P1 - Serie 08

Lukas Batschelet (16-499-733)

Theorieaufgaben

(01) Rekursive Definition

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

Lösung

$$\begin{cases} x & \text{wenn } y = 1 \\ x \cdot x^{y-1} & \text{wenn } y > 1 \end{cases}$$

(02) Rekursive Definitionen in Java

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)

Mögliche Lösung

1. `Java_Identifier` = `Java_Letter` (Basisfall)
2. `Java_Identifier` = `Java_Identifier + (Java_Letter || Java_Digit)` (Rekursion)

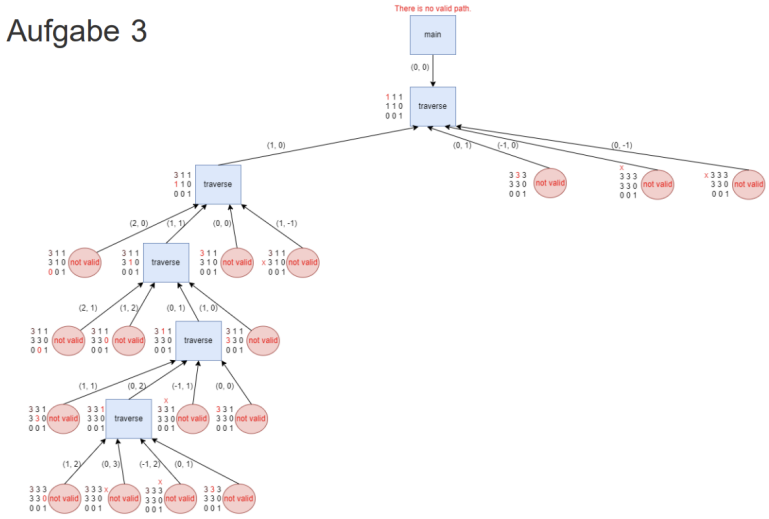
(03) Labirinth rekursiv durchqueren

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
2 | 1, 1, 0
3 | 0, 0, 1
```

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

Lösung



Implementationsaufgaben

(01) Datenstruktur `Bag`

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)))
   ```

Mögliche Lösung

```
public class Bag {

    private int value;
    private Bag innerBag;

    public Bag() {
        this.innerBag = null;
    }

    public Bag(int value) {
        this.value = value;
        this.innerBag = new Bag();
    }

    public void addValue(int value) {
        if (this.innerBag == null) {
            this.value = value;
            this.innerBag = new Bag();
        } else {
            this.innerBag.addValue(value);
        }
    }

    public String toString() {
        if (this.innerBag == null) {
            return "EMPTY";
        } else {
            return "(" + this.value + ", " + this.innerBag + ")";
        }
    }
}
```

(02) Implementierung der Fibonacci-Funktion

Schreiben Sie eine rekursive 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, ...
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?

Mögliche Lösungen

```java
public static long fib(int i) {
        switch (i) {
            case 0:
                return 0;
            case 1:
                return 1;
            default:
                return fib(i - 1) + fib(i - 2);
        }
    }

public static long fib(int n) {
        return (n < 2 ? n : fib(n - 1) + fib(n - 2));
}
```

(03) Koordination konkurrierender Aktivitäten

Wir betrachten das Problem der Koordination verschiedener konkurrierender Aktivitäten, die eine gemeinsame Ressource exklusiv nutzen. Angenommen, Sie haben ein Array `A` von `n` Aktivitäten, die zu jedem Zeitpunkt jeweils nur von einer Aktivität genutzte Ressource beanspruchen möchten (z.B. einen Hörsaal an einer Uni, eine Maschine in einem produzierenden Unternehmen, etc.). Jede Aktivität hat 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]` sind kompatibel, wenn sie sich nicht überlappen, d.h., es gilt entweder $A[i].s \geq A[j].f$ oder $A[j].s \geq A[i].f$. Im Aktivitäten-Auswahl Problem wollen wir eine maximale Teilmenge paarweise kompatibler Aktivitäten finden.

Wir setzen voraus, dass die Aktivitäten nach ihren Endzeiten aufsteigend sortiert sind: $A[0].f \leq A[1].f \leq \ldots \leq A[n].f$. Zudem ist `A[0]` eine Dummy-Aktivität mit Start- und Endzeit Null ($A[0].s = A[0].f = 0$).

Wir wollen einen rekursiven Algorithmus für dieses Problem entwickeln. Intuitiv sollten wir zu jedem Zeitpunkt die Aktivität wählen, die die Ressource für möglichst viele andere Aktivitäten freilässt, also eine Aktivität, die möglichst früh endet. Wenn diese Wahl getroffen wurde, haben wir ein verbleibendes Teilproblem zu lösen: eine maximale Menge an Aktivitäten zu bestimmen, die starten, nachdem `A[i]` fertig ist.

Der Algorithmus `Activity-Selector` erhält als Eingabe 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)

```
m = k + 1
while m < n AND A[m].s < A[k].f do
            m = m + 1
end while
if m < n then
    return {A[m]} ∪ Activity-Selector(A, m, n)
else
    return ∅
end if
```

Laden Sie von ILIAS die Dateien `Activity.java` und `ActivitySelector.java` herunter. Die Klasse `Activity` definiert eine Aktivität durch einen Namen, eine Start- und eine Endzeit. In der Klasse `ActivitySelector` 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.

Mögliche Lösung

```java
private static ArrayList<Activity> activitySelection(Activity[] activities, int k, int n) {
        ArrayList<Activity> selection = new ArrayList<Activity>();
        int index = k + 1;

        // Finds the next compatible Activity
        while (index < n && activities[index].getStartTime() < activities[k].getEndTime()){
                index++; }

        if (index < n) {
                selection.add(activities[index]);
                selection.addAll(activitySelection(activities, index, n)); // recursive call
        }
        return selection;
}
```