

Aufgabenblatt 5

Dynamische Programmierung

Zur Erinnerung

- Registrieren Sie sich unter <http://hyades.robotics.tu-berlin.de/index.php>, wenn Sie das noch nicht getan haben.
- Der Git-Dienst ist wiederhergestellt, und auch die automatisierten Tests werden jetzt wieder aufgenommen. Ab Blatt05 werden die Einreichungen mit git erfolgen.
- Sie finden Blatt05 unter <https://git.tu-berlin.de/algodat-bose21/Material>. Wenn Sie Blatt05 bereits auf ISIS eingereicht haben (mit einer Zip-Datei), machen Sie sich keine Sorgen! Wir werden das übernehmen. Wenn Sie Ihre Lösung auch auf git gepusht haben, werden wir Ihre letzte Einreichung berücksichtigen.
- Alle Übungen sind in Einzelarbeit zu erledigen. Kopieren Sie niemals Code und geben Sie Code in keiner Form weiter. Die Hausaufgaben sind Teil Ihrer Prüfungsleistung. Finden wir ein Plagiat (wir verwenden Plagiatserkennungssoftware), führt das zum Nichtbestehen des Kurses für alle Beteiligten.
- Wenn Ihre Abgabe nicht im richtigen Ordner liegt, nicht kompiliert, unerlaubte packages oder imports enthält oder zu spät abgegeben wird, gibt es **0 Punkte** auf diese Abgabe.

Abgabe (bis 06.06.2021 23:59 Uhr)

Die folgenden Dateien werden für die Bepunktung berücksichtigt:

Blatt05/src/RowOfBowls.java Aufgabe 2.1 bis 2.3

Blatt05/src/Genomics.java Aufgabe 3.1 (BONUS)

Als Abgabe wird jeweils nur die letzte Version im git gewertet.

Was Sie nach diesem Blatt wissen sollten:

- Was sind die Grundprinzipien der dynamischen Programmierung?
- Welche beiden Voraussetzungen muss ein Problem erfüllen, damit dynamische Programmierung erfolgreich angewendet werden kann? Welche beiden Varianten werden zur Speicherung der Lösungswert verwendet?
- Wie kann man für ein gegebenes Problem eine OPT-Funktion herleiten? Wie hängt das mit den Teilproblemen zusammen?

Aufgabe 1: Palindrom Teilfolge (Klausurvorbereitung)

Ein *Palindrom* ist eine Zeichenfolge die rückwärts gelesen dieselbe Folge ergibt wie vorwärts gelesen. Das Wort RENTNER ist also ein Palindrom. Eine *Palindrom-Teilfolge* ist eine Teilfolge einer Zeichenkette, die ein Palindrom ist. Dabei muss die Teilfolge nicht an einem Stück in der gegebenen Zeichenkette vorkommen.

Beispiel: DATENSTRUKTUREN enthält u.a. die Palindrom-Teilfolgen NRU KURN und ERUTURE. Auch UKU, NN und jeder einzelne Buchstabe sind (kurze) Palindrom-Teilfolgen der Zeichenkette DATENSTRUKTUREN.

Es soll ein Algorithmus entwickelt werden, der zu einer gegebenen Zeichenkette `str` die Länge der längsten Palindrom Teilfolge bestimmt.

- 1.1 Welche Laufzeit hat der *brute-force* Ansatz (alle Möglichkeiten durchprobieren)?
- 1.2 Geben Sie eine rekursive Definition der $OPT(i, j)$ an, die als Grundlage für dynamisches Programmieren verwendet werden kann.
- 1.3 Welches sind die Anfangswerte, die beim *bottom-up* Ansatz in diesem Beispiel als erstes in die Matrix geschrieben werden?
- 1.4 Was ist die Laufzeit einer Implementation mit Hilfe der OPT Funktion und dynamischer Programmierung.
- 1.5 Füllen Sie die folgende Matrix aus, in dem Sie die Werte von $OPT(i, j)$ per Hand ausrechnen. Entnehmen Sie aus der Matrix, welches die (bzw. eine) Palindrom-Teilfolge maximaler Länge ist.

	T	A	G	A	G	T	G
T							
A							
G							
A							
G							
T							
G							

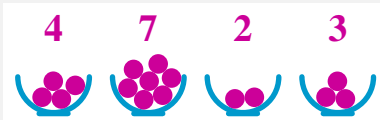
Aufgabe 2: Optimale Auswahl (Hausaufgabe)

Zwei Personen spielen folgendes Spiel. Es gibt eine (meist gerade) Anzahl von Schüsseln mit Murmeln. Die Murmelanzahl in jeder Schüssel ist von Anfang an bekannt. Abwechselnd wählen die Spieler eine der beiden äußeren Schüsseln aus und nehmen die darin befindlichen Murmeln. Am

Ende bekommt jeder Spieler die Differenz der Anzahl der eigenen Murmeln und der Murmeln des Gegenspielers, siehe Abb. 1. Ziel dieser Aufgabe ist es, eine optimale Strategie zu finden.

Abbildung 1: Schüsseln in einer Reihe

Beispiel 1:



Beispiel 2:



Die Spieler wählen abwechselnd eine der äußeren Schüsseln und nehmen die Murmeln. Am Ende zählt die Differenz der gesammelten Murmeln.

Die *Greedy* Strategie, immer die vollere Schüssel zu nehmen, führt oft nicht zum besten Ergebnis. In dem oberen Beispiel ist eine optimale Spielfolge S1: 3 – S2: 4 – S1: 7 – S2: 2, wodurch Spieler S1 eine Punktdifferenz von $(3 + 7) - (4 + 2) = 4$ erreicht. In dem unteren Beispiel ist $(3 + 1 + 8) - (4 + 5 + 2) = 1$ das Optimum.

2.1 Implementation einer rekursiven Lösung (25 Punkte)

Implementieren Sie zunächst eine einfache rekursive Lösung für die optimale Auswahl. Auf Grund der überlappenden Teilprobleme ist der Ansatz nicht effizient und kann nur für relativ kurze Folgen angewendet werden.

Implementieren Sie

```
public int maxGainRecursive(int[] values)
```

als Methode der Klasse `RowOfBowls`. Diese Methode wird einige Variablen initialisieren und dann die eigentliche rekursive Method aufrufen, die denselben Namen und eine andere Signatur haben kann. Sie dürfen Klassenvariablen einführen und benutzen. So lässt sich die Anzahl der Parameter der rekursiven Methode verringern. Die Einträge des Arrays `values` repräsentieren die einzelnen Schüsseln mit der gegebenen Anzahl an Murmeln.

Der Rückgabewert soll die Punktzahl des beginnenden Spielers (Anzahl der Murmeln minus Anzahl der Murmeln des Gegenspielers) sein. Dabei wird vorausgesetzt, dass beide Spieler für sich optimal spielen.

2.2 Effiziente Lösung mit Dynamischer Programmierung (40 Punkte)

Implementieren Sie eine effiziente Lösung mit Hilfe von Dynamischer Programmierung als Methode

```
public int maxGain(int[] values)
```

Dadurch soll eine Laufzeit und Speicherbedarf in $O(n^2)$ für Schüsselanzahl n erreicht werden. Die Matrix der Zwischenlösungen sollte in einer Klassenvariablen gespeichert werden, da sie für die nächste Teilaufgabe benötigt wird.

2.3 Optimale Spielsequenz (35 Punkte)

Implementieren Sie die Methode

```
public Iterable<Integer> optimalSequence()
```

die eine Spielsequenz zweier optimaler Spieler zurückgibt. Die Methode wird immer nach `maxGain()` aufgerufen und muss die *Indizes* der ausgewählten Schlüssel zurückgeben. In dem oberen Beispiel in Abb. 1 sind 3–0–1–2 und 3–2–1–0 optimale Spielsequenzen.

Bemerkungen:

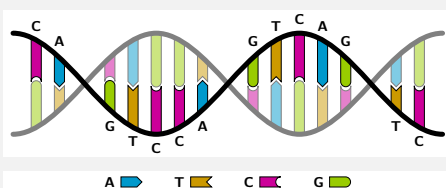
- Die optimale Spielsequenz ist nicht eindeutig. Es spielt keine Rolle, welche dieser *optimalen* Sequenzen die Methode zurückgibt.
- Achten Sie auf die Reihenfolge. Bei der Iteration über die Rückgabevariable wird erwartet, dass der erste Zug zuerst kommt.
- Die Methode `maxGain()` sollte `values` und die Matrix der Zwischenlösungen in Klassenvariablen speichern, damit sie hier zur Verfügung stehen.
- Überprüfen Sie, dass die Punkte des beginnenden Spielers, wenn gemäß der optimalen Sequenz gespielt wird, mit den Punkten, die in der `maxGain()` zurückgegeben werden, übereinstimmen.
- Der Code für unsere Lösungen ist in keiner Methode länger als 15 Zeilen. Das heißt nicht, dass Sie keinen längeren Code schreiben dürfen. Es kann Ihnen aber eine Richtlinie geben.

Aufgabe 3: Sequenzierung (Bonusaufgabe)

Diese Aufgabe ist von der Genomsequenzierung inspiriert. Sie ist stark vereinfacht, aber es wurden tatsächlich Ansätze basierend auf Dynamischer Programmierung zur DNA Analyse entwickelt.

Es sei eine lange Zeichenfolge (*strang*), sowie ein Wörterbuch (*dictionary*) mit vielen kurzen Wörtern geben. Es ist die Aufgabe zu bestimmen, auf wie viele unterschiedliche Möglichkeiten die Zeichenfolge aus Wörtern des Wörterbuches zusammengesetzt werden kann. Dabei können Wörter beliebig oft verwendet werden.

Abbildung 2: Sequenzierung



Strang: CAGTCCAGTCAGTC

Wörterbuch: AGT, CA, CAG, GTC, TC, TCA, TCC

Es gibt vier Möglichkeiten den Strang durch Wörter aus dem Wörterbuch darzustellen:

CA | GTC | CAG | TCA | GTC,

CAG | TCC | AGT | CA | GTC und zwei weitere.

3.1 Anzahl der Sequenzierungsmöglichkeiten (20 Punkte)

Implementieren Sie in der Klasse `Genomics` die Methode

```
public static long optBottomUp(String strang, String[] dictionary)
```

zur Bestimmung der Anzahl von unterschiedlichen Sequenzierungsmöglichkeiten des gegebenen Stranges in Wörter des Wörterbuches. Der Speicherbedarf soll in $O(n)$ für Stranglänge n liegen, unabhängig von der Anzahl und Länge der Wörter im Wörterbuch. Benutzen Sie den *Bottom*

Up Ansatz der Dynamischen Programmierung. Die *Top Down* Berechnung funktioniert bei sehr langen Sequenzen nicht, da die maximale Anzahl der Rekursionen überschritten werden würde.

Tipps und Hinweise:

- Die Methode der Klasse `String`

```
String.startsWith(String substring, int k)
```

prüft, ob `substring` in dem `String`-Objekt beginnend bei Index `k` enthalten ist, wobei `k=0` dem Anfang der Zeichenkette entspricht..

- Es gibt unterschiedliche Möglichkeiten, die rekursive Funktion `OPT` als Grundlage der Dynamischen Programmierung zu definieren. Eine Möglichkeit ist es, $OPT(k)$ als Anzahl der Sequenzierungsmöglichkeiten der Zeichenkette ab dem k -ten Buchstaben zu definieren. Tipp: Im Gegensatz zu vorherigen Dynamischen Programmen, ist hier eine Schleife zur Summierung notwendig, um den Wert von $OPT(k)$ zu berechnen.
- Die Anzahl der Möglichkeiten kann sehr groß werden. In der Implementation muss daher der Typ `long` verwendet werden. Für einen noch größeren Zahlenbereich stellt Java die Klasse `BigInteger` zur Verfügung. Bei den Beispielen, die in den Testes verwendet werden, reicht `long`.
- Der Code für unsere Lösung ist nicht länger als 15 Zeilen. Das heißt nicht, dass Sie keinen längeren Code schreiben dürfen. Es kann Ihnen aber eine Richtlinie geben.