Programmierpraktikum

AG Bioinformatik

SoSe 2020

Hinweis

Zu Ihrer Unterstützung sind in den entsprechenden Klassen bereits Lösungsansätze einzelner Methoden vorimplementiert. Machen Sie sich also erst einmal damit vertraut und arbeiten Sie erst dann die Aufgaben nacheinander ab. Dennoch müssen Sie in der Lage sein, bereits vorhanden Code zu verstehen und erklären zu können.

1 Einleitung

Das menschliche Genom besteht aus ca. $3,5 * 10^6$ Basen. In dieser unglaublich großen Datenmenge ist es so gut wie unmöglich, manuell diejenige Bereiche zu suchen in denen die Gene gespeichert sind, was effiziente Methoden zur automatisierten Analyse solcher riesigen Daten notwendig macht. Im Bereich der Lebenswissenschaften ist es einer der Aufgaben der Bioinformatik, Algorithmen zu entwickeln, welche das Auffinden bestimmter Bereiche auf dem Genom erleichtern. Letztere, in der Biologie auch als kodierende Bereiche bezeichnet, sollen das Ziel des vorliegenden Aufgabenteils sein.

1.1 Biologische Grundlagen

Ziel der ersten Aufgabe soll es sein, sich mit den biologischen Grundlagen vertraut zu machen. Hierzu finden Sie im Paket *ProgPrak2019/Bioinformatik/aufgabe1/* die Klasse *BioBasicsParser*. Im Laufe dieses Teils soll diese Klasse um nützliche Methoden erweitert werden, welche das Herauslesen von Informationen aus einem Wikipediaartikel ermöglichen.

1.1.1 Einlesen einer Textdatei (komplett) (5 Punkte)

Die Datei *genom.txt* enthält den Auszugs des Wikipediaartikels über das Genom[1]. Implementieren Sie eine Methode, die das zeilenweise Einlesen einer Textdatei erlaubt und speichern Sie jeden Zeile in einer Liste ab.

```
1 // Beispiel:
2 bioBasicsParser.loadEntireFile(file);
```

1.1.2 Einlesen einer Textdatei (gepuffert) (5 Punkte)

Für sehr große Dateien, bietet es sich an diese gepuffert einzulesen. Implementieren Sie eine Methode, welche mit Hilfe der Klasse *BufferedReader* die Datei *genom.txt* gepuffert einliest und speichern Sie jede Zeile in einer Liste ab.

```
1 // Beispiel:
2 bioBasicsParser.loadFileBuffered(file);
```

[1] https://de.wikipedia.org/wiki/Genom

1.2 Parsen eines Textes (15 Punkte)

Implementieren Sie eine Methode, z.B. *getInfoFor*, welche bei Übergabe eines bestimmten Schlüsselbegriffes den zugehörige Text ausgibt. Verwenden Sie Ihr Programm um mehr über die folgende Begriffe heraus zufinden:

1. Genom?
2. Codierender Abschnitt?
3. Gen?
4. Basen?

In diesem Teil der Aufgabe soll der Text aus der Datei *genom.txt* gelesen werden .

```
// Beispiel: (Quelle: genome.txt -> useFile = true)
List<String> info =
    bioBasicsParser.getInfoFor("Viral", 80, true);
System.out.println(info[0]);
// Ausgabe: Virale Genome sind sehr klein, da...
```

Hinweis: Verwenden Sie der Einfachheit halber Methoden aus der Klasse String, z.B. *indexOf*.

1.3 Parsen eines beliebigen Artikels (15 Punkte)

Um das Parsen eines beliebigen Wikipediaartikels zu erlauben, enthält die Klasse *BioBasicsParser* die noch fertig zu implementierende Methode *loadWikiPage(String url)* welche eine Url als Parameter übergeben werden muss und die Webseite (HTML) zeilenweise auf der Konsole ausgibt.

In HTML verfasste Texte enthalten sog. *Tags*, z.B.

```
<button>Klick mich!</button>
```

Diese Tags bestehen aus öffnenden und schließenden Elementen, die wiederum weitere Tags enthalten können. Da diese Elemente die Lesbarkeit sehr beeinträchtigen, soll eine Funktion hinzugefügt werden, die das Ersetzen dieser Sonderzeichen erlaubt:

```
// Beispiel:
String text =
    replaceTags("<a href=\"foo\">Prokaryoten</a>");
text.equals("Prokaryoten");
```

Verwenden Sie Ihren Ansatz aus Aufgabe 1.2 um die Indices von öffnenden ("<") und schließenden (">") Elementen zu erhalten und ersetzen Sie alle Vorkommen mit einer leeren Zeichenkette. D.h. der Tag "<p>" soll durch "" ersetzt werden. *Hinweis: Lassen Sie sich nicht von den vielen HTML-Tags auf der Webseite verwirren, es ist ausreichend wenn Ihre Implementation den wesentlichen Inhalt in lesbare Sätze übersetzt.*

Ausblick: Reguläre Ausdrücke sind ein mächtiges Werkzeug um in Texten nach Vorkommen bestimmter Muster zu suchen. In diesem Tutorial[2] können Sie nachlesen, wie Sie Aufgaben dieser Art in Zukunft eleganter lösen können.

[2] https://www.tutorialspoint.com/java/java_regular_expressions.htm

1.4 BioBasicsParser 2.0 (10 Punkte)

Ändern Sie die Klasse *BioBasicsParser* so ab, dass Ihre Lösung aus Aufgabe 1.2 nicht mehr die Datei als Quelle verwendet sondern die Funktion aus Aufgabe 1.3.

```
// Beispiel: (Quelle: URL -> useFile = false)
List<String> info =
    bioBasicsParser.getInfoFor("Viral", 80, false);
System.out.println(info[0]);
// Ausgabe: Virale Genome sind sehr klein, da...
```

2 API-Verwendung einer bioinformatischen Datenbank

Genome, welche im Rahmen von universitären Forschungsprojekten entschlüsselt wurden (für mehr Details siehe hier[3]), stehen meist in öffentlichen Datenbanken zur Verfügung. So ist gewährleistet, dass auch andere Wissenschaftler, insbesondere Bioinformatiker, uneingeschränkt mit diesen Daten arbeiten können sowie genomweite Studien *in silico* durchführen können.

Einer der wichtigsten Datenbank entschlüsselter Genome unterschiedlichster Organismen wird vom *National Center for Biotechnology Information* (NCBI[4]) bereitgestellt. Neben der Möglichkeit Daten über die Weboberfläche herunterzuladen, stellt das NCBI eine Programmierschnittstelle (API[5]) zur Verfügung, die es erlaubt Daten auch softwareseitig abzurufen.

Ziel dieses Programmteils soll es sein, mit Hilfe von Java und der öffentlichen Programmierschnittstelle vollständige Genome aus der NCBI herunter zu laden. Dazu werden vorimplementierte Methoden, d.h. Methoden welche bereits mit der NCBI API kommunizieren können, schrittweise so angepasst, dass nach Eingabe eines Organismus, z.B. das Bakterium *Escherichia coli*, die Ausgabe im FASTA-Format[6] erfolgt. Dieses Format zeichnet sich dadurch aus, dass die erste Zeile, beginnend mit ">", als Kommentarzeile dient und vor allem für Meta-Information verwendet wird. Ab der zweiten Zeile folgt die eigentliche Sequenz:

```
> die erste Zeile enthält Meta-Information
CATTGCTCAGGGATCTTCTGAACGCTCAAGGATCTTCTGAA
GATGCGACCACTGGCGTGCGCGTTACTCAGGATCTTCTGAA
GTGGCGTTGGCGGTGCGCTGCTGGAGCAAGGATCTTCTGAA
TATCGACTTACGTGTCTGCGGTGTTGCCAGGATCTTCTGAA
```

FASTA Dateien können beliebig viele dieser Sequenzblöcke enthalten, für diese Aufgaben beschränken wir uns auf nur eine.

2.1 Parsen der Genom ID (5 Punkte)

Die Klasse *GenomeDownloader* enthält die Methode *parseGenomeId(String... organism)* und nimmt den Namen eines Organismus entgegen und gibt den damit verknüpften Genomidentifizierer auf der Kommandozeile aus. Passen Sie die Methode so an, dass die ID als String zurück gegeben wird. Diese befindet sich innerhalb des ID-Tags:

```
<Id>167</Id>
```

```
// Beispiel:
String id = genomeDownloader
```

[3] https://en.wikipedia.org/wiki/Whole_genome_sequencing
[4] https://www.ncbi.nlm.nih.gov/
[5] Application Programming Interface
[6] https://de.wikipedia.org/wiki/FASTA-Format

```
    .parseGenomeId("escherichia", "coli");
id.equals("167");
```

Hinweis: Versuchen Sie ihre Lösung aus dem ersten Aufgabenteil wiederzuverwenden.

2.2 Parsen der Sequenz ID (5 Punkte)

Jeder Eintrag in der Genomdatenbank verlinkt zu dem entsprechenden Eintrag der Sequenz, d.h. Datensätze, die von z.b. verschiedenen Arbeitsgruppen eingereicht wurden. Die Methode *parseSequenceID(String id)* nimmt die ID aus Aufgabe 2.1 entgegen und druckt das Ergebnis wiederum auf die Standardausgabe. Passen Sie die Methode so an, dass alle IDs aus dem Ergebnis gefiltert, in ein String-Array gespeichert und als Ergebnis zurückgegeben werden.

```
// Beispiel:
List<String> seqIDs = genomeDownloader.getSequenceId(id);
assert seqIDs.length > 0;
seqIDs[0].equals("1434970144");
```

2.3 Download der Fasta-Datei (5 Punkte)

Nun folgt der wesentliche Schritt. Alle Sequenz IDs aus Aufgabe 2.2 verweisen auf die eigentlichen Sequenzeinträge. Die Methode *getSequence(String seqID)* nimmt eine ID aus Aufgabe 2.2 entgegen und gibt den FASTA-Eintrag aus. Passen Sie diese Methode so an, dass die Kommentarzeile verworfen wird und die eigentliche Sequenz in einem langen String gespeichert und als Ergebnis zurück gegeben wird. Java bietet hierfür bspw. den *StringBuilder* oder die Methode *String.join()*. Achten Sie darauf, dass auch eventuelle Zeilenumbrüche verworfen werden.

```
// Beispiel:
String finalSeq = genomeDownloader.getSequence(seqIDs[0]);
System.out.println("My huge genome: " + finalSeq);
// Ausgabe: "My huge genome: CATTGC...TCTTCTGAA"
```

2.4 Auffinden von codierenden Bereichen

In diesem Teil soll nun ein detaillierterer Einblick in die Daten gewonnen werden. Dazu sollen codierende Abschnitte (vgl. Aufgabe 1.2) in der Sequenz gefunden werden. Formal kann man hier die vier Basen als Alphabet $\Sigma = A, T, G, C$ betrachten.

2.4.1 Hash-Map (5 Punkte)

Erweitern Sie Ihre Funktion aus Aufgabe 2.3 so, dass anstelle der Sequenz eine Hash-Map zurück gegeben wird, die neben der Sequenz auch die Meta-Information zur Sequenz enthält.

```
// Beispiel:
Map<String, String> seqMap = genomeDownloader.getSequence(seqIDs[0]);
seqMap.get("info").equals("> die erste Zeile enthält...");
seqMap.get("seq").equals("CATTGC...TCTTCTGAA");
```

2.4.2 Die Code-Sonne als Hash-Map (10 Punkte)

In Abbildung 1 finden Sie eine sog. Code-Sonne. Diese beinhaltet alle möglichen Wörter die aus Σ gebildet werden können. Dazu folgt man einfach den entsprechenden Basen von innen nach außen und man erhält das entsprechende Triplett, bspw. "CCA". Diese Tripletts werden in der Biologie als Codon bezeichnet und *in vivo* in Aminosäuren übersetzt. Implementieren Sie eine Funktion *getAminoAcid(String triplet)* in der Klasse *GenomeParser* die für den übergebenen Parameter, ein Basen-Triplett, die entsprechende Aminosäure zurück gibt. Verwenden Sie hierfür eine Hash-Map.

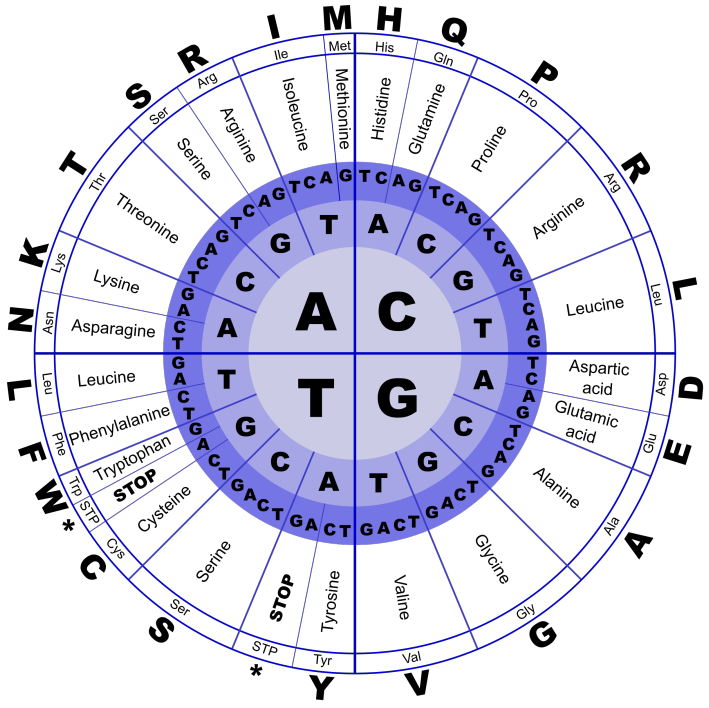

Figure 1: Mit Hilfe der Code-Sonne kann man die Basenfolge für Aminosäuren oder des Startcodons ablesen. (Quelle: `https://openclipart.org/image/2400px/svg_to_png/95191/genetic-code.png`)

```
// Beispiel:
List<String> triplets = genomeParser.getAminoAcid("AGC");
```

Weiterhin soll eine weitere Funktion implementiert werden, welche mit dem Namen einer Aminosäure aufgerufen werden kann und die entsprechende Sequenzen ausgibt.

```
// Beispiel:
genomeParser.getCodon("G").equals("GGC")
```

2.5 Objektorientierung (10 Punkte)

Implementieren Sie die Klasse *AminoAcid* und jeweils eine Klasse für jede Aminosäure, z.B. *GlutamicAcid*. Hierfür bietet sich ein kleines Python- oder Perlskript an, um sich ein wenig Arbeit zu sparen, Sie können es aber auch händisch erledigen. Lassen Sie die jeweiligen Klassen der Aminosäuren von *AminoAcid* erben und ändern Sie Ihre Implementationen aus Aufgabe 2.4.2 so ab, das diese nun mit den entsprechenden Objekten aufgerufen werden können.

```
GenomeParserOO genomeParserOO =
    new GenomeParserOO(seqMap.get("seq"));
```

```
Serine s = genomeParserOO.getAminoAcid("AGC")
String codon = genomeParserOO.getCodon(new Glycine())
```

Vergessen Sie nicht die entsprechenden Methoden (”Getter”) hinzuzufügen, die den Namen der jeweiligen Aminosäuren zurückgeben können.

```
// Beispiel:
s.getName().equals("Serine");
s.getOneLetterName().equals("S");
```

2.6 Auffinden der codierenden Bereiche (10 Punkte)

Für diese Aufgabe muss die Sequenz in Abschnitte der Länge 3 unterteilt werden, z.b. ”ATT”. Diese Tripletts werden im lebenden Organismus über komplizierte Prozesse in Aminosäuren übersetzt. Wichtig in diesem Zusammenhang sind nur die Start- und Stopcodons zwischen denen sich codierende Bereiche erstrecken. Implementieren Sie zunächst eine Funktion, die eine Sequenz entgegen nimmt und diese in Codons aufteilt. Verwenden Sie als Testdatensatz die Sequenz, die Ihre Funktion *getSequence* aus Aufgabe 2.3 mit der ID 38638184 wiedergibt, da diese nicht so groß ist.

```
// Beispiel:
List<String> codons =
    genomeParser.getCodons(seqMap.get("seq"));
```

Erweitern Sie nun Ihre Funktion so, dass anstelle eines String-Arrays ein Array aus Aminosäuren zurück gegeben wird.

```
// Beispiel:
List<AminoAcid> aminoAcids =
    genomeParserOO.getCodons(seqMap.get("seq"));
```

Suchen Sie innerhalb der Sequenz nun nach allen Start- (”ATG”, bzw. ”M”) und Stopcodons und geben Sie die gefundenen Bereiche auf der Kommandozeile aus.

```
// Beispiel:
List<String> codingRegions = genomeParserOO.getCodingRegions(aminoAcids);
codingRegions[0].toString().equals("MTGEDLX")
codingRegions[i].toString().equals("M...X")
```

Ihre gefunden Sequenzen können wichtige Bereiche des Genoms sein, welche schließlich als Bauanleitung dienen und mit Hilfe der Proteinbiosynthese[7] in Proteine übersetzt werden. Hier spielen aber viel mehr Faktoren eine Rolle als nur die Tatsache, dass die Sequenz mit einem Startcodon beginnt und mit einem Stopcoden aufhört. Mit diesen Fragen beschäftigt sich die Epigenetik[8].

2.7 Häufigkeitsverteilung der Sequenzlängen

Abschließend können Sie Ihre Daten visuell aufbereitet betrachten. Eine bereits implementierte Methode nimmt eine Liste aus codierenden Bereichen, die Anzahl der Intervalle (”bins”) sowie die Breite dieser (”width”) entgegen und plottet die Häufigkeitsverteilung der Sequenzlängen. Jeder Stern steht für ein Sequenz, deren Länge sich innerhalb des jeweiligen Intervals befindet:

[7] https://de.wikipedia.org/wiki/Proteinbiosynthese
[8] https://de.wikipedia.org/wiki/Epigenetik

```
String textualHistogram = genomeParserOO
    .getLengthDistribution(codingRegions, 15, 20);
System.out.println(textualHistogram);
[0 20) **************
[20 40) *******
[40 60) **
[60 80) *
[80 100)
[100 120) *
[120 140) *
[140 160)
[160 180) *
[180 200)
[200 220)
[220 240)
[240 260)
[260 280)
[280 300)
```

Ein Teil der Funktion die Sie im Rahmen dieser Aufgaben selbst erarbeitet haben, finden sich bereits implementiert im Paket *BioJava*[9]. Dieses Paket bietet zusätzlich viele andere Methoden, welche u.a. das Arbeiten mit DNA- und Proteinsequenzen in einem Javaprogramm um einiges erleichtern. Für *ad hoc* Analysen werden aber meist Skriptsprachen wie R (`https://www.r-project.org/`), Perl (`https://www.perl.org/`) oder Python (`https://www.python.org/`) verwendet. Insbesondere für erstere hat die Bioinformatikgemeinschaft im Laufe der Zeit unzählige Pakete bereitgestellt, mit denen bioinformatische sowie statistische Analysen durchgeführt werden können.

[9] `https://biojava.org/index.html`