Penerapan Algoritma *String Matching* Dalam *Personal Assistant* Google

Ferdy Irawan Firdaus - 13519030
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13519030@std.stei.itb.ac.id

***Abstract*—Teknologi berkembang begitu pesat dan memberi banyak kemudahan bagi kehidupan manusia disemua bidang kehidupan. Salah satunya adalah teknologi *personal assistant* seperti Google *assistant* buatan Google, di mana Google *assistant* dapat diterapkan di *smartphone, smart TV, smart home*, dan lainnya. Dengan adanya Google *assistant,* pengguna bisa mengontrol sesuatu (*smartphone, smart TV, smart home*) hanya dengan suara, misalnya membuka suatu aplikasi tertentu di dalam *smartphone*, hingga menyalakan lampu di dalam rumah yang sudah mendukung *smart home system*. Pertama Google akan menerima masukan dari pengguna melalui suara atau proses *voice recognition* lalu perintah yang berupa suara akan diubah menjadi teks atau proses *speech-to-text*, kemudian Google *assistant* menerapkan *string matching* sebelum menjalankan perintah (membuka aplikasi tertentu) dari pengguna. Dengan pattern adalah nama semua aplikasi yang terdapat di dalam *smartphone*, maka sebenarnya cukup dengan memanggil atau input *pattern* tersebut saja sehingga Google sudah bisa menjalankan perintah, namun bisa juga dalam teks *input* ditambahkan kata lainnya seperti "tolong buka youtube", maka proses *string matching* akan dilakukan pencocokkan antara teks tersebut dengan *pattern* adalah nama salah satu aplikasi yaitu "youtube". Setelah itu, jika ditemukan *pattern* tersebut di dalam teks yang diinput dari pengguna maka akan dibuka aplikasi Youtube.**

Keywords*—google; string matching; Boyer Moore; Knuth-Moris-Path; Brute Force; assistant

I. Pendahuluan

Dewasa ini perkemabangan teknologi berkembang sangat pesat, banyak sekali teknologi atau inovasi yang diciptakan atau dibuat manusia dalam rangka untuk mempermudah kehidupan manusia dalam segala aspek kehidupan, seperti mobil yang dapat menyetir secara otomatis, robot-robot yang sudah banyak dipakai di luar negeri untuk asistan rumah tangga, maupun teknologi seperti *personal assistant* yang diimplementasikan dalam *smartphone, smart TV*, dan *smart home system*. Dalam *smartphone, smart TV*, dan *smart home system* merupakan contoh dari implementasi dari asisten pribadi cerdas (*intelligent personal assistant*).

Smart home sejatinya merupakan cerminan rumah berbasis teknologi. Di mana teknologi yang disematkan berfungsi untuk mengatur dan mengontrol rumah secara otomatis dari jarak jauh, dari mana saja dan kapan saja. Pengaturan dilakukan tentunya dengan mengandalkan koneksi internet atau *wifi* dengan perangkat seluler (seperti *smarthphone*) atau bisa juga dengan lisan seperti "Ok Google" sebagai media/remotnya, oleh karena itu sistem rumah pintar atau *smart home system* menghubungkan seluruh perangkat yang ada di rumah, sehingga memungkinkan penghuninya untuk mengontrol beragam fungsi seperti akses keamanan ke rumah, suhu ruangan, pencahayaan, mengaktifkan AC, mematikan TV bahkan mengganti warna lampu yang ada di suatu ruangan.

Gambar 1. Tampilan Google *assistant*

Seperti yang kita ketahui, dahulu televisi hanya bisa digunakan untuk melihat acara televisi di sebuah stasiun televisi dengan media remot, namun seiring berkembanganya teknologi televisi sudah dilengkapi dengan fitur *smart TV* seperti *video streaming* (Youtube dan Netflix), pesan instant dan *social media* hanya dengan memanfaatkan internet maupun *wifi*, selain itu untuk *smart TV* yang berbasis Android juga bisa dikontrol dengan media suara seperti "Ok Google" lalu mengucapkan apa yang ingin *assistant* tersebut lakukan seperti "Buka Netflix". Hal tersebut sebenarnya sudah dikenal sejak lama dalam *smartphone*, laptop, dan tablet baik sistem operasi Android, Windows maupun IOS. Setiap sistem operasi mempunyai *intelligent personal assistant* masing-masing, seperti pada Google terdapat Google *assistant*, IOS memiliki Siri, dan Windows memiliki Cortana. Masing-masing dapat mengenali atau mendukung berbagai bahasa namun hanya Google *assistant* yang mendukung Bahasa Indonesia. Seringkali pada saat menggunakan Google *assistant* didapati beberapa kali pengenalan kata yang kurang tepat sehingga terjadi kesalahan melakukan aksi dari perintah tersebut atau

beberapa kali tidak melakukan apa-apa saat kita memanggil "Ok Google" dengan frasa yang lain, namun frasa tersebut bisa diganti.

II. Teori Dasar

String matching atau pencocokan *string* merupakan salah satu metode pencarian yang paling populer. Algoritma *string matching* (pencocokan string) adalah algoritma yang digunakan untuk melakukan pencarian semua kemunculan string pola atau pendek (*pattern*) yang muncul dalam teks (*text*). Teks (*text*) yaitu *long string* yang panjangnya n karakter. *Pettern* yaitu string dengan panjang m karakter (asumsi m <<< n). *String* adalah tipe data untuk teks yang merupakan gabungan huruf, angka, *whitespace* (spasi), dan berbagai karakter. Terdapat dua konsep *string*, yaitu *suffix* dan *prefix*. *Prefix* adalah huruf atau beberapa karakter (*substring*) yang ada di awal sebuah kata dan dapat merubah arti kata yang asli. *Suffix* adalah huruf atau beberapa karakter (*substring*) yang ada di akhir sebuah kata dan dapat merubah arti kata yang asli. Misalkan S adalah sebuah *string* dengan ukuran (*size*) m, maka *prefix* adalah *substring* dari S mulai indeks 0 sampai k dan *suffix* adalah *substring* dari S mulai indeks k sampai m-1, dengan k adalah sebarang indeks di antara 0 dan m-1. Contoh: string S yaitu "algoritma", maka semua kemungkinan *prefix* dari S adalah "a", "al", "alg", "algo", "algor", "algori", "algorit", "algoritm", dan "algoritma". Semua kemungkinan suffix dari S adalah "a", "ma", "tma", "itma", "ritma", "oritma", "goritma", "lgoritma", dan "algoritma".

Dalam keberjalanannya terdapat beberapa algoritma yang dapat digunakan untuk mengimplementasikan metode *string matching*. Berikut ini akan dibahas tiga algoritma yang populer digunakan untuk mengimplementasikan *string matching*.

A. Algoritma Brute Force atau Starightforward Matching

Proses kerja algoritma Brute Force adalah sebagai berikut: Awalnya dibandingkan karakter pertama dari *pattern* dengan karakter pertama yang ada di dalam teks. Jika kedua karakter sama maka pengecekan akan dilanjutkan ke karakter selanjutnya (satu indeks berikutnya) baik dari teks maupun *pattern*, dilakukan sampai keseluruhan karakter di dalam string *pattern* telah dicocokkan dengan karakter pada teks atau jika menemukan karakter yang tidak sama. Jika menemukan karakter yang tidak sama atau cocok maka pattern akan bergeser satu indeks ke kanan dan dilakukan pengecekan mulai dari awal karakter *pattern*. Contoh:

Teks: KAKIKU KAKU

Pattern: KAKU

	K	A	K	I	K	U		K	A	K	U
1	K	A	K	U							
2		K	A	K	U						
3			K	A	K	U					
4				K	A	K	U				
5					K	A	K	U			
6						K	A	K	U		
7							K	A	K	U	
8								K	A	K	U

Jumlah perbandingan: 32 kali perbandingan

Dengan panjang atau jumlah karakter dari teks adalah m dan panjang atau jumlah karakter dari *pattern* adalah m, maka ada beberapa kemungkinan kasus kompleksitas dari algoritma Brute Force.

- *Worst case* atau kemungkinan terburuk dari algoritmna Brute Force adalah dengan jumlah perbandingan m(n-m+1) = O(mn). Contoh:

  Teks: aaaaaaaaaaaaaaaaaaaaaaaaab

  Pattern: aaab

- *Best case* atau kemungkinan terbaik dari algoritmna Brute Force adalah ketika karakter pertama *pattern* P tidak pernah sama dengan karakter teks T yang dicocokkan. Kompleksitas kasus terbaik algoritma Brute Force adalah O(n). Contoh:

  Teks: Bagian dari internet yang mengandung informasi-informasi disebut www

  Pattern: www

- *Average case* atau kasus rata-rata memiliki kompleksitas waktu O(m+n). Hal ini berarti dalam kasus rata-rata, pencarian *string* menggunakan algoritma Brute Force berjalan dengan sangat cepat. Contoh:

  Teks: strategi algoritma sangat menyenangkan untuk dipelajari

  Pattern: tidur

Algoritma Brute Force sangat cepat jika diimplementasikan untuk pencarian string di mana karakter dari teks sangat besar misalnya A…Z, a…z, 1…9, dan lainnya. Namun algoritma Brute Force akan sangat lambat jika diimplementasikan dalam teks yang memiliki variasi karakter atau alfabet yang sedikit misalnya dalam teks hanya terdapat karakter 0 dan 1 (seperti binary files, image files, dan sebagainya).

B. Algoritma Knuth-Morris-Pratt (KMP)

Algoritma Knuth-Morris-Pratt dikembangkan secara terpisah oleh James H. Morris Bersama Vaughan R. Pratt pada tahun 1966 dan Donald E-Knuth pada tahun 1967 namun dipublikasikan Bersama pada tahun 1977.

Jika pada algoritma Brute Force diamati lebih mendalam, maka akan didapat suatu *clue* bahwa dengan mengingat berapa perbandingan yang dilakukan sebelum ditemukan karakter yang tidak cocok maka dapat menignkatkan beberapa pergeseran sekaligus, hal ini akan sangat menghemat perbandingan dan kompleksitas waktunya, yang selanjutnya akan meningkatkan kecepatan dalam pencarian *string*.

Algoritma Knuth-Morris-Pratt atau biasa disingkat algoritma KMP adalah algoritma *string matching* yang melakukan pengecekan dari kiri ke kanan *string* (mulai indeks 0 sampai m-1 pada string dengan panjang m), hal ini sama dengan algoritma Brute Force. Hal yang membedakan algoritma KMP dengan algoritma Brute Force adalah pergeseran yang dilakukan pada algoritma KMP jauh lebih efisien dari algoritma Brute Force. Algoritma Brute Force melakukan pergeseran posisi pattern satu persatu namun algoritma KMP tidak melakukan pengecekan dengan melakukan penggeseran satu persatu, tetapi menggunakan *prefix* terbesar *pattern* yang juga merupakan *suffix* dari *pattern* tersebut untuk dipakai sebagai acuan mengetahui jarak paling jauh untuk melakukan pergeseran pengecekan *pattern. Prefix* terbesar P[0..j-1] dan juga merupakan *suffix* P[1…j-1] dimana j adalah indeks dari *pattern* di mana terjadi *mismatch* atau ketidakcocokan karakter *pattern* pada indeks tersebut dengan teks. Contoh:

	0	1	2	3	4	5	6	7	8	9	
T:	.	.	a	b	a	a	b	x	.	.	
			0	1	2	3	4	5			
P:			a	b	a	a	b	c			j = 5

- Semua *prefix* dari P[0…4] = “abaab” adalah “a”, “ab”, “aba”, “abaa”, dan “abaab”
- Semua *suffix* dari P[1…4] = “baab” adalah “b”, “ab”, “aab”, dan “baab”.
- *Prefix* terbesar dari *pattern* P yang juga merupakan *suffix* dari *pattern* P adalah “ab” yang memiliki panjang 2, maka set nilai j yang baru (j_*new*) dengan 2. Dapat dihitung juga jumlah pergeseran yang harus dilakukan yaitu sebanyak *length*(abaab) – *length*(ab) = 5 – 2 = 3 pergeseran. Setelah dilakukan pergeseran:

	0	1	2	3	4	5	6	7	8	9	10	
T:	.	.	a	b	a	a	b	x	.	.	.	
			0	1	2	3	4	5				
P:			a	b	a	a	b	c				j = 5
						0	1	2	3	4	5	
P:						a	b	a	a	b	c	j_*new* = 2

Pada dasarnya algoritma KMP berusaha untuk melewati proses pengecekan yang tidak diperlukan atau ‘percuma’ dan dapat dipastikan tidak cocok jika pengecekan tersebut dilakukan. Maka pengecekan seperti itu tidak perlu dilakukan atau bisa dilewati sehingga proses pencocokan *string* dapat dilakukan secara lebih efisien dan cepat.

Di dalam algoritma KMP terdapat fungsi pinggiran (*border function* atau nama lainnya *failure function* yang disingkat *fail*), KMP memproses *pattern* untuk menemukan *prefix* yang cocok dari *pattern* dengan *pattern* itu sendiri. J merupakan posisi atau indeks *mismatch* dalam *pattern* dan k adalah posisi sebelum *mismatch* terjadi atau k = j – 1. Fungsi pinggiran b(k) dapat diartikan sebagai ukuran terbesar *prefix pattern* P[0…k] yang juga merupakan *suffix* dari *pattern* P[1…k]. Contoh fungsi pinggiran:

P: abaabc

j: 012345

- Jika j = 0, maka tidak ada indeks yang dimulai dari 0 dan berakhir pada -1. Sehingga nilai k dan b(k) untuk j = 0 tidak ada atau bisa ditandai dengan strip “-“.
- Jika j = 1 dan k = 1 – 1 = 0, maka *prefix* P[0…0] adalah “a” namun tidak ada *suffix* yang dapat dibentuk dari *pattern* P[1…0], maka b(k) bernilai 0 karena tidak ada prefix yang juga merupakan *suffix* dari *pattern* P.
- Jika j = 2 dan k = 2 – 1 = 1, maka *prefix* dari *pattern* P[0..1] = “ab” adalah “a” dan “ab”, *suffix* dari P[1..1] = “b” adalah “b“, sehingga tidak ada *prefix* dan juga merupakan *suffix* dari P, mengakibatkan b(k) bernilai 0.
- Jika j = 3 dan k = 3 – 1 = 2, maka *prefix* dari *pattern* P[0..2] = “aba” adalah “a”, “ab” dan “aba”, *suffix* dari P[1..2] = “ba” adalah “a” dan “ba“, sehingga *prefix* terbesar dari *pattern* P yang juga merupakan *suffix* dari *pattern* P adalah “a” dengan panjang 1 karakter, jadi nilai b(k) adalah 1.
- Jika j = 4 dan k = 4 – 1 = 3, maka *prefix* dari *pattern* P[0..3] = “abaa” adalah “a”, “ab”, “aba” dan “abaa”, *suffix* dari P[1..3] = “baa” adalah “a”, “aa” dan “baa“, sehingga *prefix* terbesar dari *pattern* P yang juga merupakan *suffix* dari *pattern* P adalah “a” dengan panjang 1 karakter, jadi nilai b(k) adalah 1.
- Jika j = 5 dan k = 5 – 1 = 4, maka *prefix* dari *pattern* P[0..4] = “abaab” adalah “a”, “ab”, “aba”, “abaa” dan “abaab”, *suffix* dari P[1..4] = “baab” adalah “b”, “ab”, “aab” dan “baab“, sehingga *prefix* terbesar dari *pattern* P yang juga merupakan *suffix* dari *pattern* P adalah “ab” dengan panjang 2 karakter, jadi nilai b(k) adalah 2.

Hasil dari pencarian tersebut dapat direpresentasikan ke dalam tabel

j	0	1	2	3	4	5
P[j]	a	b	a	a	b	c
k	-	0	1	2	3	4
b(k)	-	0	0	1	1	2

Secara sistematis, Langkah-langkah yang dilakukan algoritma Knuth-Morris-Pratt pada saat melakukan pencocokan *string* adalah sebagai berikut:

1. Algoritma Knuth-Morris-Pratt mulai mencocokkan karakter awal *pattern* dengan karakter awal teks.
2. Dari kiri ke kanan, algoritma KMP akan mencocokkan karakter per karakter *pattern* dengan teks bersesuaian, samapi salah satu kondisi di bawah ini terpenuhi:
   - Karakter di *pattern* dan teks yang dibandingkan atau dicocokkkan tidak cocok (*mismatch*). Jika *mismatch* terjadi maka akan dilakukan pergeseran dan pemeriksaan karakter pada *pattern* dilakukan mulai indeks j_*new* yang didapat dari fungsi pembatas (b(k)) dengan karakter pada teks yang terjadi *mismatch.*
   - Semua karakter di *pattern* sudah selesai dicocokkan dan ditemukan kecocokan antara *pattern* dengan teks maka akan mengembalikan posisi atau indeks ditemukannya *pattern* di dalam teks.
3. Langkah 2 diulangi sampai *pattern* berada pada ujung teks dan jika sampai ujung teks tidak ditemukan kecocokan maka *pattern* tersebut tidak ada di dalam teks.

Contoh algotima KMP:

T:	a	b.	a	b	a	a	b	a	a	b	c
P:	a	b	a	a	b	c					
			a	b	a	a	b	c			
						a	b	a	a	b	c

Jumlah perbandingan karakter: 16 kali perbandingan

Dengan panjang atau jumlah karakter dari teks adalah m dan panjang atau jumlah karakter dari *pattern* adalah m, maka kompleksitas waktu algoritma KMP adalah sebagai berikut:

- Menghitung fungsi pinggiran: O(m)
- Pencarian *string*: O(n)
- Kompleksitas waktu algoritma KMP adalah O(m+n), yang mana sangat cepat dibandingkan dengan algoritma Brute Force.

Kelebihan algoritma KMP adalah algoritma ini tidak membutuhkan berjalan mundur dalam teks *input* yang mana hal ini membuat algoritma ini baik untuk pemrosesan *file* yang sangat besar yang dibaca dari perangkat eksternal atau melalui sebuah aliran jaringan. Sedangkan kekurangan dari algoritma KMP adalah KMP tidak bekerja dengan baik seiring dengan meningkatnya ukuran dari *string*. Lebih banyak kemungkinan *mismatch* atau ketidakcocokan. Ketidakcocokan cenderung terjadi di awal *pattern*, tetapi KMP lebih cepat ketika *mismatch* atau ketidakcocokan terjadi nantinya.

C. Algoritma Boyer-Moore

Algoritma Boyer-Moore merupakan algoritma yang bisa dikatakan memiliki pendekatan berbeda dari algoritma Brute Force dan juga algoritma Knuth-Morris-Pratt (KMP) di mana algoritma ini didasarkan pada dua Teknik, yaitu sebagai berikut:

- Teknik *looking-glass*

  Algoritma Boyer-Moore melakukan pencarian pattern P dalam teks T dimuali dari kanan ke kiri atau bergerak mundur.

- Teknik *character-jump*

  Teknik ini digunakan berdasarkan kemunculan karakter pada teks T yang *mismatch* ketika dilakukan pengecekan dengan salah satu huruf dalam *pattern* P sehingga kemudian menyesuaikan posisi atau indeks *pattern* P agar sesuai dengan karakter tersebut. teknik ini berfungsi untuk membuang proses pengecekan yang sebenarnya tidak perlu dilakukan.

Pada dasarnya algoritma Boyer-Moore ini adalah kebalikan dari algoritma Brute Force namun dengan perbaikan dalam hal proses pergeseran *pattern*. Dengan adanya dua teknik di atas membuat algoritma ini cukup efisien dan popular untuk digunakan. terdapat tiga kasus yang mungkin dalam proses pencocokan dengan algoritma Boyer-Moore, yaitu sebagai berikut:

1. Kasus 1

   Mismatch terjadi pada teks indeks ke-i dengan karakternya adalah “x” (T[i] = x) di mana pada *pattern* P terdapat juga karakter “x” di sebelah kiri posisi pengecekan saat ini atau posisi saat *mismatch* terjadi, maka kemudian akan dilakukan teknik *character-jump* untuk meyamakan karakter “x” pada *pattern* P dengan karakter “x” pada teks T. Kemudian dilakukan pengecekan lagi dimulai dari belakang. Contoh:

			(cek)			
T:			x	b		
			i			
P:	x	a	a	b		
			j			

   Mismatch terjadi pada T[i] = x di mana karakter x terdapat juga di *pattern* P dan letaknya di sebelah kiri *mismatch* terjadi atau di sebelah kiri posisi pengecekan saat ini. Sehingga *pattern* P digeser ke kanan sampai posisi karakter “x” pada *pattern* P sejajar atau bersesauian dengan karakter “x” pada teks T dan indeks i_*new* serta j_*new* dimulai dari awal atau dari belakang lagi.

						(cek)
T:		.	x	b	?	?
						i_*new*

P:			x	a	a	b
						j_*new*

2. Kasus 2

   Mismatch terjadi pada teks indeks ke-i dengan karakternya adalah “x” (T[i] = x) di mana pada *pattern* P terdapat juga karakter “x” di sebelah kanan posisi pengecekan saat ini atau posisi saat *mismatch* terjadi namun tidak ada karakter “x” di sebelah kiri posisi *mismatch* atau pengecekan saat ini, maka tidak mungkin dilakukan teknik *character-jump* sehingga dilakukan pergeseran ke kanan sejauh satu karakter saja. Kemudian dilakukan pengecekan lagi dimulai dari belakang. Contoh:

		(cek)			
T:	.	x	b	x	
		i			
P:	a	a	b	x	
		j			

   Mismatch terjadi pada T[i] = x di mana karakter x terdapat juga di *pattern* P dan letaknya di sebelah kanan *mismatch* terjadi dan di sebelah kiri posisi pengecekan saat ini tidak terdapat lagi karakter “x”. Sehingga *pattern* P digeser ke kanan sejauh satu karakter saja dan indeks i_*new* serta j_*new* dimulai dari awal atau dari belakang lagi.

					(cek)
T:	.	x	b	x	?
					i_*new*
P:		a	a	b	x
					j_*new*

3. Kasus 3

   Mismatch terjadi pada teks indeks ke-i dengan karakternya adalah “x” (T[i] = x) di mana pada *pattern* P tidak terdapat karakter “x” di sebelah kanan maupun sebelah kiri posisi pengecekan saat ini atau posisi saat *mismatch*, maka tidak mungkin dilakukan teknik *character-jump* sehingga dilakukan pergeseran ke kanan hingga posisi karakter paling kiri *patten* P sejajar dengan karakter setelah karakter *mismatch* pada teks T (sejajar dengan T[i+1] di mana i adalah posisi *mismatch* pada teks T). Kemudian dilakukan pengecekan lagi dimulai dari belakang. Contoh:

		(cek)			
T:	.	x	b		
		i			
P:	a	a	b		
		j			

   Mismatch terjadi pada T[i] = x di mana karakter x tidak terdapat di *pattern* P baik posisinya di sebelah kanan maupun sebelah kiri posisi *mismatch* terjadi. Sehingga *pattern* P digeser ke kanan hingga karakter paling kiri *pattern* P (P[j] = a) sejajar dengan T[i+i] = b dan indeks i_*new* serta j_*new* dimulai dari awal atau dari belakang lagi.

		(cek)			(cek)
T:	.	x	b	?	?
					i_*new*
P:	a	a	a	b	B
					j_*new*

Di dalam algoritma Boyer-Moore terdapat fungsi *last occurrence* (lo), algoritma Boyer-Moore memproses karakter apa saja yang terdapat di dalam teks T (A) lalu menemukan posisi atau indeks terakhir kemunculan setiap karakter tersebut (karakter pada teks T) pada pattern P, jika tidak ditemukan atau tidak terdapat pada *pattern* P maka bisa ditandai dengan -1. Contoh fungsi *last occurrence*:

T: aabcdaabxa

Sehingga A = {a, b, c, d, x}

P: abacab

x	a	b	c	d	x
L(x)	4	5	3	-1	-1

Contoh string matching menggunakan algoritma Boyer-Moore:

T: abacaabadcabacab

A = {a, b, c, d}

P: abacab

Last occurrence:

x	a	b	c	d
L(x)	4	5	3	-1

Maka,

T	a	b	a	c	a	a	b	a	d	c	a	b	a	c	a	b
P	a	b	a	c	a	b										
		a	b	a	c	a	b									
			a	b	a	c	a	b								
				a	b	a	c	a	b							
										a	b	a	c	a	b	

											a	b	a	c	a	b

Jumlah perbandingan karakter: 13 kali perbandingan

Dengan panjang atau jumlah karakter dari teks adalah m, dengan A adalah semua karakter unik yang ada di dalam teks dan panjang atau jumlah karakter dari *pattern* adalah m, maka kompleksitas waktu algoritma Boyer-Moore adalah sebagai berikut:

- *Worst case* atau kemungkinan terburuk: O(nm+A)
- Algoritma Boyer-Moore sangat cepat jika A sangat besar atau artinya karakter yang ada di dalam teks sangat beragam (seperti teks berbaha inggris, sangat cepat dengan algoritma Boyer-Moore) dan sebaliknya, algoritma ini berjalan sangat lambat jika A sangat sedikit atau semua karakter di dalam teks hampir sama seperti pada *binary files* yang hanya ada karakter 0 dan 1 saja.

III. Pembahasan

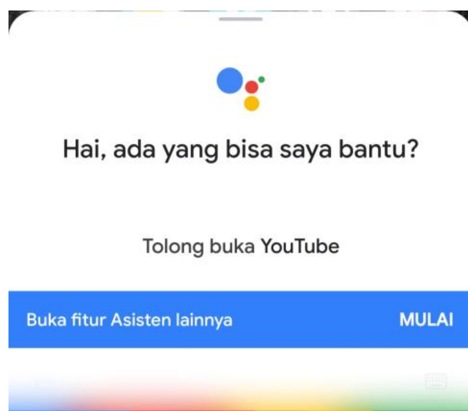
Dalam penerapannya, Google *assistant* melakukan beberapa tahap terlebih dahulu sebelum sampai pada tahap *string matching*. Tahap yang pertama adalah tahap *voice recognition*. Pengguna memberikan *input* kepada Google *assistant* berupa suara, kemudian tahap selanjutnya tahap mengubah *input* dari pengguna yang berupa suara menjadi kata-kata atau *string* biasanya disebut *speech-to-text* (fitur ini juga bisa dilihat dalam penerjemah seperti Google *translate* dan juga di Microsoft Word). Lalu proses selanjutnya adalah proses *string matching*, dan setelah dilakukan *string matching* dengan kata kunci tertentu makan akan dijalnakn sesuai perintah dari pengguna.

Sebenarnya jika meng-*input* kata kunci saja seperti "Youtube" maka Google *assistant* akan bisa langsung membuka aplikasi Youtube, hal ini berarti kata kunci atau *pattern* untuk membuka aplikasi Youtoube adalah "youtube" maka *pattern* P adalah "youtube". Untuk kepeluas tes *string matching* maka akan dibuat teks sebagai berikut:

T (Teks): tolong buka youtube

P (Pattern): youtube

Teks dan *Pattern* tersebut akan diimplementasika ke dalam tiga algoritma string matching yaitu algoritma Brute Force, Knuth-Morris-Pratt, dan Boyer-Moore.



Gambar 2. Pengguna memasukkan teks "tolong buka youtube" melalui suara

A. *Implementasi Menggunakan Algoritma Brute Force*

T: tolong buka youtube

P: youtube

Algoritma Brute Force,

t	o	l	o	n	g		b	u	k	a		y	o	u	t	u	b	e
y	o	u	t	u	b	e												
	y	o	u	t	u	b	e											
		y	o	u	t	u	b	e										
			y	o	u	t	u	b	e									
				y	o	u	t	u	b	e								
					y	o	u	t	u	b	e							
						y	o	u	t	u	b	e						
							y	o	u	t	u	b	e					
								y	o	u	t	u	b	e				
									y	o	u	t	u	b	e			
										y	o	u	t	u	b	e		
											y	o	u	t	u	b	e	
												y	o	u	t	u	b	e

Jumlah perbandingan karakter: 91 kali perbandingan

Implementasi dalam program Bahasa Python adalah sebagai berikut:

- Algoritma Brute Force

```
def BruteForce(text, pattern):
    M = len(pattern)
    N = len(text)
    for i in range(N-M+1):
        j = 0
        while j < M and pattern[j]
== text[i+j]:
            j += 1
        if j == M:
            return i
    return -1
```

- *Main program* dan *output*

```
posisi = BruteForce('tolong buka
youtube', 'youtube')
if posisi != -1:
    print("Ditemukan pada indeks ke-"
+ str(posisi))
else:
    print("Tidak ditemukan")
```

```
Ditemukan pada indeks ke-12

Process finished with exit code 0
```

Gambar 3. *Output* algoritma Brute Force

B. *Implementasi Menggunakan Algoritma Knuth-Morris-Pratt*

T: tolong buka youtube

P: youtube

Fungsi pinggiran,

- Jika j = 0, maka tidak ada indeks yang dimulai dari 0 dan berakhir pada -1. Sehingga nilai k dan b(k) untuk j = 0 tidak ada atau bisa ditandai dengan strip "-".
- Jika j = 1 dan k = 1 – 1 = 0, maka *prefix* P[0…0] adalah "y" namun tidak ada *suffix* yang dapat dibentuk dari *pattern* P[1…0], maka b(k) bernilai 0 karena tidak ada *prefix* yang juga merupakan *suffix* dari *pattern* P.
- Jika j = 2 dan k = 2 – 1 = 1, maka *prefix* dari *pattern* P[0..1] = "yo" adalah "y" dan "yo", *suffix* dari P[1..1] = "o" adalah "o", sehingga tidak ada *prefix* dan juga merupakan *suffix* dari P, mengakibatkan b(k) bernilai 0.
- Jika j = 3 dan k = 3 – 1 = 2, maka *prefix* dari *pattern* P[0..2] = "you" adalah "y", "yo" dan "you", *suffix* dari P[1..2] = "ou" adalah "u" dan "ou", sehingga tidak ada *prefix* dan juga merupakan *suffix* dari P, mengakibatkan b(k) bernilai 0.
- Jika j = 4 dan k = 4 – 1 = 3, maka *prefix* dari *pattern* P[0..3] = "yout" adalah "y", "yo", "you" dan "yout", *suffix* dari P[1..3] = "out" adalah "t", "ut" dan "out", sehingga tidak ada *prefix* dan juga merupakan *suffix* dari P, mengakibatkan b(k) bernilai 0.
- Jika j = 5 dan k = 5 – 1 = 4, maka *prefix* dari *pattern* P[0..4] = "youtu" adalah "y", "yo", "you", "yout" dan "youtu", *suffix* dari P[1..4] = "outu" adalah "u", "tu", "utu" dan "outu", sehingga tidak ada *prefix* dan juga merupakan *suffix* dari P, mengakibatkan b(k) bernilai 0.
- Jika j = 6 dan k = 6 – 1 = 5, maka *prefix* dari *pattern* P[0..5] = "youtub" adalah "y", "yo", "you", "yout", "youtu" dan "youtub", *suffix* dari P[1..5] = "outub" adalah "b", "ub", "tub", "utub" dan "outub", sehingga tidak ada *prefix* dan juga merupakan *suffix* dari P, mengakibatkan b(k) bernilai 0.

j	0	1	2	3	4	5	6
P[j]	y	o	u	t	u	b	e

k	-	0	1	2	3	4	5
b(k)	-	0	0	0	0	0	0

Maka proses algoritma KMP:

t	o	l	o	n	g		b	u	k	a		y	o	u	t	u	b	e
y	o	u	t	u	b	e												
	y	o	u	t	u	b	e											
		y	o	u	t	u	b	e										
			y	o	u	t	u	b	e									
				y	o	u	t	u	b	e								
					y	o	u	t	u	b	e							
						y	o	u	t	u	b	e						
							y	o	u	t	u	b	e					
								y	o	u	t	u	b	e				
									y	o	u	t	u	b	e			
										y	o	u	t	u	b	e		
											y	o	u	t	u	b	e	
												y	o	u	t	u	b	e

Jumlah perbandingan karakter: 91 kali perbandingan

Karena nilai b(k) untuk masing-masing nilai k semuanya bernilai 0 maka pergeseran dilakukan satu karakter ke kanan dan pengecekan dimulai dari indeks ke-0 lagi atau mulai dari awal lagi (pergeseran yang dilakukan sama dengan pergeseran pada algoritma Brute Force), hal ini berarti bahwa untuk kasus ini kompleksitas waktu yang dibutuhkan dalam algoritma KMP sama saja dengan waktu yang dibutuhkan dalam algoritma Brute Force.

Implementasi dalam program Bahasa Python adalah sebagai berikut:

- Fungsi pinggiran

```python
def computeFail(pattern):
    m = len(pattern)
    fail = [0] * m
    fail[0] = 0
    i = 1
    j = 0
    while i < m:
        if pattern[j] == pattern[i]:
            fail[i] = j + 1
            i += 1
            j += 1
        elif j > 0:
            j = fail[j-1]
        else:
            fail[i] = 0
            i += 1
    return fail
```

- Algoritma Knuth-Morris-Pratt

```python
def KMPMatch(text, pattern):
    M = len(pattern)
    N = len(text)

    fail = computeFail(pattern)

    i = 0
    j = 0
    while i < N:
        if pattern[j] == text[i]:
            if j == M - 1:
                return i - M + 1
            i += 1
            j += 1
        elif j > 0:
            j = fail[j-1]
        else:
            i += 1
    return -1
```

- *Main program* dan *output*

```python
posisi = KMPMatch('tolong buka
youtube', 'youtube')
if posisi != -1:
    print("Ditemukan pada indeks ke-"
+ str(posisi))
else:
    print("Tidak ditemukan")
```

```
Ditemukan pada indeks ke-12

Process finished with exit code 0
```

Gambar 4. *Output* algoritma KMP

C. Implementasi Menggunakan Algoritma Boyer-Moore

T: tolong buka youtube

A = {t, o, l, n, g, b, u, k, a, y, e, “ ”}

Catatan, “ ” merupakan karakter spasi atau *whitespace*

P: youtube

Last occurrence:

x	t	o	l	n	g	b	u	k	a	y	e	“ ”
L(x)	3	1	-1	-1	-1	5	4	-1	-1	0	6	-1

Maka algoritma Boyer Moore:

t	o	l	o	n	g		b	u	k	a		y	o	u	t	u	b	e
y	o	u	t	u	b	e												
							y	o	u	t	u	b	e					
												y	o	u	t	u	b	e

Jumlah perbandingan karakter: 9 kali perbandingan

Pada algoritma Boyer-Moore hanya terjadi dua kali pergeseran, pencocokan pertama yaitu karakter “ ” pada teks T dan karakter “e” pada *pattern* P namun karakter “ ” tidak ada di dalam *pattern* (L(x) = -1) sehingga *pattern* langsung digeser hingga karakter terakhir *pattern* P (yaitu “y”) sejajar dengan karakter setelah *mismatch* terjadi (yaitu karakter “b”). Setelah pergeseran, pencocokan kedua yaitu antara karakter “o” pada teks T dan karakter “e” pada *pattern* P, karakter “o” ada di dalam *pattern* P dan letaknya di sebelah kiri posisi *mismatch* terjadi atau pencocokan saat ini, maka *pattern* P digeser hingga karakter “o” pada teks T sejajar dengan karakter “o” pada *pattern* P. Pencocokan ketiga yaitu mulai dari belakang yaitu karakter “e” pada teks T dan karakter “e” pada *pattern* P dan cocok kemudian dilanjutkan karakter “b” dan cocok lagi, hal ini dilakukan hingga sampai karakter “y” dan hasilnya cocok maka sudaah ditemukan karakter “youtube” di dalam teks T.

Algoritma Boyer-Moore sangat efisien dan lebih cepat daripada algoritma Brute Force dan Knuth-Morris-Pratt dalam kasus ini karena karakter-karakter yang ada di dalam teks dan pattern bergam yang mengakibatkan banyak sekali nilai L(x) bernilai -1 sehingga jika ditemukan *mismatch* pada karakter di indeks j dan karakter tersebut tidak terdapat di dalam *pattern* (L(x) = -1) maka *pattern* akan langsung digeser sehingga karakter paling kiri pattern sejajar dengan karakter teks pada indeks ke-(j+1) atau karakter pada K[j+1].

Implementasi dalam program Bahasa Python adalah sebagai berikut:

- *Last occurrence*

```python
def buildLast(pattern):
    last = [-1]*128
    for i in range(len(pattern)):
        last[ord(pattern[i])] = i
    return last
```

- Algoritma *Boyer-Moore*

```python
def BM(text, pattern):
    last = buildLast(pattern)
    n = len(text)
    m = len(pattern)
    i = m-1
    if (i > n-1):
        return -1
    j = m-1
    if pattern[j] == text[i]:
        if j == 0:
            return i
        else:
            i -= 1
            j -= 1
    else:
        lo = last[ord(text[i])]
        i = i + m - min(j, 1+lo)
        j = m - 1
    while i <= n - 1:
        if pattern[j] == text[i]:
            if j == 0:
                return i
```

```
            else:
                i -= 1
                j -= 1
        else:
            lo = last[ord(text[i])]
            i = i + m - min(j, 1 + lo)
            j = m - 1
    return -1
```

- *Main program* dan *output*

```
posisi = BM('tolong buka youtube',
'youtube')
if posisi != -1:
    print("Ditemukan pada indeks ke-" +
str(posisi))
else:
    print("Tidak ditemukan")
```

```
Ditemukan pada indeks ke-12

Process finished with exit code 0
```

Gambar 5. *Output* algoritma Boyer-Moore

LINK VIDEO DI YOUTUBE

https://youtu.be/fXO4V8I8wDI

IV. KESIMPULAN

String matching memiliki manfaat yang banyak sekali salah satunya yaitu dalam personal assistan seperti Google *assistant* untuk mengenali *pattern* atau kata kunci dari teks yang di-*input* dari pengguna. Seperti yang diketahui bahwa terdapat tiga algoritma yang popular dalam *string matching* yaitu algoritma Brute Force, Knuth-Morris-Pratt, dan Boyer-Moore. Dengan mengambil satu contoh atau bukti nyata yang dapat dicoba langsung, penulis mencoba memasukkan teks "tolong buka youtube" dan *pattern*-nya adalah "youtube" (nama aplikasi di *smartphone*), maka penulis berhasil menggambarkan proses pencarian *pattern* di dalam teks dan menghitung berapa kali pencocokan dilakukan dari masing-masing algoritma.

V. UCAPAN TERIMA KASIH

Puji syukur penulis panjatkan kepada tuhan YME, berkat rahmat dan izin-nya makalah ini dapat diselesaikan tepat waktu. Rasa terima kasih juga penulis ucapkan kepada Ir. Rila Mandala, M.Eng.,Ph.D. atas bimbingannya selaku dosen mata kuliah strategi algoritma IF2210 kelas K-01 tahun ajaran 2020/2021. Penulis juga mengucapkan terima kasih kepada orang tua penulis, dan teman –teman atas segala doa, bantuan, masukan, sehingga masalah ini dapat selesai.

DAFTAR PUSTAKA

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf diakses pada tanggal 09 Mei 2021

[2] Levitin, Anany. 2011. Introduction to the Design and Analysis of Algorithms. Pearson: New York.

[3] https://catatananalgo.wordpress.com/2016/10/02/algoritma-string-matching-pencocokan-string/ diakses pada tanggal 09 Mei 2021

[4] https://dokumen.tips/documents/240-301-comp-eng-lab-iii-software-pattern-matching1-pattern-matching-dr.html diakses pada tanggal 09 Mei 2021

[5] https://inst.eecs.berkeley.edu/~cs61b/su06/lecnotes/lec28.pdf diakses pada tanggal 10 Mei 2021

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 11 Mei 2021

Ferdy Irawan Firdaus - 13519030