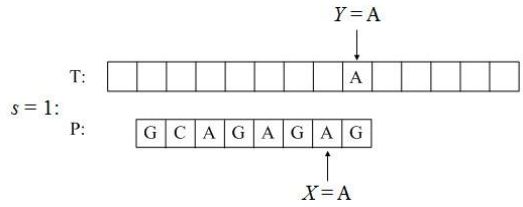
masukkan k terkecil ke tabel rcBc[Y,s]

Kondisi 3:

Jika tidak ditemukan,

masukkan m ke tabel rcBc[Y,s]

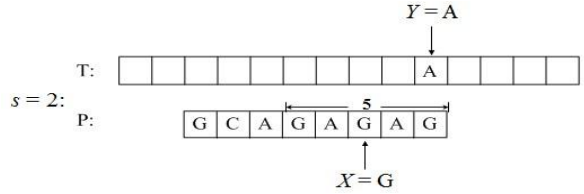
Contoh:



XY = AA tidak terdapat pada P.

rcBc[Y, 1] = m => rcBc[Y, 1] = 8

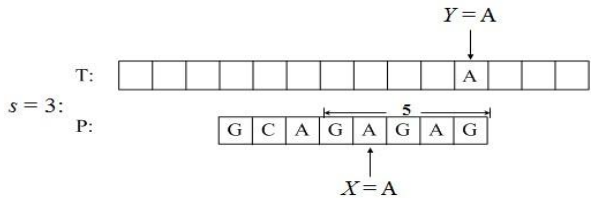
Present matched character of T (Y) \ Length of Previous Shifts (s)	1	2	3	4	5	6	7	8
A	8							
C								
G								
T								



G | | A ditemukan G C A

Cari rcBc[Y, 2] = 5

Present matched character of T (Y) \ Length of Previous Shifts (s)	1	2	3	4	5	6	7	8
A	8	5						
C								
G								
T								



A | | | A ditemukan | G C A

Cari *rcBc*[*Y*, 3] = 5

Present matched character of T (Y) \ Length of Previous Shifts (s)	1	2	3	4	5	6	7	8
A	8	5	5					
C								
G								
T								

Proses tersebut dilakukan hingga tabel terisi semua.

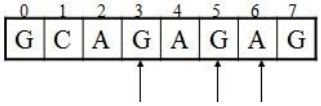
Present matched character of T (*Y*) \ Length of Previous Shifts (*s*)	1	2	3	4	5	6	7	8
A	8	5	5	3	3	3	1	1
C	8	6	6	6	6	6	6	6
G	2	2	2	4	4	2	2	2
T	8	8	8	8	8	8	8	8

2.5.2.1.2 Pencarian *special positions* dan *non-special positions*

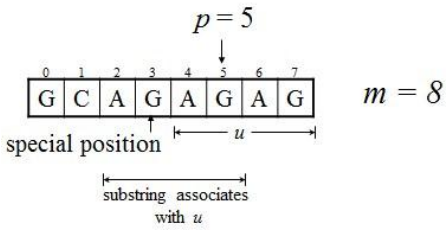
Dalam algoritma *Reverse Colussi*, pencarian *special positions* dapat dilakukan dengan menggunakan bantuan tabel rcGs[5]. Dalam membuat tabel rcGs ini, aturan mengenai *substring matching (suffix / prefix)* yang digunakan sama dengan aturan yang digunakan pada algoritma *Boyer-Moore*. Misalkan diberikan *pattern* P, tunjukkan semua posisi *suffix* sering diulang pada P atau disebut sebagai *special positions.*



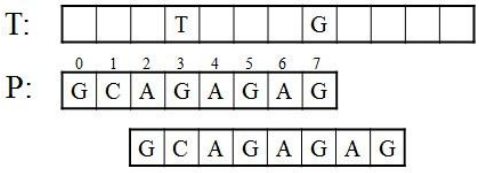
Dari *pattern* diatas, G, AG, AGAG merupakan *substring* yang sering diulang. Berikut *special positions* dari *pattern* tersebut.



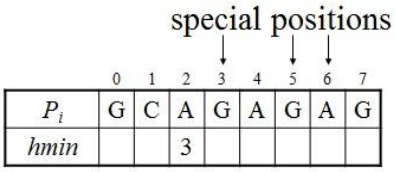
Untuk setiap *suffix* terpanjang u, sejajarkan posisi *substring* yang ada pada P. Jika *special positions* tidak cocok dengan u, geser P sejauh m-p-1 langkah dimana m adalah panjang *pattern*.



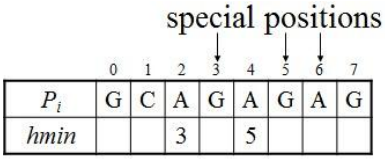
Hasil pergeseran sebesar 8-5-1 sebagai berikut:



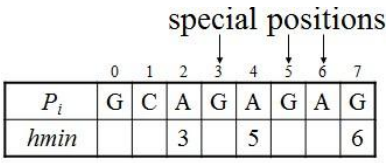
Jumlah pergeseran untuk setiap *special positions* disimpan dalam tabel yang disebut *hmin.*



Special position i = 3, simpan jumlah pergeseran (8-5-1=2) pada tabel *hmin*[2]=3

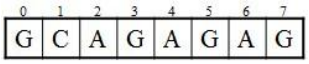


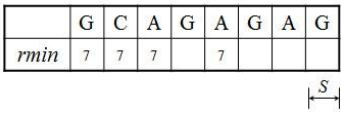
Special position i = 5, simpan jumlah pergeseran (8-3-1=4) pada tabel *hmin*[4]=5



Special position i = 6, simpan jumlah pergeseran (8-01=7) pada tabel *hmin*[6]=6 Setelah membandingkan *special positions, remainder position/non-special positions* juga harus dibandingkan. Perbandingan ini dilakukan dari kiri ke kanan. Jumlah pergeseran untuk setiap *non-special positions* disimpan dalam tabel yang disebut *rmin.*

Jika *suffix* S yang terdapat pada sisi kanan dari *nonspecial positions* ke-i sama dengan *prefix* maka *rmin(i) = m*-|*S*|. (|S| panjang S), tetapi jika tidak ada S maka *rmin(i)=m*. Contoh:





Suffix S sama dengan *prefix* yang ada di sisi kanan dari *non-special positions*, jadi nilai dari *rmin non-special positions* adalah *m*-|*S*| *(8-1)*

Setelah tabel *hmin* dan *rmin* terbuat, sekarang dapat digunakan untuk membuat tabel rcGs.

Contoh: GCAGAGAG

i	0	1	2	3	4	5	6	7	8
P_i	G	C	A	G	A	G	A	G	
hmin[*i*]			3		5			6	
rmin[*i*]	7	7	7		7				
rcGs[*i*]	0								

Pertama-tama, isi indeks dari *special positions* yang tidak kosong kedalam tabel rcGs, menghasilkan:

i	0	1	2	3	4	5	6	7	8
P_i	G	C	A	G	A	G	A	G	
hmin[*i*]			3		5			6	
rmin[*i*]	7	7	7		7				
rcGs[*i*]	0	2	4						

Kemudian isikan semua nilai dari tabel *rmin* yang tidak kosong kedalam tabel rcGs.

i	0	1	2	3	4	5	6	7	8
P_i	G	C	A	G	A	G	A	G	
hmin[*i*]			3		5			6	
rmin[*i*]	7	7	7		7				
rcGs[*i*]	0	2	4	7	7	7	7	7	

Jika *P exact match* dengan *T*, maka geser *P*. jadi *rcGs*[8]=*m*-|*S*| (8-1).

i	0	1	2	3	4	5	6	7	8
P_i	G	C	A	G	A	G	A	G	
hmin[*i*]			3		5			6	
rmin[*i*]	7	7	7		7				
rcGs[*i*]	0	2	4	7	7	7	7	7	7

2.5.2.2 Fase Pencocokan Pattern

Setelah tabel rcBc dan rcGs terbentuk, pencocokan *pattern* dilakukan berdasarkan nilai dari kedua tabel tersebut. Pada algoritma *Reverse Colussi*, perbandingan karakter dilakukan dengan menggunakan urutan tertentu yang diberikan oleh tabel *h*[5].

Untuk setiap i dimana $0 \leq i \leq m$ definisikan dua himpunan yang *disjoint*:

$Pos(i)=\{k : 0 \leq k \leq i$ dan $x[i] = x[i-k]\}$

$Neg(i)=\{k : 0 \leq k \leq i$ dan $x[i] \neq x[i-k]\}$

For $1 \leq k \leq m$, biarkan $hmin[k]$ menjadi nilai terkecil ℓ dimana $\ell \geq k$-1 dan k bukan elemen dari $Neg(i)$ untuk semua i dimana $\ell < i \leq m$-1.

For $0 \leq \ell \leq m$-1, biarkan $kmin[\ell]$ menjadi nilai terkecil k dimana $hmin[k]= \ell \geq k$ jika ada k dan $kmin[\ell]=0$ jika tidak ada k.

For $0 \leq \ell \leq m$-1, biarkan $rmin[\ell]$ menjadi nilai terkecil k dimana $r > \ell$ dan $hmin[r]=r$-1.

Nilai dari $h[0]$ diisi dengan m-1. Setelah itu, *increment* $kmin[\ell]$, semua indeks $h[1]$, ..., $h[d]$ dimana $kmin[h[i]] \neq 0$ dan isi $rcGs[i]$ dengan $kmin[h[i]]$ untuk $1 \leq i \leq d$. lalu pilih indeks $h[d+1]$, ... , $h[m-1]$ untuk di-*increment* dan isi $rcGs[i]$ dengan $rmin[h[i]]$ untuk $d < i < m$.

Percobaan

Diketahui:

Teks T, *Pattern* P:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T=	G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G

P=	G	C	A	G	A	G	A	G

s = m = 8

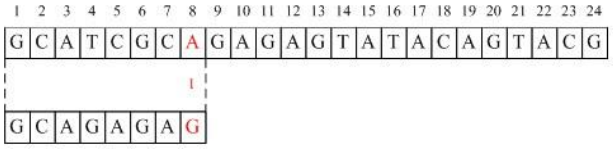
Pemrosesan awal akan menghasilkan tabel rcBc dan rcGs.

rcBc	1	2	3	4	5	6	7	8
A	8	5	5	3	3	3	1	1
C	8	6	6	6	6	6	6	6
G	2	2	2	4	4	2	2	2
T	8	8	8	8	8	8	8	8

i	0	1	2	3	4	5	6	7	8
rcGs[i]	0	2	4	7	7	7	7	7	7

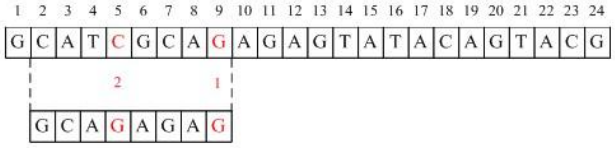
Fase Pencocokan Pattern:

Percobaan ke-1



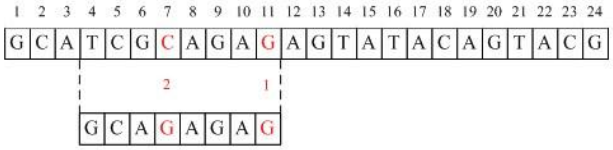
Shift by 1 (*rcBc*[A][*s*], *s* = 8), *and change* s = 1

Percobaan ke-2



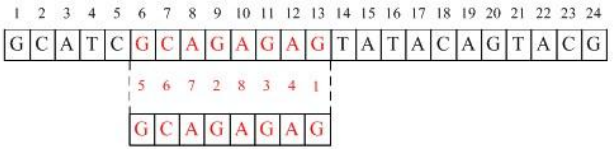
Shift by 2 (*rcGs*[1]), *and change* s = 2

Percobaan ke-3



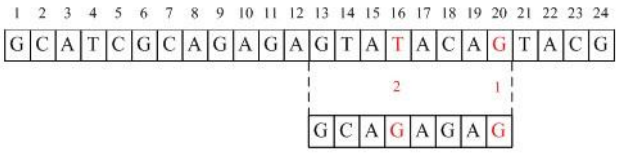
Shift by 2 (*rcGs*[1]), *and change* s = 2

Percobaan ke-4



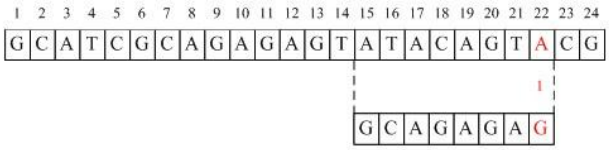
Shift by 7 (*rcGs*[8]), *and change* s = 7

Percobaan ke-5



Shift by 2 (*rcGs*[1]), *and change* s = 2

Percobaan ke-6

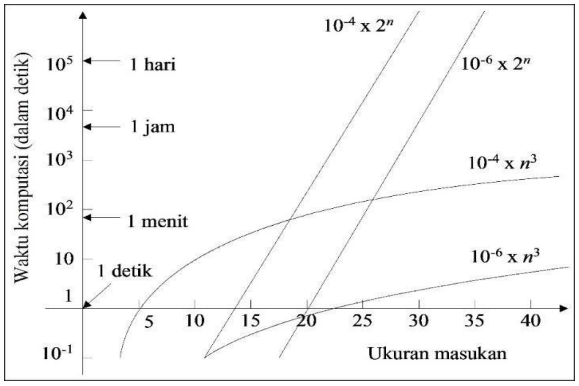


Shift by 5 (*rcBc*[A][*s*], *s* = 2), *and change s* = 5

Untuk mencocokkan pattern GCAGAGAG pada GCATCGCAGAGAGT-ATACAGTACG memerlukan percobaan sebanyak 6 kali percobaan dengan jumlah perbandingan sebanyak 16 kali.

2.6 Kompleksitas Algoritma

Sebuah algoritma tidak saja harus benar, tetapi juga harus mangkus (*efisien*). Kemangkusan algoritma diukur dari berapa jumlah waktu dan ruang (*space*) memori yang dibutuhkan untuk menjalankanya. Algoritma yang mangkus ialah algoritma yang meminimumkan kebutuhan waktu dan ruang[9]. Kebutuhan waktu dan ruang suatu algoritma bergantung pada ukuran masukan (*n*), yang menyatakan jumlah data yang diproses. Kemangkusan algoritma dapat digunakan untuk menilai algoritma yang baik. Gambar di bawah ini membuktikan alasan memerlukan algoritma yang mangkus dimana ukuran masukan mempengaruhi waktu komputasi.



Gambar 2.2 Waktu komputasi algoritma

Pengukuran waktu yang diperlukan oleh sebuah algoritma adalah dengan menghitung banyaknya operasi/instruksi yang dieksekusi. Untuk mengetahui besaran waktu pada sebuah operasi tertentu, maka dapat dihitung dengan cara berapa waktu sesungguhnya untuk melaksakan algoritma tersebut.

Contoh : Menghitung rata – rata

a_1	a_2	a_3	…	a_n

Larik bilangan bulat

```
procedure HitungRerata(input a1, a2, ..., an : integer, output r
: real){
  Menghitung nilai rata-rata dari sekumpulan elemen
  larik integer a1, a2, ..., an.
  Nilai rata-rata akan disimpan di dalam peubah r.
  Masukan: a1, a2, ..., an.
  Keluaran: r (nilai rata-rata)
}

Deklarasi
  k : integer
  jumlah : real

Algoritma
  jumlah ← 0
  k ← 1
  while k ≤ n do
    jumlah ← jumlah + ak
    k ← k+1
  endwhile
  { k > n }
  r ←  jumlah / n { nilai rata-rata }
```

Model perhitungan pada algoritma diatas yaitu :

1. Operasi pengisian nilai (jumlah ← 0, k ← 1, jumlah ← jumlah+a_k, k ← k+1, dan r ← jumlah/n) jumlah operasi pengisian nilai adalah

   $t_1 = 1 + 1 + n + n + 1 = 3 + 2n$ (2.1)
2. Operasi penjumlahan (jumlah+a_k, dan k+1) jumlah seluruh operasi penjumlahan adalah

   $t_2 = n + n = 2n$ (2.2)
3. Operasi pembagian (jumlah/n) jumlah seluruh operasi pembagian adalah

   $t_3 = 1$ (2.3)
4. Maka total kebutuhan waktu algoritma menghitung rata – rata adalah

   $t = t_1 + t_2 + t_3 = (3 + 2n)a + 2nb + c$ detik (2.4)

Model perhitungan kebutuhan waktu seperti diatas kurang dapat diterima karena dalam prakteknya tidak mempunyai informasi berapa waktu sesungguhnya untuk melakasanakan suatu operasi tertentu. Komputer dengan arsitektur yang berbeda, akan berbeda pula lama waktu untuk setiap jenis operasinya. Selain bergantung pada komputer, kebutuhan waktu sebuah program juga ditentukan oleh *compiler* bahasa yang digunakan.

Model abstrak pengukuran waktu/ruang harus independen dari pertimbangan mesin dan compiler apapun. Besaran yang dipakai untuk menerangkan model abstrak pengukuran waktu/ruang ini adalah kompleksitas algoritma. Ada dua macam kompleksitas algoritma[9], yaitu :

1. Kompleksitas waktu

   Kompleksitas waktu atau $T(n)$, diukur dari jumlah tahapan komputasi yang dibutuhkan untuk menjalankan algoritma sebagai fungsi dari ukuran masukan *n.*

2. Kompleksitas ruang

   Kompleksitas ruang atau *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/ruang algoritma, maka dapat menentukan laju peningkatan waktu/ruang yang diperlukan algoritma dengan meningkatnya ukuran masukan *n*.

2.6.1 Kompleksitas Waktu

Komplekstias waktu merupakan hal penting untuk mengukur efisiensi suatu algoritma, karena kompleksitas waktu dari suatu algoritma yang terukur adalah fungsi ukuran masalah [9]. Di dalam sebuah algoritma terdapat beberapa macam jenis operasi yaitu sebagai berikut:

1. Operasi baca / tulis.
2. Operasi aritmatika (+ ,-, *, /).
3. Operasi pengisian nilai (*assignment*).
4. Operasi pengaksesan elemen larik.
5. Operasi pemanggiilan fungsi/prosedur, dll.

Di dalam prakteknya hanya menghitung jumlah operasi khas (tipikal) yang mendasari suatu algoritma. Contoh operasi khas didalam algoritma yaitu :

1. Algoritma pencarian didalam larik
   Operasi khas : perbandingan elemen larik
2. Algoritma pengurutan
   Operasi khas : perbandingan elemen, pertukaran elemen
3. Algoritma penjumlahan 2 buah matriks
   Operasi khas : penjumlahan
4. Algoritma perkalian 2 buah matriks
   Operasi khas : perkalian dan penjumlahan

Contoh : Algoritma menghitung rata – rata sebuah larik (*array*)

```
sum ← 0
for i ← 1 to n do
  sum ← sum + a[i]
endfor
rata_rata ← sum/n
```

Operasi yang mendasar pada algoritma di atas adalah operasi penjumlahan elemen-elemen a_i (yaitu sum ← sum + a[i]) yang dilakukan sebanyak *n* kali dan mempunyai kompleksitas waktu yaitu $T(n) = n$. Kompleksitas waktu dibedakan atas tiga macam yaitu :

1. $T_{max}(n)$ yaitu kompleksitas waktu untuk kasus terburuk (*worst case*) atau kebutuhan waktu maksimum.
2. $T_{min}(n)$ yaitu kompleksitas waktu untuk kasus terbaik (*best case*) atau kebutuhan waktu minimum.
3. $T_{avg}(n)$ yaitu kompleksitas waktu untuk kasus rata-rata (*average case*) atau kebutuhan waktu secara rata – rata.

Contoh : Algoritma *sequential search*

```
procedure PencarianBeruntun(input a1, a2, ..., an : integer, x :
integer,output idx : integer)
Deklarasi
  k : integer
  ketemu : boolean
```

```
Algoritm
  k ← 1
  ketemu ← false
  while (k ≤ n) and (not ketemu) do
    if ak = x then
      ketemu ← true
    else
      k ← k + 1
    endif
  endwhile{ k > n or ketemu }
  if ketemu then { x ditemukan }
    idx ← k
  else
    idx ← 0 { x tidak ditemukan }
  endif
```

Pada algoritma diatas maka dapat dihitung kompleksitas waktunya dengan menghitung jumlah operasi perbandingan yaitu :

1. Kasus terbaik terjadi apabila $a_1 = x$

$$T_{\min}(n) = 1 \qquad (2.5)$$

2. Kasus terburuk terjadi apabila $a_n = x$ atau x tidak ditemukan.

$$T_{\max}(n) = n \qquad (2.6)$$

3. Kasus rata – rata terjadi jika x ditemukan pada posisi ke-j,maka operasi perbandingan ($a_k = x$)akan dieksekusi sebanyak j kali.

$$T\text{avg}(n) = \frac{(1 + 2 + 3 + \ldots + n)}{n} + \frac{\frac{1}{2} n\,(1 + n)}{n} = \frac{n + 1}{2} \qquad (2.7)$$

2.7 Notasi Big O

Notasi *Big O* merupakan suatu notasi matematika untuk menjelaskan batas atas dari *magnitude* suatu fungsi dalam fungsi yang lebih sederhana [8]. Dalam dunia ilmu komputer, notasi ini sering digunakan dalam analisis kompleksitas algoritma. Notasi *Big O* pertama kali diperkenalkan pakar teori bilangan Jerman, Paul Bachman tahun 1894 dalam bukunya yang berjudul *Analytische Zahlentheorie* edisi kedua[8]. Notasi tersebut kemudian dipopulerkan oleh pakar teori bilangan Jerman lainnya, Edmund Landau, dan oleh karena itu, terkadang disebut sebagai *symbol Landau*.

Untuk membandingkan kompleksitas algoritma yang satu dengan yang lain, dapat digunakan tabel 2.3 yaitu tabel jenis kompleksitas yang diurutkan berdasarkan kompleksitas yang paling baik ke yang paling buruk. Sebuah masalah yang mempunyai algoritma dengan kompleksitas polinomial kasus terburuk dianggap mempunyai algoritma yang "bagus", artinya masalah tersebut mempunyai algoritma yang mangkus, dengan catatan polinomial tersebut berderajat rendah. Jika polinomnya berderajat tinggi, waktu yang dibutuhkan untuk mengeksekusi algoritma tersebut panjang.

Tabel 2.3 Jenis Kompleksitas

Notasi	**Nama**
O(1)	*Konstan*
O(log n)	*Logaritmik*
O(n)	*Linier*
O(n log n)	*Linierithmik*
$O(n^2)$	*Kuadratik*
$O(n^3)$	*Kubik*
$O(2^n)$	*Eksponensial*
O(n!)	*Faktorial, kombinatorial*

Penjelasan masing – masing kelompok jenis kompleksitas algoritma adalah sebagai berikut :

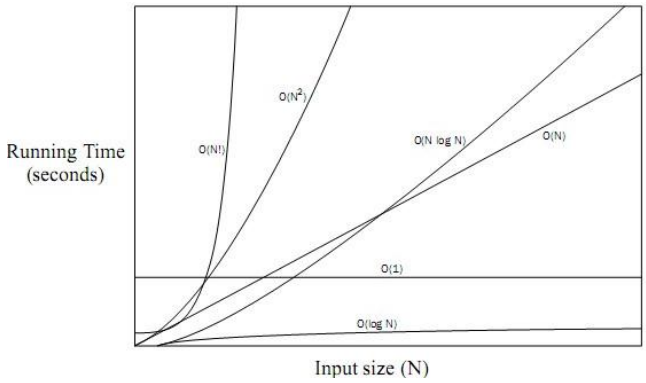
1. *O(1)* yaitu kompleksitas *O(1)* berarti waktu pelaksanaan algoritma adalah tetap, tidak bergantung pada ukuran masukan.

   Contoh prosedur *tukar* dibawah ini :

```
procedure tukar(var a:integer; var b:integer);
var
  temp:integer;
begin
  temp:=a;
  a:=b;
  b:=temp;
end;
```

   Pada contoh kasus diatas jumlah operasi penugasan (*assignment*) ada tiga buah dan tiap operasi dilakukan satu kali. Jadi *T(n)* = 3 = *O(1).*

2. $O(\log n)$ yaitu kompleksitas waktu logaritmik berarti laju pertumbuhan waktunya berjalan lebih lambat daripada pertumbuhan n. Algoritma yang termasuk kelompok ini adalah algoritma yang memecahkan persoalan besar dengan mentransformasikannya menjadi beberapa persoalan yang lebih kecil yang berukuran sama. Di sini basis algoritma tidak terlalu penting sebab bila n dinaikkan dua kali semula, misalnya, log n meningkat sebesar sejumlah tetapan.
3. $O(n)$ yaitu kompleksitas linier dimana waktu pelaksanaanya umumnya terdapat pada kasus yang setiap elemen masukanya dikenai proses yang sama. Bila n dinaikan dua kali semula.
4. $O(n \log n)$ yaitu kompleksitas loglinier dimana waktu pelaksanaan yang n log n terdapat pada algoritma yang memecahkan persoalan menjadi beberapa persoalan yang lebih kecil, menyelesaikan tiap persoalan secara independen, dan menggabung solusi masing-masing persoalan. Algoritma yang diselesaikan dengan teknik bagi dan gabung mempunyai kompleksitas asimptotik jenis ini. Bila n = 1000, maka n log n mungkin 20.000. Bila n dijadikan dua kali semua, maka n log n menjadi dua kali semula (tetapi tidak terlalu banyak).
5. $O(n^2)$ yaitu kompleksitas kuadratik dimana waktu pelaksanaannya kuadratik hanya praktis digunakan untuk persoalan yang berukuran kecil. Umumnya algoritma yang termasuk kelompok ini memproses setiap masukan dalam dua buah perulangan bersarang.
6. $O(n^3)$ yaitu kompleksitas kubik, seperti halnya algoritma kuadratik, algoritma kubik memproses setiap masukan dalam tiga buah perulangan bersarang.
7. $O(2^n)$ yaitu kompleksitas eksponensial tergolong kelompok ini mencari solusi persoalan secara seperti algoritma *Brute Force*. Bila n = 20, waktu pelaksanaan algoritma adalah 1.000.000. Bila n dijadikan dua kali semula, waktu pelaksanaan menjadi kuadrat kali semula.
8. $O(n!)$ yaitu kompleksitas faktorial. Seperti halnya pada algoritma eksponensial, algoritma jenis ini memproses setiap masukan dan menghubungkannya dengan $n - 1$. Bila n = 5, maka waktu pelaksanaan algoritma adalah 120. Bila n dijadikan dua kali semula, maka waktu pelaksanaan algoritma menjadi faktorial dari $2n$.



Gambar 2.3 Perbandingan kompleksitas algoritma

Sebagai contoh terdapat suatu program sederhana yang akan dihitung komplesitasnya dengan menggunakan Big-O seperti :

```
sum = 0;
for (i=0; i<n; i++)
  sum= sum +a[i]
```

Kemudian pada tabel di bawah ini dapat dilihat perhitungan kompleksitas dengan menggunakan Big-O dari contoh program sederhana diatas :

Tabel 2.4 Contoh Perhitungan Big-O dengan program sederhana

Program	**Notasi Big-O**	**Keterangan**
sum = 0	O(1)	Dieksekusi 1 Kali
i=0	O(1)	Dieksekusi 1 Kali
i<n	O(N)	Dieksekusi 1 Kali
i++	O(N)	Dieksekusi 1 Kali
sum = sum +a[i]	O(N)	Dieksekusi 1 Kali

Dari hasil tabel di atas maka dapat dihitung hasil dari notasi Big-O adalah sebagai berikut :

O(1) + O(1) + O(N) + O(N) + O(N) = 2+ 3N kali = O(N)

Jadi dari program sederhana di atas mempunyai hasil perhitungan dengan Big-O adalah O(N) jadi untuk waktu eksekusinya sebanding dengan jumlah data jika n=5 maka waktu eksekusinya pun 5.

2.8 Pengujian Algoritma

Pengujian Algoritma ini dijabarkan dengan memberikan pengantar pengujian perangkat lunak dan pengujian algoritma *String Matching*; memperkenalkan pengujian perangkat lunak dan berfokus pada jenis pengujian yang relevan dengan algoritma yang disebut *unit testing*; memberikan contoh spesifik dari algoritma dan memberikan beberapa aturan untuk menguji algoritma pada umumnya [12].

2.8.1 Pengujian Perangkat Lunak

Software pengujian di bidang Rekayasa Perangkat Lunak adalah proses dalam proyek perangkat lunak yang memverifikasi bahwa produk atau jasa memenuhi harapan dari kualitas dan memvalidasi bahwa perangkat lunak memenuhi persyaratan spesifikasi[12]. Pengujian perangkat lunak dilakukan untuk menemukan masalah dalam sebuah program walaupun metode pengujian yang diberikan tidak dapat menjamin untuk menemukan semua masalah. Oleh karena itu, selama suatu perangkat lunak dibangun, berbagai metode pengujian dapat digunakan, seperti melakukan *unit testing* selama pengembangan, integrasi pengujian setelah modul dan sistem selesai, dan penerimaan pengguna untuk mengevaluasi apakah kebutuhan para pengguna telah terpenuhi.

Unit pengujian adalah jenis pengujian perangkat lunak yang melibatkan persiapan yang terdefinisi dengan baik, melakukan tes prosedural fungsi diskrit dari sebuah program yang memberikan keyakinan bahwa modul atau fungsi berperilaku sesuai dengan rancangan. Unit test yang disebut sebagai *'white-box'* tes (kontras dengan tes 'kotak hitam') karena mereka ditulis dengan pengetahuan penuh dari struktur internal fungsi dan modul di bawah tes. Unit test biasanya disiapkan oleh pengembang yang menulis kode yang diuji dan biasanya otomatis, ditulis sebagai programmer kecil yang dijalankan oleh kerangka *unit testing* (seperti J unit untuk Java atau kerangka Uji di Ruby). Tujuannya bukan untuk menguji setiap jalur eksekusi dalam unit (tes lengkap), melainkan untuk fokus tes pada bidang risiko, ketidakpastian, atau kekritisan. Setiap tes berfokus pada satu aspek dari kode (uji satu hal) dan biasanya diatur dalam tes kesamaan.

Beberapa manfaat dari *unit testing* meliputi:

1. **Dokumentasi** : Penyusunan tes untuk sistem tertentu menyediakan jenis dokumentasi pemrograman melihat perilaku yang diharapkan dari fungsi dan modul dan memberikan contoh bagaimana berinteraksi dengan komponen kunci.
2. **Ketelitian** : Unit pengujian mendorong gaya pemrograman modul kecil, masukan yang jelas dan keluaran dan lebih sedikit ketergantungan antar komponen. Kode yang ditulis untuk memudahkan pengujian (*testability*) mungkin lebih mudah untuk membaca dan mengikuti.
3. **Regresi** : Secara bersamaan, tes dapat dilaksanakan sebagai uji regresi dari sistem. Otomatisasi tes berarti bahwa setiap masalah yang disebabkan oleh perubahan pada kode dengan mudah dapat diidentifikasi. Ketika masalah ditemukan, tes baru dapat ditulis untuk memastikan itu akan diidentifikasi di masa depan.

Unit test biasanya ditulis setelah program selesai. Sebuah alternatif yang populer adalah untuk mempersiapkan tes sebelum fungsionalitas dari aplikasi disiapkan, yang disebut Tes Pertama atau Test-Driven Development (TDD). Dalam metode ini, serangkaian tes yang telah ditulis kemudian dilaksanakan. Jika terjadi kegagalan pada fungsionalitas aplikasi, kemudian kegagalan tersebut ditulis untuk membuat tes lulus. Persiapan awal tes memungkinkan programmer untuk mempertimbangkan perilaku yang diperlukan dari program dan antarmuka dan fungsi program perlu mengekspos sebelum mereka ditulis.

Aturan Pengujian

Pada dasarnya, melakukan unit pengujian adalah mudah. Meski demikian, untuk menulis unit test yang baik dapat menjadi sulit mengingat adanya hubungan yang kompleks antara tes dengan kode yang diuji. Pengujian metaheuristik dan algoritma Komputasi Intelijen juga cukup sulit karena harus memberikan semacam hasil bahkan ketika diimplementasikan dengan masalah. Pedoman berikut dapat membantu ketika unit pengujian algoritma :

1. ***Start Small*** : Melakukan *unit test* beberapa kali akan lebih baik daripada tidak ada *unit test*. Setiap tes tambahan dapat meningkatkan kepercayaan dan kualitas kode. Untuk implementasi algoritma yang ada, dapat dimulai dengan menulis tes untuk perilaku kecil dan sederhana dan perlahan-lahan membangun sebuah *test suite*.
2. ***Test One Thing*** : Setiap pengujian harus fokus pada verifikasi perilaku salah satu aspek dari satu unit kode.
3. ***Test Once*** : Sebuah perilaku atau harapan hanya perlu diuji sekali, jangan mengulang ujian setiap kali suatu unit tertentu diuji.
4. ***Don't Forget The I/O*** : Ingatlah untuk menguji *input* dan *output* dari unit kode, khususnya pra-kondisi dan pasca-kondisi. Hal ini dapat memudahkan untuk fokus pada poin keputusan dalam unit dan melupakan tujuan utamanya.
5. ***Write Code For testability*** : Pengujian harus membantu membentuk kode yang diuji. Tulis fungsi kecil atau modul, berpikir tentang pengujian saat menulis kode (atau menulis tes pertama), dan kode refactor (kode update setelah fakta) untuk membuat lebih mudah dalam pengujian.
6. ***Function Independence*** : Mencoba untuk membatasi ketergantungan langsung antara fungsi, modul, objek dan konstruksi lainnya. Hal ini terkait dengan *testability* dan menulis fungsi kecil meskipun menunjukkan batas berapa banyak interaksi yang ada antara unit kode dalam algoritma.
7. ***Test Independence*** : Tes yang dilakukan harus independen satu sama lain. Kerangka memberikan kait untuk *set-up* dan *tear-down* negara sebelum pelaksanaan setiap tes, seharusnya tidak ada perlu memiliki satu tes menyiapkan data atau negara untuk tes lainnya. Tes harus dapat melaksanakan secara mandiri dan dalam urutan apapun.
8. ***Test Your Own Code*** : Hindari menulis tes yang memverifikasi perilaku kerangka kerja atau perpustakaan kode, seperti keacakan nomor acak generator atau apakah matematika atau fungsi string berperilaku seperti yang diharapkan. Fokus pada menulis tes untuk manipulasi data yang dilakukan oleh kode yang Anda tulis.

9. ***Probabilistic Testing*** : metaheuristik dan Komputasi Intelijen algoritma umumnya menggunakan stokastik atau keputusan probabilistik. Ini berarti bahwa beberapa perilaku yang tidak deterministik dan lebih sulit untuk menguji. Seperti contoh, menulis tes probabilistik untuk memverifikasi bahwa proses seperti berperilaku sebagaimana dimaksud. Mengingat bahwa tes probabilistik lebih lemah daripada tes deterministik, pertimbangkan untuk menulis tes deterministik pertama. Sebuah perilaku probabilistik dapat dibuat deterministik dengan mengganti nomor acak dengan *proxy* yang mengembalikan nilai-nilai deterministik, disebut mock. Tingkat pengujian mungkin memerlukan dampak lebih lanjut untuk kode asli untuk memungkinkan modul tergantung dan benda-benda untuk dipermainkan.
10. **Consider test First** : Menulis tes pertama dapat membantu memberi harapan ketika menerapkan algoritma dari literatur, dan membantu untuk memperkuat pikiran ketika mengembangkan atau *prototype* ide baru

2.8.2 Pengujian Akurasi Algoritma

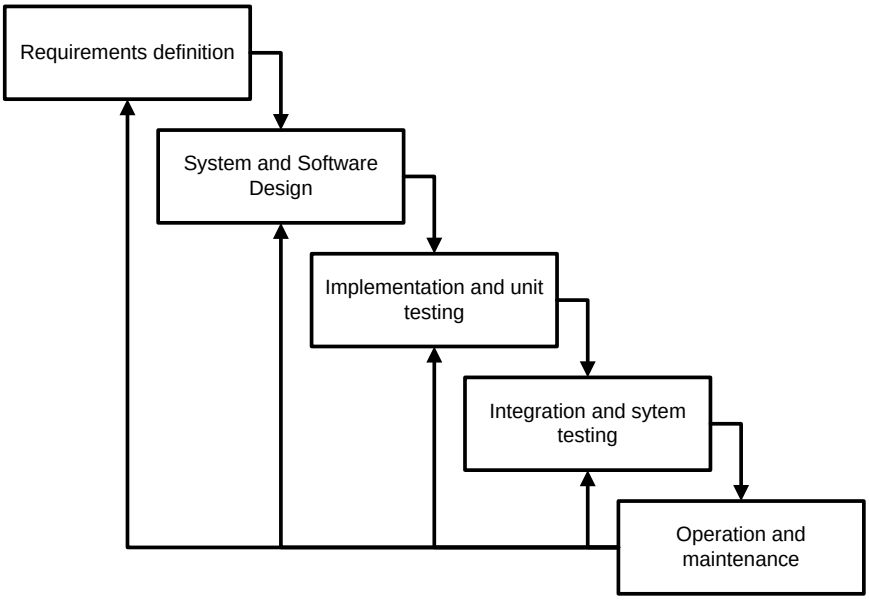
Untuk melakukan pengujian akurasi algoritma ada beberapa metode yang bisa digunakan sesuai kondisi dan studi kasus yang diteliti seperti metode berikut.

1. *Cross Validation* Pengujian standar yang dilakukan untuk memprediksi *error rate*. Data training dibagi secara random ke dalam beberapa bagian dengan perbandingan yang sama kemudian *error rate* dihitung bagian demi bagian, selanjutnya hitung rata-rata seluruh *error rate* untuk mendapatkan *error rate* secara keseluruhan.
2. *Confusion Matrix Method* ini hanya menggunakan tabel matriks terdapat dilampiran jika dataset hanya terdiri dari dua kelas, kelas yang satu dianggap sebagai positif dan yang lainnya negatif (Bramer,2007).
3. Pengujian secara manual yaitu dengan cara melakukan pembagian. hasil yang ditemukan sistem dibagi dengan fakta yang sebenarnya dan dikali 100 untuk menampilkan data dengan persentase. Kelas yang terdapat pada dataset bisa lebih dari dua.

Pada penelitian ini, untuk melakukan pengujian pengujian menggunakan metode manual dengan formula Akurasi = $\frac{\text{Pola yang ditemukan sistem}}{\text{Fakta Pola dalam Sumber String}}$ x 100% dirasa paling cocok karena kelas yang terdapat pada dataset lebih dari dua dan data training yang dilakukan ditentukan sebelum melakukan pengujian.

2.9 Metode Pembangunan Perangkat Lunak

Metode yang digunakan dalam pembangunan perangkat lunak ini adalah model *Waterfall*. Model *waterfall* melakukan pendekatan secara sistematis dan berurutan. Menurut Sommerville, model *Waterfall* mengambil kegiatan proses dasar spesifikasi, pengembangan, validasi, dan evolusi yang mewakili kegiatan tersebut sebagai fase proses terpisah seperti spesifikasi persyaratan, perancangan perangkat lunak, implementasi, pengujian dan sebagainya[8]. Tahapan-tahapan dalam model *waterfall* yang disajikan dalam gambar di bawah ini.



Gambar 2.4 *Waterfall*[8]

a. *Requirements Analysis and Definition*

   Sistem layanan, kendala dan tujuan ditetapkan melalui konsultasi dengan pengguna sistem, kemudian kebutuhan tersebut ditetapkan secara rinci dan berfungsi sebagai spesifikasi sistem.

b. *Sistem and Software Design*

Proses desain sistem mengalokasikan persyaratan, baik untuk sistem perangkat keras maupun untuk perangkat lunak, dengan mendirikan sebuah arsitektur sistem secara keseluruhan atau lengkap. Desain *software* melibatkan langkah mengidentifikasi dan menggambarkan abstraksi sistem perangkat lunak yang mendasar.

c. *Implementation and Unit Testing*

Selama tahapan ini, desain perangkat lunak disadari sebagai serangkaian program atau unit program. *Unit testing* memverifikasi setiap unit sesuai spesifikasi.

d. Integration and Sistem Testing

Program diintegrasikan dan diuji sebagai sistem yang lengkap untuk memastikan bahwa persyaratan perangkat lunak telah dipenuhi. Setelah pengujian, sistem *software* diserahkan kepada pelanggan.

e. Operation and Maintenance

Pemeliharaan melibatkan pengkoreksian kesalahan yang tidak ditemukan dalam tahap awal siklus, meningkatkan implementasi unit sistem dan peningkatan sistem sebagai kebutuhan baru ditemukan.

Dalam model *waterfall* terdapat beberapa karakteristik yang menonjol dan cenderung menjadi permasalahan. Pertama, ketika masalah muncul, maka proses akan berhenti karena tidak dapat menuju ke tahapan selanjutnya. Namun, apabila terdapat kemungkinan bahwa masalah tersebut muncul akibat kesalahan dari tahapan sebelumnya, maka proses harus membenahi tahapan sebelumnya agar masalah tersebut tidak muncul. Kedua, karena pendekatannya adalah secara *sequential*, maka setiap tahap harus menunggu hasil dari tahap sebelumnya. Hal tersebut dapat menghabiskan waktu yang cukup lama, artinya bagian lain tidak dapat mengerjakan hal lain selain hanya menunggu hasil dari tahap sebelumnya.

2.10 Diagram Konteks

Diagram konteks merupakan diagram yang mengandung satu proses yang menggambarkan hubungan keterkaitan antara sistem dengan pihak-pihak diluar lingkungan sistem dan posisi sistem didalam lingkungan tersebut. Pihak-pihak tersebut merupakan pihak-pihak yang membutuhkan informasi dan data dari sistem ataupun pihak-pihak yang menjadi sumber informasi dan data bagi sistem[11].

Hubungan keterkaitannya digambarkan sebagai aliran informasi dan data yang masuk ke dalam sistem dan keluar dari sistem.

2.11 Data Flow Diagram (DFD)

DFD adalah sebuah teknik grafik yang menggambarkan aliran data dan transformasi yang digunakan sebagai perjalanan data dari masukan menuju ke keluaran. DFD dapat diartikan juga sebagai model jaringan dari sebuah sistem. DFD dapat menggambarkan proses-proses yang terjadi dan aliran data diantaranya.

DFD sering digunakan untuk menggambarkan suatu sistem yang telah ada atau sistem baru yang akan dikembangkan secara logika tanpa mempertimbangkan lingkungan fisik dimana data tersebut mengalir atau lingkungan fisik dimana data tersebut disimpan. Secara umum DFD dapat diartikan sebagai salah satu *tools* untuk analisis sistem yang dapat bermanfaat untuk menggambarkan proses, aliran data, *entity* yang terlibat serta *data store* yang digunakan dalam sistem yang dipelajari. Dengan menuangkan hasil analisis ke dalam DFD, seorang analis dapat memahami sistem yang sedang dipelajari dengan mudah dan baik.

Dibawah ini terdapat simbol-simbol untuk DFD yang diusulkan oleh Yourdon [11]:

1. Proses, suatu proses adalah kegiatan atau kerja yang dilakukan oleh orang, prosedur atau alat yang digunakan untuk mentransformasikan data.
2. *Data Flow* (Arus Data), data yang mengalir dengan arah tertentu dari asal ke tujuan. Data yang mengalir dapat berupa dokumen, surat atau bentuk lainnya.
3. *Data Store* (Penyimpanan Data), digunakan untuk menyimpan dan mengambil data oleh proses. Data yang disimpan dapat berupa data yang terkomputerisasi maupun tidak terkomputerisasi.

4. *Source* atau *destination* atau dikenal juga dengan *external entity*, berupa orang, organisasi atau sistem lain yang berada diluar batas sistem yang berinteraksi dengan sistem yang sedang dikembangkan.

Dalam DFD terdapat beberapa aturan dasar penyusunan DFD yang dapat membantu untuk mempermudah penggambaran diagram arus data, diantaranya :

1. Setiap data yang dihasilkan atau keluar dari prosesharus didasarkan pada data yang masuk ke proses tersebut.
2. Semua aliran data yang dihasilkan harus diberi nama, nama yang diberikan harus mencerminkan aliran data antara proses, *data store* dan *entity* yang ada.
3. Hanya data yang diperlukan untuk melakukan suatu proses saja yang harus digunakan sebagai *input* suatu proses.
4. Suatu proses harus tergantung pada *input* dan *output* yang masuk ke dalam proses itu saja, tidak perlu memperhatikan apa yang terjadi pada proses lainnya.
5. Setiap proses yang ada harus merupakan proses yang berjalan terus menerus, setiap proses harus diasumsikan siap menangani fungsi atau kerja setiap proses.

Agar representasi sistem dalam DFD mudah dipahami, maka DFD disusun dalam bentuk bertingkat (*leveled*) yang merupakan rincian lanjut dari proses pada *level* yang sebelumnya. Dimulai dari tingkat yang tertinggi dilakukan identifikasi proses apa saja yang ada dan data apa saja yang mengalir antar proses yang ada.

Selanjutnya, dilakukan analisis terhadap setiap proses yang ada pada *level* tersebut, apakah sudah merupakan proses yang melakukan satu fungsi saja atau masih melakukan beberapa fungsi yang berbeda. Jika proses yang dianalisis masih melakukan lebih dari satu fungsi yang berbeda, makaperlu dilakukan pemecahan terhadap proses tersebut menjadi *level* lebih tinggi. Demikian seterusnya sampai didapat semua proses yang ada pada *level* paling tinggi dilakukan satu fungsi saja.

Jika DFD untuk suatu sistem yang dianalisis telah dibuat, sebelum melanjutkan kegiatan lainnya selalu dilakukan evaluasi atau pemeriksaan terhadap DFD tersebut. Pemeriksaan dilakukan terhadap munculnya kesalahan yang dapat diakibatkan oleh salah gambar, tidak konsistennya penggunaan simbol dan analisis terhadap proses yang terjadi.

2.12 *QT Creator*

QT Creator merupakan *cross-platdorm IDE* (*Integrated Development Environment*) yang lengkap untuk pengembangan aplikasi dengan target berbagai *platform* desktop dan berbagai *platform mobile*. *QT Creator* dapat diinstall pada Linux, OS X dan Micorosoft Windows[13].

QT Creator merupakan IDE yang menyediakan *tools* untuk mendisain dan mengembangkan aplikasi menggunakan *framework* aplikasi QT. QT merancang tampilan dan mengembangkan aplikasi satu kali, kemudian menyebarkan aplikasi tersebut ke berbagai *platform desktop* dan *platform mobile*. *QT Creator* menyediakan *tool-tool* menyeluruh dalam mengembangkan aplikasi dimulai dari memulai projek dan menyebarkan aplikasi ke berbagai target *platform*.

Salah satu kelebihan *QT Creator* adalah memungkinkan sebuah team pengembang aplikasi bekerja sama mengembangkan aplikasi dari berbagai *platform* dengan menggunakan *tool-tool* dan *debugging* yang sama.

2.13 C++

Bahasa C++ adalah bahasa komputer atau bahasa pemrograman yang merupakan pengembangan dari bahasa C kemudian disempurnakan oleh Bjarne Stroustrup pada tahun 1980 menjadi "*C with classes*" dan berganti nama menjadi C++ pada tahun 1983. Bahasa C++ diciptakan untuk mendukung pemrograman berorientasi pada objek (*Object Oriented Pragramming*/OOP) yang tidak dimiliki C. Sementara itu, C merupakan bahasa pemrograman terbaik dilingkungannya, bahasa ini tidak memiliki kemampuan OOP. Reputasi C tidak diragukan lagi dalam menghasilkan program .EXE berukuran kecil, eksekusi yang cepat, antarmuka (*interfacing*) yang sederhana dengan bahasa lain dan fleksibilitas pemrograman. Yang membuat C tampak sukar dipelajari diperkirakan karena tiadanya pemeriksaan tipe. Sebagai contoh, dapat mencampur bilangan bulat dengan string untuk menghasilkan karakter. Namun, justru disitu letak fleksibilitas C, dapat mengolah data C sebebas mengolah data dalam bahasa *assembly* [14]. Pada

penelitian ini, bahasa pemrograman yang digunakan untuk membangun aplikasi perbandingan algoritma *Reverse Colussi* dan *Zhu Takaoka* yaitu C++.

2.14 JavaScript

JavaScript adalah bahasa (pemograman) yang hebat, meskipun cenderung sulit untuk dipahami, akan tetapi kemampuan inti yang dimiliki oleh JavaScript sangat menarik untuk didalami. dengan Javascript anda dapat membuat aplikasi - aplikasi hebat seperti Google Maps. JavaScript telah merubah pandangan dunia terhadap internet (Web) [15].

Keunggulan javascript yang juga dikenal dengan nama ECMAScript yaitu dapat berjalan di semua *platform* dengan *browser* yang mendukung JavaScript, dan hampir semua *platform* dan *browser* saat ini mendukung JavaScript. Contoh dari aplikasi yang dibangun dengan JavaScript adalah Google Maps yang dapat berjalan di atas Linux, Windows, dan Mac OS. JavaScript juga semakin dan terus berkembang, seperti pertumbuhan pustaka (*library*) yang memudahkan untuk menavigasi dokumen, memlilih elemen DOM, membuat animasi, menangani *event* dan mengembangkan aplikasi Ajax. JavaScript adalah bahasa pemograman *client-side* yang *cross-platform* (berjalan di banyak *platform*) dan bersifat bebas (untuk dimodifikasi dan gratis tentunya) juga diadopsi secara *universal*.

2.15 *Code Editor*

Code Editor adalah sebuah program editor teks yang dirancang khusus untuk membuat atau mengedit *source code* (code program) dari komputer oleh programmer. *Code Editor* bisa jadi merupakan sebuah *stand alone application* atau dapat dibangun menjadi sebuah *Integrated Development Environment* (IDE). *Code Editor* merupakan alat pemrograman yang paling mendasar. Contohnya adalah atom, bluefish, Bracket, ICEcoder, Notepad ++, sublime, dan lain-lain[16].

Untuk membangun aplikasi perbandingan algoritma *Reverse Colussi* dan *Zhu Takaoka* menggunakan code editor yaitu : Sublime 3.0, Notepad ++.

2.16 *Web browser*

Web browser adalah aplikasi perangkat lunak yang digunakan untuk mencari, mengambil dan juga menampilkan informasi di *World Wide Web*, termasuk halaman Web, gambar, video dan file lainnya. Sebagai model klien / server, klien pada komputer meminta informasi dengan mengontak server Web melalui *Web browser*. *Web server* mengirimkan informasi kembali ke browser Web yang menampilkan hasilnya pada komputer atau internet perangkat yang mendukung browser. Contohnya adalah Microsoft Internet Explorer, Google Chrome, Apple Safari dan Opera, Netscape Navigator, Mozilla Firefox[17].

Pada penelitian ini, keluaran dari aplikasi berupa file yang bisa dibuka oleh *web browser*, dan *web browser* yang digunakan pada penelitian ini adalah Google Chrome.