Tag 17 - Clumsy Crucible

Lösungshinweise Teil 1

- Die Aufgabe ist ein "Shortest Path Problem" ("Kürzester Pfad Problem") und lässt sich z. B. mit dem Dijkstra-Algorithmus lösen!
- Die Lösung ist allerdings trotzdem nicht trivial, denn es genügt nicht, die Ziffern in der 2D-Matrix als einzige Knoten aufzufassen. Die Einschränkung, dass Wege maximal 3 Schritte lang in einer Richtung verlaufen dürfen, zwingt einen dazu, jede Koordinate als insgesamt 12 mögliche Knoten aufzufassen. Beispiel zu einer beliebigen Koordinate mitten im Feld: Je nachdem, ob ich die Koordinate von links, rechts, oben oder unten betreten habe und jeweils zuvor 1, 2 oder 3 Schritte lang geradeaus lief, kann die Koordinate unterschiedliche Distanzen aufweisen.
- Ein gutes Beispiel zu diesem Tag, das man auf reddit finden kann, ist folgender kleiner Input:

```
112999
911111
```

Das korrekte Ergebnis muss 7 sein! Dazu muss dein Algorithmus in der Lage sein, auch den zunächst nachteilig erscheinenden Schritt horizontal in die "2" zu betrachten.

Der nachfolgende Lösungsvorschlag ist mangels Zeit leider weder besonders performant, noch besonders schön aufgeräumt oder kommentiert. Es gibt ein zweidimensionales Array, das **Koordinaten** speichert. **Koordinaten** sind eine eigene Klasse. Pro Koordinate werden 4 Arrays à 3 **Blöcken** gespeichert. Jedes Array ist für eine Richtung, in die man sich auf dem jeweiligen Block bewegt. Jeder **Block** ist ebenso eine eigene Klasse, die nochmals einige Informationen speichert und insbesondere die individuelle Distanz und den Hitzeverlust speichert.

Lösungsvorschlag Klasse Block

```
public class Block
{
    private int hitzeVerlust;
    private char richtung;
    private int distanz;
    private boolean besucht;
    private int x, y;

    /**
     * Konstruktor für Objekte der Klasse Block
     */
    public Block(int x, int y, int hitzeVerlust, char richtung)
    {
        this.hitzeVerlust = hitzeVerlust;
        this.richtung = richtung;
        this.distanz = Integer.MAX_VALUE;
```

```
        this.besucht = false;
        this.x = x;
        this.y = y;
    }

    public int getHitzeVerlust() {
        return this.hitzeVerlust;
    }

    public char getRichtung() {
        return this.richtung;
    }

    public void setDistanz(int d) {
        this.distanz = d;
    }

    public int getDistanz() {
        return this.distanz;
    }

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }

    public void setBesucht() {
        this.besucht = true;
    }

    public boolean istBesucht() {
        return this.besucht;
    }
}
```

Lösungsvorschlag Klasse Koordinaten

```
public class Koordinate
{
    // je nach Richtung, in die man sich bewegt
    // je drei Blöcke [0] für 1 Schritt, [1] für 2 Schritte, [2] für 3
Schritte in dieselbe Richtung
    private Block[] l = new Block[3];
    private Block[] r = new Block[3];
    private Block[] u = new Block[3];
```

```
    private Block[] d = new Block[3];

    /**
     * Konstruktor für Objekte der Klasse Koordinate
     */
    public Koordinate(int x, int y, int hitzeverlust)
    {
        for (int i = 0; i < 3; i++) {
            l[i] = new Block(x, y, hitzeverlust, 'l');
            r[i] = new Block(x, y, hitzeverlust, 'r');
            u[i] = new Block(x, y, hitzeverlust, 'u');
            d[i] = new Block(x, y, hitzeverlust, 'd');
        }
    }

    public Block getBlock(char richtung, int schritte) {
        if (richtung == 'l') {
            return l[schritte-1];
        } else if (richtung == 'r') {
            return r[schritte-1];
        } else if (richtung == 'u') {
            return u[schritte-1];
        } else {
            return d[schritte-1];
        }
    }

    public Block smallestBlock() {
        Block b = new Block(0, 0, 0, 'd');
        b.setDistanz(Integer.MAX_VALUE);
        for (int i = 0; i < 3; i++) {
            if (r[i].getDistanz() < b.getDistanz() && !r[i].istBesucht()) {
                b = r[i];
            }
            if (l[i].getDistanz() < b.getDistanz() && !l[i].istBesucht()) {
                b = l[i];
            }
            if (u[i].getDistanz() < b.getDistanz() && !u[i].istBesucht()) {
                b = u[i];
            }
            if (d[i].getDistanz() < b.getDistanz() && !d[i].istBesucht()) {
                b = d[i];
            }
        }
        return b;
    }
}
```

Lösungsvorschlag Teil 2

```
public int partOne() {
    breite = inputLines.get(0).length();
    hoehe = inputLines.size();

    map1 = new Koordinate[breite][hoehe];
    // übertrage eingabedaten in Koordinaten-Blöcke
    for (int y = 0; y < hoehe; y++) {
        String line = inputLines.get(y);
        for (int x = 0; x < breite; x++) {
            map1[x][y] = new Koordinate(x, y,
Character.getNumericValue(line.charAt(x)));
        }
    }

    int x = 0;
    int y = 0;

    // setze zu Beginn alle ersten möglichen Distanzen nach rechts und nach
unten
    int kleinsteDistanz = Integer.MAX_VALUE;
    Block b;
    Block nextBlock = new Block(0, 0, 0, 'r'); // dummy Initialisierung
    int vorherigerHitzeVerlust = 0;
    for (int sx = 1; sx <= 3; sx++) {
        b = map1[sx][0].getBlock('r', sx);
        b.setDistanz(vorherigerHitzeVerlust + b.getHitzeVerlust());
        vorherigerHitzeVerlust += b.getHitzeVerlust();

        if (b.getDistanz() < kleinsteDistanz) {
            kleinsteDistanz = b.getDistanz();
            x = sx;
            y = 0;
            nextBlock = map1[x][y].getBlock('r', sx);
        }
    }

    vorherigerHitzeVerlust = 0;
    for (int sy = 1; sy <= 3; sy++) {
        b = map1[0][sy].getBlock('d', sy);
        b.setDistanz(vorherigerHitzeVerlust + b.getHitzeVerlust());
        vorherigerHitzeVerlust += b.getHitzeVerlust();

        if (b.getDistanz() < kleinsteDistanz) {
            kleinsteDistanz = b.getDistanz();
            x = 0;
            y = sy;
            nextBlock = map1[x][y].getBlock('d', sy);
        }
    }
```

```
    while (!alleLetzteBlocksBesucht()) {
        nextBlock = getKleinsteDistanz();

        // setze aktuellen Block auf "besucht"
        nextBlock.setBesucht();

        // aktualisiere Distanz zu allen möglichen 3 Nachbarn "links" und
"rechts"
        // die noch NICHT besucht wurden
        aktualisiereNachbarn(nextBlock.getX(), nextBlock.getY(), nextBlock);
    }

    int distanzZumZiel = Integer.MAX_VALUE;
    for (int i = 1; i <= 3; i++) {
        if (map1[breite-1][hoehe-1].getBlock('r', i).getDistanz() <
distanzZumZiel) {
            distanzZumZiel = map1[breite-1][hoehe-1].getBlock('r',
i).getDistanz();
        }
        if (map1[breite-1][hoehe-1].getBlock('d', i).getDistanz() <
distanzZumZiel) {
            distanzZumZiel = map1[breite-1][hoehe-1].getBlock('d',
i).getDistanz();
        }
    }
    return distanzZumZiel;
}

/**
 * Gibt true zurück, wenn die letzte koordinate (rechts unten) in allen r
und d Blöcken besucht wurde
 */
private boolean alleLetzteBlocksBesucht() {
    for (int i = 1; i <= 3; i++) {
        if (!map1[breite-1][hoehe-1].getBlock('r', i).istBesucht() ||
!map1[breite-1][hoehe-1].getBlock('d', i).istBesucht()) {
            return false;
        }
    }
    return true;
}

private void aktualisiereNachbarn(int x, int y, Block b) {
    int puffer = b.getDistanz();
    if (b.getRichtung() == 'u' || b.getRichtung() == 'd') { // gehe nach
rechts & links
        for (int i = 1; i <= 3 && x+i < breite; i++) {
            Block neighbor = map1[x+i][y].getBlock('r', i);
            if (!neighbor.istBesucht() && puffer +
neighbor.getHitzeVerlust() < neighbor.getDistanz()) {
```

```
                neighbor.setDistanz(puffer + neighbor.getHitzeVerlust());
            }
            puffer = neighbor.getDistanz();
        }
        puffer = b.getDistanz();
        for (int i = 1; i <= 3 && x-i >= 0; i++) {
            Block neighbor = map1[x-i][y].getBlock('l', i);
            if (!neighbor.istBesucht() && puffer +
neighbor.getHitzeVerlust() < neighbor.getDistanz()) {
                neighbor.setDistanz(puffer + neighbor.getHitzeVerlust());
            }
            puffer = neighbor.getDistanz();
        }
    } else { // nach oben & unten
        puffer = b.getDistanz();
        for (int i = 1; i <= 3 && y+i < hoehe; i++) {
            Block neighbor = map1[x][y+i].getBlock('d', i);
            if (!neighbor.istBesucht() && puffer +
neighbor.getHitzeVerlust() < neighbor.getDistanz()) {
                neighbor.setDistanz(puffer + neighbor.getHitzeVerlust());
            }
            puffer = neighbor.getDistanz();
        }
        puffer = b.getDistanz();
        for (int i = 1; i <= 3 && y-i >= 0; i++) {
            Block neighbor = map1[x][y-i].getBlock('u', i);
            if (!neighbor.istBesucht() && puffer +
neighbor.getHitzeVerlust() < neighbor.getDistanz()) {
                neighbor.setDistanz(puffer + neighbor.getHitzeVerlust());
            }
            puffer = neighbor.getDistanz();
        }
    }
}

/**
 * Finde die Koordinaten des Knotens der noch nicht besucht wurde und die
kleinste Distanz hat
 */
private Block getKleinsteDistanz() {
    Block smallestBlock = new Block(0, 0, 0, 'd');
    smallestBlock.setDistanz(Integer.MAX_VALUE);

    for (int y = 0; y < hoehe; y++) {
        for (int x = 0; x < breite; x++) {

            Block b = map1[x][y].smallestBlock();
            if (b.getDistanz() < smallestBlock.getDistanz()) {
                smallestBlock = b;
```

```
                }
            }
        }
        return smallestBlock;
}
```

Lösungshinweise Teil 2

Für Teil 2 müssen die Wege-Einschränkungen aus Teil 1 nur minimal überarbeitet werden: jeder Abschnitt muss genau zwischen 4-10 Schritte lang in einer Richtung verlaufen.