Aplikasi Algoritma String Matching pada DNA untuk Mendeteksi Penyakit Genetik

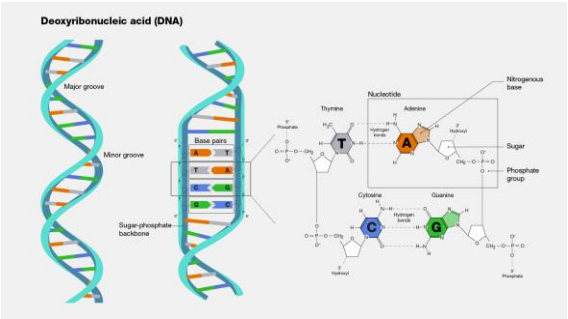
Fedrianz Dharma - 13522090
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): Fedrianzd@gmail.com

***Abstract*—Pencarian suatu *pattern* genetik pada DNA seseorang merupakan hal yang penting untuk dapat mendeteksi penyakit genetik yang mungkin dimiliki. Karena panjang dari DNA seseorang yang mencapai 3 miliar pasangan basa, maka diperlukan algoritma yang efisien sehingga dapat melakukan pencarian dengan cepat. Makalah ini akan membanding 3 algoritma *string matching*, yaitu Knuth-Morris-Pratt, Boyer-Moore, dan Aho-Corasick untuk mengetahui algoritma yang paling baik untuk digunakan pada *string matching* untuk DNA.**

Keywords—Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick, String Matching, DNA

I. PENDAHULUAN

DNA adalah molekul kompleks yang menyimpan informasi genetik makhluk hidup. DNA terdiri dari rangkaian nukleotida yang menetukan karakteristik masing-masing individu [1]. Informasi yang terkandung pada DNA bersifat unik dan dapat digunakan untuk berbagai aplikasi, salah satunya adalah mendeteksi penyakit genetik. Untuk mendapatkan informasi unik yang terdapat pada DNA, dapat dilakukan proses yang dinamakan DNA Sequencing.



Gambar 1. DNA
(sumber: https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid)

DNA Sequencing merupakan proses yang dilakukan untuk mendapatkan urutan nukleotida pada DNA. Hal ini mencakup semua teknik atau teknologi yang digunakan untuk menentukan urutan dari 4 basa, yaitu Adenine, Guanine, Cytosine, dan Thymine yang menyusun rantai DNA [2]. Setelah rantai DNA diekstrak dengan DNA Sequencing, DNA tersebut dapat dianalisis untuk menemukan *pattern* gen yang bermutasi atau menyebabkan penyakit.

Jika sudah diketahui *pattern-pattern* dari gen yang menyebabkan suatu penyakit genetik, maka DNA seseorang dapat diekstrak untuk dites apakah rantai DNA-nya mengandung pattern dari penyakit genetik yang sudah diketahui.

Melakukan pencarian *pattern* pada rantai DNA tidaklah mudah, mengingat genome pada manusia yang memiliki sekitar 3 miliar pasangan basa yang terdistribusi pada 23 kromosom [3]. Pencarian tersebut akan mungkin akan memerlukan waktu dan kekuatan pemrosesan yang besar. Oleh karena itu diperlukan algoritma yang dapat melakukan pencarian *pattern* dengan sangkil dan mangkus, seperti algoritma Knuth-Morris-Pratt dan algoritma Boyer-Moore. Selain itu, jika ingin dilakukan pencarian beberapa *pattern* sekaligus dapat menggunakan algoritma Aho-Corasick.

II. DASAR TEORI

A. *String Matching*

String matching atau *pattern matching* adalah kategori algoritma yang dapat digunakan untuk mencocokkan suatu pola atau *pattern* dalam teks yang memiliki jumlah karakter yang lebih banyak dibandingkan dengan pola tersebut. Algoritma *string matching* akan mengembalikan lokasi pertama di dalam teks yang bersesuaian dengan pola. Algoritma *string matching* memiliki berbagai macam aplikasi, seperti pencarian di dalam *text editor*, *web search engine*, analisis citra, dan bioinformatika.

Algoritma *string matching* dapat diklasifikasikan menjadi algoritma yang menggunakan satu *pattern* dan algoritma yang menggunakan lebih dari satu *pattern*. Contoh dari algoritma yang menggunakan satu *pattern* adalah algoritma Knuth-Morris-Pratt (KMP) dan algoritma Boyer-Moore (BM). Sedangkan algoritma yang menggunakan lebih dari satu *pattern* adalah algoritma Aho-Corasick.

B. *Algoritma Knuth-Morris-Pratt*

Algoritma KMP adalah algoritma *pattern-matching* yang dikembangkan oleh James H. Morris, Donald Knuth, dan Vaughan Pratt. Ide dibalik algoritma ini adalah ketika terjadi *mismatch*, posisi pada *pattern* sudah memberikan informasi yang cukup untuk melakukan pergeseran yang menghindari pencocokan yang tidak perlu dilakukan [4]. Hal ini dapat dicapai dengan menggunakan fungsi pinggiran atau *border function*.

$j = index\ terjadinya\ mismatch$ pada P[]

$k = index\ sebelum\ mismatch$

$$k = j - 1 \quad (1)$$

Border function b(k) didapatkan dari hasil preproses pada *pattern* untuk mencari prefix terbesar P[0..k] yang merupakan suffix P[1..k]. Hasil dari *b(k)* merupakan index pada *pattern* untuk memulai kembali pencocokan. Oleh karena itu, pencocokan tidak selalu dimulai kembali dari index 0, melainkan dari hasil *b(k)*.

Misalkan terdapat sebuah *pattern* DNA 'GTAGAGAG', maka *border function*-nya adalah sebagai berikut:

Tabel 1 *Border Function*

Pattern = GTAGAGAG

Length = 8

j	0	1	2	3	4	5	6	7
P[j]	G	T	A	G	A	G	A	G

k	0	1	2	3	4	5	6	-
b(k)	0	0	0	1	0	1	0	-

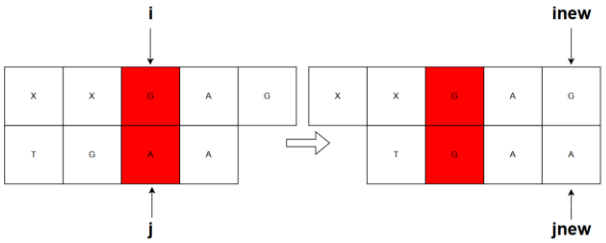
Misalkan terjadi *mismatch* pada index j = 6, maka *b(k) = b(5)* = 1. Pencocokan kembali akan mulai dilakukan dengan index 1 pada *pattern* dan bukan dari awal sehingga pencocokan yang tidak diperlukan dapat dikurangi.

Algoritma KMP memiliki kompleksitas waktu *O(m+n)* dengan kompleksitas waktu untuk menghitung *border function O(m)* dan kompleksitas waktu untuk pencarian pada teks *O(n)*. Kompleksitas ruang dari algoritma KMP adalah *O(m)*. Algoritma KMP bekerja dengan baik semakin sedikit variasi karakter yang ada pada teks. Semakin sedikit variasi karakter yang ada, kemungkinan *mismatch* akan terjadi pada bagian belakang *pattern* semakin tinggi [5]. Oleh karena itu, algoritma ini sangat cocok digunakan untuk melakuakan pencocokan DNA karena variasi karakter yang ada pada DNA hanyalah 4.

C. Algoritma Boyer-Moore

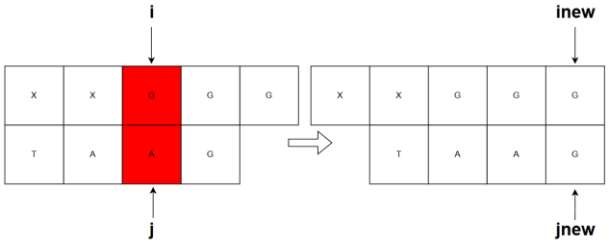
Algoritma Boyer-Moore adalah algoritma *string matching* yang dikembangkan oleh Robert S. Boyer dan J Strother Moore pada tahun 1977. Algoritma ini menggunakan 2 teknik, yaitu *looking-glass techinique* dan *character-jump technique*. *Looking-glass technique* adalah cara pencocokan yang dimulai dari index paling akhir dari *pattern* dan maju ke index paling depan dari *pattern*. Idenya adalah dengan melakukan pencocokan mulai dari kanan hingga ke kiri akan didapatkan lebih banyak informasi [6]. *Character-jump technique* adalah teknik pergeseran yang dilakukan ketika terjadi *mismatch* pada P[j] dengan T[i]. Terdapat 3 kasus pergeseran yang dapat dilakukan [5].

Kasus pertama adalah jika P mengandung G, geser P sehingga G yang terakhir pada *pattern* P sejajar dengan T[i].



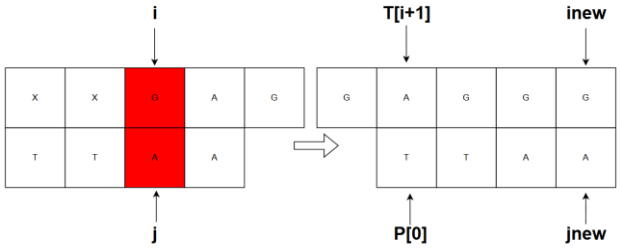
Gambar 2. Boyer-Moore Kasus Pertama

Kasus kedua adalah jika P mengandung G, namun pergeseran P ke kanan tidak memungkinkan, maka geser *pattern* sejauh 1.



Gambar 3. Boyer-Moore Kasus Kedua

Kasus Ketiga adalah jika kasus pertama dan kasus kedua tidak terjadi, maka geser P sehingga P[0] sejajar dengan T[i+1].



Gambar 4. Boyer-Moore Kasus Ketiga

Algoritma Boyer-Moore melakukan preproses pada *pattern* dan variasi karakter pada teks untuk membangun *Last Occurance Function L(x)*. *L(x)* didefinisikan sebagai index terbesar i sehingga P[i] = x atau -1 jika x tidak ada dalam *pattern*.

Misalkan terdapat sebuah *pattern* DNA 'GTAGAGAG', maka *Last Occurance Function*-nya adalah sebagai berikut:

Tabel 2. *Last Occurance Function*

A = {G, T, A}

P = GTAGAGAG

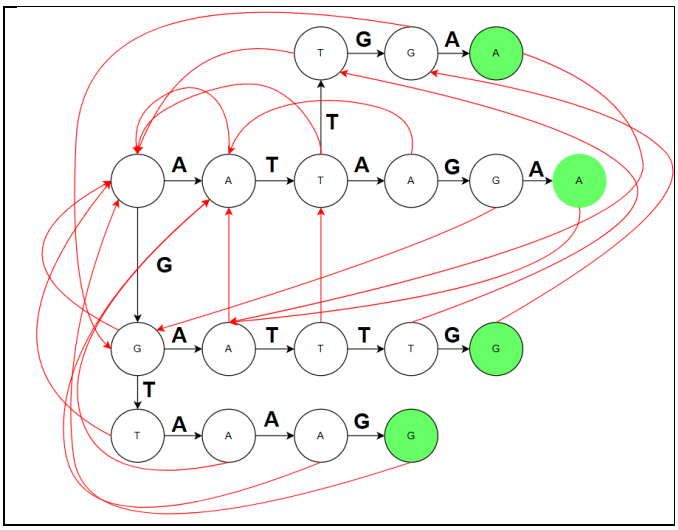
x	G	T	A	other
L(x)	7	1	6	-1

Algoritma Boyer-Moore memiliki komplesitas waktu terburuk *O(nm + A)*. Namun, algoritma Boyer-Moore cepat ketika variasi karakter pada teks atau alfabet (A) besar, dan lambat ketika alfabetnya kecil [5]. Oleh karena itu, untuk pencarian DNA, algoritma Boyer-Moore seharusnya tidak terlalu baik karena alfabet pada DNA hanya 4, yaitu A, G, C, T.

D. *Algoritma Aho-Corasick*

Algoritma Aho-Corasick adalah algoritma *string matching* yang diciptakan oleh Alfred V. Aho dan Margaret J. Corasick pada tahun 1975 [7]. Algoritma ini dapat mencari beberapa *pattern* sekaligus dalam suatu teks. Cara kerja algoritma ini mirip seperti algoritma KMP yang melakukan pencocokan mulai dari kiri ke kanan. Algoritma ini akan membangun *finite state pattern machine* untuk mencari semua *pattern* yang ada pada sebuah teks.

Misalkan terdapat sebuah *pattern* 'ATTGA', 'GTAAG', ATAGA', 'GATTG', maka akan terbentuk *finite state machine* yang menyerupai Trie sebagai berikut:



Gambar 5. *Finite State Machine Aho-Corasick*

Node yang bewarna hijau merupakan node terakhir dari sebuah *pattern*. Jika saat traversal node bewarna hijau dikunjungi, hal ini menandakan *pattern* yang dicari ditemukan pada teks. Panah bewarna merah menunjukkan *failure links* yang dapat digunakan untuk berpindah node ketika terjadi *mismatch*. *Failure links* didapatkan dari *suffix* terbesar yang dapat digunakan untuk traversal dari *root* saat terjadi *mismatch* pada suatu node [8]. Misalkan *pattern-pattern* tersebut akan dicari pada suatu teks 'GATAGA', maka traversal akan dilakukan pada pattern 'GATTG'. Pada saat terjadi *mismatch* pada index 3, yaitu 'T' pada pattern 'GATTG' dan 'A' pada teks, traversal akan berpindah ke index 1 pada pattern 'ATAGA', yaitu node 'T' yang kebetulan mempunyai *child node 'A'*. Traversal akan dilanjutkan hingga node bewarna hijau pada pattern 'ATAGA' yang menunjukkan bahwa pattern tersebut ada pada teks.

Algoritma Aho-Corasick mempunyai komplesitas waktu *O(n+m+z)* dengan z adalah total kemunculan pattern pada teks [9]. Algoritma ini sangat baik jika dibandingkan dengan algoritma KMP untuk melakukan pencarian beberapa pattern dalam satu teks. Algoritma KMP akan memiliki kompleksitas waktu *O(n*k + m)* [9]. Algoritma Aho-Corasick sangat cocok untuk digunakan pada pencarian beberapa pattern genetik penyakit sekaligus pada DNA seseorang.

III. IMPLEMENTASI

Algoritma KMP, BM, dan Aho-Corasick akan diimplementasikan menggunakan bahasa pemrograman Java.

A. *Algoritma KMP*

Pada implementasi algoritma KMP akan dibuat 2 fungsi, yaitu fungsi untuk membangun *border function* dan fungsi untuk melakukan *searching*. C*lass KMP* juga akan memiliki sebuah atribut *static* yang digunakan untuk menyimpan *pattern* apa saja yang sudah ditemukan.

```
static Set<String> found = new HashSet<>();
```

1. *borderFunc*

Fungsi *borderFunc* akan membuat *border* function yang memanfaatkan array statis. Fungsi ini akan selalu dijalankan sebelum pencarian dilakukan untuk preproses pattern terlebih dahulu dan membentuk *border function*. *Border function* digunakan untuk menentukan index pencocokan yang baru pada pattern ketika terjadi *mismatch*.

```
public static int[] borderFunc(String pattern){
        // inisialisasi array border
        int border[] = new int[pattern.length()];
        border[0] = 0;
        int m = pattern.length();
        int j = 0;
        int i = 1;
        while (i < m){
          if(pattern.charAt(j) == pattern.charAt(i)){
            border[i] = j + 1;
            i++; j++;
          }else if(j > 0){
            j = border[j-1];
          }else{
            border[i] = 0;
            i++;
          }
        }
        return border;
}
```

2. *kmpMatch*

Fungsi *kmpMatch* akan memanfaatkan fungsi *borderFunc* untuk membangun *border function*. Fungsi ini akan mengembalikan index pada text dimana pattern ditemukan. Fungsi ini akan mengembalikan -1 jika pattern tidak ditemukan pada text dan mengembalikan 1 jika pattern ditemukan.

```
public static int kmpMatch(String pattern, String text) {
        int n = text.length();
        int m = pattern.length();
        int border[] = borderFunc(pattern);
        int j = 0;
        int i = 0;
        while (i < n) {
           if (pattern.charAt(j) == text.charAt(i)) {
              if (j == m - 1) {
                 System.out.println("Found " + pattern + " in text at index " + (i-m+1));
                 found.add(pattern);
                 j = border[j-1];
              }
              i++;
              j++;
           } else if (j > 0) {
              j = border[j - 1];
           } else {
              i++;
           }
           if (i == n) {
              break;
           }
        }
        if(found.isEmpty()){
              return -1;
           }else{
              return 1;
           }
}
```

B. Algoritma Boyer-Moore

Pada implementasi algoritma BM akan dibuat 2 fungsi, yaitu fungsi untuk membangun *last occurance function* dan fungsi untuk melakukan *searching*. C*lass BM* juga akan memiliki sebuah atribut *static* yang digunakan untuk menyimpan *pattern* apa saja yang sudah ditemukan.

```
static Set<String> found = new HashSet<>();
```

1. BuildLast

Fungsi *BuildLast* akan membuat *last occurance function* yang memanfaatkan HashMap. Fungsi ini akan selalu dijalankan sebelum pencarian dilakukan untuk preproses pattern terlebih dahulu dan membentuk *last occurance function*. *Last occurance function* digunakan untuk menentukan index pencocokan yang baru ketika terjadi *mismatch*.

```
public static Map<Character, Integer> BuildLast(String pattern){
        Map<Character, Integer> last = new HashMap<Character, Integer>();
        for (int i = 0; i < pattern.length(); i++)
        {
           last.put(pattern.charAt(i), i);
        }
        return last;
}
```

2. BMmatch

Fungsi *BMmatch* akan memanfaatkan fungsi *BuildLast* untuk membangun *last occurance function*. Fungsi ini akan mengembalikan index pada text dimana pattern ditemukan. Fungsi ini akan mengembalikan -1 jika pattern tidak ditemukan pada text dan mengembalikan 1 jika pattern ditemukan.

```
public static int BMmatch(String pattern, String text){
        Map<Character, Integer> last = BuildLast(pattern);
        int n = text.length();
        int m = pattern.length();
        if (m > n){
           return - 1;
        }
        int i = m - 1;
        int j = m - 1;
        do{
           if(pattern.charAt(j) == text.charAt(i)){
              if(j == 0){
                 System.out.println("Found " + pattern + " in text at index " + (i));
                 found.add(pattern);
                  int lo = last.getOrDefault(text.charAt(i), -1);
                  i = i + m - Math.min(j, lo + 1);
                  j = m - 1;
              }else{
                 j--; i--;
```

```
            }
        }else{
            int lo = last.getOrDefault(text.charAt(i), -1);
            i = i + m - Math.min(j, lo + 1);
            j = m - 1;
        }
    } while(i <= n - 1);
    if(found.isEmpty()){
            return -1;
    }else{
            return 1;
}}
```

C. Aho-Corasick

Pada implementasi algoritma Aho-Corasick akan dibuat sebuah *class* yang memiliki 2 fungsi dan 1 prosedur, yaitu fungsi untuk membuat *finite state machine*, fungsi untuk menentukan *state* berikutnya, dan fungsi untuk melakukan *search*. *Class AhoCorasick* juga memiliki beberapa atribut yang akan digunakan, yaitu MAXS, MAXC, out, fail, go, enumMap, dan found.

```
public class AhoCorasick {
        static int MAXS = 500; // maximum number of states
        static int MAXC = 4; // jumlah alphabet
        static List<List<Integer>> out = new ArrayList<>();
        static int fail[] = new int[MAXS];
        static int go[][] = new int[MAXS][MAXC];
        static Map<Character, Integer> enumMap = Map.ofEntries(Map.entry('A', 0), Map.entry('G', 1), Map.entry('C', 2), Map.entry('T', 3));
        static Set<String> found = new HashSet<>();
```

Atribut MAXS merupakan jumlah maksimum dari state. Nilai dari atribut MAXS akan sama dengan jumlah dari panjang pattern yang akan dicari. Atribut MAXC merupakan jumlah dari variasi alfabet yang digunakan. Nilai dari atribut MAXC adalah 4 karena pada DNA hanya ada 4 alfabet, yaitu ‘A’, ‘G’, ‘C’, ‘T’. Atribut out digunakan untuk menyimpan pattern yang ditemukan pada suatu state. Atribut fail digunakan untuk melakukan perpindahan state bila state saat ini tidak memiliki *child* untuk suatu karakter masukan. Atribut go digunakan untuk melakukan perpindahan state secara normal dari masing-masing pattern yang ada. Atribut enumMap digunakan untuk melakukan enumerasi alfabet ‘A’, ‘G’, ‘C’, ‘T’ menjadi 0, 1, 2, 3.

1. buildMachine

Fungsi buildMachine akan membentuk *finite state machine* dengan menggunakan matrix. Matrix go digunakan untuk menentukan hubungan antara suatu state dengan state berikutnya dengan masukan suatu alfabet. Misalnya go[0][1] = 2, artinya state 0 memiliki *child node* state 2 dengan masukan alfabet A.

Atribut *fail* digunakan untuk melakukan perpindahan state ketika terjadi *failure*. Misalnya go[2][0] dan go[2][1] terdefinisi, sedangkan go[2][2] dan go[2][3] tidak terdefinisi. Jika state saat ini adalah 2 dan mendapat masukan C, maka akan digunakan atribut *fail* untuk berpindah state.

Atribut *out* digunakan untuk menyimpan pattern yang ditemukan jika state dicapai. Misalnya, jika out.get(3) terdefinisi pattern “CG”, maka jika state berhasil mencapai state 3, artinya pattern “CG” ditemukan.

```
static int buildMachine(String arr[], int k){
        for(int i = 0; i < MAXS; i++){
            out.add(new ArrayList<>());
        }
        for(int i = 0; i < MAXS; i++){
            Arrays.fill(go[i], -1);
        }
        int states = 1;
        for(int i = 0; i < k; i++){
            String pattern = arr[i];
            int currState = 0;
            for(int j = 0; j < pattern.length(); j++){
                int ch = enumMap.get(pattern.charAt(j));
                if(go[currState][ch] == -1){
                    go[currState][ch] = states++;
                }
                currState = go[currState][ch];
            }
            out.get(currState).add(i); // tambahkan index word ke output
        }
        // untuk setiap character yang tidak punya edge dari root
        // dikasih edge dari root ke root
        for(int ch = 0; ch < MAXC; ch++){
            if(go[0][ch] == -1){
                go[0][ch] = 0;
            }
        }
        Arrays.fill(fail, -1);

        Queue<Integer> queue = new LinkedList<>();

        for(int ch = 0; ch < MAXC; ch++){
            if(go[0][ch] != 0){
                fail[go[0][ch]] = 0;
                queue.add(go[0][ch]);
```

```
            }
        }
        while(!queue.isEmpty()){
            int state = queue.poll(); // state sebelum
            for(int ch = 0; ch < MAXC; ch++){
                // terdefinisi child node dari state sebelum untuk ch
                if(go[state][ch] != -1){
                // FAIL STATE DARI STATE sebelum
                int failureState = fail[state];
                while(go[failureState][ch] == -1){
                    // pergi ke fail statenya fail state dari state sebelum
                    failureState = fail[failureState];
                }
                // fail state untuk state saat ini
                failureState = go[failureState][ch];
                fail[go[state][ch]] = failureState;
                // gabungkan output
                out.get(go[state][ch]).addAll(out.get(failureState));
                // masukkan ke dalam queue untuk diproses selanjutnya
                queue.add(go[state][ch]);
            }
            }
        }
        return states;
    }
```

2. nextState

Fungsi nextState digunakan untuk menentukan state berikutnya dari suatu state dengan suatu masukan alfabet. Fungsi ini akan memanfaatkan atribut *go* dan *fail*. Jika *go* untuk state saat ini dan input alfabet tidak terdefinisi, maka akan digunakan atribut *fail* untuk menentukan state berikutnya.

```
static int nextState(int currState, char currInput){
        int next = currState;
        int input = enumMap.get(currInput);
        while(go[next][input] == -1){
            next = fail[next];
        }
        next = go[next][input];
        return next;
}
```

3. search

Prosedur *search* akan menerima *array of pattern* yang akan dicari pada suatu *string* teks. Prosedur ini akan memanfaatkan fungsi *buildMachine* dan *nextState*. Pada saat melakukan traversal pada *finite state machine*, atribut *out* untuk state tersebut tidak kosong, maka pattern yang ditemukan akan dicetak pada layar.

```
static void search(String[] arrPattern, int k, String text){
        int numOfStates = buildMachine(arrPattern, k);
        int currState = 0;
        for(int i = 0; i < text.length(); i++){
            currState = nextState(currState, text.charAt(i));
            if(out.get(currState).isEmpty()){
                continue;
            }
            for(int index: out.get(currState)){
                System.out.println("Pattern " + arrPattern[index] + " found in text at index " + (i - arrPattern[index].length() + 1));
                found.add(arrPattern[index]);
            }
        }
        out.clear();
}
```

D. Auxiliary Function

Fungsi *generateRandomDNA* digunakan untuk menghasilkan rantai DNA secara *random* dengan panjang sesuai dengan masukan pengguna. Fungsi *generateRandomPattern* digunakan untuk menghasilkan *pattern* unik sebanyak *count* dan masing-masing *pattern* memiliki panjang *length*.

```
public static String generateRandomDNA(int length) {
    Random random = new Random();
    StringBuilder dna = new StringBuilder();
    String nucleotides = "ACGT";
    for (int i = 0; i < length; i++) {
        int randomIndex = random.nextInt(nucleotides.length());
        char randomNucleotide = nucleotides.charAt(randomIndex);
        dna.append(randomNucleotide);
    }
    return dna.toString();
}
public static String[] generateRandomPattern(int count, int length) {
    String[] res = new String[count];
    Set<String> patterns = new HashSet<>();
    Random random = new Random();
    StringBuilder pattern = new StringBuilder();
    String nucleotides = "ACGT";
    while (patterns.size() < count) {
        for (int i = 0; i < length; i++) {
            int randomIndex = random.nextInt(nucleotides.length());
            char randomNucleotide = nucleotides.charAt(randomIndex);
            pattern.append(randomNucleotide);
        }
        patterns.add(pattern.toString());
        pattern.setLength(0);
    }
    int i = 0;
    for (String pat : patterns) {
        res[i++] = pat;
    }
    return res;

}
```

E. *Cara Pemakaian*

Misalnya dimiliki suatu dataset yang berisi nama penyakit dan pattern genetiknya sebagai berikut:

Tabel 3. Nama penyakit dan contoh patternya (random)

Nama Penyakit	Contoh Pattern
Cystic Fibrosis	AGCTGCA
Anemia	TGCAGTA
Huntington's Disease	AACGTTC
Hemophilia	CGTACGA
Tay-Sachs Disease	GATCAGC

Data tersebut dimasukkan ke dalam HashMap untuk mengetahui penyakit apa saja yang mungkin diderita pemilik DNA. Rantai DNA di-*generate* secara *random* dengan contoh panjang 1000 karakter. Contoh penggunaan untuk algoritma KMP adalah sebagai berikut:

```
public static void main(String[] args) {
    Map<String, String> disease = Map.ofEntries(
        Map.entry("AGCTGCA", "Cystic Fibrosis"),
        Map.entry("TGCAGTA", "Anemia"),
        Map.entry("AACGTTC", "Huntington's Disease"),
        Map.entry("CGTACGA", "Hemophilia"),
        Map.entry("GATCAGC", "Tay-Sachs Disease"));
    String arr[] = { "AGCTGCA", "TGCAGTA", "AACGTTC", "CGTACGA", "GATCAGC" };
    String text = AuxFunc.generateRandomDNA(1000);

    long startTime;
    long stopTime;

    startTime = System.nanoTime();
    for (String pattern : arr) {
        KMP.kmpMatch(pattern, text);
    }
    stopTime = System.nanoTime();
    if(!KMP.found.isEmpty()){
        System.out.println("Person A diduga mempunyai penyakit: ");
        for(String pattern : KMP.found){
            System.out.println("    -" + disease.get(pattern));
        }
    }
    System.out.println("Waktu yang dibutuhkan: " + (stopTime-startTime)/1000 + " \u00B5s");
}
```

Untuk menggunakan algoritma lain, hanya perlu mengganti bagian algoritma *searching*-nya saja.

```
Found TGCAGTA in text at index 881
Found AACGTTC in text at index 369
Found GATCAGC in text at index 646
Person A diduga mempunyai penyakit:
    -Tay-Sachs Disease
    -Anemia
    -Huntington's Disease
Waktu yang dibutuhkan: 12091 µs
```

Gambar 6. Contoh hasil keluaran algoritma KMP

```
Found TGCAGTA in text at index 881
Found AACGTTC in text at index 369
Found GATCAGC in text at index 646
Person A diduga mempunyai penyakit:
    -Tay-Sachs Disease
    -Anemia
    -Huntington's Disease
Waktu yang dibutuhkan: 13490 µs
```

Gambar 7. Contoh hasil keluaran algoritma BM

```
Pattern AACGTTC found in text at index 369
Pattern GATCAGC found in text at index 646
Pattern TGCAGTA found in text at index 881
Person A diduga mempunyai penyakit:
    -Tay-Sachs Disease
    -Anemia
    -Huntington's Disease
Waktu yang dibutuhkan: 13843 µs
```

Gambar 8. Contoh hasil keluaran algoritma Aho-Corasick

Berdasarkan hasil keluaran, terlihat bahwa orang tersebut mempunyai 3 penyakit genetik, yaitu Tay-Sachs Disease, Anemia, dan Huntington's Disease. Hasil keluaran juga menunjukkan lokasi index pattern genetik tersebut ditemukan, serta waktu eksekusi dari masing-masing algoritma.

IV. Analisis Hasil Pengujian

Tiap algoritma akan dilakukan percobaan berdasarkan jumlah pattern dan panjang dari text terhadap waktu eksekusi.

Tabel 4. Hasil Percobaan Waktu Eksekusi Tiap Algoritma Berdasarkan Jumlah Pattern

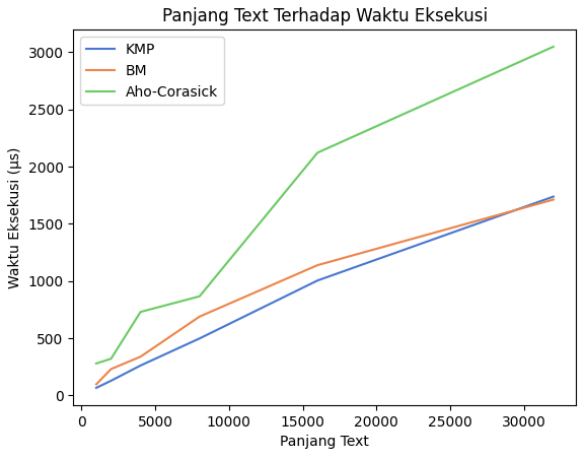
Panjang Text	Jumlah Pattern	Waktu Eksekusi (µs)		
		KMP	BM	Aho-Corasick
1000	1	63	105	218
1000	2	131	208	267
1000	4	291	286	256
1000	8	511	542	266
1000	16	1007	1527	338
1000	32	2633	3033	435

Tabel 5. Hasil Percobaan Waktu Eksekusi Tiap Algoritma Berdasarkan Panjang Text

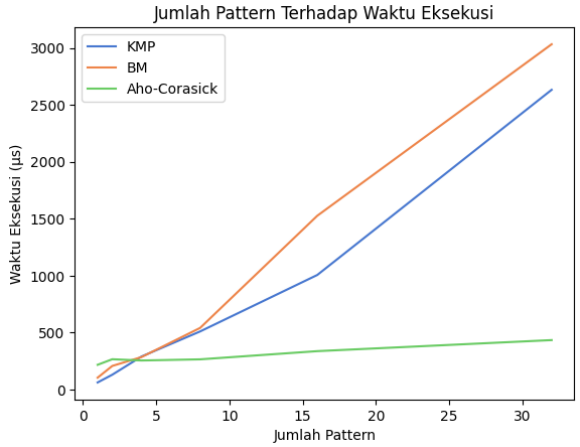
Panjang Text	Jumlah Pattern	Waktu Eksekusi (µs)		
		KMP	BM	Aho-Corasick
1000	1	67	98	280
2000	1	129	231	321
4000	1	262	339	730
8000	1	498	689	866
16000	1	1004	1138	2120
32000	1	1737	1711	3046

Tabel 6. Hasil Percobaan Waktu Eksekusi Tiap Algoritma Berdasarkan Panjang Text dan Jumlah Pattern

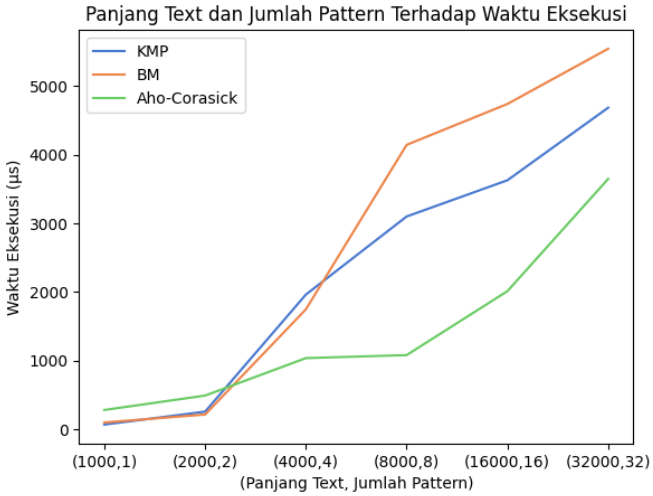
Panjang Text	Jumlah Pattern	Waktu Eksekusi (µs)		
		KMP	BM	Aho-Corasick
1000	1	67	98	280
2000	2	256	214	490
4000	4	1962	1747	1036
8000	8	3100	4144	1080
16000	16	3628	4740	2012
32000	32	4686	5545	3648



Gambar 9. Grafik Panjang Text Terhadap Waktu Eksekusi



Gambar 10. Grafik Jumlah Pattern Terhadap Waktu Eksekusi



Gambar 11. Grafik Panjang Text dan Jumlah Pattern Terhadap Waktu Eksekusi

Berdasarkan hasil percobaan, waktu eksekusi tiap algoritma berbanding lurus dengan panjang text untuk jumlah pattern pada text yang sama, seperti yang terlihat pada Gambar 9. Semakin besar panjang text, waktu ekesekusi akan bertambah besar sebanding dengan pertambahan panjang text.

Berdasarkan hasil percobaan, waktu eksekusi algoritma KMP dan BM berbanding lurus dengan jumlah pattern untuk panjang text yang sama. Namun, untuk algoritma Aho-Corasick, waktu eksekusinya relatif linear terhadap pertambahan jumlah pattern. Pertambahan waktu eksekusi untuk algoritma KMP dan BM setara dengan pertambahan jumlah pattern. Hal ini dapat dilihat dari Gambar 10.

Dari hasil percobaan juga terlihat bahwa untuk pencarian 1 pattern, algoritma Aho-Corasick memiliki waktu eksekusi paling lama. Namun, untuk pencarian dengan lebih dari 1 pattern, algoritma Aho-Corasick memiliki waktu eksekusi paling cepat. Pada Gambar 11 juga terlihat perbandingan waktu eksekusi ketiga algoritma untuk panjang text dan jumlah pattern yang terus meningkat. Terlihat pada Gambar 11 awalnya algoritma Aho-Corasick memiliki waktu eksekusi yang lebih lama, namun seiring pertambahan panjang text dan jumlah pattern, algoritma Aho-Corasick menjadi algoritma dengan waktu eksekusi yang paling cepat.

V. KESIMPULAN

Algoritma KMP dapat digunakan untuk melakukan *string matching* pada DNA untuk mencari pattern genetik yang menyebabkan penyakit dengan cukup efisien. Dibandingkan dengan algoritma BM, algoritma KMP memiliki waktu eksekusi yang lebih cepat. Namun, untuk pencarian beberapa pattern genetik, algoritma Aho-Corasick lebih efisien karena dapat mencari banyak pattern dalam sekali jalan. Algoritma Aho-Corasick memiliki waktu eksekusi yang paling cepat untuk pencarian beberapa pattern sekaligus di antara ketiga algoritma. Oleh karena itu, jika ingin melakukan pencarian hanya satu *pattern* gunakan algoritma KMP. Namun, jika ingin melakukan pencarian untuk beberapa pattern gunakan algoritma Aho-Corasick.

VIDEO LINK AT YOUTUBE

Video demo penjelasan makalah ini dapat dilihat pada *link* berikut:

https://youtu.be/ZEttrvw7hOI

GITHUB LINK

Source code yang digunakan pada makalah ini dapat dilihat pada *link* berikut:

https://github.com/FedrianzD/Makalah-Stima

UCAPAN TERIMA KASIH

Puji dan syukur kepada Tuhan Yang Maha Esa karena dengan anugerah-Nya penulis dapat menyelesaikan makalah ini dengan baik, tanpa kendala, dan tepat waktu. Penulis juga ingin mengucapkan terima kasih kepada Ibu Dr. Nur Ulfa Maulidevi, sebagai dosen pengampu mata kuliah IF2211 Strategi Algoritma tahun 2023/2024 kelas K02 atas bimbingannya sepanjang satu semester ini. Tak lupa penulis juga ingin mengucapkan terima kasih kepada orang tua yang senantiasa memberikan dukungan, serta seluruh pihak yang telah membantu dan mendukung penulis dalam menyelesaikan makalah ini.

REFERENSI

[1] S. A. Bates, "Deoxyribonucleic Acid (DNA)", Genome.gov, 2024. https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid. (diakses pada 10 Juni 2024)

[2] D. Adams, "DNA Sequencing," Genome.gov, Oct. 09, 2023. https://www.genome.gov/genetics-glossary/DNA-Sequencing#:~:text=DNA%20sequencing%20refers%20to%20the (diakses pada 10 Juni 2024)

[3] S. A. Bates, "Base Pair," Genome.gov, 2019. https://www.genome.gov/genetics-glossary/Base-Pair (diakses pada 10 Juni 2024)

[4] D. Knuthf, J. Morris, and V. Pratt, "FAST PATTERN MATCHING IN STRINGS*," SIAM Journal on Computing, vol. 6, no. 2, 1977, Available: https://www.cs.jhu.edu/~misha/ReadingSeminar/Papers/Knuth77.pdf (diakses pada 11 Juni 2024)

[5] Munir, Rinaldi, "Pencocokan String (String/Pattern Matching)." Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf (diakses pada 11 Juni 2024)

[6] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Communications of the ACM, vol. 20, no. 10, pp. 762–772, Oct. 1977, doi: https://doi.org/10.1145/359842.359859. (diakses pada 11 Juni 2024)

[7] A. V. Aho and M. J. Corasick, "Efficient string matching," Communications of the ACM, vol. 18, no. 6, pp. 333–340, Jun. 1975, doi: https://doi.org/10.1145/360825.360855. (diakses pada 11 Juni 2024)

[8] Niema Moshiri, "Advanced Data Structures: Aho-Corasick Automaton," YouTube, Apr. 26, 2020. https://www.youtube.com/watch?v=O7_w001f58c&ab_channel=NiemaMoshiri (diakses pada 11 Juni 2024)

[9] "Aho-Corasick Algorithm for Pattern Searching," *GeeksforGeeks*, Feb. 29, 2016. https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/ (diakses pada 11 Juni 2024)

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, 12 Juni 2024

Fedrianz Dharma 13522090