Mississippi

Aufgabe 3, Runde 2

Marian Dietz

1 Lösungsidee

1.1 Definitionen

Das Alphabet wird als $\sum$ bezeichnet. Dieses besteht, da es sich um Genome handelt, nur aus `A`, `C`, `G` und `T`:

$$\sum = \{A, C, G, T\}$$

Ein Wort ist die Konkatenation von Elementen aus $\sum$. Die Konkatenation von zwei Wörtern a und b wird durch ab dargestellt. Die Menge aller Wörter, die man aus dem Alphabet $\sum$ bilden kann, ist $\sum^*$ (Kleenesche Hülle). $|w|$ ist die Länge des Wortes w. w_i ist der i-te Buchstabe von w, wobei $i \in \mathbb{N}$ und $1 \leq i \leq |w|$.

Eine Teilzeichenkette eines Wortes w wird als Infix bezeichnet, d. h. ein Wort b ist ein Infix von w, wenn es zwei Wörter a und c gibt, sodass $abc = w$. Ein Suffix ist ein Infix am Ende des Wortes w, d. h. ein Wort b ist ein Suffix von w, wenn es ein Wort a gibt, sodass $ab = w$. Ein Präfix ist ein Infix am Anfang des Wortes w, d. h. ein Wort a ist ein Präfix von w, wenn es ein Wort b gibt, sodass $ab = w$. Für Infixe gilt, dass sowohl das leere Wort ein Infix von w, als auch w selber ein Infix von w ist. Das gilt ebenso für Suffixe und Präfixe, da sie auch Infixe sind.

Das Suffix s_i ist das Suffix, das mit dem i-ten Element von w beginnt und laut Definition bei $w_{|w|}$ endet, wobei $i \in \mathbb{N}$ und $1 \leq i \leq |w|$. $s_1 = w$, denn ein Wort ist stets auch Suffix von sich selbst. Das leere Suffix wird hier nicht definiert, da es für den Algorithmus nicht benötigt wird. Auch im Suffixarray ist es nicht enthalten. Wieso das nicht nötig ist, wird später an geeigneter Stelle erklärt.

l ist die Mindestlänge der Teilzeichenketten. $l \geq 1$ und $l \in \mathbb{N}$, da Teilzeichenketten, die weniger als ein Zeichen lang sind, weniger sinnvoll sind. k ist die Mindestanzahl der Teilzeichenketten. $k \geq 2$ und $k \in \mathbb{N}$, da nach **wiederholt** vorkommenden Basenfolgen gesucht werden soll, was nicht bei $k < 2$ nicht der Fall wäre.

1.2 Aufbau eines Suffixarrays

Für das Suchen von Teilzeichenketten werden Suffixarrays verwendet. Wie das funktioniert, soll an dem in der Aufgabenstellung genannten Beispiel, der Zeichenkette `CAGGAGGATTA` erklärt werden. Dabei ist $k = 2$ und $l = 2$, es müssen also Infixe gefunden werden,

die mindestens zwei Zeichen lang sind und mindestens 2 Mal in der Zeichenkette vorkommen.

Ein Suffixarray enthält alle möglichen Suffixe einer Zeichenkette. Die Tabelle zeigt das (noch unsortierte) Suffixarray für `CAGGAGGATTA`.

i	s_i
1	`CAGGAGGATTA`
2	`AGGAGGATTA`
3	`GGAGGATTA`
4	`GAGGATTA`
5	`AGGATTA`
6	`GGATTA`
7	`GATTA`
8	`ATTA`
9	`TTA`
10	`TA`
11	`A`

Damit nach mehrfach vorkommenden Teilzeichenketten gesucht werden kann, muss das Suffixarray sortiert werden, sodass es in einer lexikographischen Reihenfolge vorliegt. Für die vier Buchstaben des Genoms wird dabei einfach das normale Alphabet verwendet. D. h. `A` kommt vor `C`, `C` vor `G` und `G` vor `T`. Ein Präfix p eines anderen, längeren Wortes w erscheint in lexikographischer Reihenfolge vor w, da alle Buchstaben von p mit denen von w bis zum Ende von p übereinstimmen, p jedoch kürzer als w ist. Genauso ist es bspw. in Wörterbücher, denn darin erscheint „Informatik“ vor „Informatiker“.

Das Sortieren kann z. B. durch Mergesort geschehen. Welcher Sortieralgorithmus aber genau verwendet wird, ist egal und wirkt sich nicht auf die restlichen Schritte aus.

Das sortierte Suffixarray für `CAGGAGGATTA` würde dementsprechend folgendermaßen aussehen:

Zeile	i	s_i
1	11	`A`
2	2	`AGGAGGATTA`
3	5	`AGGATTA`
4	8	`ATTA`
5	1	`CAGGAGGATTA`
6	4	`GAGGATTA`
7	7	`GATTA`
8	3	`GGAGGATTA`
9	6	`GGATTA`
10	10	`TA`
11	9	`TTA`

1.3 Finden von Teilzeichenketten

Wichtig zu wissen ist, dass jedes Infix eines Wortes w ein Präfix eines Suffixes von w ist. Das ist einfach zu erkennen, kann aber dadurch begründet werden, dass es für jeden Buchstaben w_i von w ein Suffix s_i gibt. Wenn ein Infix mit w_i beginnt, kann es, im Gegensatz zu dem zugehörigen Suffix s_i, an einem beliebigen Buchstaben w_j aufhören, wobei $i \leq j$. w_j ist auch im Suffix enthalten, da das Ende vom Suffix s_i bei $w_{|w|}$, dem letzten Buchstaben von w liegt. Das Infix startet also mit dem Anfang von s_i und endet irgendwo im Suffix bei w_j. Daher ist jedes Infix ein Präfix eines Suffixes von w. Das bedeutet wiederum, dass jedes Infix bzw. jede Teilzeichenkette im Suffixarray zu finden ist, und zwar am Anfang von mindestens einem Suffix.

Das kann man sich zunutze machen, um wiederholt auftretende Infixe im Suffixarray zu suchen. Dafür beginnt man mit dem letzten Eintrag aus dem Suffixarray und verarbeitet alle Zeilen von unten nach oben. Bei jeder Zeile wird das Präfix p des Suffixes (und damit Infix der gesamten Zeichenkette) aus dieser Zeile mit der Länge l erstellt. Jetzt wird gezählt, wie oft das Präfix p in der Tabelle enthalten ist. Da das Array lexikographisch geordnet ist, muss man dafür nur nachzählen, wie viele der Suffixe in den darüberstehenden Zeilen mit dem Präfix p beginnen. Dafür wird nach oben hin für eine Zeile nach der anderen überprüft, ob sie mit dem Präfix p beginnt. Es kann aufgrund der lexikographischen Reihenfolge abgebrochen werden, wenn eine Zeile nicht das Präfix p besitzt. Die dabei entstehende Zahl n gibt an, wie oft p in der kompletten Zeichenkette insgesamt auftritt. Dies kann man daran begründen, dass p, wie oben beschrieben, ein Infix der Zeichenkette ist und durch dieses Vorgehen alle weiteren Infixe bzw. Präfixe von Suffixen gesucht werden, die genau gleich sind.

Ist $n \geq k$, dann muss das Präfix mit der zugehörigen Zahl n gespeichert werden, damit später überprüft werden kann, ob p maximal ist. Außerdem wird, wenn $n \geq k$, l um 1 erhöht. Anschließend wird nochmal nachgezählt, wie oft das daraus entstehende Präfix in der Tabelle auftaucht usw. Das ganze wird so oft gemacht, bis irgendwann ein Präfix nicht oft genug auftritt, also $n < k$. Dann wird das vorherige Element aus dem Suffixarray genommen und verarbeitet, wobei l natürlich wieder auf den Ursprungswert zurückgesetzt wird.

Soll irgendwann n für ein Präfix p berechnet werden, das schon einmal verarbeitet wurde, dann kann n, wenn es neu berechnet wird, nicht mehr korrekt sein, da immer nur in den darüberstehenden Zeilen nach p Ausschau gehalten wird, es jedoch in einer Zeile darunter schon einmal dieses Präfix p gegeben haben muss. Daher darf n niemals in unterschiedlichen Zeilen für dasselbe p berechnet werden, sondern muss immer in der untersten Zeile, in der es das erste Mal auftaucht, verarbeitet werden.

Elemente des Arrays, die nicht mehr genug Elemente vor sich haben (bzw. als Tabelle dargestellt Zeilen, die nicht mehr genügend Zeilen über sich haben), sodass durch sie keine mehrfach enthaltenen Infixe herausgefunden werden können, weil n nicht so groß wie k werden kann, können dabei einfach übersprungen werden.

Hier wird übrigens auch klar, wieso das leere Suffix nicht im Suffixarray stehen muss. Ein leeres Suffix mit der Länge 0 kann nur ein Präfix haben, welches ebenfalls die Länge 0 besitzt. Es wäre also sinnlos, sich damit zu beschäftigen, da keine Teilzeichenketten

mit der Länge 0 gesucht werden.

Folgendermaßen würde das für das Beispiel `CAGGAGGATTA` aussehen:

1. Es geht los mit dem letzten Element des Suffixarrays, in der Tabelle wäre das die Zeile 11. Dort steht s_9 mit dem Wert `TTA`. Da $l = 2$, müssen die Infixe mindestens die Länge zwei haben. Der Algorithmus startet also mit den ersten 2 Zeichen von s_9, `TT`. Es wird nachgezählt, wie viele Zeilen dieses Präfix besitzen. Dabei entsteht $n = 1$, diese Zeichenkette taucht also nur insgesamt einmal auf. Da $n < k$, darf `TT` nicht als valide Teilzeichenkette abgespeichert werden und es geht mit der nächsten Zeile weiter.

2. Es wird mit Zeile 10 fortgefahren. Dies ist s_{10} mit dem Wert `TA`. Auch hier gibt es keinen Erfolg, da das vorherige Suffix nicht mit `TA` beginnt.

3. Als nächstes wird s_6, `GGATTA` überprüft. Dabei wird herausgefunden, dass $n = 2$ für das Präfix `GG`, denn die vorherige Zeichenkette, `GGAGGATTA`, beginnt auch mit `GG`. Es wurde also ein Infix gefunden, das 2 Mal in der originalen Zeichenkette vorkommt. Es wird gespeichert, damit später überprüft werden kann, ob es maximal ist. Da die Überprüfung erfolgreich war und $n \geq k$, wird l um 1 erhöht. D. h. es wird jetzt n für `GGA` berechnet. Es entsteht $n = 2$, also ist auch `GGA` 2 Mal in der Zeichenkette enthalten und muss gespeichert werden. Es wird noch `GGAT` überprüft, diese Zeichenkette taucht jedoch nicht mehrmals auf ($n = 1$).

4. Es wird Zeile 8 berechnet. Darin ist s_3 mit `GGAGGATTA` enthalten. Das Präfix `GG` wurde schon einmal bearbeitet, daher darf n nicht noch einmal berechnet werden. Dasselbe gilt für `GGA`. Daher wird fortgefahren mit `GGAG`. Es entsteht $n = 1$ und es geht mit Zeile 7 weiter.

5. ...

6. Die Zeile 1 der Tabelle kann übersprungen werden, da n ohnehin nicht größer als 1 werden könnte, es wird jedoch ein Wert von mindestens 2 benötigt.

Wurden auf diese Weise alle Zeilen bearbeitet, dann entstehen die Infixe `GG`, `GGA`, `GA`, `AG`, `AGG` und `AGGA` die alle jeweils nur 2 Mal auftauchen.

1.4 Rausfiltern von nicht maximalen Teilzeichenketten

Jetzt müssen noch alle Teilzeichenketten rausgefiltert werden, die nicht maximal sind. Dafür werden einfach von jeder gespeicherten Teilzeichenkette alle Infixe gebildet.

Dies funktioniert für jede Teilzeichenkette w folgendermaßen: Das erste Infix startet bei w_1 und endet auch bei w_1. Für das nächste Infix wird der Endindex um 1 hochgezählt usw., bis die Endposition bei $w_{|w|}$ ankommt. Das nächste Infix startet bei w_2 und endet bei w_2. Für das nachfolgende Infix wird der Endindex wieder um 1 hochgezählt usw. Das ganze geht bis zum letzten Infix, dieses startet bei $w_{|w|}$ und endet bei $w_{|w|}$.

Immer, wenn ein Infix von w erstellt wurde, wird überprüft, ob dieses ebenfalls als Teilzeichenkette gespeichert ist, und wenn ja, ob es genauso oft auftritt wie w selbst. Ist

dies der Fall, dann ist es nicht maximal und kann gelöscht werden. Für das Infix, das bei w_1 startet und bei $w_{|w|}$ endet, wird natürlich nichts überprüft, da dieses Infix dem eigentlichen Wort w entspricht.

Im Beispiel sind von der Teilzeichenkette `AGGA` die Wörter `A`, `G`, `AG`, `GG`, `GA`, `AGG` und `GGA` Teilzeichenketten (wobei es eigentlich noch mehr gibt, sie sehen aber genauso aus wie diese Teilzeichenketten und werden daher ignoriert). Davon sind `AG`, `GG`, `GA`, `AGG` und `GGA` alle ebenfalls als Teilzeichenketten gespeichert, die wie `AGGA` je zwei Mal auftreten, und müssen daher gelöscht werden. Es bleibt zum Schluss also nur noch `AGGA` übrig.

Alle Teilzeichenketten, die jetzt gespeichert sind, halten alle Regeln ein: sie sind mindestens l Zeichen lang, treten mindestens k mal auf und sind maximal. Sie können ausgegeben werden.

2 Umetzung

Das Programm wurde objektorientiert in der Programmiersprache Java geschrieben. Zum Ausführen muss der Befehl „java -jar [Pfad]“ in einem Terminal eingegeben werden. Es wird Java 7 oder neuer benötigt. Es können auf gewöhnlichen Computern Zeichenketten der Länge 100000 in unterschiedlichen Konfigurationen (l und k) in nur wenigen Sekunden berechnet werden. Auch längere Zeichenketten sind problemlos möglich, jedoch wird, je länger die Zeichenkette ist, auch mehr Zeit in Anspruch genommen.

2.1 Klassen

Der Code besteht aus folgenden Klassen:

- `Mississippi` - liest die Daten ein, erstellt einen `SequenceFinder` und gibt die darin berechneten Teilzeichenketten aus.
- `SequenceFinder` - erstellt ein `SuffixArray` und sucht anhand von diesem alle Teilzeichenketten, die alle Bedingungen einhalten.
- `SuffixArray` - stellt ein Suffixarray dar. Es wird mit der Zeichenkette erstellt und erstellt anhand dieser ein Array aus Suffixen (`Suffix[]`).
- `Suffix` - stellt ein Suffix dar. Dafür speichert es den die gesamte Zeichenkette und den Index, an dem das Suffix startet.

2.2 `Mississippi`

Das Programm startet in der Methode `start()` der Klasse Mississippi. Von dort aus werden die Methoden `requestGenome()`, `requestL()` und `requestK()` aufgerufen, die das Genom, l und k mithilfe eines `Scanner`s aus dem Package `java.util` einlesen. Wenn die eingegebene Zeichenkette des Genoms ein Pfad zu einer Datei ist, wird der Inhalt aus dieser Datei mithilfe von Klassen aus dem Package `java.io` eingelesen, ansonsten wird die Eingabe selbst einfach als Genom verwendet. Wie genau das geschieht, ist jedoch für

den Algorithmus irrelevant. l und k sind dabei `int`s. Danach wird ein `SequenceFinder` erstellt. Die entstandenen Zeichenketten, die dieser in einem `Set<String>` durch die Methode `find()` zurückgibt, werden dann in der Methode `printSequences()` ausgegeben.

2.3 `SequenceFinder`

Ein `SequenceFinder` wird mit den drei zuvor in `Mississippi` eingelesenen Werten l, k und dem Genom erstellt. Es wird gleich im Konstruktor ein `SuffixArray` für das Genom erstellt.

Durch die Methode `find()` wird das Suchen der Teilzeichenketten gestartet. Dies geschieht, indem zuerst `sequences()` aufgerufen wird. Diese Methode sucht alle Teilzeichenketten, die mindestens l Zeichen lang sind und mindestens k Mal auftreten. Die zurückgegebene `Map<String, Integer>`, die jeweils eine Teilzeichenkette als Schlüssel und die zugehörige Anzahl des Auftretens als Wert enthält, wird dann der Methode `removeUnnecessarySequences()` übergeben, die die nicht maximalen Teilzeichenketten daraus entfernt. Ist auch das abgeschlossen, gibt die Methode `find()` das `keySet()` der `Map` zurück, da nur die Teilzeichenketten und nicht die zugehörigen Anzahlen benötigt werden.

Die Methode `sequences()` arbeitet folgendermaßen: Zusätzlich zur `Map<String, Integer>`, in die nacheinander alle validen Teilzeichenketten eingefügt werden, wird ein `Set<String>` erstellt. Dort werden immer alle bereits bearbeiteten Teilzeichenketten eingefügt, damit n, die Anzahl des Auftretens einer Teilzeichenkette, für kein Präfix mehrmals berechnet wird, was zu falschen Ergebnissen führen würde. Die Teilzeichenketten werden gesucht, indem in einer Schleife alle Elemente des Suffixarrays abgearbeitet werden. Für jedes dieser Suffixe werden solange die Präfixe davon bearbeitet und deren Anzahl durch die Methode `count()` des Suffixarrays berechnet, bis diese Anzahl irgendwann kleiner als k ist. Dann wird mit dem nächsten Suffix fortgefahren.

In `removeUnnecessarySequences()` wird ein `Set<String>` erstellt, das alle Teilzeichenketten enthalten soll, die gelöscht werden sollen. Um dieses `Set` aufzubauen, werden einfach alle Teilzeichenketten der `Map` durchlaufen. Von diesen werden wiederum in verschachtelten Schleifen alle Teilzeichenketten gebildet und dem `Set` hinzugefügt, wenn sie genauso oft auftreten. Ist das abgeschlossen, werden alle Teilzeichenketten, die jetzt im `Set` enthalten sind, aus der `Map` mit allen validen Teilzeichenketten gelöscht.

2.4 `SuffixArray`

Ein Suffixarray wird einfach mit einer Zeichenkette des Genoms erstellt. In einer Schleife werden alle möglichen Suffixe davon abgearbeitet und einem Array aus Suffixen (`Suffix[]`) hinzugefügt. Jedes Suffix wird mit derselben Instanz des Genoms erstellt, daher besteht kein erhöhter Speicherplatzbedarf. Zusätzlich dazu erhält jedes Suffix den zugehörigen Startindex. Danach wird das Array durch die Methode `Arrays.sort()` aus dem Package `java.util` mit dem Mergesort-Algorithmus sortiert.

Die Methode `count()`, die vom `SequenceFinder` benötigt wird, funktioniert mit einer Schleife, die alle Suffixe im Array vor dem angegebenen Suffix durchläuft und bei jedem

Durchgang den Zähler `count` hochzählt. Dieser wird zurückgegeben, wenn ein Suffix nicht mit dem gegebenen Präfix beginnt oder es keine vorherigen Suffixe mehr gibt.

2.5 `Suffix`

`Suffix` implementiert das Interface `Comparable<Suffix>`, damit die Suffixe im `SuffixArray` einfach sortiert werden können. Die Methode `compareTo()` vergleicht alle Zeichen von zwei Suffixen. Tritt irgendwann ein Unterschied auf (d. h. ein `char` ist größer bzw. kleiner als ein anderer), dann wird dementsprechend -1 oder $+1$ zurückgegeben, basierend darauf, welches `char` größer ist. Dies kann gemacht werden, da `char`s in Java intern durch Zahlen dargestellt werden. Dabei sind Buchstaben, die weiter vorne im Alphabet erscheinen, kleiner als andere Buchstaben. Tritt kein Unterschied bis zu dem Ende eines Suffixes auf, dann wird geguckt, welches Suffix kürzer ist und dementsprechend ein Wert zurückgegeben.

3 Beispiele

Die für die Beispiele verwendeten Genome, die auf der Website des Bundeswettbewerbs stehen, sind als eigene Dateien im Ordner dieser Aufgabe zu finden.

3.1 Beispiel 1

Dieses Beispiel zeigt das Ergebnis der schon in der Aufgabenstellung genannten und auch in dieser Dokumentation verwendeten Zeichenkette `CAGGAGGATTA` für $k = 2$ und $l = 2$. Auf einem gewöhnlichen Computer wurden ca. 5 Millisekunden für die Berechnung benötigt.

```
Zeichenkette oder Pfad zu einer Datei mit einer Zeichenkette
    eingeben:
CAGGAGGATTA
Wie oft sollen die Teilzeichenketten mindestens auftreten (k)?
2
Wie lang sollen die Teilzeichenketten mindestens sein (l)?
2
Sequenzen werden gesucht...

Es gibt 1 valide Teilzeichenkette(n):
AGGA
```

3.2 Beispiel 2

Dies sind die Teilzeichenketten vom ersten Beispiel, das auf der Website vom Bundeswettbewerb Informatik steht, für $l = 6$ und $k = 25$. Die Zeichenkette ist verfügbar unter `http://www.bundeswettbewerb-informatik.de/fileadmin/templates/`

`bwinf/aufgaben/bwinf33/chrM.fa`. Auf einem gewöhnlichen Computer wurden ca. 320 Millisekunden für die Berechnung benötigt.

```
Zeichenkette oder Pfad zu einer Datei mit einer Zeichenkette
    eingeben:
[...]/chrM.fa
Wie oft sollen die Teilzeichenketten mindestens auftreten (k)?
25
Wie lang sollen die Teilzeichenketten mindestens sein (l)?
6
Sequenzen werden gesucht...

Es gibt 10 valide Teilzeichenkette(n):
AACCCC, ACCCCC, CTACTC, ACCCTA, CCCCAT, AAACCC, CCTAGC, AAAAAA,
    CACCCT, CCACCC
```

3.3 Beispiel 3

Dieses Beispiel zeigt eine Konfiguration für die Zeichenkette aus Beispiel 2, zu der es keine Teilzeichenketten gibt: $k = 10$ und $l = 10$. Diese Berechnung benötigte auf einem gewöhnlichen Computer ca. 190 Millisekunden.

```
Zeichenkette oder Pfad zu einer Datei mit einer Zeichenkette
    eingeben:
[...]/chrM.fa
Wie oft sollen die Teilzeichenketten mindestens auftreten (k)?
10
Wie lang sollen die Teilzeichenketten mindestens sein (l)?
10
Sequenzen werden gesucht...

Es gibt keine validen Teilzeichenketten.
```

3.4 Beispiel 4

Bei diesem Beispiel wird eine Zeichenkette verarbeitet, die 100000 Zeichen lang ist. Sie ist auf der Website vom BwInf unter `http://www.bundeswettbewerb-informatik.de/fileadmin/templates/bwinf/aufgaben/bwinf33/E.coli.100000` zu erreichen. Es werden $l = 9$ und $k = 10$ verwendet. Ein gewöhnlicher Computer benötigte zur Berechnung nur ca. 3,4 Sekunden.

```
Zeichenkette oder Pfad zu einer Datei mit einer Zeichenkette
    eingeben:
[...]/E.coli.100000
Wie oft sollen die Teilzeichenketten mindestens auftreten (k)?
```

```
10
Wie lang sollen die Teilzeichenketten mindestens sein (l)?
9
Sequenzen werden gesucht...

Es gibt 5 valide Teilzeichenkette(n):
CCGGATGCG, CGCTGGCGC, CGCTGGAAG, GCGCTGGCG, GGCGCTGGC
```

3.5 Beispiel 5

Hier wird wieder die Zeichenkette aus Beispiel 4 verwendet. Daran kann man sehen, dass es insgesamt nur 2 Teilzeichenketten gibt, die mindestens 2 Mal auftreten ($k = 2$) und mindestens 50 Zeichen lang sind ($l = 50$). Es wurden ca. 2,8 Sekunden auf einem gewöhnlichen Computer zur Berechnung benötigt.

```
Zeichenkette oder Pfad zu einer Datei mit einer Zeichenkette
    eingeben:
[...]/E.coli.100000
Wie oft sollen die Teilzeichenketten mindestens auftreten (k)?
2
Wie lang sollen die Teilzeichenketten mindestens sein (l)?
50
Sequenzen werden gesucht...

Es gibt 2 valide Teilzeichenkette(n):
TGCCGGATGCGCTTTGCTTATCCGGCCTACAAAATCGCAGCGTGTAGGCC,
    GTAGGCCTGATAAGACGCGCCAGCGTCGCATCAGGCGTTGAATGCCGGATGCGCTTT-
    GCTTATCCGGCCTACAAAATCGCAGCG
```

3.6 Beispiel 6

Dieses Beispiel zeigt anhand der 100000er-Zeichenkette aus den Beispielen 4 und 5, wie viele wiederholte und maximale Teilzeichenketten es insgesamt gibt, ohne weitere Einschränkungen ($k = 2$ und $l = 1$). Dies sind genau 54544. Die vielen Teilzeichenketten können hier aus Platzgründen nicht gezeigt werden, sind aber als Datei im Ordner dieser Aufgabe zu finden. Diese Berechnung nahm auf einem gewöhnlichen Computer ca. 20 Sekunden in Anspruch.

```
Zeichenkette oder Pfad zu einer Datei mit einer Zeichenkette
    eingeben:
[...]/E.coli.100000
Wie oft sollen die Teilzeichenketten mindestens auftreten (k)?
2
Wie lang sollen die Teilzeichenketten mindestens sein (l)?
```

```
1
Sequenzen werden gesucht...

Es gibt 54544 valide Teilzeichenkette(n):
[...]
```

4 Quelltext

Listing 1: `Mississippi`

```java
public class Mississippi {

    public static void main(String[] args) {
        new Mississippi().start();
    }

    /**
     * Startet das Programm. Liest Daten ein, lässt die Teilzeichenketten
     * berechnen und gibt diese aus.
     */
    private void start() {

        // ...
        // genome (Zeichenkette), k und l wurden eingelesen

        Set<String> sequences = new SequenceFinder(genome, k, l).find();

        // Teilzeichenketten ausgeben...

    }

}
```

Listing 2: `SequenceFinder`

```java
/**
 * Sucht Teilzeichenketten einer Zeichenkette, die die Bedingungen k und l
 * einhalten.
 */
public class SequenceFinder {

    private SuffixArray array;
    private int k, l;

    public SequenceFinder(String genome, int k, int l) {
        this.array = new SuffixArray(genome);
        this.k = k;
```

```
        this.l = l;
    }

    /**
     * Sucht nach Teilzeichenketten, wofür zuersten alle Teilzeichenketten
     * berechnet werden. Danach werden diejenigen Teilzeichenketten
     * gelöscht, die nicht maximal sind.
     */
    public Set<String> find() {
        Map<String, Integer> sequences = sequences();
        removeUnnecessarySequences(sequences);
        return sequences.keySet();
    }

    /**
     * Berechnet alle Teilzeichenketten, die k und l einhalten.
     */
    private Map<String, Integer> sequences() {
        // Teilzeichenketten mit der Anzahl des Auftretens:
        Map<String, Integer> sequences = new HashMap<>();
        Set<String> checked = new HashSet<>(); // bereits überprüfte Präfixe

        int length = array.length();

        // Bei dem letzten Suffix beginnen und zu den vordersten Elementen
        // des Arrays alle durcharbeiten. Suffixe mit i < k - 1 müssen
        // nicht abgearbeitet werden, da sie nicht genug Suffixe vor sich
        // im Array haben können, um Präfixe zu haben, die oft genug
        // auftauchen.
        for (int i = length - 1; i >= k - 1; i--) {
            Suffix suffix = array.get(i);

            // Das Präfix des aktuellen Suffixes muss mindestens l Zeichen
            // lang sein und darf nicht länger als das eigentliche Suffix
            // sein.
            for (int prefixLength = l; prefixLength <= suffix.length();
                 prefixLength++) {

                // Präfix des Suffixes abfragen:
                String prefix = suffix.genome(prefixLength);

                // bereits bearbeitete Präfixe werden übersprungen:
                if (checked.contains(prefix))
                    continue;

                checked.add(prefix);

                // die Teilzeichenkette muss mindestens k-mal auftreten:
                int count = array.count(i, prefix);
```

```
            if (count < k) break;

            sequences.put(prefix, count);

        }
    }

    return sequences;
}

/**
 * Löscht alle nicht maximalen Zeichenketten aus der übergebenen Map.
 */
private void removeUnnecessarySequences(Map<String, Integer> sequences) {

    // die zu löschenden Zeichenketten:
    Set<String> remove = new HashSet<>();

    for (Map.Entry<String, Integer> entry : sequences.entrySet()) {

        // Wenn diese Zeichenkette schon gelöscht werden soll, muss sie
        // nicht überprüft werden:
        if (!remove.contains(entry.getKey())) {
            String string = entry.getKey();

            // Alle Teilzeichenketten von string bilden:
            for (int i = 0; i < string.length(); i++) {
                for (int j = i; j <= string.length(); j++) {

                    // j+1, da der Endindex von substring exklusiv ist.
                    String sub = string.substring(i, j + 1);

                    // sub.length() < string.length(), damit die
                    // aktuelle Zeichenkette nicht mit sich selbst
                    // verglichen wird. Objects.equals(), da sequences
                    // .get() auch null zurückgibt, ist die
                    // Teilzeichenkette nicht enthalten (das kann nur
                    // sein, wenn die Teilzeichenkette kürzer als l ist
                    // und daher nicht in die Map gekommen ist).
                    if (sub.length() < string.length() && Objects
                            .equals(sequences.get(sub), entry.getValue()))
                        remove.add(sub);
                }
            }
        }
    }

    // nicht maximale Zeichenketten rauslöschen:
    for (String string : remove)
```

```
            sequences.remove(string);
    }

}
```

Listing 3: `SuffixArray`

```
public class SuffixArray {

    private Suffix[] suffixes;

    /**
     * Erstellt ein neues Suffixarray anhand der übergebenen Zeichenkette.
     */
    public SuffixArray(String genome) {

        int length = genome.length();
        this.suffixes = new Suffix[length];

        for (int i = 0; i < length; i++)
            suffixes[i] = new Suffix(genome, i);

        Arrays.sort(suffixes);
    }

    public int length() {
        return suffixes.length;
    }

    public Suffix get(int i) {
        return suffixes[i];
    }

    /**
     * Zählt die Anzahl der Suffixe, die mit dem übergebenen Präfix
     * anfangen. Dafür werden die Suffixe vor dem Suffix mit dem
     * übergebenen Index überprüft, bis ein Suffix auftaucht, wofür die
     * Bedingung nicht eintritt.
     */
    public int count(int i, String prefix) {

        // Es wird davon ausgegegangen, dass das Suffix i mit Präfix startet:
        int count = 1;

        // Die Überprüfung startet mit dem vorherigen Suffix (i--), endet
        // spätestens beim ersten Suffix, nach jedem Durchgang wird i um eins
        // runtergezählt, damit das vorherige Suffix genommen wird und der
        // Zähler wird inkrementiert.
        for (i--; i >= 0; i--, count++)
            if (!suffixes[i].genome().startsWith(prefix))
```

```
                return count;

        return count;
    }

}
```

Listing 4: `Suffix`

```
public class Suffix implements Comparable<Suffix> {

    private String genome;
    private int index;

    /**
     * @param genome das gesamte Genom.
     * @param index der Startindex
     */
    public Suffix(String genome, int index) {
        this.genome = genome;
        this.index = index;
    }

    @Override
    public int compareTo(Suffix o) {
        if (this == o) return 0;

        // Die chars können nur bis zum letzten Zeichen des kürzeren
        // Suffixes verglichen werden:
        int length = Math.min(length(), o.length());

        for (int i = 0; i < length; i++) {
            if (this.charAt(i) > o.charAt(i))
                return +1;
            if (this.charAt(i) < o.charAt(i))
                return -1;
        }

        // Sind alle chars gleich, erscheint das kürzere Suffix weiter vorne.
        return this.length() - o.length();
    }

    public int length() {
        return genome.length() - index;
    }

    private char charAt(int charIndex) {
        return genome.charAt(index + charIndex);
    }
```

```
    public String genome() {
        return genome.substring(index);
    }

    public String genome(int end) {
        return genome.substring(index, index + end);
    }

}
```