5. Übung

Abgabe bis 27.11.2017, 10:00 Uhr

Einzelaufgabe 5.1: Rekursion: Gray-Code — 10 EP

In der Vorlesung wurde der sog. Gray-Code vorgestellt. In dieser Aufgabe soll von Ihnen ein Generator für Gray-Codes implementiert werden. Erstellen Sie vorab eine neue Klasse `GrayCode`.

a) Implementieren Sie zunächst die statische Klassenmethode `int prevLength(int len)`. Diese Methode soll die Länge des nächst **kleineren** Gray-Codes zurückgeben. Hat ein Gray-Code die Länge len, so hat der nächst kleinere Gray-Code die Länge $\frac{len}{2}$, wenn len durch zwei teilbar ist. Ansonsten ist die Länge des nächst kleineren Gray-Codes durch die größte 2er-Potenz bestimmt, die kleiner len ist.

b) Ergänzen Sie danach `String[] generate(GrayCodeControl gcc, int len)`. Diese Methode erzeugt mittels Rekursion zuerst den nächst kleineren Gray-Code, um diesen anschließend zu einem Gray-Code $\mathcal{G}_{len}$ der Länge len zu erweitern. Die Basisfälle stellen hierbei die Gray-Codes $\mathcal{G}_1 = \{"0"\}$ sowie $\mathcal{G}_2 = \{"0", "1"\}$ dar. Für die Länge $len = 4$ soll demnach die Zeichenkettenreihung $\mathcal{G}_4 = \{"00", "01", "11", "10"\}$ zurückgegeben werden.
ACHTUNG – WICHTIG: Gleich zu Beginn Ihrer `generate` **müssen** Sie die Methode `gcc.logGenerate(gcc, len)` aufrufen und ihr die Argumente des aktuellen Aufrufs übergeben. Diese Methode dient der Überprüfung Ihrer Implementierung und ist für die automatische Bewertung im EST essentiell! Prüfen Sie Ihre Lösung mit dem öffentlichen Test!

Geben Sie Ihre Lösung als `GrayCode.java` über EST ab.

Einzelaufgabe 5.2: Induktionsbeweis – Catalan-Zahlen — 17 EP

Gegeben sei folgendes rekursives Programmfragment in der Sprache Java:

```java
static long cn(long n) {
    return n == 0 ? 1 : (4 * (n - 1) + 2) * cn(n - 1) / (n + 1);
}
```

a) Um welche Art der Rekursion handelt es sich dabei?

b) Beweisen Sie mittels vollständiger Induktion, dass die Methode `cn(n)` für beliebige $n \geq 0$ die n-te Catalan-Zahl C_n berechnet:

$$C_n = \frac{(2n)!}{(n+1)! \cdot n!}$$

Sie dürfen dabei annehmen, dass es *keinen* Überlauf im Rückgabewert gibt.

c) Geben Sie eine geeignete Terminierungsfunktion an und skizzieren Sie den Nachweis der Terminierung, indem Sie Ihre Wahl genau begründen.

Geben Sie Ihre Lösung als `InduktionCatalan.pdf` über EST ab.

Gruppenaufgabe 5.3: Skyline — 13 GP

Im Folgenden sollen Sie das Skyline-Problem lösen, das Sie aus der Vorlesung kennen. Erstellen Sie eine gleichnamige Klasse in der Datei `SkylineSolver.java` und ergänzen Sie sie wie folgt:

a) Implementieren Sie eine Methode `int area(int[] skyline)`, in der Sie die Fläche unter der Skyline berechnen. Die Skyline ist als Folge von Koordinaten und Höhen gegeben. Sowohl die Koordinaten als auch die Höhen einer Skyline sind garantiert nicht-negativ und die letzte Höhe ist immer 0, wie in der Vorlesung.

b) Die Methode `int[][] divide(int[][] orig, int num, boolean isLeft)` bekommt die Reihung `orig` mit Gebäuden übergeben. Jedes Gebäude in `orig` ist durch eine `int`-Reihung $\{L, H, R\}$ (linke Koordinate, Höhe, rechte Koordinate) definiert.
Im Fall `isLeft` soll die Methode die ersten `num` Gebäude (ansonsten die restlichen Gebäude) jeweils in einer **neuen** Reihung zurückgeben.

c) Implementieren Sie in `int[] conquer(SkylineSolverHelper ssh, int[][] b)` die eigentliche Berechnung der Skyline. Ihre Berechnung **muss rekursiv** sein und die in dieser Aufgabe implementierten Methoden `divide()` und `conquer()`, sowie die vorgegebene Methode `ssh.merge()` verwenden. Achten Sie darauf, dass Ihre Methode `conquer` den Aufwand von $\mathcal{O}(n \cdot \log n)$ bei n Gebäuden nicht übersteigt.

Geben Sie Ihre Lösung als `SkylineSolver.java` über EST ab.

Gruppenaufgabe 5.4: Fibonacci-Verallgemeinerung — 20 GP

Eine der (vielen) möglichen Verallgemeinerungen der Fibonacci-Reihe wird wie folgt definiert:

$$F_n = \begin{cases} n, & \text{falls } 0 \leq n < c \\ aF_{n-1} + F_{n-c}, & \text{falls } n \geq c \text{ und gerade} \\ bF_{n-1} + F_{n-c}, & \text{falls } n \geq c \text{ und ungerade} \end{cases} \quad \Bigg| \quad \text{mit } a, b \in \mathbb{R}, a \neq 0 \wedge b \neq 0, c \in \mathbb{N}^+$$

Sie sollen in der Klasse `GenericFib` verschiedene Implementierungsvarianten umsetzen, die jeweils unterschiedlich viel Rechenzeit bzw. Speicher benötigen.

a) Naive Implementierung: Setzen Sie die Formel ***ohne*** weitere Optimierungen in der Klassenmethode `double fibNaiveRec(GenericFibKontrolle gfk, double a, double b, int c, int n)` um. **WICHTIG:** Rufen Sie gleich zu Beginn der Methode ***unbedingt*** `gfk.fibNaiveRecLog(gfk, a, b, c, n);` auf!

b) Aufrufe zählen: Um welche Rekursionsform[1] handelt es sich bei der naiven Umsetzung (Sie können die Frage stattdessen ebenso anhand der obigen Formel beantworten)? Wie oft wird `fibNaiveRec` mit den Parameter `n == 2` aufgerufen, wenn die Methode einmal mit $a = 1.5 \wedge b = -0.5 \wedge c = 2 \wedge n = 7$ aufgerufen wird?

c) Dynamische Programmierung: Deklarieren Sie ein Klassenfeld `dp`, um Zwischenergebnisse zu speichern, wie es bei der dynamischen Programmierung üblich ist.
Ergänzen Sie anschließend eine Klassenmethode `void initDP(int nMax)`, die das Array `dp` so initialisiert, dass es Zwischenergebnisse für genau $0 \leq$ `n` $\leq$ `nMax` speichern kann. Die

[1] Lineare Rekursion, Endrekursion, Kaskadenartige Rekursion, Verschränkte Rekursion, Verschachtelte Rekursion

Methode muss zusätzlich alle Einträge als „noch unbelegt" markieren – verwenden Sie dazu den Wert `Double.NaN`.
Implementieren Sie nun die eigentliche optimierte Berechnung in der Methode `fibDP`. Sie soll die gleiche Signatur wie `fibNaiveRec` haben und die gleichen Ergebnisse wie `fibNaiveRec` ermitteln, aber das Array `dp` verwenden, um die Berechnung zu beschleunigen. Stellen Sie sicher, dass die eigentliche Berechnung für jedes `n` höchstens einmal (rekursiv) durchgeführt wird. **WICHTIG:** Rufen Sie gleich zu Beginn der Methode ***unbedingt*** `gfk.fibDPLog(gfk, a, b, c, n);` auf!

d) Durchreichen von Ergebnissen: Bei dieser Variante sollen Sie ohne explizite Prüfung davon ausgehen, dass $1 \leq c \leq 3$ ist. Erstellen Sie die Methode `fibDvE`, die die gleiche Signatur wie `fibNaiveRec` hat und ebenfalls die gleichen Ergebnisse (nur für $1 \leq c \leq 3$) ermitteln soll. Implementieren Sie die Hilfsmethode `fibDvEHelper`, deren Signatur identisch beginnt, aber vier zusätzliche Parameter `int i, double mem1, double mem2, double mem3` bekommt. Beim Ausführen dieser Methode soll sie sich in jedem Zweig maximal einmal selbst aufrufen, d.h. sie muss linear rekursiv sein.
Der Parameter `n` der Methode `fibDvEHelper` gibt an, mit welchem `n` die ursprüngliche Methode `fibDvE` aufgerufen wurde, also zu welchem Glied der Folge der Wert berechnet werden soll. Der zusätzliche Parameter `i` gibt hingegen an, welches Folgenglied aktuell in der Methode berechnet wird. Die weiteren Parameter `mem1`, `mem2` und `mem3` geben den Wert des Folgeglieds mit den um 1, 2 bzw. 3 kleineren Index an. Beim ersten Aufruf sind die Werte mit `Double.NaN` gefüllt. Beim Aufruf der nächsten Aufrufebene müssen die richtigen Werte übergeben werden. Für noch nicht bekannte Werte soll `Double.NaN` durchgereicht werden. **WICHTIG:** Rufen Sie gleich zu Beginn Ihrer Methode `fibDvEHelper` ***unbedingt*** `gfk.fibDvELog(gfk, a, b, c, n, i, mem1, mem2, mem3);` auf!

ACHTUNG – WICHTIG: Sie dürfen **ausschließlich** die vorangehend geforderten Methoden (5) und Attribute (1) in Ihrer Klasse implementieren – darüber hinaus **keine einzigen** weiteren! Behalten Sie die Reihenfolge der rekursiven Aufrufe wie in der Formel bei (also zuerst für $n - 1$ und erst danach für $n - c$)! Prüfen Sie Ihre Lösung mit dem öffentlichen Test! Geben Sie Ihre Lösung als `GenericFib.pdf` bzw. `GenericFib.java` über EST ab.

27 EP + 33 GP = 60 Punkte