Programmieren 1
Übungsserie 8

Stoff

- Bis zu Kapitel 12
- Fokus: Rekursion

Allgemeine Informationen zur Abgabe

- Die Abgabe erfolgt online auf ILIAS.
- Quellcode zu den Implementationsaufgaben muss als `*.zip` Datei abgegeben werden. Exportieren Sie hierzu Ihr Projekt direkt aus Eclipse. Quellcode, den wir nicht kompilieren können, wird nicht akzeptiert.
- Lösungen zu den Theorieaufgaben muss als `*.pdf` Datei abgegeben werden. Andere Formate werden nicht akzeptiert.
- Arbeit in Zweiergruppen: Geben Sie jeweils nur ein Exemplar der Lösung pro Gruppe ab. Geben Sie in jeder Quellcode-Datei die *Namen und Matrikelnummern* beider Gruppenmitglieder in den ersten beiden Zeilen als Kommentar an.
- Vorbesprechung: *09.12.2022*
- Abgabe: *16.12.2022 13:00 Uhr*

Theorieaufgaben

1. Schreiben Sie eine rekursive Definition von x^y, wobei x und y ganze Zahlen sind und $y > 0$ gilt.

2. Gegeben seien folgende Definitionen:

   - `Java_Letter` = {`A, B, C, ..., Z, a, b, c, ..., z, _, $`}
   - `Java_Digit` = {`0, 1, 2,... ,9`}

   Schreiben Sie eine rekursive Definition für `Java_Identifier` für gültige Java Bezeichner. Sie müssen nur eine Definition angeben und keinen (Pseudo)-Code schreiben. (vgl Skript Kapitel 12.1 s 252)

3. Illustrieren Sie die rekursiven Aufrufe der Methode `traverse` (ähnlich zu Abb. 12.7 im Skript) für das folgende Labyrinth der Grösse 3×3:

   ```
   1,1,1
   1,1,0
   0,0,1
   ```

   Es müssen alle Aufrufe beachtet werden, also auch die, die ausserhalb der Matrix oder auf einem bereits besuchten Feld sind.

Implementationsaufgaben

1. Setzen Sie für folgende rekursive Definition der Datenstruktur `Bag` in Java um: Ein `Bag` ist entweder leer oder enthält ein Objekt vom Typ `Integer` und einen `Bag`.

   - Schreiben Sie zwei Konstruktoren `public Bag()` (für einen leeren `Bag`) und `public Bag(int value)` (für einen `Bag` mit einem initialen Wert).
   - Schreiben Sie eine Methode `public void addValue(int value)`, welche einen neuen Wert zum `Bag` hinzufügt.
   - Schreiben Sie eine Methode `public String toString()` zur Ausgabe des Inhaltes von `Bag`. Die Ausgabe von folgendem Testprogramm

     ```
     Bag b1 = new Bag();
     System.out.println(b1);

     Bag b2 = new Bag(1);
     System.out.println(b2);

     Bag b3 = new Bag(1);
     b3.addValue(2);
     b3.addValue(3);
     System.out.println(b3);
     ```

     soll sein

     ```
     EMPTY
     (1, EMPTY)
     (1, (2, (3, EMPTY)))
     ```

2. Schreiben Sie eine *rekursiv* Methode `static long fib(int i)`, die die i-te Zahl der Folge

   $$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots$$

   berechnet. Z.B. soll der Aufruf `fib(7)` den Wert `13` liefern. Die erste Zahl der Folge ist `0` (entsprechend `fib(0)`, die zweite `1` (`fib(1)`, danach ist jede Zahl die Summe ihrer beiden Vorgänger, z.B. `fib(8) = fib(7) + fib(6)`. Allgemein: `fib(n) = fib(n-1) + fib(n-2)` (für `n >= 2`)

   Schreiben Sie dazu eine passende `main`-Methode, die die ersten `50` Zahlen der Folge am Bildschirm ausgibt. Was stellen Sie beim Ausführen des Programms fest?

3. Wir betrachten das Problem der Koordination verschiedener konkurrierender Aktivitäten, die eine gemeinsame Ressource exklusiv nutzen. Nehmen Sie an, Sie haben ein Array A von n Aktivitäten anstehend, die eine zu jedem Zeitpunkt jeweils nur von einer Aktivität verwendbare Ressource benutzen möchten (z.B. einen Hörsaal an einer Uni, eine Maschine in einem produzierenden Unternehmen, etc.). Jede Aktivität besitzt zwei Attribute s und f, welche die Start und Endzeit der Aktivität definieren ($A[i].s$ ist zum Beispiel die Startzeit der i-ten Aktivität). Es gilt: $0 \leq A[i]s < A[i]f < \infty$.

Zwei Aktivitäten $A[i|$ und $A[j]$ werden als kompatibel bezeichnet, wenn sie sich nicht überlappen, d.h. entweder $A[i].s \geq A[j].f$ oder $A[j].s \geq A[i].f$ gilt. Im *Aktivitäten-Auswahl Problem* wollen wir eine maximale Teilmenge paarweise zueinander kompatibler Aktivitäten finden.

Wir setzen im folgenden voraus, dass die Aktivitäten in aufsteigender Reihenfolge nach ihren Endzeiten sortiert sind:

$$A[0].f \leq A[1].f \leq \ldots \leq A[n].f$$

Zudem legen wir fest, dass $A[0]$ eine *Dummy* Aktivität ist, mit Start- und Endzeit Null ($A[0].s = A[0].f = 0$).

Wir wollen nun einen rekursiven Algorithmus für dieses Problem entwickeln. Folgen wir der Intuition sollten wir zu jedem Zeitpunkt die Aktivität wählen, die die Ressource für so viele andere Aktivitäten wie möglich frei lässt. Das bedeutet, wir sollten zu jedem Zeitpunkt eine Aktivität auswählen, die möglichst früh endet.

Wir wählen also zu jedem Zeitpunkt die Aktivität $A[i]$, welche die frühste Endzeit $A[i].f$ besitzt (und somit auch maximal viel Platz für weitere/zukünftige Aktivitäten lässt). Wenn diese Wahl gemacht wurde, haben wir ein verbleibendes Teilproblem zu lösen, nämlich eine maximale Menge an Aktivitäten zu bestimmen, die starten, nachdem $A[i]$ fertig ist (also mit Startzeiten nach $A[i].f$).

Diese Idee ist in folgendem rekursiven Algorithmus formalisiert. Als Eingabe erhält der Algorithmus ein Array A mit Aktivitäten, den Index k, der das zu lösende Teilproblem definiert und die Grösse n des ursprünglichen Problems. Der erste Aufruf von Activity-Selector erfolgt mit den Parametern Activity-Selector$(A, 0, A.length)$

Algorithm 1 Activity-Selector(A, k, n)

```
1: m = k + 1
2: while m < n AND A[m].s < A[k].f do
3:     m = m + 1
4: end while
5: if m < n then
6:     return {A[m]} ∪ Activity-Selector(A, m, n)
7: else
8:     return ∅
9: end if
```

Laden Sie von ILIAS die Dateien `Activity.java` und `ActivitySelector.java` herunter. Die Klasse `Activity` definiert eine Aktivät durch einen Namen, eine Start- und eine Endzeit. In der Klasse `AcitivitySelector` wird in der Methode `main` ein Array mit 12 Aktivitäten erzeugt und an die rekursive Methode `activitySelection` übergeben. Ihre Aufgabe ist es, diese Methode zu implementieren und zu testen.