Technische Universität Wien
Institut für Computergraphik und Algorithmen
Arbeitsbereich für Algorithmen und Komplexität

186.866 Algorithmen und Datenstrukturen VU

Programmieraufgabe E7

---

1 Übersicht

Ihre Aufgabe ist es eine optimale Strategie für das "Coin Game" zu entwickeln. Dabei sollen Sie errechnen, was die maximale (und damit optimale) Anzahl von Münzen ist, die Sie in einer gegebenen Runde "Coin Game" erreichen können. Die optimale Strategie soll dabei einmal naiv, d.h. durch Aufzählen aller Spielzustände, und einmal durch dynamische Programmierung, also durch das clevere Ausnutzen der überlappenden Subprobleme, gelöst werden.

Das benötigte Framework ("AlgoDat_E7.zip") steht im TUWEL zu Verfügung. Implementieren Sie in der Datei "src/exercise/E7.java" folgende Methoden:

- `naiveGame(Integer[] bowls)` löst das Problem naiv.
- `dynGame(Integer[] bowls)` löst das Problem durch ein dynamisches Programm.

2 Coin Game

Im "Coin Game" spielen zwei Spieler gegeneinander. Vor ihnen stehen Schalen in einer Reihe nebeneinander. Jede Schale enthält eine Anzahl von Münzen. Die Spieler dürfen abwechselnd entweder die Schale am **rechten** oder **linken** Rand nehmen und alle Münzen darin auf ihre Seite legen. Die Spieler wiederholen dies solange, bis keine Schale mehr übrig ist. Gewonnen hat am Ende der Spieler mit den **meisten** Münzen auf seiner Seite.

Beachten Sie, dass dies ist ein Spiel mit perfekter Information ist. Das bedeutet, dass jeder Spieler zu jedem Zeitpunkt weiß wie viele Münzen beide Spieler bisher auf ihrer Seite haben, sowie wie viele Münzen sich in **jeder einzelnen** Schale befinden. Wenn Sie ihre optimalen Strategie für das "Coin Game" entwickeln, nehmen Sie an, dass Ihr Gegner ebenfalls **optimal** spielt und **Sie immer beginnen**.

Hinweis: Ihr Gegner wird seine Züge so wählen, dass er eine maximale Anzahl von Münzen erlangt und Sie damit eine **minimale Anzahl**.

Spiel zwischen Spieler A und B	
Aufgabe:	Spieler A beginnt. Finden Sie die maximale Anzahl an Münzen, die Spieler A erreichen kann.
Input:	Ein Array $[4, 6, 2, 3]$
Lösung:	9 (A:3, B:4, A:6, B:2)

3 Implementierung

Beginnen Sie mit der Methode `naiveGame(Integer[] bowls)`, wobei **bowls** ein unsortiertes Integer-Array ist. Jeder Eintrag im Array steht für die Anzahl Münzen in einer Schale. Hierbei ist Index 0 die Schale am linken Ende und der letzte Eintrag im Array die Schale am rechten Ende. Lösen Sie das Problem, indem Sie alle möglichen Spielzüge und alle möglichen darauf folgenden Spielzüge Ihres Gegners aufzählen und das optimale Ergebnis berechnen. Denken Sie daran, dass Ihr Gegner ebenfalls **optimal** spielt.

In der Methode `dynGame(Integer[] bowls)` sollen Sie das Problem durch dynamische Programmierung lösen, indem Sie die **überlappenden Subprobleme** und deren optimale Lösung ausnutzen.

4 Testen

In der Datei "E7.java" befinden sich zwei boolean Variablen, die kennzeichnen, welche Methoden Sie implementiert haben. Wenn Sie diese Variablen auf *true* setzen, wird Ihre Implementierung dieser Methode getestet, sonst nicht.

- `NAIVE_GAME_IS_IMPLEMENTED = true` kennzeichnet, dass die Methode `naiveGame()` implementiert ist.
- `GAME_IS_IMPLEMENTED = true` kennzeichnet, dass die Methode `dynGame()` implementiert ist.

Sie müssen **beide Methoden implementieren und die Fragen beantworten, um die Aufgabe vollständig zu lösen**. Achten Sie darauf, dass in Ihrer finalen Abgabe beide Variablen auf *true* gesetzt sind.

Wenn Sie die Datei "AlgoDat_E7.java" ausführen und eine der fünf Instanzen auswählen erhalten Sie entweder eine Nachricht, dass die ausgewählte Instanz korrekt von allen implementierten Methoden gelöst und eine dementsprechende *.csv*-Datei im *solutions* Ordner erzeugt wurde, oder eine Fehler-Nachricht, die Ihnen mitteilt welche Methode ein falsches Ergebnis erzeugt hat, bei welcher Instanz und wie das richtige Ergebnis lauten sollte.

5 Auswertung

Wenn Sie die Datei "AlgoDat_E7.java" ausführen, können Sie, wie im Tutorial beschrieben, in der Konsole eine Testinstanz auswählen.

Es stehen Ihnen fünf verschiedene Instanzen zur Verfügung. Die "01_debug.csv" besteht nur aus trivialen Testfällen mit einer, bzw. zwei Schalen. "02_small.csv" kann immer noch von Hand nachvollzogen werden. Die Instanzen "03_medium.csv" und "04_abgabe.csv" sollten von einer korrekten Lösung in ca. 2 Minuten oder schneller gelöst werden. Die Instanz "05_big.csv" soll aufzeigen wie schnell die Diskrepanz zwischen der naiven Lösung und dem dynamischen Programm wächst. Sie sollte von dem dynamischen Programm immer noch schnell gelöst werden können, die naive Herangehensweise braucht hier jedoch bis zu einer halben Stunde. **Für die Abgabe relevant ist nur die Instanz "04_abgabe.csv"**.

Es wird eine *.csv-Datei* im *solutions* Ordner erstellt, welche Sie mit der Datei "E7/Evaluation/plot.html" auswerten können. Sie bekommen einen Vergleich der Laufzeiten relativ zur Instanzgröße für die beiden von Ihnen implementierten Varianten präsentiert. **Beachten Sie wie immer, dass die Laufzeiten der sehr kleinen Instanzen nicht aussagekräftig sind**. Beantworten Sie folgende Fragen:

- Wie haben Sie die naive Version implementiert? Beschreiben sie ihre Implementierung.
- Welche Veränderungen haben sie vorgenommen um das dynamische Programm zu erstellen?
- Welchen Unterschied sehen Sie zwischen den beiden Implementierungen bezüglich ihrer Laufzeiten? Begründen Sie Ihre Antwort.