33. Bundeswettbewerb Informatik

Mobile (Aufgabe 2)

Johannes Heinrich, Marian Dietz

Lösungsidee

1. Textuelle Darstellung eines Mobiles

Die textuelle Darstellung eines Mobiles sieht wie folgt aus:

- Eine Balken wird in der Form „{Position=Komponente, Position=Komponente}“ dargestellt. Der Inhalt wird durch zwei geschweifte Klammern eingeschlossen. Darin sind die verschiedenen Komponenten durch Kommata getrennt enthalten. Die Position ist dabei stets der Aufhängepunkt, also wo die zugehörige Komponente hängt. Jede Komponente ist entweder ein weiterer Balken oder eine Figur. Dabei können auch mehr als zwei Komponenten am Balken hängen, laut Aufgabenstellung sind dies jedoch maximal vier. Die Reihenfolge der Komponenten am Balken ist beliebig.
- Eine Figur wird lediglich durch das eigene Gewicht als Zahl dargestellt.

2. Das Programm

Das Programm besteht hauptsächlich aus zwei Teilen: den Balken und den Figuren. Die Ausbalancierung besteht aus einem einfachen Algorithmus, welcher im Folgenden erklärt wird. n entspricht dabei der Anzahl der Figuren.
Wenn ein Balken B_1 erstellt wird, wird zunächst überprüft, ob alle n Figuren an B_1 „passen“. Laut Aufgabenstellung dürfen dies höchstens vier sein. Sollte dies *nicht* der Fall sein, werden für den Balken lediglich die drei Figuren w_1, w_2 und w_3 genommen, der Rest $w_4, \ldots, w_n$ wird an einen anderen Balken B_2 gehängt (wobei auch wieder überprüft wird, ob die Grenze von vier Figuren nicht überschritten wird und evtl. noch weitere Balken erstellt werden), welcher dann an den ursprünglichen Balken B_1 kommt.

Da nun sichergestellt wurde, dass an jedem Balken höchstens vier Komponenten hängen, kann der Algorithmus zur Ausbalancierung der am Balken B_x hängenden Komponenten folgen. Dazu wird, damit der Balken ausbalanciert ist, vorausgesetzt, dass die Summe der Komponenten mal deren Position gleich null ist: $\sum_{i=1}^{n} w_i \cdot p_i = 0$, wobei n der Anzahl der Komponenten am Balken B_x entspricht, w_i ist das Gewicht der Komponente i und p_i ist die Position der Komponente i.

Sollen nun zwei Komponenten am Balken B_x hängen, muss eine Komponente auf der positiven Seite (links) und eine Komponente auf der negativen Seite (rechts) hängen. Dabei muss $p_1 \cdot w_1 = -(p_2 \cdot w_2)$, damit beide Seiten ausgeglichen sind (die rechte Seite wird negiert, da die Positionen der rechten Seite stets negativ sind). Da w_1 und w_2 bereits vorgegeben sind, müssen lediglich p_1 und p_2 generiert werden. Die eben genannte Gleichung wird zu einer wahren Aussage, indem $p_1 = w_2$ und $p_2 = -w_1$ ist. Denn wenn wir diese Gleichungen in die obere einsetzen, wird daraus folgenden Gleichung: $w_2 \cdot w_1 = -(-w_1 \cdot w_2) = w_1 \cdot w_2$. Dies ist aufgrund des

Kommutativgesetzes eine wahre Aussage. Das bedeutet, dass bei zwei Komponenten diese an folgende Stellen gehängt werden:

- Komponente 1 kommt an der linken Seite an das Gewicht von Komponente 2.
- Komponente 2 kommt an der rechten Seite an das Gewicht von Komponente 1.

Ein ähnlicher Algorithmus wird auch bei vier Komponenten verwendet. Die Gleichung dazu lautet $p_1 \cdot w_1 + k \cdot p_3 \cdot w_3 = -(p_2 \cdot w_2 + k \cdot p_4 \cdot w_4)$. Die ersten beiden Komponenten werden wieder an folgenden Stellen angebracht: $p_1 = w_2$ und $p_2 = -w_1$. Die Gleichungen der beiden letzten Komponenten lauten $p_3 = k \cdot w_4$ und $p_4 = -(k \cdot w_3)$. Hier taucht eine neue Variable k auf. Diese wird genau dann verwendet, wenn mindestens eine der Positionen w_3 oder w_4 bereits durch p_1 oder p_2 besetzt ist, schließlich können nicht mehrere Komponenten an einer Position hängen. Daraus resultieren folgende Werte für k:

1. Wenn $w_3 \neq w_1$ und $w_4 \neq w_2$ ist $k = 1$ (kann also weggelassen werden, da es keine Überscheidungen mit den Positionen gibt).
2. Wenn $w_3 = w_1$ oder $w_4 = w_2$, ist $k = 2$. w_3 und w_4 werden also einfach verdoppelt, damit die zugehörigen Komponenten nicht mit p_1 oder p_2 ins Gehege kommen.
3. Wenn ($w_3 = w_1$ oder $w_4 = w_2$) und ($w_3 = 2 \cdot w_1$ oder $w_4 = 2 \cdot w_2$), ist $k = 3$. Wenn also immer noch nach der Verdoppelung unter Punkt 2 Positionen doppelt belegt sein sollten, werden die Positionen verdreifacht.

Eine höhere Zahl als 3 kann k nicht annehmen, denn es können nicht mehr als zwei Positionen bereits vergeben sein. Dies liegt darin begründet, dass vor den letzten beiden Komponenten w_3 und w_4 erst die zwei Komponenten w_1 und w_2 eingefügt wurden, da lediglich vier Komponenten am Balken hängen sollen.
Dies sind also die Positionen der verschiedenen Komponenten:

- Komponente 1 kommt an der linken Seite an das Gewicht von Komponente 2.
- Komponente 2 kommt an der rechten Seite an das Gewicht von Komponente 1.
- Komponente 3 kommt an der linken Seite an das Gewicht von Komponente 4 (möglicherweise multipliziert mit 2 oder 3, wenn diese Position bereits durch Komponente 1 besetzt wird bzw. die Position von Komponente 4 bereits durch Komponente 2 besetzt wird).
- Komponente 4 kommt an der rechten Seite an das Gewicht von Komponente 3 (möglicherweise multipliziert mit 2 oder 3, wenn diese Position bereits durch Komponente 2 besetzt wird bzw. die Position von Komponente 3 bereits durch Komponente 1 besetzt wird).

Die letzte Möglichkeit tritt ein, wenn drei Komponenten an einen Balken gehängt werden sollen. Dabei soll folgende Gleichung gelten, damit der Balken ausbalanciert ist:
$p_1 \cdot w_1 + p_2 \cdot w_2 = -(p_3 \cdot w_3)$. Diese Gleichung zeigt, dass die ersten beiden Komponenten auf die linke Seite kommen, die dritte Komponente auf die rechte Seite. Dabei sind die Positionen wie folgt verteilt: $p_1 = w_3$, $p_2 = w_3 \cdot 2$ und $p_3 = -(w_1 + w_2 \cdot 2)$. Eingesetzt in die erste Gleichung:
$w_3 \cdot w_1 + w_3 \cdot 2 \cdot w_2 = -(-(w_1 + w_2 \cdot 2) \cdot w3)$, äquivalent umgeformt:
$w_3 \cdot w_1 + w_3 \cdot 2 \cdot w_2 = w_1 \cdot w_3 + w_2 \cdot 2 \cdot w_3$. Daraus ergeben sich folgende Positionen:

- Komponente 1 kommt an der linken Seite an das Gewicht von Komponente 3.
- Komponente 2 kommt an der linken Seite an das Gewicht von Komponente 3 multipliziert mit zwei.
- Komponente 3 kommt an der rechten Seite an das Gewicht von Komponente 1 plus das Gewicht von Komponente 2 multipliziert mit zwei.

Nun ist der Balken komplett ausbalanciert und funktionsfähig. Jedoch kann es vorkommen, dass die Positionen bei vielen Zahlen sehr hoch sind. Deshalb wird nun versucht, alle Positionen noch einmal zu „kürzen“.
Der euklidische Algorithmus wird angewendet, um den größten gemeinsamen Teiler der Positionen zu berechnen. Sollten mehr als zwei Komponenten am Balken hängen, wird der ggT zuerst für die ersten beiden Komponenten berechnet. Dann wird nacheinander für jede Position der ggT mit dem davor errechneten ggT und der jeweiligen Position erstellt. Schließlich werden alle Positionen durch den ggT geteilt.

Damit ist nun der komplette Algorithmus abgeschlossen. Er findet selbstverständlich für jeden Balken einzeln statt, bis alle Balken abgearbeitet sind. Sollte ein Mobile nur aus maximal vier Figuren bestehen, wird der Vorgang also nur einmal ausgeführt.

Umsetzung

Das Programm wurde objektorientiert in der Programmiersprache *Java* geschrieben. Ausgeführt werden kann das Programm in einer Konsole / einem Terminal mit dem Befehl „java -jar [Pfad zur .jar-Datei]“. Darin sind folgende Klassen enthalten:
- `Mobile` - Hauptklasse, liest Daten ein und startet die Generierung des Mobiles,
- `Component` - `interface`, stellt eine Komponente, die an einem Balken hängen kann, dar, dies kann entweder ein weiterer Balken oder eine Figur sein.
- `Figure` - Figur, implementiert `Component`, besteht lediglich aus einem Gewicht.
- `Bar` - Balken, implementiert `Component`, besteht aus maximal vier weiteren Komponenten und deren Positionen.

Nachdem die Hauptklasse `Mobile` die Gewichte eingelesen hat, erstellt sie daraus Figuren und übergibt diese einer neuen Instanz von `Bar`. Dieser Balken berechnet, ob er einen weiteren Balken erstellen muss oder ob alle Figuren an ihn „passen“. Dies können, wie schon unter dem Abschnitt „Lösungsidee“ beschrieben, maximal vier Figuren sein. Ist dies nicht der Fall, erstellt er einen weiteren Balken und übergibt ihm die überschüssigen Figuren. Auch dieser überprüft die Bedingungen usw. Durch diesen Ablauf kann an jedem Balken höchstens **ein** weiterer Balken hängen. Nun wird jeder Balken in der Methode `balance()` ausbalanciert. Abhängig von der Anzahl der am Balken hängenden Komponenten werden folgende Methoden aufgerufen:
- `balance(Map<Integer, Component> balanced, Component component1, Component component2)` wird bei zwei Komponenten 1 Mal (mit allen beiden Komponenten) bzw. vier Komponenten 2 Mal (einmal mit den ersten beiden Komponenten, einmal mit den letzten beiden Komponenten) aufgerufen
- `balance(Map<Integer, Component> balanced, Component component1, Component component2, Component component3)` wird bei drei Komponenten 1 Mal (mit allen drei Komponenten) aufgerufen

Diese Methoden balancieren die übergebenen Komponenten aus und fügen sie der `Map` hinzu, welche als erster Parameter übergeben wird. Die Schlüssel dieser `Map` entsprechen der jeweiligen Position, der Wert ist die Komponente, welche an diese Postion gehängt wird.

Zur Feststellung des Gewichts einer Komponente verfügt das `interface Component` über eine Methode `getWeight()`.

- Ist die Komponente eine Figur, wird lediglich die Eigenschaft `weight`, also das Gewicht der Figur zurückgegeben.
- Ist die Komponente ein Balken, wird das Gewicht aller Komponenten, welche an diesem Balken hängen, zusammengerechnet und zurückgegeben. Alle Komponenten eines Balkens sind in der `Map components` gespeichert (Eigenschaft der Klasse `Bar`).

Schließlich wird in der Methode `ggT()` die größte Zahl berechnet, durch die alle Positionen des Balkens teilbar sind und diese Division wird am Ende der ersten `balance()`-Methode noch ausgeführt. Damit ist die Generierung des Mobiles abgeschlossen.

Das Erstellen der textuellen Darstellung eines Mobiles wird über die Methode `toString()` erledigt. Eine Figur gibt einfach das eigene Gewicht als `String` zurück, ein Balken gibt `components.toString()` zurück. Dies ist die Methode `toString()` vom `interface Map`. Der Rückgabewert davon ist bereits so formatiert, wie wir es für unsere textuelle Darstellung benötigen.

Beispiele

1. Textuelle Darstellung eines Mobiles

Beispiel 1

```
{4={2=1, -1=2}, 1=2, -2=3, -4=2}
```

Dieses Beispiel zeigt die textuelle Darstellung des Mobiles in der Aufgabenstellung. Es besteht aus zwei Balken. Am ersten hängen drei Figuren mit den Gewichten 2, 3 und 2. Außerdem hängt an der Position vier ein weiterer Balken, an welchem wiederum zwei Figuren mit den Gewichten 1 und 2 hängen. Die Gleichung zur Überprüfung der Balanciertheit des kleineren Balkens lautet $1 \cdot 2 + 2 \cdot (-1) = 0$, für den größeren Balken ist dies $(1 + 2) \cdot 4 + 2 \cdot 1 + 3 \cdot (-2) + 2 \cdot (-4) = 0$.

2. Das Programm

Beispiel 1

```
Figuren durch Leerzeichen getrennt eingeben:
1 2 3
{3=1, -5=3, 6=2}
```

Bei dieser Ausführung des Programmes wurden die drei Gewichte 1, 2 und 3 eingegeben. Dazu wurde ein Mobile generiert, das aus nur einem Balken besteht.
$1 \cdot 3 + 3 \cdot (-5) + 2 \cdot 6 = 0$

Beispiel 2

```
Figuren durch Leerzeichen getrennt eingeben:
2 5 10 1 3
{-2=5, 4=10, 5=2, -10={1=3, -3=1}}
```

Hier wurde ein Mobile mit den Gewichten 2, 5, 10, 1 und 3 generiert. Es besteht aus einem zusätzlichen Balken, der am Größeren hängt, da nicht alle fünf Figuren an *einen* Balken gehängt werden können.
$3 \cdot 1 + 1 \cdot (-3) = 0$ (kleiner Balken)
$5 \cdot (-2) + 10 \cdot 4 + 2 \cdot 5 + (3 + 1) \cdot (-10) = 0$ (großer Balken)

Beispiel 3

```
Figuren durch Leerzeichen getrennt eingeben:
{-1=1000, 1=1000, -2=1000, 2=1000}
```

Dieses Beispiel zeigt, wie die Positionen, welche eigentlich 2000, 1000, -1000 und -2000 wären, am Ende stark „gekürzt“ wurden. Außerdem tritt der Sonderfall bei vier Figuren an einem Balken ein, bei dem die Positionen der beiden letzten Komponenten mit zwei multipliziert werden, da ansonsten Positionen mehrfach belegt werden würden.
$1000 \cdot (-1) + 1000 \cdot 1 + 1000 \cdot (-2) + 1000 \cdot 2 = 0$

Quellcode

`Component` — Stellt eine Komponente (Balken oder Figur) dar.

```java
public interface Component {

    public int getWeight();

}
```

`Figure` — Stellt eine Figur dar.

```java
public class Figure implements Component {

    private int weight;

    public Figure(int weight) {
        this.weight = weight;
    }

    @Override
    public int getWeight() {
        return weight;
    }

    @Override
    public String toString() {
        return Integer.toString(weight);
    }

}
```

`Bar` — Stellt einen Balken dar.

```
public class Bar implements Component {

    /**
     * Die Komponente, welche an diesem Balken hängen.
     * Schlüssel: die Position der Komponente, positiv oder negativ
     * Wert: Die Komponenten, welche an der Position (Schlüssel) hängt
     */
    private Map<Integer, Component> components;

    /**
     * Erstellt eine neue Instanz eines Balkens. Dafür wird aus den gegebenen
     * Figuren ein ausbalancierter Balken erstellt.
     *
     * @param figures die Figuren, welche an diesem Balken hängen sollen
     * @throws java.lang.IllegalArgumentException wenn weniger als 2 Komponenten
     *                                            übergeben wurden
     */
    public Bar(Figure[] figures) {
        if (figures.length < 2)
            throw new IllegalArgumentException("length of figures must be at " +
                    "least 2");
        components = generateBar(figures);
    }

    /**
     * Es werden alle Gewichte der Komponenten zusammengezählt und zurückgegeben.
     */
    @Override
    public int getWeight() {
        int weight = 0;

        for (Component component : components.values())
            weight += component.getWeight();

        return weight;
    }

    @Override
    public String toString() {
        return components.toString();
    }

    // ...

}
```

`Bar.generateBar()`

```
/**
 * Generiert einen balancierten Balken anhand der angegebenen Figuren.
 *
 * @param figures die Figuren, welche am Balken hängen sollen
 * @return eine {@link java.util.Map} mit den Positionen als Schlüssel und
 * den Komponenten als Werte
 */
private Map<Integer, Component> generateBar(Figure[] figures) {
    // Unbalancierte Komponente generieren. Wenn die Anzahl der Figuren
    // kleiner oder gleich 4 ist, koennen diese alle am Balken hängen.
    // Ansonsten koennen nur 3 Figuren am Balken hängen und die restlichen
    // werden an einen anderen Balken gehaengt, welcher dann die vierte
    // Komponente ist.
    List<Component> unbalanced = generateComponents(figures,
            figures.length <= 4 ? 4 : 3);

    return balance(unbalanced);
}
```

Bar.generateComponents()

```
/**
 * Generiert eine unbalancierte Liste mit allen Komponenten, welche am Balken
 * hängen sollen.
 *
 * @param figures    die dazu zu verwendenen Figuren
 * @param maxFigures die maximale Anzahl von Figuren, die am obersten Balken
 *                   hängen (sollten weitere Figuren übrig bleiben, werden
 *                   diese an einen weiteren Balken gehängt)
 * @return eine unbalancierte Liste mit den am Balken hängenden Komponenten
 */
private List<Component> generateComponents(Figure[] figures, int maxFigures) {
    if (figures.length <= maxFigures) // alle koennen an EINEN Balken
        return Arrays.asList((Component[]) figures);

    else { // ein weiterer Balken muss erstellt werden

        // components: fertige Komponente
        List<Component> components = new ArrayList<Component>();

        // newMobileFigures: Figuren, die an einen anderen Balken müssen
        Figure[] newMobileFigures = new Figure[figures.length - maxFigures];

        for (int i = 0; i < figures.length; i++)
            if (i < maxFigures) // kann einfach an den Balken
                components.add(figures[i]);
            else // muss an den anderen Balken
                newMobileFigures[i - maxFigures] = figures[i];

        // Den fertigen Komponenten wird der neue Balken mit allen
        // restlichen Figuren hinzugefuegt.
        components.add(new Bar(newMobileFigures));

        return components;
    }
}
```

Bar.balance(List<Component> unbalanced)

```
/**
 * Balanciert die Liste mit unbalancierten Komponenten aus.
 * Dabei wird unterschieden, ob es zwei, drei oder vier auszubalancierende
 * Komponenten gibt.
 *
 * @param unbalanced die unbalancierten Komponenten
 * @return die balancierten Komponenten
 */
private Map<Integer, Component> balance(List<Component> unbalanced) {
    Map<Integer, Component> balanced = new HashMap<Integer, Component>();

    switch (unbalanced.size()) {
        case 2:
            // lediglich beide Komponente zusammen ausbalancieren
            balance(balanced, unbalanced.get(0), unbalanced.get(1));
            break;
        case 3:
            // alle drei Komponente zusammen ausbalancieren
            balance(balanced, unbalanced.get(0), unbalanced.get(1),
                    unbalanced.get(2));
            break;
        case 4:
            // erst die beiden ersten Komponenten ausbalancieren,
            // danach die beiden anderen
            balance(balanced, unbalanced.get(0), unbalanced.get(1));
            balance(balanced, unbalanced.get(2), unbalanced.get(3));
            break;
    }

    // wenn möglich, werden alle Positionen durch den ggT geteilt, um das
    // Mobile kleiner zu machen
    int ggT = ggT(balanced);
    Map<Integer, Component> divided = new HashMap<Integer, Component>();
    for (Map.Entry<Integer, Component> entry : balanced.entrySet())
        divided.put(entry.getKey() / ggT, entry.getValue());

    return divided;
}
```

Bar.balance(Map<Integer, Component> balanced, Component component1, Component component2)

```
/**
 * Balanciert zwei Komponente aus und steckt diese in die Map balanced.
 *
 * @param balanced   die {@code Map}, in welche die balancierten Komponenten
 *                   kommen
 * @param component1 die erste Komponente
 * @param component2 die zweite Komponente
 */
private void balance(Map<Integer, Component> balanced, Component component1,
                     Component component2) {

    // Bei 4 Komponenten wird diese Methode 2 Mal aufgerufen. Daher können beim
    // zweiten Aufruf Positionen bereits belegt sein. Aus diesem Grund wird
    // hier zunächst ein Faktor generiert, mit dem die Positionen des zweiten
    // Aufrufs multipliziert werden, ohne die Positionen des ersten Aufrufs
    // erneut zu belegen. Dieser Faktor kann höchstens 3 sein, da nur zwei
    // Positionen vorher besetzt wurden. Im Normalfall ist er aber nicht
    // notwendig ist und daher gleich 1.
    int multiplication = 1;
    while (balanced.containsKey(component1.getWeight() * multiplication) ||
            balanced.containsKey(-component2.getWeight() * multiplication))
        multiplication++;

    // Komponente 2 kommt an die Stelle von Gewicht von Komponente 2
    balanced.put(component1.getWeight() * multiplication, component2);

    // Komponente 1 kommt an die Stelle von -Gewicht von Komponente 1
    balanced.put(-component2.getWeight() * multiplication, component1);
}
```

Bar.balance(Map<Integer, Component> balanced, Component component1, Component component2, Component component3)

```
/**
 * Balanciert drei Komponenten aus und steckt diese in die Map balanced.
 * Hier ist es nicht nötig, einen Faktor zu generieren,
 * mit dem die Positionen multipliziert werden, da diese Methode nur ein
 * Mal aufgerufen werden kann (s. {@link #balance(java.util.List)})
 *
 * @param balanced   die {@code Map}, in welche die balancierten Komponente kommen
 * @param component1 die erste Komponente
 * @param component2 die zweite Komponente
 * @param component3 die dritte Komponente
 */
private void balance(Map<Integer, Component> balanced, Component component1,
                     Component component2, Component component3) {

    // Komponente 1 kommt an die Stelle von Gewicht von Komponente 3
    balanced.put(component3.getWeight(), component1);

    // Komponente 2 kommt an die Stelle von Gewicht von Komponente 3 mal 2
    balanced.put(component3.getWeight() * 2, component2);

    // Komponente 3 kommt an die Stelle auf der anderen Seite des Balkens,
    // um das Gewicht auszugleichen
    balanced.put(-component1.getWeight() - component2.getWeight() * 2,
            component3);
}
```

`Bar.ggT(Map<Integer, Component> balanced)`
`Bar.ggT(int a, int b)`

```
/**
 * Berechnet die größte Zahl (ggT), durch die alle Positionen dividiert
 * werden können, sodass immer noch ein valides Mobile besteht.
 *
 * @param balanced die ausbalancierten Komponenten
 * @return der ggT, durch den alle Positionen geteilt werden können
 */
private int ggT(Map<Integer, Component> balanced) {
    Integer[] positions = balanced.keySet().toArray(new Integer[balanced.size()]);

    // ggT von allen Zahlen in positions berechnen
    int ggT = ggT(positions[0], positions[1]);
    if (positions.length > 2)
        ggT = ggT(ggT, positions[2]);
    if (positions.length > 3)
        ggT = ggT(ggT, positions[3]);

    return ggT;
}

/**
 * Berechnet den ggT der zwei gegeben Zahlen rekursiv anhand des "euklidischen
 * Algorithmus'".
 *
 * @param a Zahl 1
 * @param b Zahl 2
 * @return den ggT der zwei gegebenen Zahlen
 */
private int ggT(int a, int b) {
    if (b == 0)
        return a;
    else
        return ggT(b, a % b);
}
```

Code, der zum Starten des Programmes in der Klasse `Mobile` verwendet wird. Vorher wurden die Figuren eingelesen und in der Variable `figures` gespeichert.

```
System.out.println(new Bar(figures));
```