technische universität dortmund

Fakultät für Informatik
Lehrstühle 2, 11, 13

Jun.-Prof. Dr. Maike Buchin
I. Kamehkhosh, N. Kriege, D. Kurz, M. Ludewig, B. Zey

Sommersemester 2018
5. Juni

DAP2 Praktikum – Blatt 10

Abgabe: 18.–22. Juni

Wichtig: Der Quellcode ist natürlich mit sinnvollen Kommentaren zu versehen. Überlegen Sie außerdem, in welchen Bereichen Invarianten gelten müssen, und überprüfen Sie diese ggf. an sinnvollen Stellen mit *Assertions* (siehe Hinweis auf Blatt 2).

Langaufgabe 10.1: Dynamische Programmierung (5 Punkte)

Gegeben sei ein Integer-Array A der Länge n. Das *Maximum-Subarray-Problem* besteht darin, ein zusammenhängendes Subarray von A zu finden, dessen Zahlen die größte Summe haben. Ein solches Subarray nennen wir *maximales Subarray*. Bedenken Sie, dass auch das leere Subarray eine korrekte Lösung sein kann. Überzeugen Sie sich zunächst von folgenden Aussagen:

- Es kann mehrere maximale Subarrays geben.
- Hat A nur nicht-negative Zahlen, so ist die Antwort trivial.
- Es gibt einen naiven Algorithmus, der in $O(n^2)$ Zeit das Problem löst.
- Sei $A[i..j]$ ein maximales Subarray von $A[1..j]$. Dann ist $A[i..j+1]$ ein maximales Subarray von $A[1..j+1]$, genau dann wenn $A[j+1] \geq 0$.

Wir schreiben nun einen Algorithmus, der von links nach rechts das Array durchwandert und dabei das maximale Subarray von $A[1..j]$ für alle j berechnet. Dazu wird neben der besten bisher gefundenen Lösung auch ein Kandidatenarray verwaltet. Ein *Kandidatenarray* im Schritt j ist ein Subarray $A[i..j]$ von A mit der Eigenschaft $\sum_{k=i}^{j} A[k] \geq 0$. Im Schritt $j+1$ wird das Kandidatenarray aus dem vorigen Schritt j um das Element $A[j+1]$ erweitert, falls $\sum_{k=i}^{j+1} A[k] \geq 0$. Das maximale Subarray von $A[1..j+1]$ ist dann entweder das erweiterte Kandidatenarray oder die beste zuvor gefundene Lösung. Falls $\sum_{k=i}^{j+1} A[i] < 0$, verwerfen wir das aktuelle Kandidatenarray und fahren wir mit einem leeren Kandidatenarray fort.

(a) Implementieren Sie den naiven Algorithmus, der das Problem in Zeit $O(n^2)$ löst.

(b) Implementieren Sie den oben beschriebenen Algorithmus, der auf dynamischer Programmierung beruht.

(c) Welche theoretische Laufzeit hat dieser Algorithmus?

(d) Messen und vergleichen Sie die Laufzeiten der beiden Algorithmen. Erstellen Sie dazu Arrays unterschiedlicher Größe mit positiven und negativen Zufallszahlen.

Langaufgabe 10.2: Türme von Hanoi (3 Punkte)

Die Türme von Hanoi ist ein bekanntes Spiel, das sehr elegant rekursiv gelöst werden kann. Das Spiel besteht aus drei Stäben A, B und C, auf die mehrere gelochte Scheiben gelegt werden, die alle verschieden groß sind. Zu Beginn liegen alle Scheiben auf Stab A, der Größe nach geordnet, mit der größten Scheibe unten und der kleinsten Scheibe oben. Ziel des Spiels ist es, den kompletten Scheiben-Stapel von A nach C zu versetzen. Stab B dient dabei als Hilfsstapel.

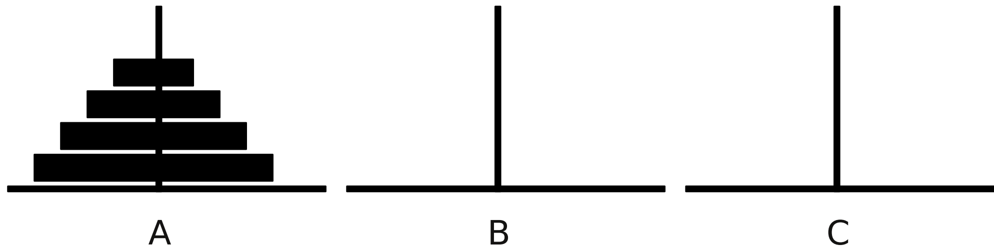

Abbildung 1: Die Türme von Hanoi mit vier Scheiben, Ausgangssituation

Spielregeln: Bei jedem Zug darf die oberste Scheibe eines beliebigen Stabes auf einen der beiden anderen Stäbe gelegt werden. Zu jedem Zeitpunkt des Spieles müssen die Scheiben auf jedem Stab der Größe nach geordnet sein. Es darf also nie eine größere Scheibe auf einer kleineren Scheibe abgelegt werden.

Schreiben Sie eine rekursive Funktion `void move(int quantity, char start, char help, char target)`, die das Verschieben von `quantity` Scheiben vom Stab `start` auf den freien Zielstab `target` modelliert. Dabei soll Stab `help` als Hilfsstapel verwendet werden. Aufgabe der Funktion ist es, eine Anleitung zum Lösen des Spiels auf dem Bildschirm auszugeben.

Beispielsweise könnte die Ausgabe für $n = 4$ wie folgt aussehen:

```
Verschiebe oberste Scheibe von A nach B
Verschiebe oberste Scheibe von A nach C
Verschiebe oberste Scheibe von B nach C
Verschiebe oberste Scheibe von A nach B
Verschiebe oberste Scheibe von C nach A
Verschiebe oberste Scheibe von C nach B
Verschiebe oberste Scheibe von A nach B
Verschiebe oberste Scheibe von A nach C
Verschiebe oberste Scheibe von B nach C
Verschiebe oberste Scheibe von B nach A
Verschiebe oberste Scheibe von C nach A
Verschiebe oberste Scheibe von B nach C
Verschiebe oberste Scheibe von A nach B
Verschiebe oberste Scheibe von A nach C
Verschiebe oberste Scheibe von B nach C
```

Orientieren Sie sich dabei an folgender Idee:

(1) Falls nur eine Scheibe von `start` nach `target` verschoben werden soll, so ist dies unmittelbar möglich. Die durchgeführte Verschiebung soll durch Verwendung einer Ausgabeanweisung protokolliert werden.

(2) Falls mehrere Scheiben verschoben werden sollen, so geht man wie folgt vor:

   (a) Zunächst werden `quantity - 1` Scheiben von Stab `start` nach Stab `help` verschoben (*Rekursionsaufruf*). Dabei nimmt Stab `target` die Rolle als Hilfsstapel ein.
   (b) Anschließend wird – wie in Fall (1) beschrieben – die größte (und einzige) Scheibe von `start` nach `target` bewegt.
   (c) Abschließend werden `quantity - 1` Scheiben von Stab `help` nach Stab `target` verschoben (*Rekursionsaufruf*). Dabei nimmt Stab `start` die Rolle als Hilfsstapel ein.

Hinweis: Sie sollen in dieser Aufgabe nicht versuchen, den konkreten Inhalt eines Stapels in einer Variable zu speichern und in dieser den Ablauf der Spielzustände zu simulieren. Rekursive Aufrufe der Funktion und passende Ausgaben sind ausreichend, um die Aufgabe zu lösen.

Schreiben Sie eine `main`-Funktion, in der zunächst die Anzahl der Scheiben in einer Variablen n eingelesen wird und dann ein Aufruf von `move(n, 'A', 'B', 'C')` erfolgt. In Folge des Aufrufs werden die Züge der Strategie durch die Ausgaben in der rekursiven `move`-Funktion protokolliert.