// Pseudocode Brute Force:

// procedure PencocokanString(input P : string, T : string, n, m : integer, output idx : integer)

// { Masukan: pattern P yang panjangnya m dan teks T yang panjangnya n. Teks T direpresentasikan sebagai string (array of character)

// Keluaran: lokasi awal kecocokan (idx)

// }

// Deklarasi

// i : integer

// ketemu : boolean

// Algoritma:

// i←0

// ketemu←false

// while (i ≤ n-m) and (not ketemu) do

// j←1

// while (j ≤ m) and (P\_j=T\_(i+j) ) do

// j←j+1

// endwhile

// { j > m or P\_j=T\_(i+j) }

// if j = m then { kecocokan string ditemukan }

// ketemu←true

// else

// i←i+1 {geser pattern satu karakter ke kanan teks }

// endif

// endwhile

// { i > n – m or ketemu }

// if ketemu then return i+1 else return -1 endif

Brute Force in Java:

public static int brutematch(String T, String P)

{ int n = T.length(); // n is length of text

int m = P.length();; // m is length of pattern

int j;

for(int i=0; i <= (n-m); i++) {

j = 0;

while ((j < m) && (T.charAt(i+j)== P.charAt(j)) ) {

j++;

}

if (j == m)

return i; // match at i

}

return -1; // no match

}

// end of brutematch()

// Pseudocode KMP:

// procedure PencocokanString(input P : string, T : string, n, m : integer, output idx : integer)

// { Masukan: pattern P yang panjangnya m dan teks T yang panjangnya n. Teks T direpresentasikan sebagai string (array of character)

// Keluaran: lokasi awal kecocokan (idx)

// }

// Deklarasi

// i, j, lps[m] : integer

// ketemu : boolean

// Algoritma:

// i ← 0

// j ← 0

// ketemu ← false

// BuatLongestProperSuffixArray(P, m, lps) { Membuat longest proper suffix array menggunakan fungsi BuatLongestProperSuffixArray }

// while (i < n) do

// if P[j] = T[i] then

// j ← j + 1

// i ← i + 1

// if j = m then { kecocokan string ditemukan }

// ketemu ← true

// idx ← i - j

// break

// endif

// else

// if j ≠ 0 then

// j ← lps[j-1]

// else

// i ← i + 1

// endif

// endif

// endwhile

// if ketemu then

// return idx

// else

// return -1

// endif

// function BuatLongestProperSuffixArray(P : string, m : integer, lps : integer[])

// { Masukan: pattern P yang panjangnya m

// Keluaran: longest proper suffix array (lps) }

// Deklarasi

// len ← 0

// i ← 1

// lps[0] ← 0

// while (i < m) do

// if P[i] = P[len] then

// len ← len + 1

// lps[i] ← len

// i ← i + 1

// else

// if len ≠ 0 then

// len ← lps[len-1]

// else

// lps[i] ← 0

// i ← i + 1

// endif

// endif

// endwhile

// }

KMP in Java:

public static int kmpMatch(String text,

String pattern)

{

int n = text.length();

int m = pattern.length();

int b[] = computeBorder(pattern);

int i=0;

int j=0;

while (i < n) {

if (pattern.charAt(j) == text.charAt(i)) {

if (j == m - 1)

return i - m + 1; // match

i++;

j++;

}

else if (j > 0)

j = b[j-1];

else

i++;

}

return -1; // no match

}// end of kmpMatch()

public static int[] computeBorder(String pattern)

{

int b[] = new int[pattern.length()];

fail[0] = 0;

int m = pattern.length();

int j = 0;

int i = 1;

while (i < m) {

if (pattern.charAt(j) == pattern.charAt(i)) {

//j+1 chars match

b[i] = j + 1;

i++;

j++;

}

else if (j > 0) // j follows matching prefix

j = b[j-1];

else { // no match

b[i] = 0;

i++;

}

}

return fail;

}// end of computeBorder()

// Pseudocode BM:

// procedure PencocokanString(input P : string, T : string, n, m : integer, output idx : integer)

// { Masukan: pattern P yang panjangnya m dan teks T yang panjangnya n. Teks T direpresentasikan sebagai string (array of character)

// Keluaran: lokasi awal kecocokan (idx)

// }

// Deklarasi

// i, j, skip : integer

// ketemu : boolean

// badCharTable[256] : integer

// Algoritma:

// i ← 0

// ketemu ← false

// BuatBadCharTable(P, m, badCharTable) { Membuat bad character table menggunakan fungsi BuatBadCharTable }

// while (i ≤ n - m) do

// j ← m - 1

// while (j ≥ 0) and (P[j] = T[i + j]) do

// j ← j - 1

// endwhile

// if j < 0 then { kecocokan string ditemukan }

// ketemu ← true

// idx ← i

// break

// else

// skip ← max(1, j - badCharTable[T[i + j]])

// i ← i + skip

// endif

// endwhile

// if ketemu then

// return idx

// else

// return -1

// endif

// function BuatBadCharTable(P : string, m : integer, badCharTable : integer[])

// { Masukan: pattern P yang panjangnya m

// Keluaran: bad character table (badCharTable) }

// Deklarasi

// i : integer

// for i ← 0 to 255 do

// badCharTable[i] ← m

// endfor

// for i ← 0 to m - 2 do

// badCharTable[P[i]] ← m - 1 - i

// endfor

// }

Boyer-Moore in Java:

public static int bmMatch(String text,

String pattern)

{

int last[] = buildLast(pattern);

int n = text.length();

int m = pattern.length();

int i = m-1;

if (i > n-1)

return -1; // no match if pattern is

// longer than text int j = m-1;

do {

if (pattern.charAt(j) == text.charAt(i))

if (j == 0)

return i; // match

else { // looking-glass technique

i--;

j--;

}

else { // character jump technique

int lo = last[text.charAt(i)]; //last occ

i = i + m - Math.min(j, 1+lo);

j = m - 1;

}

} while (i <= n-1);

return -1; // no match

} // end of bmMatch()

public static int[] buildLast(String pattern)

/\* Return array storing index of last

occurrence of each ASCII char in pattern. \*/

{

int last[] = new int[128]; // ASCII char set

for(int i=0; i < 128; i++)

last[i] = -1; // initialize array

for (int i = 0; i < pattern.length(); i++)

last[pattern.charAt(i)] = i;

return last;

} // end of buildLast()

// Pseudocode Rabin-Karp:

// procedure PencocokanString(input P : string, T : string, n, m : integer, output idx : integer)

// { Masukan: pattern P yang panjangnya m dan teks T yang panjangnya n. Teks T direpresentasikan sebagai string (array of character)

// Keluaran: lokasi awal kecocokan (idx)

// }

// Deklarasi

// i, j, pHash, tHash, h : integer

// ketemu : boolean

// prime : integer

// Algoritma:

// prime ← 101 { Bilangan prima yang digunakan dalam Rabin-Karp }

// i ← 0

// ketemu ← false

// pHash ← hitungHash(P, m, prime) { Menghitung hash dari pattern P menggunakan fungsi hitungHash }

// tHash ← hitungHash(T, m, prime) { Menghitung hash dari substring pertama teks T menggunakan fungsi hitungHash }

// h ← power(prime, m - 1) mod prime { Nilai h yang akan digunakan dalam perhitungan rolling hash }

// while (i ≤ n - m) do

// if pHash = tHash then

// j ← 0

// while (j < m) and (P[j] = T[i + j]) do

// j ← j + 1

// endwhile

// if j = m then { kecocokan string ditemukan }

// ketemu ← true

// idx ← i

// break

// endif

// endif

// if i < n - m then

// tHash ← (prime \* (tHash - T[i] \* h) + T[i + m]) mod prime

// if tHash < 0 then

// tHash ← tHash + prime

// endif

// endif

// i ← i + 1

// endwhile

// if ketemu then

// return idx

// else

// return -1

// endif

// function hitungHash(s : string, len, prime : integer) : integer

// { Masukan: string s dengan panjang len, bilangan prima prime

// Keluaran: nilai hash dari string s }

// Deklarasi

// hashVal, i : integer

// hashVal ← 0

// for i ← 0 to len - 1 do

// hashVal ← (hashVal \* prime + s[i]) mod prime

// endfor

// return hashVal

// }

Rabin-Karp in Java:

public static int rabinKarpMatch(String text, String pattern) {

int last[] = buildLast(pattern);

int n = text.length();

int m = pattern.length();

int i = m - 1;

if (i > n - 1)

return -1; // no match if pattern is longer than text

int j = m - 1;

do {

if (pattern.charAt(j) == text.charAt(i)) {

if (j == 0)

return i; // match

else { // looking-glass technique

i--;

j--;

}

} else { // character jump technique

int lo = last[text.charAt(i)]; // last occurrence

i = i + m - Math.min(j, 1 + lo);

j = m - 1;

}

} while (i <= n - 1);

return -1; // no match

}

public static int[] buildLast(String pattern) {

int last[] = new int[128]; // ASCII char set

for (int i = 0; i < 128; i++)

last[i] = -1; // initialize array

for (int i = 0; i < pattern.length(); i++)

last[pattern.charAt(i)] = i;

return last;

}