UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA

Hak cipta dan penggunaan kembali:

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

Copyright and reuse:

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

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

BAB II

LANDASAN TEORI

2.1. String Searching

Permasalahan pencarian s*tring* (s*tring searching*) pada dasarnya adalah mencari sebuah *string* yang terdiri dari beberapa karakter (yang biasa disebut *pattern*) dalam sejumlah besar *text*. Pencarian *string* juga biasa digunakan untuk mencari pola bit dalam sejumlah besar *file binary.* Contoh permasalahannya adalah dalam program *text editor* seperti Microsoft Word, atau dalam lingkup yang lebih besar adalah pencarian *web* seperti Google, Bing, dan Yahoo. (Hartoyo, Vembrina, & Meilana, 2010).

Proses pencocokan *string* yang merupakan bagian dalam proses pencarian *string* memegang peranan penting untuk mendapatkan dokumen yang sesuai dengan kebutuhan informasi. Pencocokan *string* secara garis besar dapat dibedakan menjadi dua yaitu pencocokan *string* secara eksak/sama persis (*exact string matching*) dan pencocokan string berdasarkan kemiripan (*inexact string matching/fuzzy string matching*). Pencocokan string berdasarkan kemiripan masih dapat dibedakan menjadi dua yaitu berdasarkan kemiripan penulisan (*approximate string matching*) dan berdasarkan kemiripan ucapan (*phonetic string matching*). Contoh *phonetic string matching* adalah kata *step* akan menunjukkan kecocokan dengan kata *step, sttep, stepp, sstep, stepe, steb*. Sedangkan bila kita menggunakan *exact string matching* kata *step* hanya akan menunjukkan kecocokan dengan kata *step* saja (Syaroni, 2005).

Studi Perbandingan Implementasi..., Vina Sagita, FTI UMN, 2012

Algoritma pencarian *string* atau sering disebut juga pencocokan *string* adalah algoritma untuk melakukan pencarian semua kemunculan *string* pendek *pattern*[0..n − 1] yang disebut *pattern* di *string* yang lebih panjang teks[0..m − 1] yang disebut teks (Wiramuda, 2008).

Hingga saat ini algoritma pencarian string terbagi atas 3 kategori berdasarkan arah pencarian *string* yaitu dari kiri ke kanan, kanan ke kiri, dan dari arah yang ditentukan secara spesifik. Metode-metode tersebut adalah (Kumara, 2009):

1. Metode pencocokan dari kiri ke kanan merupakan metode yang paling natural karena sesuai dengan arah membaca. Contoh algoritma yang membaca *string* dari kiri ke kanan adalah algoritma Brute Force dan Knuth, Morris, dan Pratt.
2. Pencocokan *string* dari kanan ke kiri merupakan metode yang dianggap paling efisien dalam praktiknya. Contoh algoritma dengan metode ini adalah algoritma Boyer-Moore, yang kemudian dikembangkan menjadi algoritma Turbo Boyer-Moore, Tuned Boyer-Moore, dan Zhu-Takaoka.
3. Pencocokan *string* dari arah yang ditentukan secara spesifik merupakan algoritma yang memiliki hasil yang paling baik secara teoritis. Contoh algoritma yang mencocokan *string* dengan metode ini adalah Colussi, Galil-Seiferas, dan Crochemore-Perrin.

UNIVERSITAS
MULTIMEDIA
NUSANTARA

2.2. Algoritma

Menurut Silvina Irwanti pada jurnalnya yang berjudul Implementasi Algoritma *Backtracking* dalam Perancangan Perangkat Lunak *Game* Anagram, kata algoritma diambil dari nama ilmuwan muslim dari Uzbekistan Abu Ja'far Muhammad bin Musa Al-Khuwārizmi (780-846M), sebagaimana tercantum pada terjemahan karyanya dalam bahasa latin dari abad ke-12 "*Algorithmi de numero Indorum*". Awalnya kata algoritma adalah istilah yang merujuk kepada aturan-aturan aritmatis untuk menyelesaikan persoalan dengan menggunakan bilangan numerik arab. Pada abad ke-18, istilah ini berkembang menjadi algoritma, yang mencakup semua prosedur atau urutan langkah yang jelas dan diperlukan untuk menyelesaikan suatu permasalahan. Pemecahan sebuah masalah pada hakekatnya adalah menemukan langkah-langkah tertentu yang jika dijalankan efeknya akan memecahkan masalah tersebut.

Dalam matematika dan komputasi, algoritma atau algoritme merupakan kumpulan perintah untuk menyelesaikan suatu masalah. Kompleksitas dari suatu algoritma merupakan ukuran seberapa banyak komputasi yang dibutuhkan algoritma tersebut untuk menyelesaikan masalah. Secara informal, algoritma yang dapat menyelesaikan suatu permasalahan dalam waktu yang singkat memiliki kompleksitas yang rendah, sementara algoritma yang membutuhkan waktu lama untuk menyelesaikan masalahnya mempunyai kompleksitas yang tinggi (Simanullang, 2011).

Ada 5 komponen utama dalam algoritma yaitu *finiteness*, *definiteness*, *input*, *output* dan *effectiveness* (Irwanti, 2010):

1. *Finiteness*.

   Sebuah algoritma harus selalu berakhir setelah sejumlah langkah berhingga.

2. *Definiteness*.

   Setiap langkah dari sebuah algoritma harus didefinisikan secara tepat, tindakan yang di muat harus dengan teliti dan sudah jelas ditentukan untuk setiap keadaan.

3. *Input*.

   Sebuah algoritma memiliki nol atau lebih masukan, sebagai contoh, banyaknya masukan diberikan di awal sebelum algoritma mulai.

4. *Output*.

   Sebuah algoritma memiliki satu atau lebih keluaran, sebagai contoh, banyaknya keluaran memiliki sebuah hubungan yang ditentukan terhadap masukan.

5. *Effectiveness*.

   Pada umumnya sebuah algoritma juga diharapkan untuk efektif.

2.3. Algoritma Boyer-Moore

Menurut Edward Rompah pada artikelnya yang membahas tentang algoritma Boyer-Moore, algoritma Boyer-Moore dipublikasikan oleh Robert S. Boyer, dan J. Strother Moore pada tahun 1977. Ide utama dari algoritma ini adalah dengan melakukan pencocokan dari paling kanan *string* yang dicari. Dengan menggunakan algoritma ini, secara rata-rata proses pencarian akan lebih cepat dibandingkan dengan proses pencarian lainnya. Ide dibalik algoritma ini adalah

bahwa dengan memulai pencocokan karakter dari kanan, dan bukan dari kiri, maka akan lebih banyak informasi yang didapat (Mufthy, 2011).

Algoritma Boyer-Moore melakukan perbandingan dimulai dari kanan ke kiri, tetapi pergeseran *window* tetap dari kiri ke kanan. Jika terjadi kecocokan maka dilakukan perbandingan karakter teks dan karakter pola yang sebelumnya, yaitu dengan sama-sama mengurangi indeks teks dan pola masing-masing sebanyak satu (Kumara, 2009).

Pseudocode algoritma Boyer-Moore:

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

Algoritma:
  for (i := 0 to ASIZE-1)
     bmBc[i] := m;
  endfor
  for (i := 0 to m - 2)
     bmBc[P[i]] := m - i - 1;
  endfor
```

Gambar 2.1 *Pseudocode* penghitungan tabel *bad-character* (Kumara, 2009)

UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA

```
procedure preSuffixes(
    input P : array[0..n-1] of char,
    input n : integer,
    input/output suff : array[0..n-1] of integer
)

Deklarasi:
  f, g, i: integer

Algoritma:
  suff[n - 1] := n;
  g := n - 1;
  for (i := n - 2 downto 0) {
     if (i > g and (suff[i + n - 1 - f] < i - g))
        suff[i] := suff[i + n - 1 - f];
     else
        if (i < g)
           g := i;
        endif
        f := i;
        while (g >= 0 and P[g] = P[g + n - 1 - f])
           --g;
        endwhile
        suff[i] = f - g;
     endif
  endfor
```

Gambar 2.2 *Pseudocode* perhitungan tabel *suffix* (Kumara, 2009)

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

  preSuffixes(x, n, suff);

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

Gambar 2.3 *Pseudocode* perhitungan tabel *good-suffix* (Kumara, 2009)

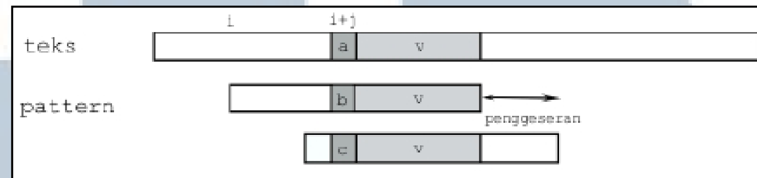
```
procedure BoyerMooreSearch(
    input m, n : integer,
    input P : array[0..n-1] of char,
    input T : array[0..m-1] of char,
    output ketemu : array[0..m-1] of boolean
)

Deklarasi:
i, j, shift, bmBcShift, bmGsShift: integer
BmBc : array[0..255] of interger
BmGs : array[0..n-1] of interger

Algoritma:
preBmBc(n, P, BmBc)
preBmGs(n, P, BmGs)
i:=0
while (i<= m-n) do
    j:=n-1
    while (j >=0 n and T[i+j] = P[j]) do
       j:=j-1
    endwhile
    if(j < 0) then
       ketemu[i]:=true;
    endif
    bmBcShift:= BmBc[chartoint(T[i+j])]-n+j+1
    bmGsShift:= BmGs[j]
    shift:= max(bmBcShift, bmGsShift)
    i:= i+shift
```

Gambar 2.4 *Pseudocode* Algoritma Boyer-Moore (Kumara, 2009)

Misalnya ada sebuah percobaan pencocokan yang terjadi pada T[i..i+n-1], dan anggap ketidakcocokan pertama terjadi diantara T[i+j] dan P[j], dengan 0<j<n. Berarti, T[i+j+1..i+n-1]=P[j+1..n-1] dan a=T[i+j]≠b=P[j]. Jika v adalah akhiran dari *pattern* setelah b dan u adalah sebuah awalan dari *pattern*, maka pergeseran-pergeseran yang mungkin adalah pergeseran *good-suffix* dan pergeseran *bad-character* (Kumara, 2009). Dalam sumber lain *Bad-Character shift* disebut *Occurence Heuristic* dan *Good-Suffix shift* disebut *Match Heuristic*. (Rompah, 2009).

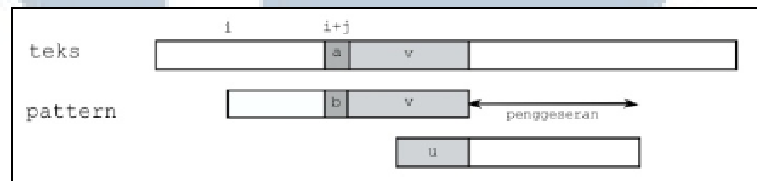
Pergeseran *good-suffix* yang terdiri atas mensejajarkan potongan T[i+j+1..i+n-1]=P[j+1..n-1]=v dengan kemunculannya paling kanan di *pattern*

yang didahului oleh karakter yang berbeda dengan P[j], Pergeseran tersebut diilustrasikan di bawah ini (Kumara, 2009):



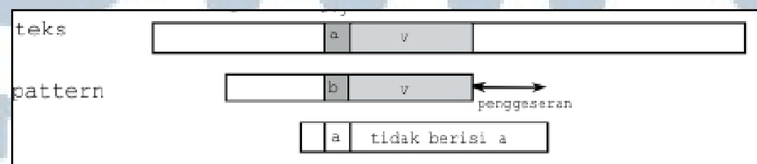
Gambar 2.5 Pergeseran *good-suffix* bila ada potongan *u* yang sama. (Kumara, 2009)

Namun, jika tidak ada potongan seperti itu, maka algoritma akan mensejajarkan akhiran dari v dari T[i+j+1..i+n-1] dengan awalan u dari *pattern* yang sama, seperti yang diilustrasikan di gambar berikut (Kumara, 2009):



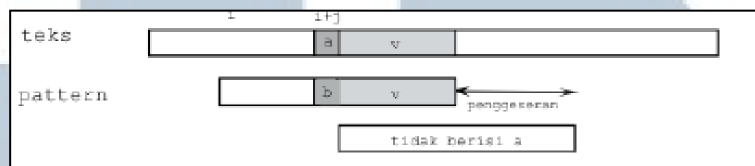
Gambar 2.6 Pergeseran *good-suffix* jika hanya ada awalan *v* dari pattern yang sama dengan akhiran *u* (Kumara, 2009)

Pergeseran *bad-character* yang terdiri dari mensejajarkan T[i+j] dengan kemunculan paling kanan karakter tersebut di *pattern*, Pergeseran ini diilustrasikan sebagai berikut (Kumara, 2009):



Gambar 2.7 Pergeseran *bad-character* bila di *pattern* terdapat karakter *a* (Kumara, 2009)

Dan bila karakter tersebut tidak ada di *pattern*, maka *pattern* akan disejajarkan dengan T[i+n+1], seperti yang diilustrasikan oleh (Kumara, 2009):



Gambar 2.8 Pergeseran *bad-character* bila *pattern* tidak mengandung karakter *a* (Kumara, 2009)

Pergeseran *bad character* ini akan sering terjadi pada pencocokan *string* dengan ruang alfabet yang besar dan dengan *pattern* yang pendek yang sering terjadi di praktek pada umumnya. Hal ini terjadi karena akan banyak karakter di teks yang tidak muncul di *pattern*. Namun, untuk *file* biner, yang mempunyai alfabet Σ={0, 1}, Pergeseran ini kemungkinan besar tidak akan membantu sama sekali. Hal ini dapat diatasi dengan membandingkan beberapa bit sekaligus (Kumara, 2009).

2.4. Algoritma Turbo Boyer-Moore

Algoritma Turbo Boyer-Moore adalah sebuah variasi dari algoritma Boyer-Moore. Bila dibandingkan dengan algoritma Boyer-Moore, algoritma ini membutuhkan ruang lebih tapi tidak memerlukan pemrosesan ekstra. Ruang ekstra yang diperlukan berguna untuk mengingat faktor dari teks yang cocok dengan akhiran dari *string* yang dicari selama *attempt* terakir dan hanya jika *good-suffix shift* dilakukan (Charras & Lecroq, 2009).

Dengan demikian, teknik ini mempunyai dua keunggulan (Charras & Lecroq, 2009):

1. Teknik ini memungkinkan untuk melompati faktor dari teks tersebut.

2. Teknik ini mengijinkan Pergeseran turbo.

Salah satu contoh aplikasi yang mengimplementasikan algoritma Turbo Boyer-Moore dalam pencarian *string*-nya adalah MySQL versi 4.0 keatas. MySQL akan melakukan pencarian menggunakan algoritma ini ketika *string* dalam frasa LIKE panjangnya lebih dari 3 karakter (AB., 2005).

Pergeseran *bad character* dan *good suffix* pada algoritma Turbo-Boyer-Moore sama dengan Pergeseran yang dilakukan algoritma Boyer-Moore. Sedangkan sebuah Pergeseran *turbo* dapat terjadi bila pada *attempt* yang sedang dilakukan, akhiran dari *pattern* yang cocok dengan teks lebih pendek dari bagian dari teks yang diingat dari *attempt* sebelumnya. Pada kasus ini, anggap v1 adalah faktor yang diingat dari *attempt* sebelumnya, dan v2 adalah bagian dari *pattern* yang cocok pada *attempt* yang sedang dilakukan, sehingga v1zv2 adalah akhiran dari *pattern.* Lalu anggap a adalah karakter di teks dan b adalah karakter dari *pattern* yang sedang dicocokan pada *attempt* tersebut. Maka av2, adalah akhiran dari *pattern*, dan juga akhiran dari v1 karena $|v2|<|v1|$. Dua karakter a dan b muncul dengan jarak p di teks, dan akhiran dari v1zv2 dari *pattern* mempunyai periode $p=|zv2|$, karena v1 merupakan pinggiran dari v1zv2, sehingga tidak mungkin melewati dua kemunculan karakter a dan b di teks (Kumara, 2009).

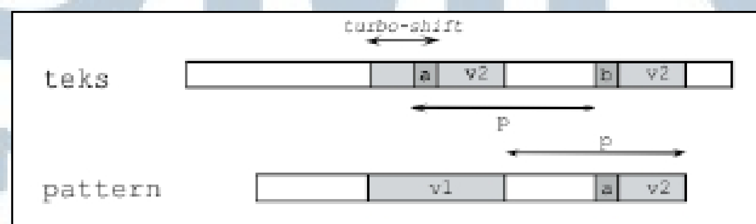
Pseudocode algoritma Turbo Boyer-Moore:

UNIVERSITAS MULTIMEDIA NUSANTARA

```
Procedure TurboBoyerMooreSearch(
        input m, n : integer,
        input P : array[0..n-1] of char,
        input T : array[0..m-1] of char,
        output ketemu : array[0..m-1] of boolean
)

Deklarasi:
 i, j, v1, v2, shift, bmBcShift, bmGsShift, turboShift: integer
 BmBc : array[0..255] of integer
 BmGs : array[0..n-1] of integer

Algoritma:
 preBmBc(n, P, BmBc);
 preBmGs(n, P, BmGs);
 i := v1 := 0
 shift := n
 while (i<= m-n) do
        j := n-1
        while (j>=0 and T[i+j] = P[j]) do
                j:=j-1
                if(v1 != 0 and (i = m-1-shift))
                        j := j-v1
        endwhile
        if(j<0) then
                ketemu[i]:=true
                shift:=bmGs[0]
                v1 = n-shift
        else
                v2 := n-1-j
                turboShift := v1-v2
                bmBcShift := BmBc[chartoint(T[i+j])]-n+j+1
                bmGsShift := BmGs[j]
                shift := max(bmBcShift, bmGsShift)
                shift := max(shift, turboShift)
                if(shift = bmGs[i]) then
                        v1 := min(m-shift, v2)
                else
                        if(turboShift < bmBcShift)
                                shift := max(shift, v1+1)
                        endif
                        v1 := 0
                endif
        endif
        i := i+shift
 endwhile
```

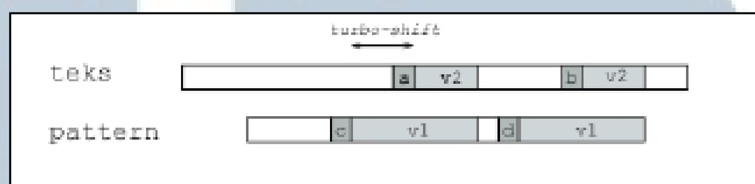
Gambar 2.9 *Pseudocode* Turbo Boyer-Moore (Kumara, 2009)

Pergeseran terkecil yang mungkin dilakukan adalah sebesar |v1|-|v2|, yang disebut sebagai pergeseran turbo dan diilustrasikan oleh (Kumara, 2009):



Gambar 2.10 Pergeseran *turbo* (Kumara, 2009)

Jika terjadi kasus dimana |v2|>|v1|, dan panjang dari pergeseran *bad-character* lebih besar dari pergeseran *good-suffix* maupun Pergeseran turbo. Pada kasus ini, seperti yang diilustrasikan pada gambar berikut (Kumara, 2009):



Gambar 2.11 Pergeseran harus lebih dari */v1/+1* (Kumara, 2009)

Dua karakter c dan d pastilah berbeda karena disyaratkan bahwa jika v1≠0 maka pergeseran sebelumnya adalah pergeseran *good suffix*. Sebagai akibatnya, jika pergeseran dengan panjang yang lebih besar dari pergeseran *turbo* namun lebih kecil dari |v1|+1 maka c dan d akan disejajarkan dengan karakter yang sama di teks. Oleh karena itu, dalam kasus ini panjang Pergeseran minimal adalah |v1|+1 (Kumara, 2009).

2.5. Algoritma Tuned Boyer-Moore

Algoritma Tuned Boyer-Moore dapat dilihat sebagai implementasi yang efisien algoritma Horspool (Jansen, Margraf, Mastrolilli, & Rolim, 2003). Fitur utama dari algoritma ini adalah simplifikasi dari algoritma Boyer-Moore, mudah untuk diimplementasikan, hanya menggunakan *bad-character shift*, dan sangat cepat dalam praktiknya (Lee & Cheng, 2007).

Pada dasarnya dalam algoritma ini yang menjadi fokus utama adalah karakter terakhir dari *pattern* dan menggeser *pattern* untuk mencocokan bagian paling akhir dari *pattern* tersebut (Lee & Cheng, 2007).

Anggap *pattern* P dengan panjang m. Setiap perulangan dari Tuned Boyer-Moore dapat dibagi menjadi 2 fase yaitu *last character localization* dan *matching phase*. Fase pertama mencari persamaan dari P[m-1], dengan melakukan tiga kali *blind shift* (sesuai dengan aturan perhitungan *bad-character shift* klasik) sampai dibutuhkan. Fase pencocokan mencoba mencocokan sisa dari *pattern* P[0..m-2] dengan menggunakan karakter-karakter dari teks, tetap dilakukan dari kanan ke kiri. Di akhir fase pencocokan, kemajuan *shift* dihitung berdasarkan aturan *bad-character* Horspool (Jansen, Margraf, Mastrolilli, & Rolim, 2003).

Selanjutnya, dengan tujuan menghitung *shift* terakhir dengan benar, pertama-tama algoritma menempatkan m *copies* sebanyak P[m-1] di akhir teks sebagai penjaga (Jansen, Margraf, Mastrolilli, & Rolim, 2003).

Pseudocode algoritma Tuned Boyer-Moore:

δ-TUNED-BOYER-MOORE(x, m, y, n, δ)

1 ▷ Preprocessing
2 **for** all $a \in \Sigma$
3   **do** *shift*$[a] \leftarrow \min\{\{m - i \mid x[i] \stackrel{\delta}{=} a\} \cup \{m\}\}$
4 $s \leftarrow \min\{\{m - i \mid x[i] \stackrel{2\delta}{=} x[m]$ and $0 < i < m\} \cup \{m\}\}$
5 $y[n + 1..n + m] \leftarrow (x[m])^m$
6 ▷ Searching
7 $j \leftarrow m$
8 **while** $j \leq n$
9   **do** $k \leftarrow$ *shift*$[y[j]]$
10   **while** $k \neq 0$
11   **do** $j \leftarrow j + k$
12                 $k \leftarrow$ *shift*$[y[j]]$
13                 $j \leftarrow j + k$
14                 $k \leftarrow$ *shift*$[y[j]]$
15                 $j \leftarrow j + k$
16                 $k \leftarrow$ *shift*$[y[j]]$
17   **if** $x[1..m - 1] \stackrel{\delta}{=} y[j - m + 1..j - 1]$ and $j \leq n$
18   **then** REPORT$(j - m + 1)$
19         $j \leftarrow j + s$

Gambar 2.12 *Pseudocode* Tuned Boyer-Moore (Crochemore, Lecroq, Plandowski, Iliopoulos, Pinzon, & Rytter, 2003)

Ilustrasi dari algoritma Tuned Boyer-Moore adalah sebagai berikut:

Langkah pertama adalah menghitung Boyer-Moore *Preprocessing Table*. Dalam algoritma ini yang digunakan adalah tabel *Occurence Heuristic* atau *Bad-Character Shift*. Contohnya seperti ini:

6 5 4 3 2 1
P=AGCAGAC

	A	C	G	T
bmBC	1	4	2	7

Gambar 2.13 Contoh tabel *preprocessing* Boyer-Moore (Lee & Cheng, 2007)

Selanjutnya, mencocokan *pattern* untuk mencari *string* dalam suatu teks besar yang diberikan.

- Text string
  T=GCGAGCAGACGTGCGAGTACG
- Pattern string
  P=AGCAGAC

	A	C	G	T
tbmBC	1	4	2	7

tbmBC[*A*]=1, shift=1

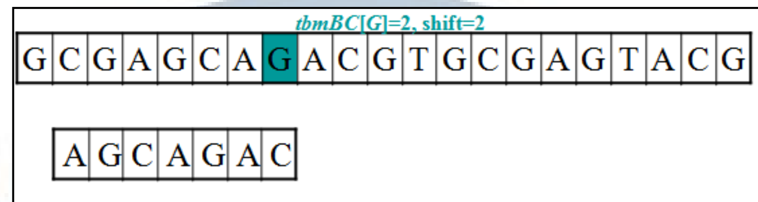
G	C	G	A	G	C	A	G	A	C	G	T	G	C	G	A	G	T	A	C	G

A	G	C	A	G	A	C

Gambar 2.14 Proses mencocokan *pattern* dengan *string* (*blind shift* pertama) (Lee & Cheng, 2007)

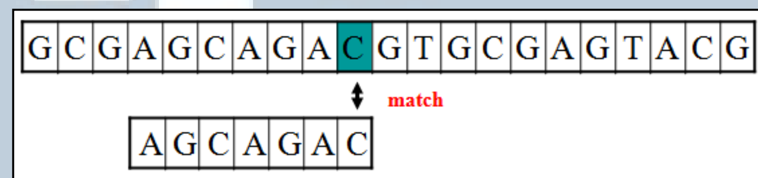
Jika ditemukan ketidakcocokan, maka *pattern* akan digeser sesuai dengan hasil yang ada di tabel *preprocessing*.

UNIVERSITAS
MULTIMEDIA
NUSANTARA



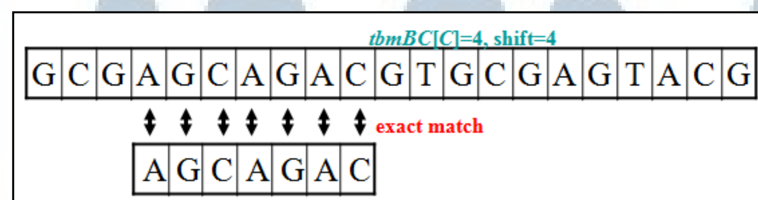
Gambar 2.15 Proses mencocokan *pattern* setelah *shift* 1 (*blind shift* ke-2) (Lee & Cheng, 2007)

Hal ini dilakukan terus menerus hingga didapatkan *string* yang cocok.



Gambar 2.16 Proses mencocokan *pattern* setelah *shift* 2 (*blind shift* ke-3) (Lee & Cheng, 2007)

Dilanjutkan dengan mengecek *string* lainnya, dimulai dari paling belakang. Jika ditemukan ketidakcocokan, maka akan dilakukan pergeseran atau lompatan bergantung pada letak ketidakcocokan.



Gambar 2.17 Proses pencarian berhasil (Lee & Cheng, 2007)