BAB 2
LANDASAN TEORI

2.1. Permainan Word Search Puzzle

Permainan *word search puzzle* atau permainan pencarian kata adalah permainan permainan berbasis *puzzle* untuk mencari kata-kata yang disusun dalam bentuk *array* dua dimensi atau yang lebih dikenal dengan matriks. Kata-kata tersebut dapat disusun secara horizontal, vertical maupun diagonal dan dapat ditulis dalam posisi terbalik maupun tidak. Strategi umum yang digunakan oleh pemain untuk menyelesaikan permainan *puzzle* ini adalah dengan mencari huruf pertama dari kata yang dicari dalam kumpulan huruf kemudian mencari huruf kedua yang terletak disebelah kanan, kiri, atas, bawah atau diagonal yang cocok dan seterusnya sampai huruf-huruf yang ditemukan membentuk kata yang dicari. Gambar 2.1 merupakan contoh permainan *word search puzzle.*



Gambar 2.1 Word search puzzle
Sumber (http://en.wikipedia.org/wiki/Word_search)

Dalam melakukan pencarian satu kata, pemain harus mencari melalui delapan jalur yang mungkin, yaitu :

1. Horizontal ke kanan
2. Horizontal ke kiri
3. Vertikal ke atas
4. Vertikal ke bawah
5. Diagonal ke kiri atas
6. Diagonal ke kiri bawah
7. Diagonal ke kanan atas
8. Diagonal ke kanan bawah

2.2. Algoritma

Menurut Goodman Hadet Niemi algoritma adalah urutan-urutan terbatas dari operasi-operasi yang terdefinisi dengan baik, yang masing-masing membutuhkan *memory* dan waktu yang terbatas untuk menyelesaikan masalah. Langkah-langkah dalam algoritma harus logis dan harus dapat ditentukan bernilai salah atau benar. Dalam beberapa konteks, algoritma adalah spesifikasi urutan langkah untuk melakukan pekerjaan tertentu. Pertimbangan dalam pemilihan algoritma adalah pertama, algoritma haruslah efektif yang artinya artinya algoritma akan memberikan keluaran yang dikehendaki dari sejumlah masukan yang diberikan.

Pertimbangan kedua yang harus diperhatikan adalah efektif yaitu harus mengetahui seberapa baik hasil yang dicapai oleh algoritma tersebut. Hal ini penting terutama pada algoritma untuk menyelesaikan masalah yang memerlukan *aproksimasi* hasil (hasil yang hanya berupa pendekatan). Ketiga adalah efisiensi algoritma, efisiensi algoritma dapat ditinjau dari 2 hal yaitu efisiensi waktu dan memori. Meskipun algoritma memberikan keluaran yang benar (paling mendekati), tetapi jika harus menunggu berjam-jam untuk mendapatkan keluaran dari algoritma tersebut biasanya tidak akan dipakai, setiap orang menginginkan keluaran yang cepat. Begitu juga dengan memori,

semakin besar memori yang terpakai maka semakin buruklah algoritma tersebut.

2.2.1. Ciri dan Sifat Algoritma

Dalam setiap penggunaan algoritma, penggunaan algoritma juga mempunyai ciri dan sifat. Menurut Donald E. Knuth dalam bukunya *the art of computer programming*, algoritma harus mempunya 5 ciri penting, yaitu :

1. Algoritma mempunyai awal dan akhir, artinya suatu algoritma harus berhenti setelah mengerjakan serangkaian tugas atau 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, yaitu :

1. *Input* yaitu suatu algoritma memliki *input* atau kondisi awal sebelum algoritma dilaksanakan dan bisa berupa nilai-nilai pengubah yang diambil dari himpunan khusus.
2. *Output* yaitu suatu algortima akan menghasilkan *output* setelah dilaksanakan, dimana nilai *output* diperoleh dari nilai *input* yang telah diproses melalui algoritma.
3. *Definiteness* yaitu langkah-langkah yang dituliskan dalam algoritma terdefinisi dengan jelas sehingga mudah dilaksanakan oleh sistem.
4. *Finiteness* yaitu 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* yaitu setiap langkah dalam algoritma bisa dilaksanakan dalam selang waktu tertentu sehingga pada akhirnya memberi solusi sesuai yang diharapkan.

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

2.2.2. String

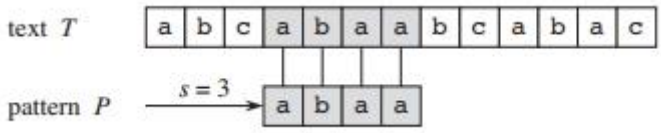
String dalam ilmu komputer dapat diartikan dengan sekuens dari karakter. Walaupun sering juga dianggap sebagai data abstrak yang menyimpan sekuens nilai data, atau biasanya berupa *bytes* yang mana merupakan elemen yang digunakan sebagai pembentuk karakter sesuai dengan *encoding* karakter yang disepakati seperti ASCII, ataupun EBCDIC [4].

2.2.3. String Matching

String matching dalam bahasa Indonesia dikenal dengan istilah pencocokan *string*. Berikut 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 $\sum$ abjad terbatas. Contoh, diketahui $\sum$ = {0,1} atau $\sum$ = {a,b,…,z}. Sedangkan karakter susunan P and T dapat didefinisikan
sebagai kata (*string*) pada banyak karakter.

Pola yang dimaksud adalah dimana *P* muncul dengan shift s dalam text *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

2.2, 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 [14].



Gambar 2.2 Kasus *String* Matching

2.2.4. 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. Algoritma pencocokan *string* dapat diklasifikasikan menjadi tiga bagian menurut arah pencarianya, yaitu :

1. Dari arah yang paling alami, dari kiri ke kanan, yang merupakan arah membaca, algoritma yang termasuk kategori ini adalah :
   1) Algoritma *Brute Force*

      Cara yang dilakukan dengan membandingkan seluruh elemen karakter pada pola dengan kalimat atau teks panjang, dimulai pada elemen karakter pertama pada kalimat tersebut. Jika tidak sesuai maka pembandingan dimulai dengan elemen kedua dari kalimat tersebut.
   2) Algoritma *Knuth-Morris-Pratt*

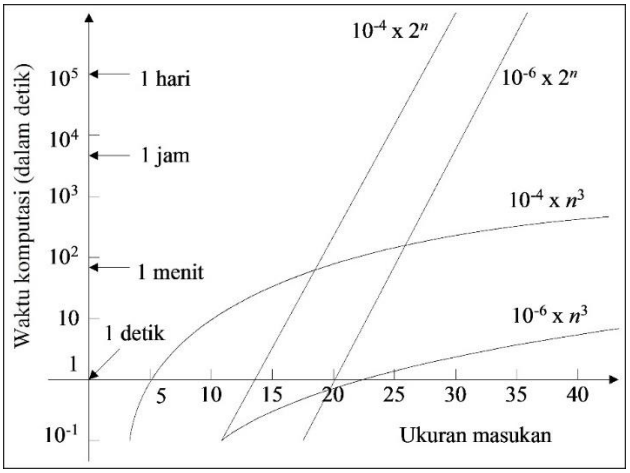
      Menurut Robert Baase cara yang dilakukan algoritma *Knuth-Morris-Pratt* adalah dengan menghitung fungsi pinggiran dari pola terlebih dulu dan kemudian akan dilakukan perbandingan antara pola dan elemen pertama dari kalimat, jika tidak sesuai, maka perbandingan tidak dilakukan pada elemen kedua, namun tergantung dari nilai yang akan dikeluarkan oleh fungsi pinggiran tersebut.
2. Dari kanan ke kiri, arah yang biasanya menghasilkan hasil terbaik secara praktikal, contohnya adalah :

1) 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 *teoritis*, algoritma yang termasuk kedalam kategori ini adalah :
   1) Algoritma *Colussi*
   2) Algoritma *Chrochemore-Perrin*

2.3. Kompleksitas Algoritma

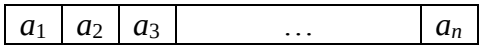
Sebuah algoritma tidak saja harus benar, tetapi juga harus mangkus (*efisien*). Algoritma yang bagus adalah algoritma yang mangkus, kemangkusan algirmta diukur dari berapa jumlah waktu dan ruang (*space*) memori yang dibutuhkan untuk menjalankanya. Algoritma yang mangkus ialah algoritma yang meminimumkan kebutuhan waktu dan ruang. 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. Pada gambar 2.3 membuktikan bahwa kenapa memerlukan algoritma yang mangkus dimana ukuran masukan mempengaruhi waktu komputasi.



Gambar 2.3 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



Gambar 2.4 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

$$t_1 = 1 + 1 + n + n + 1 = 3 + 2n \quad (2.1)$$

2. Operasi penjumlahan (`jumlah+`a_k, dan `k+1`) jumlah seluruh operasi penjumlahan adalah

$$t_2 = n + n = 2n \quad (2.2)$$

3. Operasi pembagian (`jumlah/n`) jumlah seluruh operasi pembagian adalah

$$t_3 = 1 \qquad (2.3)$$

Maka total kebutuhan waktu algoritma menghitung rata – rata adalah

$$t = t_1 + t_2 + t_3 = (3 + 2n)\text{a} + 2nb + c \text{ detik} \qquad (2.4)$$

Model perrhiutngan kebutuhan waktu seperti diatas kurang dapat diterima karena dalam prakteknya tidak mempunyai informasi berapa waktu sesungguhnya untuk melakasanakan suatu operasi tertentu dan 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, yaitu

1. Komplekstias waktu

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

2.3.1. Kompleksitas Waktu

Komplekstias waktu merupakan hal penting untuk mengukur efisiensi suatu algoritma karena kompleksitas waktu dari suatu algoritma yang terukur adalah fungsi ukuran masalah. Didalam sebuah algoritma terdapat beberapa macam jenis operasi yaitu

1. Operasi baca/tulis.

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

Didalam prakeknya 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 khar : 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 diatas 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    { bernilai true jika x ditemukan atau
false jika x tidak ditemukan }
Algoritma:
  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 $a_1 = x$

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

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

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

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

$$T_{\text{avg}}(n) = \frac{(1+2+3+\ldots+n)}{n} = \frac{\frac{1}{2}n(1+n)}{n} = \frac{(n+1)}{2} \tag{2.7}$$

2.4. Notasi Big O

Notasi *Big O* merupakan suatu notasi matematika untuk menjelaskan batas atas dari *magnitude* suatu fungsi dalam fungsi yang lebih sederhana. 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, pada bukunya yang berjudul *Analytische Zahlentheorie* edisi kedua. 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 2.1 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 [20].

Tabel 2.1 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;
```

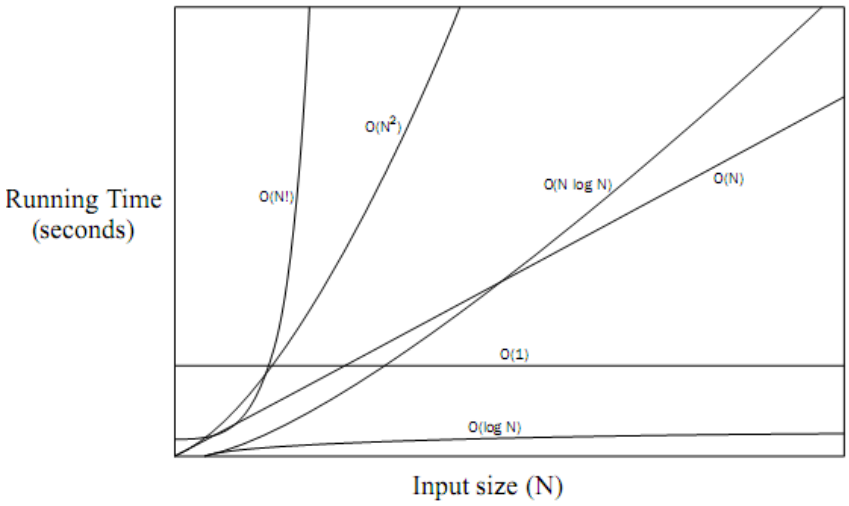
   Pada contoh kasus diatas jumlah operasi penugasan (*assignment*) ada tiga buah dan tiap operasi dilakukan satu kali. Jadi $T(n) = 3 = O(1)$.

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 persoalana 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 $2n$.

Berikut pada gambar 2.5 adalah beberapa perbandingan kompleksitas algoritma



Gambar 2.5 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 2.2 dibawah ini dapat dilihat perhitungan kompleksitas dengan menggunakan Big-O dari contoh program sederhana diatas :

Tabel 2.2 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 2.2 di atas maka dapat dihitung hasil dari notasi Big-O adalah sebagai berikut :

$$O(1) + O(1) + O(N) + O(N) + O(N) = 2+ 3N \text{ kali} = O(N)$$

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.

2.5. Algoritma Knuth-Morris-Pratt

Algoritma *Knuth-Morris-Pratt* adalah salah satu algoritma pencarian *string*, dikembangkan terpisah oleh oleh D. E. Knuth pada tahun 1967 dan James H. Morris bersama Vaughan R. Pratt pada tahun 1966. Namun keduanya mempublikasikanya secara bersamaan pada tahun 1977.

Pada persoalan pencocokan *string* umumnya diberikan dua buah hal utama, yaitu :

1. Teks yaitu *string* yang relatif panjang yang akan menjadi bahan yang akan dicari kecocokanya (dimisalkan panjang dari teks adalah *n*).
2. Pola (*pattern*) yaitu *string* dengan panjang relatif yang lebih pendek dari teks (misalkan panjang pola adalah *m*, maka $m < n$). Pola akan digunakan sebagai *string* yang dicocokan ke dalam teks.

Umumnya pada persoalan pencarian adalah apakah mencari pola yang diberikan pada teks tertentu dan dari jawaban persoalan itu bisa berupa ada atau tidak, berapa kali ditemukan ataupun diposisi berapa pola tersebut ditemukan. Maka dari itu dasar utama dari algoritma *Knuth-Morris-Pratt* adalah untuk mereduksi pergeseran *pointer* pencocokan *string* lebih jauh daripada harus menggesernya satu persatu layaknya yang dilakukan pada metode pencarian *string* dengan algoritma *Brute Force*. Uniknya, berbeda dengan cara konvensional (algoritma *Brute Force*) yang melakukan pergeseran sebanyak satu langkah. Algoritma *Knuth-Morris-Pratt* menyimpan informasi dari pola tersebut untuk menentukan jumlah pergeseran yang relatif lebih jauh. Dimana dalam hal pencocokan *string*, pola yang dicocokkan berawal dari kiri teks. Algoritma Knuth-Morris-Pratt mempunyai kompleksitas algoritma $O(m+n)$.

2.5.1. Cara Kerja Algoritma Knuth-Morris-Pratt

Beberapa definisi pada algoritma ini, yaitu :

Misalkan *A* adalah alfabet dan $x = x_1 x_2 \ldots x_k$ adalah *string* yang panjangnya *k* yang dibentuk dari karakter-karakter di dalam alfabet *A*.

1. Awalan (*prefix*) dari *x* adalah *substring u* dengan

$$u = x_1 x_2 \ldots x_k \, , j \in \{1,2, \ldots, k\} \tag{2.8}$$

   dengan kata lain, *x* diawali dengan *u*.

2. Akhiran (*suffix*) dari *x* adalah *substring u* dengan

$$u = x_{j-b} x_{j-b+1} \ldots x_k \, , j \in \{1,2, \ldots, k\} \tag{2.9}$$

   dengan kata lain, *x* diakhiri dengan *u*.

3. Pinggiran (*border*) dari *x* adalah *substring r* sehingga

$$u = x_1 x_2 \ldots x_{j-1} \tag{2.10}$$

dan

$$u = x_{j-b} x_{j-b+1} \ldots x_j \, , j \in \{1,2,\ldots, k\} \tag{2.11}$$

dengan kata lain, pinggiran dari *x* adalah *substring* yang keduanya awalan dan juga akhiran sebenarnya dari *x.*

Misalkan ditentukan sebuah pola (*pattern*) adalah

$$x : abacab \tag{2.12}$$

Maka awalan dari x adalah

$$[] \, , a, ab, aba, abac, abaca \tag{2.13}$$

Akhiran dari x adalah

$$[], a, ab, cab, acab, bacab \tag{2.14}$$

Pinggiran [] (*string* kosong) mempunyai panjang 0, pinggiran *ab* mempunyai panjang 2. Jika pada pencocokan *string* ditemukan ketidakcocokan pada teks, maka algoritma *Knuth-Morris-Pratt* mengeliminasi sejumlah besar pergeseran yang tidak perlu yakni sepanjang

awalan terbesar yang juga merupakan akhiran dari *pattern* sesuai dengan posisi terakhir pencocokan *string*.

Secara sistematis, langkah-langkah yang dilakukan algoritma *Knuth-Morris-Pratt* pada saat mencocokan *string* terdapat 2 tahap yaitu adalah sebagai berikut :

2.5.1.1 Tahap Preprocessing

Pada proses awal (*preprocessing*), algoritma melakukan proses awal (preprocessing) terhadap *pattern* P dengan menghitung fungsi pinggiran yang mengindikasikan pergeseran yang mungkin dengan menggunakan perbandingan yang dibentuk sebelum pada tahap pencarian *string*. Dalam literatur lain fungsi ini disebut dengan *overlap function, failure function*, fungsi awalan, tabel *next* dan lainya. Fungsi tersebut akan menghasilkan *output* berupa *array integer* yang merupakan angka-angka pinggiran untuk setiap posisi iterasi pada *pattern* . Sebagai contoh, pada tabel 2.3 tinjau *pattern P =abcabdabc*. Nilai *b*(*j*) untuk setiap karakter di dalam *P* adalah sebagai berikut:

Tabel 2.3 Tabel *pattern*

P	a	b	c	a	b	d	a	b	c
P[j]	1	2	3	4	5	6	7	8	9

Dari *pattern* pada tabel 2.2 kemudian diambil kemungkinan dari *substring pattern* pada tabel 2.4 tersebut yaitu :

Tabel 2.4 Tabel sub*string pattern*

1	a
2	ab
3	abc
4	abca
5	abcab
6	abcabd
7	abcabda
8	abcabdab
9	abcabdabc

Kemudian dari setiap *substring pattern* tersebut diambil semua kemungkinan awalan (*prefix)* dan akhiran (*suffix)* pada tabel 2.5.

Tabel 2.5 Kemungkinan prefix dan *suffix*

P	Prefixes	*Suffix*es
a	No prefix	No *suffix*
ab	[a]	[b]
abc	[a,ab]	[c,bc]
abca	[a,ab,abc]	[a,ca,bca]
abcab	[a,ab,abc,abca]	[b,ab,cab,bcab]
abcabd	[a,ab,abc,abca,abcab]	[d,bd,abd,cabd,bcabd]
abcabda	[a,ab,abc,abca,abcab,abcabd]	[a,da,bda,abda,cabda,bcabda]
abcabdab	[a,ab,abc,abca,abcab,abcabd,abcabda]	[b,ab,dab,bdab,abdab,cabdab,bcabdab]
abcabdabc	[a,ab,abc,abca,abcab,abcabd,abcabda,abcabdab]	[c,bc,abc,dabc,bdabc,abdabc,cabdabc,bcabdabc,abcabdabc]

Setelah diambil dari setiap kemungkinan *prefix* dan *suffix* dari *substring pattern* , maka lihat panjang *prefix* yang yang sama dengan akhiran *suffix* pada tabel 2.6 *:*

Tabel 2.6 Tabel prefix dan *suffix*

a	0
ab	0
abc	0
abc**a**	1 [a]
abc**ab**	2 [ab]
abcabd	0
abcabd**a**	1 [a]
abcabd**ab**	2 [ab]
abcabd**abc**	3 [abc]

Maka dari setiap kemungkinan tersebut didapatkan hasil pada tabel 2.7, sebagai berikut :

Tabel 2.7 Tabel fungsi pinggiran

P	a	b	c	a	b	d	a	b	c
P[j]	1	2	3	4	5	6	7	8	9
b(j)	0	0	0	1	2	0	1	2	3

Ket : j = Index
P[j] = Pattern
B[j] = Nilai pinggiran

2.5.1.1. Tahap Pencarian

Setelah didapatkan nilai pinggiran dari *pattern* pada tahap *preprocessing* tersebut barulah kemudian dapat diproses pencocokan antara *pattern* dan teks yang diberikan pada tahap kedua yaitu tahap pencarian. Sebagai contoh diberikan sebuah teks=*abacaabacabcabdabc* dan *pattern* =*abcabdabc* yang sudah diketahui nilai pinggiranya pada tabel 2.8, setelah didapatkan nilai pinggiran *pattern* maka proses pencarian *string* dapat diproses sebagai berikut :

Tabel 2.8 Tabel pencarian

Teks :	a	b	a	c	a	a	b	a	c	a	b	c	a	b	d	a	b	c
Pattern :	a	b	c	a	b	d	a	b	c									
Nilai Pinggiran	0	0	0	1	2	0	1	2	3									

Berikut tahap proses pencarian *string* terhadap teks yang sudah diketahui nilai pinggiran dari *pattern* :

1) Pada tabel 2.9 *pattern* *a* cocok dengan dengan teks *a* begitupun *b* cocok dengan teks *b*, namun pada saat posisi *pattern* *c* terjadi ketidak cocokan dengan teks *a* maka lihat nilai pinggiran *pattern* c pada tabel 2.8 adalah 0, karena pada pencocokan *pattern a,b* terjadi kecocokan dengan teks *a,b*. Maka *pattern* digeser sebesar jumlah kecocokan – nilai pinggiran yaitu 2 – 0 = 2. Kemudian pattern digeser sejumlah 2 karakter.

Tabel 2.9 Pencarian iterasi ke-1

Teks	a	b	a	c	a	a	b	a	c	a	b	c	a	b	d	a	b	c
Pattern	a	b	c	a	b	d	a	b	c									

Ket :

	Cocok (*match*)
	Tidak cocok (*missmatch*)

2) Pada tabel 2.10 *pattern a* cocok dengan dengan teks *a*, namun pada saat posisi *pattern b* terjadi ketidak cocokan dengan teks *c* maka lihat nilai pinggiran *pattern b* pada tabel 2.8 adalah 0, karena pada pencocokan *pattern a* terjadi kecocokan dengan teks *a*. Maka *pattern* digeser sebesar jumlah kecocokan – nilai pinggiran yaitu 1 – 0 = 1. Kemudian pattern digeser sejumlah 1 karakter.

Tabel 2.10 Pencarian iterasi ke-2

Teks	a	b	a	c	a	a	b	a	c	a	b	c	a	b	d	a	b	c
Pattern			a	b	c	a	b	d	a	b	c							

Ket :

	Cocok (*match*)
	Tidak cocok (*missmatch*)

3) Pada tabel 2.11 *pattern a* tidak cocok dengan dengan teks *c*, maka lihat nilai pinggiran *pattern* a pada tabel 2.8 adalah 0. Karena tidak terjadi ketidakcocokan, maka *pattern* digeser sejumlah 1 karakter.

Tabel 2.11 Pencarian iterasi ke-3

Teks	a	b	a	c	a	a	b	a	c	a	b	c	a	b	d	a	b	c
Pattern				a	b	c	a	b	d	a	b	c						

Ket :

	Cocok (*match*)
	Tidak cocok (*missmatch*)

4) Pada tabel 2.12 *pattern a* cocok dengan dengan teks *a*, namun pada saat posisi *pattern b* terjadi ketidak cocokan dengan teks *a* maka lihat nilai pinggiran *pattern* c pada tabel 2.8 adalah 0, karena pada pencocokan *pattern a* terjadi kecocokan dengan teks *a*. Maka *pattern* digeser sebesar jumlah kecocokan – nilai pinggiran yaitu 1 – 0 = 1. Kemudian pattern digeser sejumlah 1 karakter.

Tabel 2.12 Pencarian iterasi ke-4

Teks	a	b	a	c	a	a	b	a	c	a	b	c	a	b	d	a	b	c
Pattern					a	b	c	a	b	d	a	b	c					

Ket :

	Cocok (*match*)
	Tidak cocok (*missmatch*)

5) Pada tabel 2.13 *pattern a* cocok dengan dengan teks *a* begitupun *b* cocok dengan teks *b*, namun pada saat posisi *pattern c* terjadi ketidak cocokan dengan teks *a* maka lihat nilai pinggiran *pattern* c pada tabel 2.8 adalah 0, karena pada pencocokan *pattern a,b* terjadi kecocokan dengan teks *a,b*. Maka *pattern* digeser sebesar jumlah kecocokan – nilai pinggiran yaitu 2 – 0 = 2. Kemudian pattern digeser sejumlah 2 karakter.

Tabel 2.13 Pencarian iterasi ke-5

Teks	a	b	a	c	a	a	b	a	c	a	b	c	a	b	d	a	b	c
Pattern						a	b	c	a	b	d	a	b	c				

Ket :

	Cocok (*match*)
	Tidak cocok (*missmatch*)

6) Pada tabel 2.14 *pattern a* cocok dengan dengan teks *a*, namun pada saat posisi *pattern b* terjadi ketidak cocokan dengan teks *c* maka lihat nilai pinggiran *pattern b* pada tabel 2.8 adalah 0, karena pada pencocokan *pattern a* terjadi kecocokan dengan teks *a*. Maka *pattern* digeser sebesar jumlah kecocokan – nilai pinggiran yaitu 1 – 0 = 1. Kemudian pattern digeser sejumlah 1 karakter.

Tabel 2.14 Pencarian iterasi ke-6

Teks	a	b	a	c	a	a	b	a	c	a	b	c	a	b	d	a	b	c
Pattern								a	b	c	a	b	d	a	b	c		

Ket :

	Cocok (*match*)
	Tidak cocok (*missmatch*)

7) Pada tabel 2.15 *pattern a* tidak cocok dengan dengan teks *c*, maka lihat nilai pinggiran *pattern* a pada tabel 2.8 adalah 0. Karena tidak terjadi ketidakcocokan, maka *pattern* digeser sejumlah 1 karakter.

Tabel 2.15 Pencarian iterasi ke-7

Teks	a	b	a	c	a	a	b	a	c	a	b	c	a	b	d	a	b	c
Pattern									a	b	c	a	b	d	a	b	c	

Ket :

	Cocok (*match*)
	Tidak cocok (*missmatch*)

8) Pada tabel 2.16 *pattern a* cocok dengan dengan teks *a* begitupun *b* cocok dengan teks *b*, *pattern c* cocok dengan *c* dan posisi *pattern a* cocok teks *a* dan seterusnya. Ternyata dari semua *pattern* itu cocok dengan teks maka pencarian selesai dilakukan.

Tabel 2.16 Pencarian iterasi ke-8

Teks	a	b	a	c	a	a	b	c	a	a	b	c	a	b	d	a	b	c
Pattern										a	b	c	a	b	d	a	b	c

Ket :

	Cocok (*match*)
	Tidak cocok (*missmatch*)

2.6. Algoritma *Boyer-Moore*

Algoritma *Boyer-Moore* adalah salah satu algoritma pencarian *string* yang dipublikasikan oleh Robert S. Boyer, dan J. Strother Moore pada tahun 1977. Rinaldi [1] mendefinisikan bahwa Algoritma *Boyer-Moore* merupakan variasi lain dari pencarian *string* dengan melompat maju sejauh mungkin. Tetapi, algoritma *Boyer-Moore* berbeda dengan algoritma *Knuth-Morris-Pratt*, dimana algoritma *Boyer-Moore* melakukan perbandingan *pattern* mulai dari kanan sedangkan algoritma *Knuth-Morris-Pratt* melakukan perbandingan dari kiri [2]. Analisis kompleksitas algoritma *Boyer-Moore* menyangkut 3 hal yaitu komplekstitas waktu, kompleksitas ruang, dan waktu pemrosesan. Kasus terbaik pada algoritma *Boyer-Moore* terjadi jika pertama kali masing-masing percobaan membandingkan simbol teks tidak cocok dengan yang ada di *pattern*. Sehingga kompleksitas waktu yang terbaik dengan notasi big O yaitu *O(N/M)*. Kompleksitas waktu untuk kasus terburuk adalah *O(Nm)*. Pada umumnya, kompleksitas waktu yang dibutuhkan untuk kasus rata-rata adalah *O(N/M)*.

Berikut langkah-langkah yang dilakukan algoritma *Boyer-Moore* pada saat mencocokkan *string* adalah :

1. Algoritma *Boyer-Moore* mulai mencocokkan *pattern* pada awal teks.
2. Dari kanan ke kiri, algoritma ini akan mencocokkan karakter per karakter *pattern* dengan karakter di teks yang bersesuaian, sampai salah satu kondisi berikut dipenuhi:
   a) Karakter di *pattern* dan di teks yang dibandingkan tidak cocok (*mismatch*).
   b) Semua karakter di *pattern* cocok, kemudian algoritma akan memberitahukan penemuan di posisi ini.
3. Algoritma kemudian menggeser *pattern* dengan memaksimalkan nilai penggeseran *good-suffix* dan penggeseran *bad-character*, lalu mengulangi langkah 2 sampai *pattern* berada di ujung teks.

2.6.1. Cara Kerja Algoritma *Boyer-Moore*

Secara sistematis, langkah-langkah yang dilakukan algoritma *Boyer-Moore* sama dengan algoritma *Knuth-Morris-Pratt* pada saat mencocokan *string* dimana terdapat 2 tahap yaitu sebagai berikut :

2.6.1.1. Tahap Preprocessing

Berbeda dengan algoritma Knuth-Morris-Pratt pada tahap *preprocessing* di algoritma *Boyer-Moore* menggunakan dua fungsi *shift* (pergeseran) yaitu *bad-chacarter shift* dan *good-suffix shift*. Dimana dari dua fungsi tersebut akan mencari nilai untuk pergeseran *pattern* pada pada teks. Fungsi pertama yaitu *bad-character shift*, fungsi tersebut akan menghitung banyaknya pergeseran yang harus dilakukan berdasarkan identitas karakter yang menyebabkan kegagalan pencocokan *string* yang akan menghasilkan nilai *occurrence heuristic (*OH*)*, dimana nilai tersebut akan disimpan pada tabel *boyer-moore bad character (*BmBc*)*. Fungsi kedua yaitu *good-suffix shift*, fungsi tersebut juga akan menghitung banyaknya pergeseran yang harus dilakukan, tetapi berdasarkan jumlah karakter yang berhasil dicocokan sebelum pencocokan *string* tersebut gagal yang akan menghasilkan nilai

match heuristic (MH), dimana nilai tersebut disimpan pada tabel boyer-moore good suffix (BmGs).

Tahap pertama menentukan *pattern* atau kata yang akan dicari dengan menghitung *bad–character shift* yang kemudian disimpan pada tabel 2.17 yaitu tabel pencacahan karakter *Boyer-Moore Bad Character* (*BmBc*) untuk seterusnya akan diisi sesuai dengan nilai yang didapat dari hasil perhitungan.

Contoh *pattern* : GCAGAGAG

Tabel 2.17 Tabel pencacahan karakter untuk *BmBc*

Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai OH	?	?	?	?	?	?	?	?

Ket : OH = Occurrence Huristic
? = Belum diketahui

Lakukan pencacahan disetiap karakter pada *pattern* sebagai berikut :

1. Pencacahan dimulai dari karakter ke-2 paling kanan (*index* - panjang (*pattern*)-2) dari *pattern* .
2. Bandingkan setiap karakter yang dicacah terhadap tabel 2.17 yaitu tabel *BmBc*, jika karakter yang dicacah tidak ditemukan dalam tabel tersebut, maka tambahkan karakter tersebut ke dalam tabel *BmBc* dan nilai OH sama dengan jumlah pergeseran karakter yang dilakukan sebelumnya.
3. Lakukan langkah ke-2 kembali, dengan melakukan perpindahan 1 karakter ke kiri hingga mencapai karakter paling kiri (*index* = 0) secara terus menerus.
4. Jika *index* = 0, cacah karakter paling kanan lalu bandingkan terhadap tabel *BmBc*, lakukan hal yang sama jika karakter ditemukan sesuai dengan langkah ke-2.

Berikut cara menghitung tabel *Bad-Character Shift* yang diterapkan prosesnya dengan langkah-langkah pencarian di atas :

1. Pada iterasi ke-1, proses yang paling awal yaitu melakukan pencacahan dari karakter ke-2 paling kanan (*index* - panjang(*pattern*)-2) dari *pattern* didapatkan karakter *A* dilanjutkan dengan langkah ke-2 dengan membandingkan karakter yang dicacah, sehubungan langkah pertama maka nilai *shift* (pergeseran) ke satu maka masukan nilai 1 (*default*) untuk karakter A pada tabel 2.18.

Tabel 2.18 Tabel iterasi ke-1 pencarian nilai OH pada *BmBc*

						Mulai : *Index* = panjang(*pattern*) - 2	
G	C	A	G	A	G	A	G
Move = 1							

Tabel 2.19 Tabel comparator *BmBc* iterasi ke-1

Bandingkan "A" dengan "Null		**Hasil**	
***BmBc*(sebelum)**		***BmBc*(sesudah)**	
Karakter	Null	Karakter	A
Nilai OH	Null	Nilai OH	1

2. Pada iterasi ke-2 ini seperti yang telah dijelaskan pada langkah ke-3 pada langkah-langkah diatas. Sehubungan karakter *G* belum memiliki nilai maka masukan *G* dengan nilai OH (*Occurrence heuristic*) = 2 pada tabel 2.20, dilanjutkan dengan melakukan perpindahan 1 karakter ke kiri hingga mencapai karakter paling kiri yaitu (*index*=0) secara terus menerus.

Tabel 2.20 Tabel iterasi ke-2 pencarian nilai OH pada *BmBc*

					Mulai: *Index* = panjang(*pattern*) - 3		
G	C	A	G	A	G	A	G
Pindah = 2							

Tabel 2.21 Tabel comparator *BmBc* iterasi ke-2

Bandingkan "G" with "A"		**Hasil**		
***BmBc*(sebelum)**		***BmBc*(sesudah)**		
Karakter	A	Karakter	A	G
Nilai OH	1	Nilai OH	1	2

3. Pada proses iterasi ke-3 pada tabel 2.22 lakukan langkah ke-3. Sehubungan karakter *A*sudah memiliki nilai OH (*Occurrence heuristic*) =

1, maka dilanjutkan dengan melakukan perpindahan ke kiri kiri hingga mencapai karakter paling kiri yaitu (*index*=0) secara terus menerus.

Tabel 2.22 Tabel iterasi ke-3 pencarian nilai OH pada *BmBc*

				Mulai: *Index* = panjang(*pattern*)-4			
G	C	A	G	A	G	A	G
Pindah = 3							

Tabel 2.23 Tabel comparator *BmBc* iterasi ke-3

Bandingkan "A" dengan "A,G"			**Hasil**		
***BmBc*(sebelum)**			***BmBc*(sesudah)**		
Karakter	A	G	Karakter	A	G
Nilai OH	1	2	Nilai OH	1	2

4. Pada proses iterasi ke-4 pada tabel 2.24 lakukan langkah ke-3. Sehubungan karakter *G* sudah memiliki nilai OH (*Occurrence heuristic*) = 2 maka dilanjutkan dengan melakukan perpindahan ke kiri kiri hingga mencapai karakter paling kiri yaitu (*index*=0) secara terus menerus.

Tabel 2.24 Tabel iterasi ke-4 pencarian nilai OH pada *BmBc*

			Mulai: *Index* = panjang(*pattern*)-5				
G	C	A	G	A	G	A	G
Pindah = 4							

Tabel 2.25 Tabel comparator *BmBc* iterasi ke-4

Bandingkan "G" dengan "A,G"			**Hasil**		
***BmBc*(sebelum)**			***BmBc*(sesudah)**		
Karakter	A	G	Karakter	A	G
Nilai OH	1	2	Nilai OH	1	2

5. Pada proses iterasi ke-5 pada tabel 2.26 lakukan langkah ke-3. Sehubungan karakter *A* sudah memiliki nilai OH (*Occurrence heuristic*) = 1 maka dilanjutkan dengan melakukan perpindahan ke kiri kiri hingga mencapai karakter paling kiri yaitu (*index*=0) secara terus menerus.

Tabel 2.26 Tabel iterasi ke-5 pencarian nilai OH pada *BmBc*

		Mula: *Index* = panjang(*pattern*)-6					
G	C	A	G	A	G	A	G
Pindah = 5							

Tabel 2.27 Tabel comparator *BmBc* iterasi ke-5

Bandingkan "A" dengan"A,G"			**Hasil**		
***BmBc*(sebelum)**			***BmBc*(sesudah)**		
Karakter	A	G	Karakter	A	G
Nilai OH	1	2	Nilai OH	1	2

6. Pada proses iterasi ke-6 pada tabel 2.28 lakukan langkah ke 3. Sehubungan karakter *C* belum memiliki nilai maka masukan *C* dengan nilai OH (*Occurrence heuristic*) = 6 sesuai nilai index dari kanan ke kiri, dilanjutkan dengan melakukan perpindahan ke kiri kiri hingga mencapai karakter paling kiri yaitu (*index*=0) secara terus menerus .

Tabel 2.28 Tabel iterasi ke-6 pencarian nilai OH pada BmBc

	Mulai: *Index* = panjang(*pattern*)-7						
G	C	A	G	A	G	A	G
Pindah = 6							

Tabel 2.29 Tabel comparator *BmBc* iterasi ke-6

Bandingkan "C" dengan"A,G"			**Hasil**			
***BmBc*(sebelum)**			***BmBc*(sesudah)**			
Karakter	A	G	Karakter	A	G	C
Nilai OH	1	2	Nilai OH	1	2	6

7. Pada proses iterasi ke-7 pada tabel 2.30 lakukan langkah ke 3. Sehubungan karakter *G* sudah memiliki nilai OH (*Occurrence heuristic*) = 2 maka dilanjutkan dengan melakukan perpindahan ke kiri kiri hingga mencapai karakter paling kiri yaitu (*index*=0) secara terus menerus

Tabel 2.30 Tabel iterasi ke-7 pencarian nilai OH pada BmBc

Mulai: *Index* = Panjang(*pattern*)-1							
G	C	A	G	A	G	A	G
Pindah = 7							

Tabel 2.31 Tabel comparator *BmBc* iterasi ke-7

Bandingkan "G" dengan "A,G,C"				**Hasil**			
***BmBc*(sebelum)**				***BmBc*(sesudah)**			
Karakter	A	G	C	Karakter	A	G	C
Nilai OH	1	2	6	Nilai OH	1	2	6

8. Pada proses iterasi ke-7 pada tabel 2.32 telah mencapai *index*=0, maka lakukan langkah ke-4 yaitu jika *index* = 0, cacah karakter paling kanan (*index* = panjang(*pattern*)-1) pada tabel 2.23 lalu bandingkan kembali denga tabel comparator, Sehubungan karakter *G* sudah memiliki nilai OH (*Occurrence heuristic*) = 2 dan *index* = 0 maka didapatkan nilai dari hasil perhitungan tersebut seperti yang terdapat pada tabel 2.33.

Tabel 2.32 Tabel iterasi ke-8 pencarian nilai OH pada BmBc

							Akhir: *Index* = Panjang(*pattern*)-1
G	C	A	G	A	G	A	G
							Pindah = 8

Tabel 2.33 Tabel comparator *BmBc*

Bandingkan "G" dengan "A,G,C"				**Hasil**			
***BmBc*(sebelum)**				***BmBc*(sesudah)**			
Karakter	A	G	C	Karakter	A	G	C
Nilai OH	1	2	6	Nilai OH	1	2	6

Tabel 2.34 Tabel hasil perhitungan *bad-character shift*

BmBc			
Karakter	A	G	C
Nilai OH	1	2	6

Setelah mendapatkan hasil akhir dari perhitungan *bad-character shift* maka dilanjutkan membuat tabel akhiran/*suffix* dari *pattern* yang dapat dilihat pada tabel 2.34, untuk mencari nilai *good-suffix shift*. Pembuatan tabel *suffix* dari kanan ke kiri berfungsi untuk dijadikan *suffix* comparator sedangkan pembuatan tabel *suffix* dari kiri ke kanan berfungsi sebagai *suffix* yang nanti akan dibandingkan dan pencarian nilai MH (*Match Heuristic*), tabel tersebut dapat dilihat pada tabel 2.35.

Tabel 2.35 *Suffix* kanan ke kiri dan *Suffix* kiri ke kanan

Pattern : GCAGAGAG (kanan ke kiri)			
idx	Prefix	*Suffix*	Length
7	G	GCAGAGA	7

Pattern : GCAGAGAG (kiri ke kanan)			
Idx	Prefix	*Suffix*	Length
0	G	CAGAGAG	7

6	AG	GCAGAG	6		1	GC	AGAGAG	6
5	GAG	GCAGA	5		2	GCA	GAGAG	5
4	AGAG	GCAG	4		3	GCAG	AGAG	4
3	GAGAG	GCA	3		4	GCAGA	GAG	3
2	AGAGAG	GC	2		5	GCAGAG	AG	2
1	CAGAGAG	G	1		6	GCAGAGA	G	1
0	GCAGAGAG	*NULL*	0		7	GCAGAGAG	NULL	0

Tabel 2.36 *Suffix* lengkap beserta *suffix* comparator

Suffix									
Index	0	1	2	3	4	5	6	7	Move
Prefix	G	C	A	G	A	G	A	G	
Suffix	CAGAGAG	AGAGAG	GAGAG	AGAG	GAG	AG	G	NULL	
Suffix comparator	GCAGAGA								1
	GCAGAG								2
	GCAGA								3
	GCAG								4
	GCA								5
	GC								6
	G								7
	NULL								8
MH value	?	?	?	?	?	?	?	?	

Tabel 2.37 Tabel *BmGs* (*Boyer-Moore* Good *Suffix*)

Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai MH	?	?	?	?	?	?	?	?

Kemudian lakukan pemecahan akhiran pada *suffix* terhadap *suffix comparator* tabel 2.35. Berikan nilai MH (*Match Heuristic*) pada tabel 2.36 diatas yaitu tabel *BmGs* dengan *moving long* yang berkesuaian. Berikut langkah-langkah untuk mencari nilai *good-suffix shift* :

1. Cacah *suffix* pada masing-masing *index* terhadap seluruh *suffix comparator.*

2. Jika panjang kedua *suffix* tida sama panjang, potong salah satu *suffix* yang memiliki panjang terbesar, besaran pemotongan *suffix* sesuai dengan nilai length terkecil kearah paling kanan karakter. Jika ditemukan kecocokan antar *suffix*, bandingkan prefix kedua *suffix* jika masing-masing *suffix* memiliki prefix. Jika prefix yang diperbandingkan adalah sama, maka kecocokan tidak dapat diterima.
3. Lakukan langlah ke-2 hingga *index suffix* mencapai 0.
4. Untuk *index suffix* terbesar, secara default akan diberikan nilai 1 pada MH value, sedangkan yang lainya diberikan nilai sesuai dengan nilai move yang diraih.

Berikut proses menghitung tabel *Good-suffix shift* :

Tabel 2.38 Tabel perhitungan *Good-suffix shift* iterasi ke-1

Suffix Tabel		
Index	7	Move
Prefix	G	
Suffix	NULL	
Suffix comparator	GCAGAGA	1
	GCAGAG	2
	GCAGA	3
	GCAG	4
	GCA	5
	GC	6
	G	7
	NULL	8
Nilai MH	1	

Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai MH	?	?	?	?	?	?	?	1

Tabel 2.39 Tabel perhitungan *Good-suffix shift* iterasi ke-2

Suffix Tabel		
Index	6	Move
Prefix	A	
Suffix	G	
Suffix comparator	GCAGAGA	1
	GCAGAG	2
	GCAGA	3
	GCAG	4
	GCA	5
	GC	6
	G	7
	NULL	8
Nilai MH	7	

Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai MH	?	?	?	?	?	?	7	1

Tabel 2.40 Tabel perhitungan *Good-suffix shift* iterasi ke-3

Suffix Tabel		
Index	5	Move
Prefix	G	
Suffix	AG	
Suffix comparator	GCAGAGA	1
	GCAGGAG	2
	GCAGA	3
	GCAG	4
	GCA	5
	GC	6
	G	7
	NULL	8
Nilai MH	4	

Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai MH	?	?	?	?	?	4	7	1

Tabel 2.41 Tabel perhitungan *Good-suffix shift* iterasi ke-4

Suffix Tabel		
Index	4	Move
Prefix	A	
Suffix	GAG	
Suffix comparator	GCAGAGA	1
	GCAGAGAG	2
	GCAGA	3
	GCAG	4
	GCA	5
	GC	6
	G	7
	NULL	8
Nilai MH	7	

Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai MH	?	?	?	?	7	4	7	1

Tabel 2.42 Tabel perhitungan *Good-suffix shift* iterasi ke-5

Suffix Tabel		
Index	3	Move
Prefix	G	
Suffix	AGAG	
Suffix comparator	GCAGAGA	1
	GCAGAGAG	2
	GCAGA	3
	GCAG	4
	GCA	5
	GC	6
	G	7
	NULL	8
Nilai MH	2	

Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai MH	?	?	?	2	7	4	7	1

Tabel 2.43 Tabel perhitungan *Good-suffix shift* iterasi ke-6

Suffix Tabel		
Index	2	Move
Prefix	A	
Suffix	GAGAG	
Suffix comparator	GCAGAGA	1
	GCAGAG	2
	GCAGA	3
	GCAG	4
	GCA	5
	GC	6
	G	7
	NULL	8
Nilai MH	7	

Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai MH	?	?	7	2	7	4	7	1

Tabel 2.44 Tabel perhitungan *Good-suffix shift* iterasi ke-7

Suffix Tabel		
Index	1	Move
Prefix	C	
Suffix	AGAGAG	
Suffix comparator	GCAGAGA	1
	GCAGAG	2
	GCAGA	3
	GCAG	4
	GCA	5
	GC	6
	G	7
	NULL	8
Nilai MH	7	

Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai MH	?	7	7	2	7	4	7	1

Tabel 2.45 Tabel perhitungan *Good-suffix shift* iterasi ke-8

Suffix Tabel		
Index	0	Move
Prefix	G	
Suffix	CAGAGAG	
Suffix comparator	GCAGAGA	1
	GCAGAG	2
	GCAGA	3
	GCAG	4
	GCA	5
	GC	6
	G	7
	NULL	8
Nilai MH	7	

Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai MH	7	7	7	2	7	4	7	1

Setelah proses perhitungan pencarian nilai *good-suffix shift*, maka didapatkan Nilai MH yang dapat dilihat pada tabel 2.46.

Tabel 2.46 Tabel hasil perhitungan *good-suffix shift*

BmGs								
Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai MH	7	7	7	2	7	4	7	1

2.6.1.2. Tahap Pencarian

Setelah proses pada tahap *preprocessing* didapatkan nilai OH dan MH, Kemudian lakukan pencarian pada teks dengan menggunakan nilai OH dan MH yang telah dicari sebelumnya dengan aturan sebagai berikut :

1. Jika karakter yang dituju tidak terdapat pada tabel 2.33 yaitu tabel *BmBc*, maka nilai OH = panjang(*pattern*).
2. Cari selisih antara OH dengan jumlah karakter yang telah cocok.
3. Bandingkan OH dan MH, ambil nilai terbesar sebagai keputusan pergeseran.

Berikut penjelasan implementasi dalam sebuah contoh kasus pada tabel 2.47 yang sudah diketahui nilai dari OH atau *Bad-character shift* dan MH atau *Good-suffix shift* :

Tabel 2.47 Tabel *BmBc* dan *BmGs* hasil akhir berserta nilai

BmBc & BmGs								
Index	0	1	2	3	4	5	6	7
Karakter	G	C	A	G	A	G	A	G
Nilai OH	2	6	1	2	1	2	1	2
Nilai MH	7	7	7	2	7	4	7	1

Pattern yang telah diketahui nilai OH dan MH tersebut akan diimplentasikan pada contoh kasus teks yang terdapat pada tabel 2.48 berikut ini :

Tabel 2.48 Tabel teks yang akan dicari

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G

Langkah-langkah dalam pencarian sebagai berikut :

1. Pada iterasi ke-1, *pattern* akan membandingkan dari posisi terakhir *pattern* yaitu dari kanan, di posisi ke 7 pada tabel 2.49, karakter pada teks *A* tidak cocok dengan *pattern G*, maka lihat besar nilai MH pada tabel *Good-suffix shift* dan OH pada tabel *Bad-character shift* di tabel 2.47. Kemudian bandingkan nilai MH dan OH berdasarkan nilai maksimal untuk melakukan pergeseran.

$$BmGs\ [7] = BmBc[A]\ -7+7\ atau\ 1 = 1-7+7 = 1 \quad 5)$$

Sehingga geser *pattern* sebesar 1 posisi (nilai maksimal dari kedua tabel pergeseran).

Tabel 2.49 tabel pencarian *Boyer-Moore* iterasi ke-1

Index :	0	1	2	3	4	5	6	7		8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Teks :	G	C	A	T	C	G	C	A		G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
Pattern :	G	C	A	G	A	G	A	G																	

Ket :

	Cocok (*match*)
	Tidak cocok (*missmatch*)

2. Pada iterasi ke-2 *pattern* akan membandingkan dari posisi terakhir *pattern* yaitu dari kanan, di posisi ke 8 pada tabel 2.50, karakter pada teks *G* dan *A* terjadi kecocokan dengan *pattern G* dan *A*, maka lihat besar nilai MH pada tabel *Good-suffix shift* dan OH pada tabel *Bad-character shift* di tabel 2.47. Kemudian bandingkan nilai MH dan OH berdasarkan nilai maksimal untuk melakukan pergeseran

$$BmGs\ [7] = BmBc[A]\ -7+5\ atau\ 4 = 6-7+5 = 4 \quad (2.16)$$

Sehingga geser *pattern* sebesar 4 posisi (nilai maksimal dari kedua tabel pergeseran).

Tabel 2.50 tabel pencarian *Boyer-Moore* iterasi ke-2

Index :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Teks :	G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
Pattern :		G	C	A	G	A	G	A	G															

Ket :

	Cocok (*match*)
	Tidak cocok (*missmatch*)

3. Pada iterasi ke-3 *pattern* akan membandingkan dari posisi terakhir *pattern* yaitu dari kanan, di posisi ke 8 pada tabel 2.51, karakter pada teks *G,A,G,A,G,A,C* dan *G* cocok dengan *pattern* *G,A,G,A,G,A,C* dan *G* maka pencarian selesai.

Tabel 2.51 tabel pencarian *Boyer-Moore* iterasi ke-3

Index :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Teks :	G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
Pattern :						G	C	A	G	A	G	A	G											

Ket :

	Cocok (*match*)
	Tidak cocok (*missmatch*)

2.7. Object Oriented Programming (OOP)

Analisis dan desain berorientasi objek adalah cara baru dalam memikirkan suatu masalah dengan menggunkan model yang dibuat menurut konsep sekitardunia nyata. Dasar pembuatan adalah objek, yang merupakan kombinasi antara struktur data dan perilaku dalam suatu entitas. Model berorientasi objek bermanfaat untuk memahami masalah, komunikasi dengan ahli aplikasi, pemodelan suatu organisasi, menyiapkan dokumentasi serta perancangan program dan basis data. Pertama-tama suatu model analisis dibuat untuk menggambarkan aspek dasar dari domain aplikasi, dimana model tersebut berisi objek yang terdapat dalam domain aplikasi termasuk deskripsi dari keterangan objek dan perilakunya. Secara spesifik, pengertian berorientai objek berarti bahwa mengorganisasi perangkat lunak sebagai kumpulan dari objek tertentu yang memiliki struktur data dan perilakunya. Hal ini yang membedakan dengan pemograman konvensional dimana struktur data dan perilaku hanya berhubungan secara terpisah. Terdapat beberapa cara untuk menentukan karateristik dalam pendekatan berorientasi objek, tetapi secara umum mencakup empat hal, yaitu identifikasi, klasifikasi, *polymorphism* (polimorfisme) dan *inheritance* (pewarisan).

1. Karateristik dari objek

Identitas berarti bahwa data diukur mempunyai nilai tertentu yang membedakan entitas dan disebut objek. Suatu paragraf dari dokumen, suatu windows dari workstation, dan raja putih dari buah catur adalah contoh dari objek. Objek dapat kongkrit, seperti halnya arsip dalam sistem, atau konseptual seperti kebijakan penjadualan dalam multiprocessing pada sistem operasi. Setiap objek mempunyai sifat yang melekat pada identitasnya. Dua objek dapat berbeda walaupun bila semua atributnya identik Klasifikasi berarti bahwa suatu kegiatan mengumpulkan data (atribut) dan perilaku (operasi) yang mempunyai struktur data sama ke dalam satu grup yang disebut kelas. Paragraf, window, buah catur adalah contoh dari kelas. Kelas merupakan abstraksi yang menjelaskan sifat penting pada suatu aplikasi dan mengabaikan yang lain. Setiap kelas menunjukan suatu kumpulan infinite yang mungkin dari objek. Suatu objek dapat dikatakan sebagai instans dari kelas, setiap instansi dari kelas mempunyai nilai individu untuk setiap nama atribut dan operasi, tetapi memiliki bersama atribut dan operasi dengan instansi lain dalam kelas.

2. Karateristik Metodologi Berorientasi Objek

Metodologi pengembangan sistem berorientasi objek mempunyai tiga karateristik utama yaitu:

1. *Encapsulation*

Encapsulation (pengkapsulan) merupakan dasar untuk pembatasan ruang lingkup program terhadap data yang diproses. Data dan prosedur atau fungsi dikemas dalam bersama-sama dalam suatu objek, sehingga prosedur atau fungsi lain dari luar tidak dapat mengaksesnya. Data terlindung dari prosedur atau objek lain kecuali prosedur yang berada dalam objek itu sendiri.

2. *Inheritance*

Inheritance (pewarisan) adalah teknik yang menyatakan bahwa anak dari objek akan mewarisi atribut dan metoda dari induknya langsung. Atribut dan

metoda dari objek induk diturunkan kepada anak objek, demikian seterusnya. Pendefinisian objek dipergunakan untuk membangun suatu hirarki dari objek turunannya, sehingga tidak perlu membuat atribut dan metoda lagi pada anaknya, karena telah mewarisi sifat induknya.

3. *Polymorphism*

Polymorphism (polimorfisme) yaitu konsep yang menyatakan bahwa sesuatu yang sama dapat mempunyai bentuk dan perilaku berbeda. Polimorfisme mempunyai arti bahwa operasi yang sama mungkin mempunyai perbedaan dalam kelas yang berbeda.

2.8. Java

Pada 1991, sekelompok insinyur Sun dipimpin oleh Patrick Naughton dan James Gosling ingin merancang bahasa komputer untuk perangkat konsumer seperti *cable TV Box*. Karena perangkat tersebut tidak memiliki banyak memori, bahasa harus berukuran kecil dan mengandung kode yang liat. Juga karena manufaktur-manufaktur berbeda memilih *processor* yang berbeda pula, maka bahasa harus bebas dari manufaktur manapun. Proyek diberi nama kode *Green*. Niklaus Wirth, pencipta bahasa Pascal telah merancang bahasa portabel yang menghasilkan *intermediate code* untuk mesin hipotesis. Mesin ini sering disebut dengan mesin maya (*virtual machine*). Kode ini kemudian dapat digunakan di sembarang mesin yang memiliki *interpreter*. Proyek *Green* menggunakan mesin maya untuk mengatasi isu utama tentang netral terhadap arsitektur mesin. Karena orang-orang di proyek Green berbasis C++ dan bukan Pascal maka kebanyakan sintaks diambil dari C++, serta mengadopsi orientasi objek dan bukan prosedural. Mulanya bahasa yang diciptakan diberi nama *Oak* oleh James Gosling yang mendapat inspirasi dari sebuah pohon yang berada pada seberang kantornya, namun dikarenakan nama Oak sendiri merupakan nama bahasa pemrograman yang telah ada sebelumnya, kemudian Sun menggantinya dengan Java.

Nama Java sendiri terinspirasi pada saat mereka sedang menikmati secangkir kopi di sebuah kedai kopi yang kemudian dengan tidak sengaja salah satu dari mereka menyebutkan kata Java yang mengandung arti asal bijih kopi. Akhirnya mereka sepakat untuk memberikan nama bahasa pemrograman tersebut dengan nama Java. Berdasarkan white paper resmi dari Sun, Java memiliki karakteristik berikut :

1) Sederhana (*Simple*)

Bahasa pemrograman Java menggunakan Sintaks mirip dengan C++ namun sintaks pada Java telah banyak diperbaiki terutama menghilangkan penggunaan pointer yang rumit dan *multiple inheritance*. Java juga menggunakan *automatic memory allocation* dan *memory garbage collection*.

2) Berorientasi objek (*Object Oriented*)

Java mengunakan pemrograman berorientasi objek yang membuat program dapat dibuat secara modular dan dapat dipergunakan kembali. Pemrograman berorientasi objek memodelkan dunia nyata kedalam objek dan melakukan interaksi antar objek-objek tersebut.

3) Terdistribusi (*Distributed*)

Java dibuat untuk membuat aplikasi terdistribusi secara mudah dengan adanya *librari networking* yang terintegrasi pada Java.

4) *Interpreted*

Program Java dijalankan menggunakan interpreter yaitu Java Virtual Machine (JVM). Hal ini menyebabkan *source code* Java yang telah dikompilasi menjadi Java *bytecodes* dapat dijalankan pada platform yang berbeda-beda.

5) Robust

Java mempuyai reliabilitas yang tinggi. *Compiler* pada Java mempunyai kemampuan mendeteksi *error* secara lebih teliti dibandingkan bahasa pemrograman lain. Java mempunyai *runtime-Exception handling* untuk membantu

mengatasi *error* pada pemrograman.

6) *Secure*

Sebagai bahasa pemrograman untuk aplikasi internet dan terdistribusi, Java memiliki beberapa mekanisme keamanan untuk menjaga aplikasi tidak digunakan untuk merusak sistem komputer yang menjalankan aplikasi tersebut.

7) *Architecture Neutral*

Program Java merupakan platform independent. Program cukup mempunyai satu buah versi yang dapat dijalankan pada platform berbeda dengan *Java Virtual Machine*.

8) *Portable*

Source code maupun program Java dapat dengan mudah dibawa ke platform yang berbeda-beda tanpa harus dikompilasi ulang.

9) *Performance*

Performance pada Java sering dikatakan kurang tinggi. Namun *performance* Java dapat ditingkatkan menggunakan kompilasi Java lain seperti buatan.

2.9. Netbeans

Netbeans adalah sebuah aplikasi *Integrated Development Environment* (IDE) yang berbasiskan Java dari Sun Microsystems yang berjalan di atas *swing*. *Swing* merupakan sebuah teknologi Java untuk pengembangan aplikasi dekstop yang dapat berjalan pada berbagai macam platform seperti windows, linux, Mac OS X dan Solaris. Sebuah IDE merupakan lingkup pemrograman yang di integrasikan ke dalam suatu aplikasi perangkat lunak yang menyediakan *graphic user interface* (GUI), suatu kode editor atau text, suatu *compiler* dan suatu *debugger*.

Netbeans juga dapat digunakan progammer untuk menulis, meng*compile*, mencari kesalahan dan menyebarkan program netbeans yang ditulis dalam bahasa pemrograman java namun selain itu dapat juga mendukung bahasa pemrograman lainnya dan program ini pun bebas untuk digunakan dan untuk membuat *professional dekstop*, *enterprise*, *web*, dan *mobile applications* dengan Java *language*, C/C++, dan bahkan *dynamic languages* seperti PHP, JavaScript, Groovy, dan Ruby.

NetBeans merupakan sebuah proyek kode terbuka yang sukses dengan pengguna yang sangat luas. *Sun Microsystems* mendirikan proyek kode terbuka NetBeans pada bulan Juni 2000 dan terus menjadi sponsor utama. Dan saat ini pun netbeans memiliki 2 produk yaitu platform netbeans dan netbeans IDE. *Platform* netbeans merupakan *framework* yang dapat digunakan kembali (*reusable*) untuk menyederhanakan pengembangan aplikasi desktop dan platform netneans juga menawarkan layanan-layanan yang umum bagi aplikasi dekstop, mengijinkan pengembang untuk fokus ke logika yang spesifik terhadap aplikasi. Fitur fitur yang terdapat dalam netbeans antara lain:

1. *Smart Code Completion* yaitu untuk mengusulkan nama variabel dari suatu tipe, melengkapi *keyword* dan mengusulkan tipe parameter dari sebuah method.
2. *Bookmarking* yaitu fitur yang digunakan untuk menandai baris yang suatu saat hendak kita modifikasi.
3. *Go to commands* yaitu fitur yang digunakan untuk jump ke deklarasi variabel, *source code* atau file yang ada pada project yang sama.
4. *Code generator* yaitu jika kita menggunakan fitur ini kita dapat menggenerate *constructor, setter* dan *getter method* dan yang lainnya.
5. *Error stripe* yaitu fitur yang akan menandai baris yang *error* dengan memberi *highlight* merah.