Stefan Kurtz **Universität Hamburg**
Abteilung für Genominformatik **Zentrum für Bioinformatik**

Grundlagen der Sequenzanalyse
Wintersemester 2022/2023
Übungen zur Vorlesung: Ausgabe am 06.12.2022

In den nun immer anspruchsvolleren Programmieraufgaben kommt es nach unserer Erfahrung häufiger vor, dass Tests nicht bestanden werden. In einem solchen Fall dokumentieren Sie unbedingt Ihren Programmcode, damit die Korrekturtutorin Ihre Lösung nachvollziehen und entsprechende Punkte vergeben kann oder sogar in der Lage versetzt wird, den Programmcode zu korrigieren.

Aufgabe 7.1 (4 Punkte) In der Vorlesung wurde gezeigt, dass man für die Berechnung der Editdistanz von zwei Sequenzen u und v der Länge m bzw. n nur $O(m)$ Speicher benötigt. Diese Eigenschaft folgt daraus, dass man zur Berechnung der Werte in der j-ten Spalte der Matrix E_δ für $j > 0$ nur die Werte der Spalte $(j-1)$-ten Spalte benötigt. D.h. man benötigt zu jedem Zeitpunkt des Algorithmus nur Speicher für jeweils 2 Spalten. Man kann das Verfahren noch weiter optimieren, indem die Berechnung nur mit einer Spalte und zwei skalaren Werten (also integer oder float-Werte, je nach Kostenfunktion), die Kostenwerte zwischenspeichern, durchgeführt wird. Dieses optimierte Verfahren soll in Form von zwei Python-Funktionen in der Datei `lsp_dist_only.py` implementiert werden. Dabei soll die Einheitskostenfunktion verwendet werden.

1. Die Funktion `nextEDtabcolumn_inplace_unitcost` hat keinen **`return`**-Wert und die folgenden drei Parameter: die vorherige Spalte `col` der Matrix E_δ als Liste von $m+1$ Werten, wobei m die Länge von u ist; die Sequenz `u`; ein Zeichen `b`. Die Funktion soll in der gegebenen Liste `col` die Werte der neuen Spalte berechnen, und zwar bzgl. des Zeichens `b` und der Sequenz `u`. Wenn `b` das j-te Zeichen der Sequenz `v` ist (bei Indizierung ab 1), dann gilt:
   - `col` enthält vor dem Aufruf der Funktion die Werte der $(j-1)$-ten Spalte.
   - `col` enthält nach dem Aufruf der Funktion die Werte der j-ten Spalte.

   In der Funktion darf keine neue Liste erzeugt werden, und es dürfen zu jedem Zeitpunkt nur maximal 2 Werte der vorherigen Spalte in lokalen Variablen zwischengespeichert werden.
2. Die Funktion `linearspace_distance_only_unitcost` hat zwei Parameter, nämlich die beiden Sequenzen `u` und `v`. In einer Liste der Länge $m+1$ wird zunächst die erste Spalte der Matrix E_δ berechnet. Dann wird nacheinander die erste Funktion für alle Zeichen `b` aus `v` aufgerufen und dabei jeweils die genannte Liste übergeben. Nachdem die letzte Spalte berechnet wurde, wird der letzte Wert der Spalte mit einer **`return`**-Anweisung zurückgeliefert.

In den Materialien finden Sie eine Datei `lsp_dist_only_template.py`. Diese enthält einen Optionsparser sowie ein Hauptprogramm. Benennen Sie diese Datei in `lsp_dist_only.py` um und implementieren Sie darin die genannten Funktionen. Durch `make test` verifizieren Sie die Korrektheit Ihres Programms für gegebene Testdaten.

Zur Unterstützung des Testens finden Sie in der Datei `matrix_major_row.py` eine Funktion zum Transponieren und eine Funktion zum Ausgeben einer Matrix. Um diese Funktion zu nutzen, müssen Sie in `linearspace_distance_only_unitcost` zunächst eine leere Liste `columns_list` erzeugen und dann jeweils die berechnete Liste `col` durch `columns_list.append(col.copy())`

kopieren und an diese Liste anhängen. Am Ende erzeugen Sie aus der Liste `columns_list` mit `matrix_transpose` eine Liste von Zeilen. Diese können Sie ausgeben. Die Ausgabe muss identisch sein mit der Matrix, die das Programm `editdistance.py` ausgibt, wenn man die Option `-s` verwendet. Hier ist ein Beispiel zur Verwendung:

```
$ ./editdistance.py -s agcgatac acgcatag > correct.txt
$ ./lsp_dist_only.py   agcgatac acgcatag > my_matrix.txt
$ diff correct.txt my_matrix.txt
```

Nachdem Sie alle Fehler korrigiert haben und die Tests erfolgreich waren, kommentieren Sie bitte die Aufrufe von `matrix_transpose` und `matrix_show` in Ihrem Programm aus.

Punkteverteilung:

- 2 Punkte für eine Implementierung der Funktion `nextEDtabcolumn_inplace_unitcost` entsprechend der obigen Spezifikation
- 1 Punkt für die korrekte Implementierung der zweiten Funktion.
- 1 Punkt für bestandene Tests.

Aufgabe 7.2 (5 Punkte) Gegeben seien zwei Sequenzen u und v der Länge m bzw. n und ein zusätzlicher Parameter $q \leq \min\{m, n\}$. In u gibt es $m - q + 1$ Substrings der Länge q und in v gibt es $n - q + 1$ Substrings der Länge q (im Folgenden q-Worte in u bzw. v genannt).

Folglich gibt es $O(m \cdot n)$ Paare von q-Worten von u und v. Für jedes Paar von q-Worten definieren wir den *match-count* (Abgekürzt *mc*-Wert) als die Anzahl von übereinstimmenden Zeichen an gleichen Positionen in den q-Worten. So ist zum Beispiel für $q = 5$ und die beiden q-Worte *acata* und *agaaa* der mc-Wert gleich 3.

Wir betrachten nun das Problem der Berechnung der mc-Werte für alle Paare von q-Worten in u und v in Form einer $(m - q + 1) \times (n - q + 1)$-Matrix MC mit

$$\mathsf{MC}(i, j) = mc(u[i + 1 \ldots i + q], v[j + 1 \ldots j + q]) \text{ für } 0 \leq i \leq m - 1, 0 \leq j \leq n - 1$$

Beispiel: Gegeben seien die Sequenzen $u = aaaba$ und $v = baaba$ sowie $q = 3$. Dann gibt es in u die drei q-Worte *aaa*, *aab* und *aba*. In v gibt es die q-Worte *baa*, *aab* und *aba*. Damit ergeben sich die folgenden Substring-Paare mit den angegebenen mc-Werten nach den beiden Substrings. Diese Information kann kompakt in der MC-Matrix dargestellt werden.

		mc()
$u[1 \ldots 3] = aaa$	$v[1 \ldots 3] = baa$	2
$u[2 \ldots 4] = aab$	$v[1 \ldots 3] = baa$	1
$u[3 \ldots 5] = aba$	$v[1 \ldots 3] = baa$	1

		mc()
$u[1 \ldots 3] = aaa$	$v[2 \ldots 4] = aab$	2
$u[2 \ldots 4] = aab$	$v[2 \ldots 4] = aab$	3
$u[3 \ldots 5] = aba$	$v[2 \ldots 4] = aab$	1

		mc()
$u[1 \ldots 3] = aaa$	$v[3 \ldots 5] = aba$	2
$u[2 \ldots 4] = aab$	$v[3 \ldots 5] = aba$	1
$u[3 \ldots 5] = aba$	$v[3 \ldots 5] = aba$	3

MC

2	2	2
1	3	1
1	1	3

Beachten Sie, dass die Strings ab 1 und die Zeilen und Spalten der Matrix ab 0 indiziert werden

Das Problem der Berechnung der Matrix MC kann leicht in $O(m \cdot n \cdot q)$ Zeit gelöst werden. Entwickeln Sie einen effizienten Algorithmus, der das Problem in $O(m \cdot n)$ Zeit löst.

Gehen Sie bei der Entwicklung Ihrer Lösung wie folgt vor:

1. Geben Sie eine Gleichung an, mit der man für $i > 0$ und $j > 0$, den Wert $\mathsf{MC}(i, j)$ aus $\mathsf{MC}(i-1, j-1)$ in konstanter Zeit berechnen kann. Dazu müssen Sie sich überlegen, welchen Zusammenhang es zwischen zwei Paaren

$$u[i+1 \ldots i+q], v[j+1 \ldots j+q]$$
$$u[i \ldots i+q-1], v[j \ldots j+q-1]$$

   von q-Worten und ihren mc-Werten gibt.

2. Spezifizieren Sie eine mögliche Reihenfolge, in der die Matrix MC ausgewertet werden kann.

3. Begründen Sie, warum Ihr Algorithmus eine Laufzeit von $O(m \cdot n)$ erreicht. Beachten Sie dabei auch den Fall, dass $i = 0$ oder $j = 0$ ist.

4. Implementieren Sie den effizienten Algorithmus als Python-Funktion `matchcount_matrix`. Diese Funktion hat drei Parameter, nämlich die beiden Sequenzen u und v sowie den Wert q. Die Funktion soll mit einer **`return`**-Anweisung ein Paar `total_cmp, mat` zurückliefern. Dabei ist `mat` die berechnete Matrix MC und `total_cmp` die Gesamtanzahl aller Zeichenvergleiche zur Berechnung der Matrix.

Ihre Funktion soll in einer Datei `matchcount.py` implementiert werden, die Sie durch Umbenennung der Datei `matchcount_template.py` aus den Materialien erhalten. Das vorhandene Hauptprogramm kann zum Testen verwendet werden (Option `--check`) und zur Ausgabe der Matrix (Option `--matrix`). Die Materialien enthalten ein Makefile mit Tests sowie die erwarteten Ergebnisse für einige Paare von Strings. Durch den Aufruf von `make test` verifizieren Sie die Korrektheit Ihres Algorithmus und seiner Implementierung für die gegebenen Testdaten.

Punkteverteilung:

- Rekurrenz: 1 Punkt
- Reihenfolge der Auswertung: 1 Punkt
- Begründung der Effizienz (inklusive Randfall): 1 Punkt
- Implementierung und Tests: 2 Punkte

Aufgabe 7.3 (4 Punkte) Ein wichtiges Problem in der Massenspektrometrie besteht darin, alle Proteinsequenzen in einer Datenbank zu finden, die ein bestimmtes Molekulargewicht haben. Formal lässt sich dieses Problem wie folgt formulieren:

Sei $\mathcal{A}$ ein Alphabet und $\sigma : \mathcal{A} \to \mathbb{N} \setminus \{0\}$ eine Gewichtsfunktion, die jedem Symbol aus $\mathcal{A}$ eine positive natürliche Zahl zuordnet. Für alle Sequenzen $u \in \mathcal{A}^*$ definieren wir das *Gewicht* $\sigma(u) = \sum_{i=1}^{|u|} \sigma(u[i])$.

Sei $S \in \mathcal{A}^n$ eine Sequenz der Länge n und $w > 0$ ein vorgegebenes Gewicht. Das *gewichtete Substring-Matching Problem* besteht darin, die folgende Menge zu bestimmen:

$$Sol(\sigma, S, w) = \{(j, \ell) \mid 1 \leq j \leq n, 1 \leq \ell \leq n+1-j, \sigma(S[j \ldots j+\ell-1]) = w\}$$

d.h. die Menge aller Substrings in S (repräsentiert durch eine Startposition j und die Länge ℓ), deren Gewicht genau w ist.

Beispiel: Sei $\mathcal{A} = \{a, c, g, t\}$ das DNA-Alphabet, sei $\sigma(a) = 1$, $\sigma(c) = \sigma(t) = 2$ sowie $\sigma(g) = 3$. Für $S = acgtagctc$ ist dann

$$Sol(\sigma, S, 9) = \{(1, 5), (3, 4), (6, 4)\}$$

Diese Menge repräsentiert die Substrings $acgta$, $gtag$ und $gctc$ und es gilt $\sigma(acgta) = \sigma(gtag) = \sigma(gctc) = 9$.

Implementieren Sie einen Algorithmus, der das gewichtete Substring-Matching Problem in $O(|S|)$ Zeit löst. Dabei sollen Sie ausnutzen, dass die Gewichte immer positive ganze Zahlen sind.

Zur Implementierung benennen Sie die Datei `massmatching_template.py` in `massmatching.py` um. Ihr Algorithmus soll durch die Funktion

```
massmatches_enum(weight_dict,sequence,weight)
```

implementiert werden. Dabei ist `weight_dict` ein Dictionary, das für die Zeichen des DNA-Alphabets die oben angegeben Gewichte enthält. `sequence` ist ein String mit Zeichen aus dem DNA-Alphabet und `weight` das vorgegebene Gewicht. Die Funktion soll mit `yield` die Paare von Startposition und Länge eines Treffers, aufsteigend sortiert nach den Startpositionen aufzählen.

Diese Funktion wird im bereits vorhandenen Programmteil aufgerufen und die Ergebnisse werden ausgegeben.

Begründen Sie, dass die Laufzeit Ihres Algorithmus in $O(|S|)$ ist.

Im Material zu dieser Übung finden Sie Testdaten und ein Makefile. Durch `make test` verifizieren Sie, dass Ihr Programm für die Testfälle die richtigen Ergebnisse liefert.

Punkteverteilung

- 2 Punkte für die Implementierung der genannten Funktion.
- 1 Punkt für eine korrekte und nachvollziehbare Begründung der linearen Laufzeit.
- 1 Punkt für bestandene Tests.

Bitte die Lösungen zu diesen Aufgaben bis zum 11.12.2022 um 22:00 Uhr an gsa@zbh.uni-hamburg.de schicken.