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

Grundlagen der Sequenzanalyse
Wintersemester 2022/2023
Übungen zur Vorlesung: Ausgabe am 22.11.2022

Bitte beachten Sie die Regeln zur Formatierung Ihrer Lösungen, inbesondere die Begrenzung der Zeilenbreite für Programmcode und Textdateien auf 80 Zeichen. Formatierungsfehler findet man leicht durch das Programm `bin/code_check.py`.

Aufgabe 5.1 (3 Punkte) Implementieren Sie in Python3 in einer Datei `editdistance.py` den in der Vorlesung vorgestellten Algorithmus zur Berechnung der Edit-Distanz von zwei Sequenzen.

Seien m und n die Längen der Sequenzen, für die die Edit-Distanz bestimmt werden soll. Die DP-Matrix `edelta` (Notation im Skript: E_δ) hat $m+1$ Zeilen und $n+1$ Spalten und kann durch die folgende Anweisung erzeugt werden:

```
edelta = [[None] * (n+1) for i in range(m+1)]
```

`edelta[i][j]` ist dann der Wert in der i-ten Zeile und j-ten Spalte von `edelta`. Man kann selbstverständlich auch `edelta[i][j]` einen Wert zuweisen.

Implementieren Sie den genannten Algorithmus als Funktion `fillDPtable`, die die DP-Matrix `edelta` für zwei Sequenzen u und v berechnet und über eine **`return`**-Anweisung zurückliefert. Als Parameter erhält `fillDPtable` die Sequenzen u und v. Als Kostenfunktionen verwenden Sie die Einheitskostenfunktion. Sie brauchen daher keinen weiteren Parameter zur Übergabe der Kostenfunktion. Beachten Sie, dass in der E_δ-Rekurrenz aus der Vorlesung auf die Strings 1-basiert zugegriffen wird, während Python-Strings 0-basiert indiziert werden.

Das Programm `editdistance.py`, in dem die Funktion `fillDPtable` aufgerufen wird, enthält genau zwei Sequenzen. Auf diese wird mit Hilfe der Liste `sys.argv` zugegriffen. Die Ausgabe erfolgt im Format

```
seq1 seq2 edist
```

Dabei sind `seq1` und `seq2` die beiden Sequenzen und `edist` die Editdistanz bzgl. der Einheitskostenfunktion. Diese drei Werte sind jeweils durch ein Tabulator-Zeichen getrennt. Durch `make test` verifizieren Sie, dass Ihr Programm für die Testfälle in der Datei `edist-testcases.tsv` die richtigen Ergebnisse liefert.

Punkteverteilung:

- 2 Punkte für eine nachvollziehbare Implementierung der Funktion `fillDPtable` (auch wenn Sie möglicherweise nicht immer die korrekten Ergebnisse liefert)
- 1 Punkte für ein Programm, dass die Tests besteht.

Aufgabe 5.2 (6 Punkte) Das exakte String Matching Problem ist ein klassisches Problem der Informatik. Es besteht darin, alle Vorkommen eines Musters p der Länge m in einem Text t der Länge n zu finden, d.h. alle Positionen j, $1 \leq j \leq n-m+1$, so dass $t[j \ldots j+m-1] = p$ ist.

Dieses Problem kann man z.B. dadurch lösen, dass man an allen möglichen Positionen in t jeweils testet, ob p zum entsprechenden Substring in t passt, siehe Alg. 1. Geben Sie ein Beispiel für p und t, so dass der naive Algorithmus $O(mn)$ Zeit benötigt. Begründen Sie Ihre Antwort.

Algorithm 1 Ein naiver Algorithmus für das exakte String Matching Problem.

Require: Text t der Länge n, Pattern p der Länge m

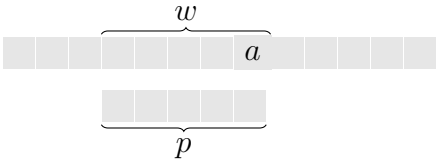
```
1:  if m ≤ n and m > 0 and n > 0 then
2:      for j ← 1 to n − m + 1 do
3:          i ← 1
4:          while i ≤ m and t[j + i − 1] = p[i] do
5:              i ← i + 1
6:          end while
7:          if i = m + 1 then
8:              print "match at position " j
9:          end if
10:     end for
11: end if
```

Entwickeln Sie einen Algorithmus, der das exakte String Matching Problem effizienter löst. Das Ziel ist, die Anzahl f der Positionen j, für die der Vergleich von $w = t[j \dots j + m - 1]$ und p durchgeführt werden muss, zu reduzieren. Es ist nicht notwendig, dass der Algorithmus auch im schlimmsten Falle eine geringere Laufzeit hat als der naive Algorithmus. (Def. f)

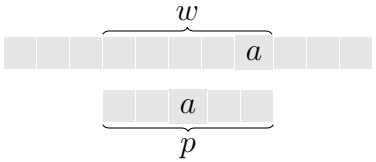
Hinweis: Ihr Algorithmus könnte (wie der naive Algorithmus) ein „Fenster“ der Länge m von links nach rechts über den Text schieben und das Muster p mit dem aktuellen Fensterinhalt $w = t[j \dots j + m - 1]$ vergleichen.

Bestimmte Substrings aus t der Länge m, bei denen klar ist, dass sie nicht zu Treffern führen, könnten übersprungen werden, und zwar dadurch, dass man das „Fenster“ mehr als eine Position nach rechts verschiebt. Die Verschiebespanne könnte man z.B. auf der Basis des letzten Buchstabens a im aktuellen Substring w von t bestimmen. Dabei muss man zwei Fälle betrachten, wie in der folgenden Abbildung dargestellt:

Fall 1: a kommt nicht in $p[1 \dots m-1]$ vor



Fall 2: a kommt in $p[1 \dots m-1]$ vor



Überlegen Sie sich nun, welche möglichst große und immer positive Verschiebespanne des Musters sinnvoll ist. Dabei sollen natürlich keine Treffer übersehen werden. Sie werden sehen, dass für die Bestimmung der Verschiebespanne nur das Muster p und das Alphabet des Textes bekannt sein muss, so dass eine Vorverarbeitung von p unabhängig vom Text möglich ist. Die Vorverarbeitung liefert eine Tabelle, in der für jedes Zeichen des Alphabets die Verschiebespanne steht. Hier ist ein konkretes Beispiel:

Beispiel: Sei $p =$ `acgat` und $t =$ `aatgacgacatacgtta`. Dann ist $m = 5$ und $n = 17$. Wir geben für die folgenden Fälle an, um wieviele Positionen man das Fenster nach rechts verschieben kann, ohne einen Treffer zu verlieren. Dabei ist das aktuelle Fenster in t jeweils unterstrichen.

aktuelles Fenster	Verschiebespanne
aatgacgacatacgtta	1
aatgacgacatacgtta	3
aatgacgacatacgtta	3
aatgacgacatacgtta	1
aatgacgacatacgtta	3
aatgacgacatacgtta	5

An den hier dargestellten 6 Positionen des Fensters muß also das Muster mit dem entsprechenden Substring w des Textes verglichen werden. Der naive Algorithmus muss jedoch an $n - m + 1 = 17 - 5 + 1 = 13$ Positionen des Fensters $O(m)$ Vergleiche von w mit dem entsprechenden Substring von t durchführen. Also muss im verbesserten Algorithmus nur an $100 \cdot \frac{6}{13} \approx 46\%$ der Fensterpositionen ein Vergleich in $O(m)$ Zeit durchgeführt werden.

Versuchen Sie nun aus diesem Beispiel und entsprechend der obigen Fallunterscheidung eine Regel abzuleiten, aus der sich dann ein Algorithmus ergibt. Dokumentieren Sie Ihre Lösung in einer Datei mit dem Namen `verschiebe_spanne.tex` oder oder `verschiebe_spanne.txt` abhängig vom verwendeten Format.

Implementieren Sie diesen Algorithmus in Python als eine Funktion `string_matching_fast` in einer Datei `string_matching.py`. Die Funktion erhält als Parameter die beiden Strings `p` und `t`. Ihre Funktion muss nicht die Positionen der Vorkommen von p und t bestimmen, sondern lediglich den Wert von f (Definition, siehe oben) berechnen. Im obigen Fall ist das 6. Der Rückgabewert der Funktion soll der Prozentwert $100 \cdot \frac{f}{n-m+1}$ sein. Im Material zu dieser Übung finden Sie ein Hauptprogramm `string_matching_run.py`, das zwei Dateien als Argumente erhält und daraus den Text und die Muster extrahiert. Schließlich wird die Funktion `string_matching_fast` aufgerufen und zu jedem Muster der genannte Prozentwert ausgegeben. Die Materialien enthalten zwei Paare von Dateien mit Mustern und Text. Durch den Aufruf von `make test` können Sie die Korrektheit Ihrer Implementierung testen.

Punkteverteilung:

- 1 Punkt: Beispiel für die $O(mn)$ Laufzeit für den naiven Algorithmus zu Stringmatching und Begründung.
- 2 Punkte: Entwicklung der algorithmischen Idee.
- 3 Punkte: Implementierung und Bestehen der Tests.

Bitte die Lösungen zu diesen Aufgaben bis zum 27.11.2022 um 22:00 Uhr an gsa@zbh.uni-hamburg.de schicken.