Stefan Kurtz **Universität Hamburg**
Emanuel Ehmki **Zentrum für Bioinformatik**

Programmierung für Naturwissenschaften 1
Wintersemester 2019/2020
Übungen zur Vorlesung: Ausgabe am 11.12.2019

... und immer wieder am Mittwoch verfügbar:
`https://feedback.informatik.uni-hamburg.de/PfN1/wise2019-2020`

Aufgabe 8.1 (4 Punkte) In dieser Aufgabe geht es um die vollständige Zerlegung einer positiven ganzen Zahl in Summanden. Sei n eine positive ganze Zahl und S eine Menge positiver ganzer Zahlen. Eine additive Zerlegung von n bzgl. S ist eine Liste $[s_1, s_2, \ldots, s_k]$ von k Zahlen aus S, mit $k \geq 1$, so dass gilt:

- $s_i \leq s_{i+1}$ für alle i, $1 \leq i \leq k-1$.
- $n = \sum_{i=1}^{k} s_i$

D.h. alle Zahlen der Zerlegung stammen aus S und die Liste enthält mindestens ein Element. Außerdem ist die Liste der Zahlen der additiven Zerlegung aufsteigend sortiert und ihre Summe ist n.

Für eine Liste L von ganzen Zahlen definieren wir

$$\mathsf{quality}(L) = \sqrt{\sum_{s \in L} (s - \mathsf{mean}(L))^2}, \text{ wobei}$$

$$\mathsf{mean}(L) = \frac{1}{|L|} \left(\sum_{s \in L} s \right)$$

das arithmetische Mittel der Zahlen aus L ist. Dabei ist $|L|$ die Anzahl der Elemente in L.

In diesen beiden Definitionen steht $\sum_{s \in L} \ldots$ für die Summierung der Werte $(s - \mathsf{mean}(L))^2$ bzw. s über alle Listenelemente s aus L. Wenn ein Listenelement mehrfach in einer Liste vorkommt, wird es auch mehrfach gezählt.

Hier ist eine Tabelle mit additiven Zerlegungen L für gegebene n und S. Falls es keine additive Zerlegung gibt, ist das Ergebnis `None`

n	S	$\mathsf{quality}(L)$	L
11	$\{2, 3, 5\}$	0.87	$[2, 3, 3, 3]$
32	$\{7, 11, 13\}$	3.46	$[7, 7, 7, 11]$
38	$\{7, 11, 13\}$	5.20	$[7, 7, 11, 13]$
45	$\{8, 9\}$	0.00	$[9, 9, 9, 9, 9]$
47	$\{11, 12, 13, 14\}$	0.87	$[11, 12, 12, 12]$
47	$\{13, 14\}$		`None`

Die Aufgabe besteht nun darin, für gegebene Werte von n und S eine additive Zerlegung L von n bzgl. S mit minimalem quality(L)-Wert zu berechnen. Falls es mehr als eine Zerlegung mit dem minimalen quality-Wert gibt, dann soll die lexikographisch kleinste dieser Zerlegungen bestimmt werden. Eine Liste L von ganzen Zahlen ist lexikographisch kleiner als eine Liste M von ganzen Zahlen, wenn es ein $q \leq \min\{|L|, |M|\}$ gibt, so dass $L[i] = M[i]$ für alle $i, 1 \leq i < q$ und entweder $q = |L| < |M|$ oder $L[q] < M[q]$. Z.B. ist die Liste $[7, 7, 11]$ lexikographisch kleiner als die Liste $[7, 11, 7]$. Wenn Sie Ihre rekursive Funktion (s.u.) so implementieren, wie vorgeschlagen, dann werden die Zerlegungen in lexikographischer Reihenfolge aufgezählt, so dass Sie z.B. immer zuerst die Zerlegung $[7, 7, 11]$ vor der Zerlegung $[7, 11, 7]$ aufzählen.

Als ersten Teil Ihrer Lösung implementieren Sie in einer Datei `splitnumber.py` eine Funktion `quality_function`, die für eine Liste von ganzen Zahlen den quality-Wert berechnet und mit einer `return`-Anweisung zurückliefert.

Im zweiten Schritt schreiben Sie eine rekursive Python3-Funktion

```
split_number_rec(terms_of_sum, best_split, remain, terms_idx, l)
```

für die gilt:

- `terms_of_sum` ist die Liste der Zahlen aus S in aufsteigend sortierter Reihenfolge.
- `best_split` ist eine Liste mit genau zwei Werten: `best_split[0]` ist eine additive Zerlegung mit minimalem quality-Wert unter allen bisher berechneten additiven Zerlegungen von n bzgl. S. `best_split[1]` ist der quality-Wert der Liste in `best_split[0]`. Falls bisher noch keine additive Zerlegung berechnet wurde, sind beide Werte `None`.
- `remain` ist eine ganze Zahl $\leq n$, für die eine additive Zerlegung der Zahlen aus der Liste `terms_of_sum[terms_idx:]` berechnet werden muss. D.h. `terms_idx` gibt an, ab welchem Index in der Liste `terms_of_sum` die Elemente, bzgl. der `remain` zerlegt werden soll, stehen.
- `l` ist eine Liste, die die in den bisherigen Aufrufen aufgesammelten Summanden aus `terms_of_sum` enthält, d.h. die Summe von `remain` und der Werte aus `l` ist n. Beachten Sie, dass Sie für jeden rekursiven Aufruf zunächst durch `new_l = l.copy()` eine Kopie von `l` erzeugen müssen, an die Sie den nächsten Summanden anhängen.
- Die Funktion `split_number_rec` hat keinen Rückgabewert. Das Ergebnis wird im Parameter `best_split` gespeichert. Das ist möglich, weil `best_list` eine Liste ist, deren Werte in der Funktion verändert werden können.

Im dritten Schritt implementieren Sie eine Funktion `split_number(number,terms_of_sum)`. Dabei ist `number` die zu zerlegende Zahl und `terms_of_sum` die oben genannte Liste von Zahlen. Diese Funktion initialisiert die Liste `best_split`, ruft `split_number_rec` mit den passenden Argumenten auf und liefert dann `best_split` als Ergebnis zurück.

Beachten Sie, dass es nicht immer eine additive Zerlegung gibt. In diesem Fall werden die beiden initialen `None`-Werte in `best_split` nicht verändert.

Im Material zu dieser Übung finden Sie eine Python3-Datei `splitnumber_mn.py`, die Ihr Modul `splitnumber.py` importiert und die genannte Funktion `split_number` aufruft. Sie müssen sich also in dieser Aufgabe nicht um die Behandlung von Benutzerfehlern kümmern. Ebenso finden Sie im Material ein bash-Skript mit Testaufrufen sowie ein Makefile. Durch `make test` verifizieren Sie, dass Ihr Programm die richtigen Ergebnisse für die Testfälle berechnet.

Aufgabe 8.2 (2 Punkte) In der Vorlesung wurde im Abschnitt *Reading and representing data matrices* gezeigt, wie man eine Matrix in ein Dictionary von Dictionaries konvertiert und dieses wieder ausgibt. Den entsprechenden Programmcode finden Sie in den beiden Dateien `data_matrix.py` und `data_matrix_main.py` (siehe Material zu dieser Aufgabe). In diesen Dateien finden Sie einige Zeilen der Form `#lst...` sowie `#lstend#`, die Sie ignorieren können.

Implementieren Sie nun in einer Datei `data_matrix_class.py` eine Klasse `DataMatrix`, die die gleiche Funktionalität bietet, wie die drei Funktionen aus `data_matrix.py`. Im Einzelnen müssen Sie die genannte Klasse mit zwei Instanz-Variablen `_matrix` und `_attribute_liste` und den folgenden Methoden implementieren:

- `__init__(self,lines,key_col=1,sep='\t')` entspricht der Methode `data_matrix_new` aus `data_matrix.py`. `__init__` liefert natürlich keine Werte über eine `return`-Anweisung, sondern speichert die Matrix und die Attributliste in den genannten Instanz-Variablen.
- Die Methode `show(self,sep,attributes,keys)` soll das Gleiche leisten wie die Funktion `data_matrix_show`.
- Die Methode `show_orig(self,sep,attributes,keys)` soll das Gleiche leisten wie die Funktion `data_matrix_show_orig`.
- Die Methode `keys(self)` liefert `self._matrix.keys()` über eine `return`-Anweisung.
- Die Methode `attribute_list(self)` liefert `self._attribute_list` über eine `return` -Anweisung.

Die beiden Instanz-Variablen sind privat und dürfen nicht außerhalb der Klassendefinition verwendet werden.

Nach der Implementierung der Klasse kopieren Sie den Programmcode aus `data_matrix_main.py` in die Datei `data_matrix_class.py` und modifizieren ihn so, dass die Funktionalität bzgl. der Datenmatrizen durch die Methoden der Klasse `DataMatrix` realisiert wird. Das Hauptprogramm soll ausgeführt werden, wenn `__name__ == "__main__"` gilt. Dadurch kann die Klasse in Zukunft weiterverwendet werden.

In den Materialien finden Sie zwei Testdateien `atom-data-mini.tsv` und `atom-data.tsv` und ein Makefile. Durch `make test` verifizieren Sie, dass Ihr Programm funktioniert.

Aufgabe 8.3 (4 Punkte) Sie haben eine neue Messmethode entwickelt, die für ein chemisches, physikalisches oder biologisches System Koordinaten im zweidimensionalen Raum liefert. Diese Messmethode kann z.B. durch die Wahl verschiedener Einstellungen variiert werden. Sie haben Ihre Methode mit verschiedenen Einstellungen ausprobiert und entsprechende Koordinaten ermittelt. Diese bilden damit Vorhersagen der Realität. Sie sollen nun die Qualität der einzelnen Messungen jeweils durch einen Vergleich Ihrer vorhergesagten Werte mit einem Goldstandard ermitteln, der durch eine sehr aufwändige aber anerkannte Messmethode ermittelt wurde.

Die Qualität Ihrer Methode soll durch die Bestimmung der Sensitivität und der Spezifität relativ zum Goldstandard bestimmt werden. Die Sensitivität macht eine Aussage über die Fähigkeit der Methode, Koordinaten entsprechend des Goldstandards richtig vorherzusagen. Die Spezifität macht eine Aussage über die Fähigkeit der Methode, keine bzgl. des Goldstandards falschen Werte vorherzusagen. Die formale Definition dieser Begriffe basiert auf drei Mengen, P, G und TP. Dabei ist P die Menge der Messwerte Ihrer Messmethode, G die Menge der Werte des Goldstandards und $TP = P \cap G$, also die Schnittmenge von P und G. TP heißt auch die Menge der *True Positives*, also der korrekten Vorhersagen. Die Sensitivität $se(P, G)$ und die Spezifität $sp(P, G)$ sind dann

definiert durch

$$se(P,G) = 100 \cdot \frac{|TP|}{|G|} \qquad\qquad sp(P,G) = 100 \cdot \frac{|TP|}{|P|}$$

Dabei bezeichnet $|S|$ die Größe einer Menge S. Da $TP \subseteq G$ und $TP \subseteq P$, liegen beide Werte zwischen 0 und 100. Eine ideale Methode erreicht jeweils Werte von 100%, d.h. sie liefert die gleichen Messwerte wie der Goldstandard. In vielen Anwendungen erreicht man optimale Sensitivität nur auf Kosten geringer Spezifität und umgekehrt. Daher kombiniert man beide Werte, indem man den harmonischen Durchschnitt berechnet. Für $0 \leq a, b \leq 100$ ist $\frac{2}{\frac{1}{a}+\frac{1}{b}}$ der harmonische Durchschnitt.

Implementieren Sie ein Programm `predictionqual.py`, das die Qualität der vorhergesagten Messdaten berechnet und formatiert ausgibt. Das Programm soll eine Option `-g/--gold_standard` mit genau einem String-Argument haben. Dieses String-Argument ist der Name der Datei mit dem Goldstandard. (z.B. `goldstandard.tsv` im Material). Alle weiteren Argumente sind die Namen der Dateien mit den Koordinaten der zwanzig Messungen (`prediction*.tsv` im Material).

Alle genannten Dateien enthalten jeweils ein Paar von x,y-Koordinaten pro Zeile, separiert durch das Zeichen \t. Zur Vereinfachung sind die Koordinaten durch ganze Zahlen repräsentiert. Ein Koordinatenpaar (a, b) ist identisch mit Koordinatenpaar (a', b'), wenn $a = a'$ und $b = b'$ ist. True Positives sind die identischen Koordinatenpaare. Die Ausgabe soll zeilenweise für jede Datei die Qualitätswerte der Messungen aufsteigend sortiert nach dem harmonischen Durchschnitt enthalten, und zwar durch das Zeichen \t separiert und in folgendem Format:

- Spalte 1: Name der Datei `(filename)`
- Spalte 2: Anzahl der True Positives `(tp)`
- Spalte 3: Sensitivität `(sens)`
- Spalte 4: Spezifität `(spec)`
- Spalte 5: Harmonischer Durchschnitt `(hmean)`

Für die Sortierung verwenden Sie die Methode `sorted`, siehe Vorlesungsfolien Seite 325ff. Die numerischen Werte sollen rechtsbündig in einem Block der Breite 6 ausgegeben werden. Die Fließkommawerte sollen mit zwei Nachkommastellen ausgeben werden. In der Datei `quality-out.tsv` finden Sie das erwartete Ergebnis.

Hinweise:

- Die Dateien enthalten jedes Koordinatenpaar jeweils genau einmal.
- Verwenden Sie die Klasse `set` zur Speicherung der Koordinaten. Diese Klasse bietet u.a. die folgenden Operationen.
  - Eine leere Menge `s` wird durch `s = set()` erzeugt.
  - Durch `s.add(x)` fügen Sie ein neues Element `x` zur Menge `s` hinzu.
  - Für Mengen `s` und `t` liefert der Ausdruck `s & t` den Durchschnitt `s∩t` von `s` und `t`.
  - Für eine Menge `s` liefert `len(s)` die Größe dieser Menge.

In den Materialien finden Sie neben den erwähnten Dateien mit dem Goldstandard und den Messungen ein Makefile. Durch `make test` verifizieren Sie, dass Ihr Programm korrekt funktioniert.

Bitte die Lösungen zu diesen Aufgaben bis zum 16.12.2019 um 18:00 Uhr an pfn1@zbh.uni-hamburg.de schicken. Die Besprechung der Lösungen erfolgt am 18.12.2019.

Effizienz der Methoden zur Translation von Codons

In den Folien zum Thema Codontranslation wird gesagt, dass die Methode, die ein Dictionary verwendet, die effizienteste ist. Es wurde gefragt, warum das so ist und was genau Effizienz in diesem Zusammenhang bedeutet.

Ich will hier kurze Antworten geben. Vielleicht mache ich es noch zu einem Thema einer Peer-Teaching Aufgabe. Das Thema Effizienz von Algorithmen wird im Modul Algorithmen und Datenstrukturen genau betrachtet.

Wenn man bei Algorithmen von Effizienz spricht, meint man meist die Laufzeit oder den Speicherbedarf. Beim Speicherbedarf zählt immer die maximale Größe, die der Algorithmus während der Laufzeit benötigt. Laufzeit und Speicher werden durch Funktionen, die abhängig von der Eingabegröße sind, ausgedrückt.

Wenn sich z.B. die Anzahl der Schritte eines Algorithmus für eine Eingabe der Größe n durch die Funktion $h(n) = 500$ ausdrücken lässt, dann spricht man von konstanter Laufzeit. Wenn sich z.B. die Anzahl der Schritte eines Algorithmus für eine Eingabe der Größe n durch die Funktion $f(n) = 20 + 5n$ berechnen läßt, sagt man: Die Laufzeit ist linear. Wenn sich die Anzahl der Schritte eines Algorithmus durch die Funktion $g(n) = 1000 + 100n + 2n^2$ berechnen lässt, dann spricht man von quadratischer Laufzeit. Es zählt also immer nur der dominierende Term. Damit werden Faktoren und Konstanten ignoriert.

Im Fall der Codon Translation ergibt sich für alle drei Varianten aus den Folien eine konstante Laufzeit pro Codon und damit eine lineare Laufzeit für die gesamte Sequenz. Der Zugriff auf ein Dictionary für einen gegebenen Schlüssel benötigt nur wenige Rechenschritte, da nur der Schlüssel (also ein Codon) in einem numerischen Wert konvertiert werden muss, aus dem man mit einer Rechenoperation eine Adresse im Speicher berechnen kann. Für beide Schritte benötigt man auf einer modernen CPU nur wenige Rechenschritte.

Bei der ersten Variante muss man im Schnitt $\frac{64}{2} = 32$ Vergleiche durchführen, um festzustellen, welcher der 64 Fälle zutrifft. Dafür benötigt man sehr viel mehr Rechenschritte als bei der dritten Variante. Bei der Variante mit den regulären Ausdrücken wird dieser in einen sogenannten endlichen Automaten transformiert, mit dem die Sequenz durchsucht wird. Das benötigt weniger Rechenschritte als die erste Variante, aber sicher mehr als die dritte Variante. Die genaue Anzahl der Schritte für die einzelnen Methoden lässt sich nur sehr schwer bestimmen und sie hängt von der Programmiersprache ab, von der Art und Weise, wie in der Programmiersprache die genannten Datenstrukturen (Dictionaries, endliche Automaten) implementiert sind, vom Betriebssystem und dem Rechner. Daher müßte man konkrete Messungen durchführen, um den Unterschied in der Praxis zu messen.