**DAP2 Praktikum für ETIT und IKT, SoSe 2018, Langaufgabe L5**

Fällig am 09.07. um 14:00

Es gelten die üblichen Programmierregeln in der erbarmungsloser Härte. Hässlicher Code führt zu Punktabzug. Assertions sind nicht notwendig.

## Langaufgabe L5, Teil 1: Dynamische Programmierung (1,4 Punkte)

Suchen Sie mit Hilfe der dynamischen Programmierung aus einem Array von Zahlen A[1..n] das zusammenhängende Sub-Array, dessen Zahlensumme maximal ist. Dieses wird „maximales Sub-Array“ genannt. Folgendes ist zu beachten:

- Auch das leere Sub-Array kann eine korrekte Lösung sein.
- Es kann mehrere maximale Subarrays geben. Sie müssen nur eines finden.
- Sind im Array nur nicht-negative Zahlen, so ist die Antwort trivial.
- Es gibt einen naiven Algorithmus, der in $\Theta(n^2)$ Zeit das Problem löst.
- Sei A[i..j] ein maximales Sub-Array von A[1..j]. Dann ist A[i..j+1] ein maximales Sub-Array 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 jeweils das maximale Sub-Array von A[1..j] für alle j berechnet. Dazu wird neben der bisher gefundenen besten Lösung (den Bereich der besten Lösung also immer merken) auch ein Kandidatenarray verwaltet. Am Anfang beinhaltet die beste Lösung natürlich noch keine Elemente.

- Ein Kandidaten-Array A[i..j] ist ein Sub-Array von A mit der Eigenschaft $\sum_{k=i}^{j} A[k] \geq 0$.
- Wir erweitern ein Kandidaten Array A[i..j] zu A[i..j+1], falls $\sum_{k=i}^{j+1} A[k] \geq 0$
- Das maximale Sub-Array von A[1..j+1] ist dann entweder die beste zuvor gefundene Lösung oder das erweiterte Kandidaten-Array. Das kann an der Summe $\sum_{k=i}^{j+1} A[k]$ im Vergleich mit der bisherigen Lösung leicht entschieden werden.
- Falls $\sum_{k=i}^{j+1} A[k] < 0$, verwerfen wir das aktuelle Kandidaten-Array und fahren mit einem zunächst leeren Kandidatenarray fort, dass genau nach dem Element startet, welches dazu führte, dass $\sum_{k=i}^{j+1} A[k] < 0$ wurde.

Es gibt folgende Aufgaben:

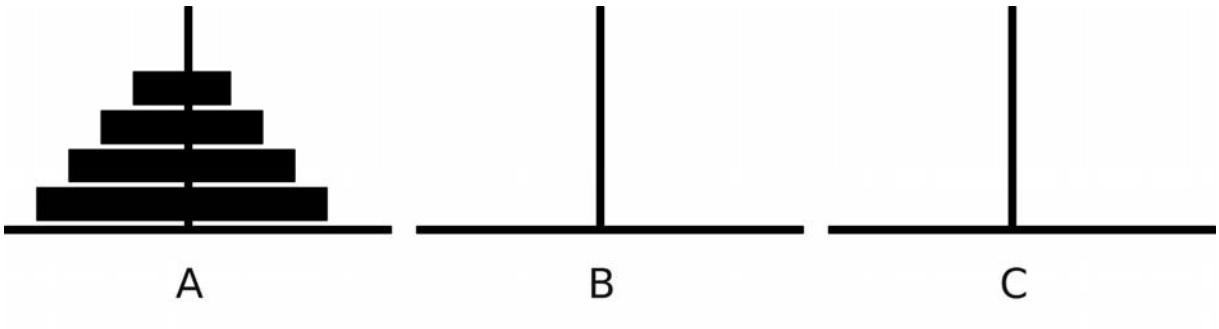
- Implementieren Sie in Form eines Templates das Unterprogramm `NaiveMaxSubsum` das das Problem löst. Die Aufrufdefinition entnehmen Sie dem Gerüst des Quelltextes.

- Implementieren Sie den oben beschriebenen Algorithmus als Template eines Unterprogramms `DynamicMaxSubsum`
- Welche theoretische Laufzeit hat der Algorithmus?
- Vergleichen Sie die Laufzeiten zwischen der naiven Variante und dem hier vorgestelltem Algorithmus. Erstellen Sie dazu Arrays unterschiedlicher Größe mit positiven und negativen Zufallszahlen.

Verwenden Sie das Ihnen zur Verfügung gestellte Programmgerüst. Editieren ist nur dort erlaubt, wo es gekennzeichnet ist.

**Langaufgabe L5, Teil 1: Türme von Hanoi (0,6 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.




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. Die Funktion `move` darf dazu Ausgaben nach `cout` machen. 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.
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. Verwenden Sie dazu das Ihnen zur Verfügung gestellte Programmgerüst. Editieren ist nur dort erlaubt, wo es gekennzeichnet ist.