Perbandingan Kemangkusan Algoritma Pencocokan String terhadap Rantai Kode DNA

Aditias Alif Mardiano / 13518039
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13518039@std.stei.itb.ac.od

***Abstrak*— Mahluk hidup terdiri dari sistem biologi yang karakteristiknya bisa dikenali dari kode genetiknya. Kode gentik ini biasa dikenali dengan kumpulan rantai asam deoksiribonukleat dan sering dikenal dengan DNA (deoxyribonucleic acid). Rantai DNA ini tersusun dari biomolekul yang susunannya dapat dikenali kemiripannya dengan melakukan pencocokan kode DNA yang nantinya bisa dipakai untuk keperluan forensik, komputasi, sampai rekayasa genetika. Rantai kode DNA ini juga memiliki kode yang sangat panjang sehingga untuk mencari suatu pola pada rantai kode dibutuhkan suatu algoritma pencocokan string untuk memangkas waktu pencarian.**

Kata Kunci—DNA; Rabin-Karp; Algoritma Pencocokan String; hashing

I. PENDAHULUAN

Algoritma pencocokan string adalah salah satu materi yang diajarkan di mata kuliah IF2211 Strategi Algoritma. Pada mata kuliah ini, mahasiswa diajarkan untuk melakukan pemecahan suatu masalah melalui pendekatan secara matematis yang akan diimplementasikan untuk persoalan sehari-hari.

Salah satu persoalan sehari-hari ini adalah pencocokan rantai kode DNA. Pencocokan rantai kode DNA digunakan untuk keperluan forensik dan rekayasa genetika Rantai kode DNA ini sangat panjang sehingga untuk mencari suatu pola pada rantai kode tersebut, dibutuhkan usaha lebih untuk melakukan pencariannya. Salah satu pendekatan yang dapat dilakukan adalah menggunakan algoritma Rabin-Karp.

Algoritma Rabin-Karp adalah algoritma pencocokan String yang cocok untuk diterapkan di pencocokan kode DNA karena melakukan pendekatan persoalan menggunakan *hashing* dalam pencarian pola. *Hashing* adalahs semacam metode untuk mengkodekan string dengan cara memberi fungsi *hash* pada tiap karakter yang dicek dan menghitung nilainya.

Penggunaan Algoritma Rabin-Karp ini sangat cocok untuk pencocokan rantai kode DNA karena pada DNA sendiri ada empat jenis asam yang teridentifikasi yang berarti hanya akan ada empat jenis karakter untuk diterapkan kode hashnya. Sehingga pemangkasan waktu menggunakan algoritma Rabin-Karp akan sangat cocok dibandingkan menggunakan algortima pencocokan string yang lain.

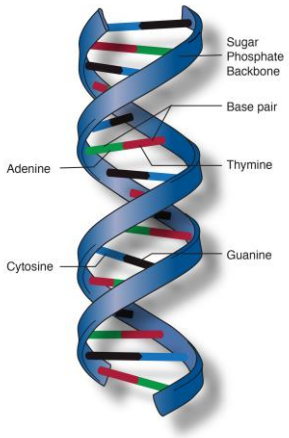
II. DASAR TEORI

A. *DNA*

DNA adalah asam amino yang menyimpan data karakteristik dari suatu sel, perlu diketahui bahwa dna memiliki kode unik kompleks yang sangat panjang yang berguna untuk kebutuhan data untuk kesehatan atau forensik.

Molekul pada DNA membentuk struktur yang seperti rantai dobel helix yang memiliki kandungan gula deoksiribosa dan fosfat. Gula deoksiribosa yang apada pada DNA ini terdiri dari Adenin (a), Sitosin (c), Guanin (g), dan Timin (t), untaian pada DNA terdiri dari gula ini dan membentuk ikatan tertentu dengan ketentuan Adenin dengan Timin membentuk ikatan 2 basa nitrogen, sedangkan Sitosin dengan Guanin membentuk ikatan 3 basa nitrogen.

Pada pengecekan rantai DNA, susunan dari gula Deoksiribosa inilah yang akan dicek menggunakan String Matching, namun pada dasarnya susunan DNA ini bisa berganti karena banyak faktor, seperti umur, terpapar radiasi, atau karena penyakit, dan lingkungan. Sehingga pengecekan DNA untuk keperluan forensik tidak akan menghasilkan data yang terlalu akurat karena data yang diambil sudah rusak karena sudah terbiarkan beberapa lama.



Gambar 2.1 Rantai dobel helix DNA

Sumber: https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid diakses pada tanggal 3 Mei 2020 pada pukul 18.37

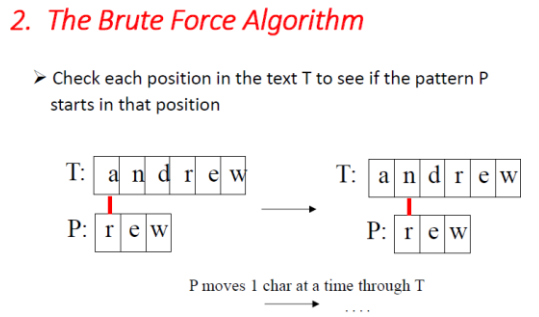
B. Algoritma Pencocokan String

Algoritma pencocokan string adalah algoritma yang digunakan dalam kehidupan sehari hari untuk memenuhi kebutuhan pencarian pola pada suatu teks atau data yang panjang dengan menggunakan solusi-solusi yang dapat memastikan ditemukan string pasti dan subset dari string. Algotima pencocokan string sendiri ada banyak di dunia, diantaranya:

1. Algoritma Brute Force

   Algoritma Brute Force adalah algoritma yang melakukan pendekatan secara sangat dasar dan algoritmanya cenderung mudah diterapkan. Algoritma ini paling tidak mangkus untuk diterapkan karena memakan waktu yang cukup lama namun tidak ada waktu untuk preprosesing.

   Teks dengan panjang m akan diproses dengan cara mengecek tiap posisi pada teks apakah ditemukan pattern dengan panjang n.



Gambar 2.2 Ilustrasi Algoritma Brute Force

Sumber: http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Pencocokan-String-(2018).pdf diakses pada tanggal 3 Mei 2020 pada pukul 11.52

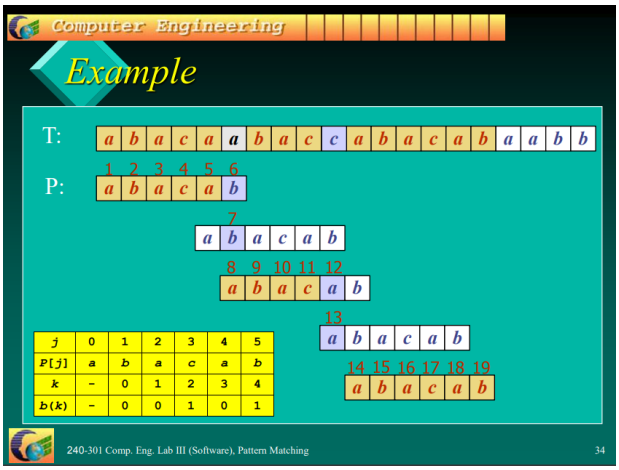
Kompleksitas waktu dari algoritma ini adalah *O(mn)*.

2. Algoritma Knuth-Morris-Pratt (KMP)

   Algoritma Knuth-Morris-Pratt adalah algoritma pencarian pattern dengan memanfaatkan *prefix, suffix,* dan fungsi gagal/border yang pendekatannya mirip dengan algoritma brute force, tetapi pergeseran dilakukan dengan lebih cerdas.

   Algortima KMP adalah sebagai berikut:

   1. Preproses pattern P dengan membuat fungsi border dengan cara mencari semua nilai *prefix* terbesar pada P[0..j-1] yang sama dengan *suffix* pada P[1..j-1] ketika terjadi mismatch pada P[j].
   2. Nilai ini lalu disimpan pada fungsi border b(k) dengan k adalah j-1
   3. Selanjutnya dilakukan pengecekan pada teks menggunakan *pattern* dengan mengecek tiap karakternya
   4. Jika terjadi *mismatch*, maka gunakan fungsi border dengan ketentuan
      a. Jika terjadi *mismatch* pada karakter pertama maka indeks pengecekan pada teks digeser sebanyak satu kali ke kanan.
      b. Jika terjadi *mismatch* selain pada karakter pertama maka dilakukan indeks pengecekan pattern akan di-*reset* dengan menggunakan nilai dari fungsi border.
   5. Jika tidak terjadi mismatch sampai indeks terakhir *pattern* maka fungsi akan mengembalikan true.
   6. Jika sampai akhir tidak ditemukan *pattern* maka algoritma akan mengembalikan false.



Gambar 2.3 Ilustrasi pencarian *pattern* pada string teks menggunakan algoritma KMP

Sumber: http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Pencocokan-String-(2018).pdf diakses pada tanggal 3 Mei 2020 pada pukul 12.22

Kompleksitas waktu dari menghitung fungsi border adalah *O(m)* dan pencarian string adalah *O(n)* dengan m adalah panjang pattern dan n adalah panjang teks sehingga algoritma ini memiliki kompleksitas waktu *O(m+n)* yang jauh lebih cepat dibandingkan Algoritma Brute Force.

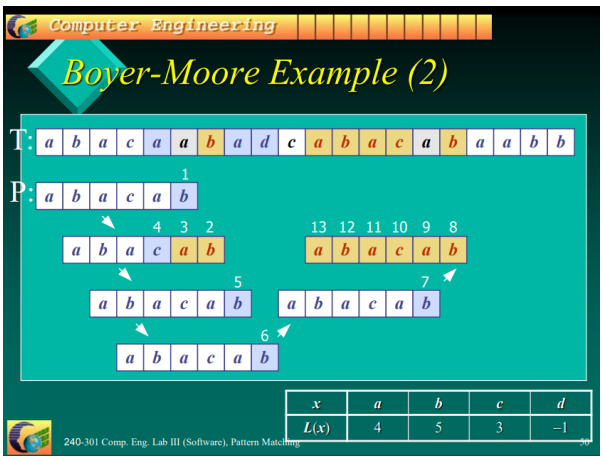
3. Algoritma Boyer-Moore (BM)

   Algoritma Boyer-Moore adalah algoritma pencarian pattern dengan memanfaatkan *last occurence*. *Last occurence* adalah indeks terbesar kemunculan suatu karakter pada *pattern*, dan pengecekan pada teks dan pattern akan dimulai dari indeks paling terakhir,

pergeseran dilakukan dengan The Character Jump Techiique.

Algoritma BM adalah sebagai berikut:

1. Preproses *last occurence* menggunakan indeks terbesar kemunculan suatu karakter pada *pattern*, jika tidak ada suatu karakter pada pattern maka nilai last occurencenya -1.
2. Character Jump Technique akan dilakukan jika terjadi mismatch pada pattern P[j] dan Teks T[i] dengan T[i] = x, terdapat 3 kasus:
    a. Jika terdapat x di P dengan indeks yang lebih kecil dari j (sebelah kiri pengecekan), maka seolah-olah geser p ke kanan agar posisi x di T[i] sejajar dengan posisi *last occurence* x di P
    b. Jika terdapat x di P dengan indeks yang lebih besar dari j (sebelah kanan pengecekan), maka seolah-olah geser p satu karakter ke kanan, agar posisi indeks terakhir di P sejajar dengan (posisi akhir T sebelumnya) + 1
    c. Jika karakter x tidak ditemukan pada P maka geser P agar posisi indeks pertama P(P[0]) sejajar dengan indeks i+1
3. Jika tidak terjadi mismatch sampai indeks pertama *pattern* maka fungsi akan mengembalikan true.
4. Jika sampai akhir tidak ditemukan *pattern* maka algoritma akan mengembalikan false.



Gambar 2.4 alstrasi pencarian *pattern* pada string teks menggunakan algoritma BM

Sumber: http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Pencocokan-String-(2018).pdf diakses pada tanggal 3 Mei 2020 pada pukul 13.07

Kompleksitas waktu algoritma BM adaah *O(mn)* dan merupakan cocok untuk pengecekan untuk karakter Bahasa Inggris.

4. Algoritma Rabin-Karp

Algoritma Rabin-Karp adalah algoritma pencarian pattern menggunakan yang lebih efisien dengan memanfaatkan *hashing value*

Algortima Rabin-Karp adalah sebagai berikut:

1. Diberikan suatu nilai pada tiap karakter yang akan digunakan untuk pencocokan dengan string
2. Dihitung nilai hash dari *pattern*, lalu dicek nilai hash dari text per panjang pattern
3. Bila nilai hash tidak sama maka dilakukan pergeseran indeks pengecekan pada teks
4. Bila nilai hash sama maka dicek apakah tiap karakter pada string sama, jika tidak maka akan bergeser, jika iya maka *pattern* ditemukan
5. Pencarian akan berhenti jika ditemukan *pattern* atau pengecekan string sudah keluar dari panjang indeks karakter pada teks.

Algoritma Rabin-Karp memiliki kompleksitas O(n+m) untuk mencari substring dengan panjang m dalam string dengan panjang n, namun algoritma ini memiliki kasus terburuk dengan kompleksitas O(nm). algoritma ini juga ditentukan oleh fungsi hash yang

Karena algoritma ini menggunakan fungsi hash untuk membandingkan pola dan teks yang disediakan. Oleh karena itu, kecepatan eksekusi program yang menggunakan algoritma ini juga ditentukan berdasarkan fungsi hash yang digunakan. fungsi hash akan mempengaruhi performa dari algoritma yang dibuat. Berikut ini adalah rolling hash dari algoritma Rabin-Karp yang digunakan:

$H = c_1a^{k-1} + c_2a^{k-2} + ... + c_ka^0$

Dengan keterangan c adalah nilai hash dari suatu karakter, a adalah suatu konstatnta, dan k adalah jumlah karakter pada string, dan H adalah nilai dari hashnya.

Pada pemrosesan DNA, ada 4 jenis basa nitrogen yaitu guanin dilambangkan dengan G, adenin dilambangkan dengan A, sitosin dilambangkan dengan C, dan timin yang dilambangkan dengan T, karena hanya ada 4 jenis karakter ini maka konstanta a yang dipakai adalah 4, nilai c adalah nilah hash, misalc pada G adalah 1, pada A adalah 2, pada C adalah 3, dan pada T adalah 4,

Contoh kasusnya adalah sebagai berikut: diberikan rantai DNA TCGA dan kita akan mencari kecocokan dengan pattern CGA, maka:

Maka nilai hash pattern-nya adalah:

$H(\text{“CGA”}) = 3 \times 4^2 + 1 \times 4^1 + 2 \times 4^0 = 48+4+2 = 54$

Lalu akan dicek pada teks nilai hash sepannjang pattern,

$H(\text{“TCG”}) = 4 \times 4^2 + 3 \times 4^1 + 1 \times 4^0 = 64+12+1 = 77$

Karena nilai hash tidak sama maka dilakukan pergeseran, kemudian dicek lagi namun memanfaatkan nilai hash sebelumnya dikurangi dengan nilai hash dari karakter pertama indeks yang dicek sebelumnya ditambah nilai hash dari karakter terakhir yang dicek sesudahnya

$H(\text{“CGA”}) = H(\text{“TCG”}) - H(\text{“T”}) + H(\text{“A”})$

,maka nilai hashnya:

$H(\text{“CGA”}) = (77 - 4 \times 4^2) \times 4 + 2 = 54$

Karena nilai dari hash sama, maka dicek lah dengan *pattern* kesamaannya, jika tidak sama maka akan dilakukan pergeseran lagi, jika iya maka fungsi akan mengembalikan true.

Sehingga dengan penerapan seperti ini maka pencarian suatu subset kode DNA pada suatu rantai yang sangat panjang akan sangat mangkus dibandingkan dengan algoritma yang lainnya.

III. Perbandingan Algoritma pencocokan String Pada Pengecekan Rantai DNA

Algoritma pencocokan string berikut akan diuji menggunakan algoritma KMP, Boyer-Moore dan Rabin-Karp. Kode yang digunakan pada makalah ini untuk algoritma KMP dan Boyer Moore merupakan dokumen dari penulis sendiri sedangkan untuk algoritma Rabin-Karp merupakan pengembangan dari algoritma yang telah ada dan bersumber dari https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/ (diakses pada tanggal 3 Mei 2020 pukul 15.28). Algoritma KMP dan BM dibuat dengan paradigma pemrograman berorientasi objek dengan menggunakan bahasa Python, Sedangkan untuk Rabin-Karp dalam bentuk prosedural.

```
class Text:
    #Constructor
    def __init__(self, text, pattern, border):
        self.text = deepcopy(text)
        self.pattern = deepcopy(pattern)
        self.border = deepcopy(border)
        self.foundPattern = [True for i in range(len(text))]
        self.characterInPattern = []
        self.lastOccurence = []
```

Gambar 3.1 Konstruktor dari teks yang akan diproses secara KMP dan BM
Sumber: dokumen penulis

A. Preprosesing

Pada bagian preprosesing, untuk algoritma KMP akan dilakukan persiapan fungsi border atau fungsi fail menggunakan kode berikut:

```
#Preproses fungsi border atau fungsi fail
def borderFunction(pattern):
    global border
    for i in range(len(pattern)):
        tempPre = ""
        tempSuf = ""
        prefix = []
        suffix = []
        b = 0
        if (i!=0):
            for j in range (i):
                tempPre = tempPre + pattern[j]
                prefix.append(tempPre)
                if (j != i-1):
                    tempSuf = pattern[i-1-j] + tempSuf
                    suffix.append(tempSuf)
            for k in range (i):
                if (k != i-1):
                    if (prefix[k] == suffix[k]):
                        b = k + 1
            border.append(b)
```

Gambar 3.2 Preprosesing fungsi border pada algoritma KMP
Sumber: dokumen penulis

Untuk algoritma BM dilakukan persiapan fungsi last occurence menggunakan kode berikut:

```
#Preproses Last Occurence menyetel karakter pada pattern
def setUniqueCharinPattern(self):
    # traverse for all elements
    for x in self.pattern:
        # check if exists in unique_list or not
        if x not in self.characterInPattern:
            self.characterInPattern.append(x)

#Preproses Last Occurence
def setlastOccurenceFunction(self):
    for j in range (len(self.characterInPattern)):
        stop = False
        temp = 0
        for index in range (len(self.pattern)):
            if (self.pattern[index] == self.characterInPattern[j]):
                temp = index
        self.lastOccurence.append(temp)
```

Gambar 3.3 Preprosesing last occurence pada algoritma BM
Sumber: dokumen penulis

Dan untuk algoritma Rabin-Karp dicari nilai hash dari *pattern* dan nilai hash pertama dari teks dengan panjang pattern menggunakan kode berikut:

```
# h = pow(d, M-1)%q
for i in range(M-1):
    h = (h*d)%q

# mengitung hash value untuk pattern dan yang dicek di awal pada teks
for i in range(M):
    p = (d*p + ord(P[i]))%q
    t = (d*t + ord(T[i]))%q
```

Gambar 3.4 Preprosesing hash *pattern* pada algoritma Rabin-Karp
Sumber: dokumen penulis

Dengan p adalah hash value dari pattern, t adalah hash value dari teks yang dicek dan q adalah bilangan prima yang dipilih.

B. *Data Uji*

Data uji yang didapat menggunakan data yang di *generate* oleh website

https://www.bioinformatics.org/sms2/random_coding_dna.html yang diakses pada tanggal 3 Mei pukul 16.00, dengan cara memasukkan panjang rantai kode, pada data uji kali ini penulis menggunakan panjang kode 1000.

```
atgcaccgcatattgtacgaagggcattctcctcatgccttccgaacagaagattggcac
tgggccgatcttgatgaactaccacttcttccggattgttgcacattaagattggcttgg
tccgaaactcgggttaaacccttctggctcaccgaccgtgaacgactcacctccgatccg
gacgctggtatacgggcagtgagtgtaggcgtcatcagtactaatgcgagcggctgtagg
gaccccaccattagcttaattgggcacctatacgcggtcagttcatatgtaagcatgaag
cccgctattgatcgagtgactatccaaatgctgatcgggcgtgtcgagataaggtcagct
tacagtcttggcgggtcacaacttaatgaggctagtcagagcgaggagttgtcttccata
tcactaacttatcttgtcaattggacgccaagagtcttttctcacgtgagtgccgtggaa
cacgccggaccttccttgaggtcgacccatattacccggccgacggtggcattggcgcca
ttgagtgtcttaaagttaaagtcgacaacatattgtggcaattgcaggtttttgccttgg
agtacggggcgcagagggagatgggtgaatatcggggttcgggggataggcgtcgtcgct
gccgaagggtgccaatacgcaggtttagatggttgggctctacacaatgcgctgccatat
tgtcaggaaaagggtgcagatggtgaacctatggcaaatttttgcgatggcccagaaaat
tctattgtagagatttcgaagggagggaggcatccccgtctgtctttggtgattcgattt
tacgatatggggcctagtgatctaagtatatacccactagaggttcgtgcgcatgtgccc
ttgctgtgtaagtcaaaagcgaccttgcccgataaagttctatcttatgcgttgctctac
ctcaccgtagtgtacggaacctcagtggcatttggaaagggagttcaggaagctatcaaa
agcctcggcggcaactgtgacctgctgtataagtggttgtggccggcagaattgcccact
actacttcgctttatgtgccacgttcgcggttcgccacgcagattgtaatagggatcggt
cagccaccacactcctttgcaagccatctttctgactgctcgaatctctccgggggaacag
tcctccgggccagatgtttcacactgccggtctcggaatcatcggccgtgtgccgccctg
catttagttttaccatctcgtcgcgcgaacctccgtactgagctgggacatggaacttttt
gtcacaatagatagtgaatttgcgctcaggtcaattaggaggagcctattgacagagtcc
atctggttctggatgcgagagaagaaaaagctaagagttcaagcagcgaggaagcgtgtt
aacaaaactcctttacgcgctcccatgcttcggcttgagatcttaacaaacttcgtatgg
gtaccgatgcgccttggttgcctgtatgctattctatcatcccgagtagggttaatcaag
accatacgaggcagtggtaaacatcataacacaggaagagtggcagttacatcctcaatg
ttgtcttttgcagagcctcgatgggtacttcgaccatgcattgttagaacgatgtgggcc
agctcggctataagcacagttgtcgatggccgaccgtacgtaacccgtagaattttccaa
gtgagttgctggcggtcggatgtcaccgtaaaacctgagatttgttctatttaccacact
gactcacaaacgcccgccaatgcggtgtcaggcctccttgagggctcactaagagacgtg
actccggcaagctcatctaactacggtttgctgcaaagtagtcaaatagtcagcttccgc
agtgccccaccgtctctaatggatccgcgtcaagcctgtgcgtatatcgtaatacgacaa
atcttggcatcgacacggcgattcgtggacatgacggacaagcgccgagggtcaacggaa
gtgcccgtaatgatccctattatattgcaactgaacctgcagtacatctcagggacaagc
caaagcctttcaacgcgggttaaattatcttccttcgaccctaaaaatgagtgccctata
```

Gambar 3.5 Data uji rantai DNA
Sumber:
https://www.bioinformatics.org/sms2/random_coding_dna.html diakses pada tanggal 3 mei 2020 pada pukl 16.00

C. Pengecekan teks

Berikut kode masing masing untuk pengecekan yang dilakukan masing-masing string matching:

1. KMP

```python
#Algoritma KMP
def KMP(self):
    for s in range (len(self.text)):
        i = 0
        temp = (self.text[s]).lower()
        while i < len(temp):
            j = 0
            while j < (len(self.pattern)):
                if(i == len(temp)):
                    self.foundPattern[s] = False
                    break
                if (temp[i] != self.pattern[j]):
                    if (j == 0):
                        i+=1
                    else:
                        j = self.border[j-1]
                else:
                    j += 1
                    i += 1
            break
```

Gambar 3.6 Algoritma KMP
Sumber: dokumen penulis

2. Boyer-Moore

```python
#Algoritma BM
def BM(self):
    self.setUniqueCharinPattern()
    self.setlastOccurenceFunction()
    for s in range (len(self.text)):
        i = 0
        temp = (self.text[s]).lower()
        while ((i < len(temp)) and (i>=0)):
            j = len(self.pattern) -1
            while j >= 0:
                if(i >= len(temp)):
                    self.foundPattern[s] = False
                    break
                if (temp[i] != self.pattern[j]):
                    lo = self.getLastOccurence(temp[i])
                    if (lo == -1):
                        i = i + len(self.pattern)
                    elif(lo < j):
                        i = i + len(self.pattern) - (lo + 1)
                    else:
                        i = i + len(self.pattern) - j
                    j = len(self.pattern) -1
                else:
                    j -= 1
                    i -= 1
            break
```

Gambar 3.7 Algoritma BM
Sumber: dokumen penulis

3. Rabin-Karp

```
# Pergeseran pada teks dilakukan satu per satu
for i in range(N-M+1):
    # jika hash value sama maka dicek
    # apakah karakter pada pattern dan teks yang dicek sudah sama
    if p==t:
        # pengecekan satu persatu
        for j in range(M):
            if T[i+j] != P[j]:
                break

        j+=1
        # jika p == t and P[0...M-1] = T[i, i+1, ...i+M-1] (sama semua)
        if j==M:
            print("Pattern ditemukan pada indeks: " + str(i))

    # menambahkan dengan hash yang baru dan mengurangi dengan yang lama
    # dikurangi dengan indeks pertama sebelumnya ditambah indeks terakhir sekarang
    if i < N-M:
        t = (d*(t-ord(T[i])*h) + ord(T[i+M]))%q

        # jika nilai t negatif, maka dipositifkan
        if t < 0:
            t = t+q
```

Gambar 3.8 Algoritma Rabin-Karp
Sumber: dokumen penulis

D. Cara Kerja Program

Program akan bekerja dengan cara memasukkan pattern yang ingin dicari lalu memasukkan file eksternal yang akan dicari patternya, hasil keluaran pada program ini adalah apakah ketemu atau tidak dan lama waktu pemrosesannya

E. Perbandingan Waktu Pemrosesan

Dilakukan pengecekan pada teks dna_cek.txt dan dicek waktu eksekusi masing-masing algoritma, pattern yang dicari adalah "atctggttctggatgcgagagaagaaaaagctaagagttcaagcagcgaggaagcgtgtt" yang ada pada teks.
Berikut keluaran dari program yang telah dibuat

1. KMP

```
F:\TugasMakalah>python KMP.py
Masukkan pattern string: atctggttctggatgcgagagaagaaaaagctaagagttcaagcagcgaggaagcgtgtt

Masukkan directory file eksternal: F:\TugasMakalah\dna_cek.txt
Menggunakan KMP:
Ketemu, di baris ke23
Total waktu yang dibutuhkan: 0.008747577667236328 s
```

Gambar 3.9 Hasil *screenshot* untuk waktu pemrosesan teks menggunakan KMP
Sumber: dokumen penulis

Waktu pemrosesan : 0,008747577667236328 s

2. Boyer-Moore

```
F:\TugasMakalah>python BM.py
Masukkan pattern string: atctggttctggatgcgagagaagaaaaagctaagagttcaagcagcgaggaagcgtgtt

Masukkan directory file eksternal: F:\TugasMakalah\dna_cek.txt
Menggunakan BM:
Ketemu, di baris ke23
Total waktu yang dibutuhkan: 0.009040355682373047 s
```

Gambar 3.10 Hasil *screenshot* untuk waktu pemrosesan teks menggunakan Boyer-Moore
Sumber: dokumen penulis

Waktu pemrosesan : 0,009040355682373047 s

3. Rabin-Karp

```
F:\TugasMakalah>python Rabin-Karp.py
Masukkan pattern string: atctggttctggatgcgagagaagaaaaagctaagagttcaagcagcgaggaagcgtgtt

Masukkan directory file eksternal: F:\TugasMakalah\dna_cek.txt
Menggunakan Rabin-Karp:
Pattern ditemukan pada indeks: 1380
Total waktu yang dibutuhkan: 0.004191875457763672 s
```

Gambar 3.9 Hasil *screenshot* untuk waktu pemrosesan teks menggunakan Rabin-Karp
Sumber: dokumen penulis

IV. KESIMPULAN

Berdasarkan pengujian yang telah dilakukan oleh penulis, dapat dibuktikan bahwa dalam kasus pengecekan rantai DNA menggunakan algoritma pengecekan string KMP, Boyer-Moore, dan Rabin-Karp, bahwa pengecekan menggunakan algoritma Rabin-Karp lebih mangkus daripada yang lain karena pengecekan dari Rabin-Karp akan dilakukan saat fungsi hashnya sama dan karena rantai dna yang panjang maka nilai dari hashnya akan varitif sehingga akan sangat jarang untuk ditermukan hit atau hash yang sama dengan isi yang berbeda.

VIDEO LINK PENJELASAN YOUTUBE

Berikut link untuk video penjelasan makalah ini: https://youtu.be/Mz7bA8vPHL0

REFERENSI

[1] Munir, Rinaldi, Diktat Kuliah IF2211 Strategi Algoritma. Bandung: Program Studi Teknik Informatika Institut Teknologi Bandung, 2009

[2] Alberts, dkk, Molecular Biology of the Cell. 4th edition. New York: Garland Science; 2002.

[3] https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid diakses pada tanggal 3 Mei 2020

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, 3 Mei 2020

Aditias Alif Mardiano
13518039