Tag 18 - Lavaduct Lagoon

- Variante 1

Ich habe für Teil 1 und Teil 2 verschiedene Lösungswege genutzt.

Lösungshinweise Teil 1

- Finde zunächst heraus, wie groß der auszugrabende Bereich in die Breite und die Höhe sein muss.
- Erstelle ein char[][]-Array mit der passenden Größe.
- Trage den Input-Befehlen folgend ein 't' für "Trench" (Graben) in das char[][]-Array an jeder nötigen Stelle ein. Dadurch erhältst du den kompletten geschlossenen Verlauf des äußeren Grabens.
- Abschließend füllst du von jedem äußeren Pixel an den Kanten des char[][]-Arrays per Breitensuche die leeren Felder mit einem 'o' für "outside". Danach ist jedes leere Feld ein inneres Feld. Die Summe aller Lava-Pool-Felder ist die summe aller leeren Felder und aller 't'-Felder.

Lösungsvorschlag

```
public int partOne() {
    trenchList = new ArrayList();

    int offsetX = 0;
    int offsetY = 0;
    int largestX = 0;
    int largestY = 0;

    int lastX = 0;
    int lastY = 0;
    trenchList.add(new int[]{0,0});

    // speichere den offset für x und y
    for (String line: inputLines) {
        char dir = line.split(" ")[0].charAt(0);
        int steps = Integer.parseInt(line.split(" ")[1]);
        String color = line.split("[(]")[1].split("[)]")[0];

        while (steps > 0) {
            if (dir == 'R') {
                trenchList.add(new int[]{lastX + 1, lastY});
                lastX++;
            } else if (dir == 'L') {
                trenchList.add(new int[]{lastX - 1, lastY});
                lastX--;
            } else if (dir == 'U') {
                trenchList.add(new int[]{lastX, lastY - 1});
                lastY--;
```

```
            } else {
                trenchList.add(new int[]{lastX, lastY + 1});
                lastY++;
            }
            steps--;

            if (lastX < offsetX) {
                offsetX = lastX;
            }
            if (lastY < offsetY) {
                offsetY = lastY;
            }
            if (lastX > largestX) {
                largestX = lastX;
            }
            if (lastY > largestY) {
                largestY = lastY;
            }
        }
    }

    breite = largestX - offsetX + 1;
    hoehe = largestY - offsetY + 1;
    map = new char[breite][hoehe];

    // trage für die trench ein 't' in die map ein
    for (int[] k: trenchList) {
        map[k[0]-offsetX][k[1]-offsetY] = 't';
    }

    for (int x = 0; x < breite; x++) {
        if (map[x][0] == '\u0000') {
            fillQueue.push(new int[]{x, 0});
            fillOutside();
        }
        if (map[x][hoehe-1] == '\u0000') {
            fillQueue.push(new int[]{x, hoehe-1});
            fillOutside();
        }
    }
    for (int y = 0; y < hoehe; y++) {
        if (map[0][y] == '\u0000') {
            fillQueue.push(new int[]{0, y});
            fillOutside();
        }
        if (map[breite-1][y] == '\u0000') {
            fillQueue.push(new int[]{breite-1, y});
            fillOutside();
        }
    }
```

```
    int insideCounter = 0;
    for (int x = 0; x < breite; x++) {
        for (int y = 0; y < hoehe; y++) {
            if (map[x][y] != 'o') {
                insideCounter++;
            }
        }
    }

    return insideCounter;
}

private void fillOutside() {
    while (!fillQueue.isEmpty()) {
        int[] k = fillQueue.pop();
        int x = k[0];
        int y = k[1];

        if (x < 0 || y < 0 || x == breite || y == hoehe || map[x][y] !=
'\u0000') {
            continue;
        } else {
            map[x][y] = 'o';
            fillQueue.push(new int[]{x + 1, y});
            fillQueue.push(new int[]{x - 1, y});
            fillQueue.push(new int[]{x, y + 1});
            fillQueue.push(new int[]{x, y - 1});
        }
    }
}
```

Lösungshinweise Teil 2

- Für Teil 2 werden die Zahlen zu groß, sodass die Lösung aus Teil 1 nicht mehr ausreicht. Die Hauptvorgehensweise ist folgende:

- Gehe der Reihe nach die Befehle entlang. Trage aber keine Kreuze in eine 2-dimensionale Matrix, sondern betrachte nur die einzelnen Spalten (x-Werte). Führe eine ArrayList, um pro Spalte mehrere y-Werte eintragen zu können, an denen "etwas passiert".:
  - Wenn du gerade nach **rechts** läufst, dann markierst du, dass du mit dem aktuellen y-Wert das **obere Ende** des Lava-Pools.
  - Beim nach **links** laufen markierst du mit dem y-Wert das **untere Ende** des Lava-Pools.
  - Wenn du nach **unten** läufst, dann ist dein oberer Startpunkt, das obere Ende des Lava-Pools. Als weiteren y-Wert trägst du ein, wie viele y-Werte du weiter nach unten zu gehen hast (das ist dann ein mögliches unteres Ende des Pools).
  - Beim nach **oben** laufen ist der untere Startpunkt ein unteres Ende des Pools, und beim "Ziel" des nach-oben-laufens ist das obere Ende des Pools.
- Beispiel für das rechte Bild (Annahme, dass die Erstellung des Pfades im Uhrzeigersinn geschah): in der Spalte für x=5 stehen folgende y-Werte drin:

```
0 (wegen "rechts")
0 (wegen oberes Ende von "oben")
1 (wegen "rechts")
1 (wegen unteres Ende von "oben")
3 (wegen "links")
5 (wegen "rechts")
5 (wegen oberes Ende von "unten")
7 (wegen unteres Ende von "unten")
7 (wegen "links")
```

Nun muss man zusätzlich noch unterscheiden, ob die Eintragung jeweils das obere oder untere Ende vom Pool markiert. Platzsparend kann man dies z. B. tun, indem man die Zahlen als String abspeichert und hinten eine 0 für oberes Ende und 1 für unteres Ende anhängt. Damit steht eigentlich in der Spalte:

```
00
00
10
11
31
50
50
71
71
```

Hat man diese Werte, kann man schnell die Differenzen ausrechnen und damit die Fläche über alle Spalten aufsummieren. Wichtig ist nun, die Werte noch zu bereinigen: y=1 hat die Endung 0 und 1 (kommt doppelt vor). Wir können daher y=1 gänzlich aus der Liste streichen. Ebenso können wir alle anderen Dopplungen streichen und müssen die Einträge sortieren (der Pfad kann "chaotisch" verlaufen und die Werte müssen nicht bereits automatisch aufsteigend sortiert sein. Am Ende bleibt übrig:

```
00
31
50
71
```

Die Fläche in der einen Spalte ist also: `( (3-1) + 1) + ( (7-5) + 1)`

Lösungsvorschlag Teil 2

```java
public long partTwo() {
    ArrayList<String> anweisungen = new ArrayList();
    spalten = new ArrayList();
    int breite = 0;

    for (String line: inputLines) {
        String hex = line.split("#")[1].split("[)]")[0];
        int steps = Integer.parseInt(hex.substring(0, 5), 16);
        // zähle alle schritte nach rechts
        if (hex.substring(5).equals("0")) {
            breite += steps;
        }
        anweisungen.add(steps + hex.substring(5));
    }

    ArrayList<ArrayList<String>> spalten = new ArrayList();
    for (int i = 0; i < breite; i++) {
        spalten.add(new ArrayList<String>());
    }

    int x = 0;
    int y = 0;
    for (String anweisung: anweisungen) {
        // rechts
```

```
        if (anweisung.charAt(anweisung.length()-1) == '0') {
            // 0 am Ende bedeutet: ein oberes Ende vom Pool innerhalb der
Spalte
            spalten.get(x).add(y + "" + 0);
            int steps = Integer.parseInt(anweisung.substring(0,
anweisung.length()-1));
            for (int i = 0; i < steps; i++) {
                x = (x+1) % breite;
                spalten.get(x).add(y + "" + 0);
            }
        }
        // unten
        else if (anweisung.charAt(anweisung.length()-1) == '1') {
            spalten.get(x).add(y + "" + 0);
            int steps = Integer.parseInt(anweisung.substring(0,
anweisung.length()-1));
            y += steps;
            spalten.get(x).add(y + "" + 1);
        }
        // links
        else if (anweisung.charAt(anweisung.length()-1) == '2') {
            // 1 am Ende bedeutet: ein unteres Ende vom Pool innerhalb der
Spalte

            spalten.get(x).add(y + "" + 1);
            int steps = Integer.parseInt(anweisung.substring(0,
anweisung.length()-1));
            for (int i = 0; i < steps; i++) {
                x--;
                if (x < 0) {
                    x = breite - 1;
                }
                spalten.get(x).add(y + "" + 1);
            }
        }
        // oben
        else if (anweisung.charAt(anweisung.length()-1) == '3') {
            spalten.get(x).add(y + "" + 1);
            int steps = Integer.parseInt(anweisung.substring(0,
anweisung.length()-1));
            y -= steps;
            spalten.get(x).add(y + "" + 0);
        }
    }

    long summe = 0;
    // Gehe über jede Spalte
    for (ArrayList<String> spalte: spalten) {

        // entferne alle vollständigen dopplungen (wandelt es in ein
```

```
"sortiertes" Set um (alles kann dort nur einmalig vorkommen))
        Set<String> set = new LinkedHashSet<>(spalte);
        spalte.clear();
        spalte.addAll(set);

        // sortiere
        spalte = sortArrayList(spalte);

        // prüfe nun, ob direkte Nachfolger in allen Zeichen exklusive dem
letzten übereinstimmen -> y-koordinate gleichzeitig oberes und unteres Ende
-> kann entfernt werden.
        for (int i = 0; i < spalte.size() - 1; i++) {
            if (spalte.get(i).substring(0,
spalte.get(i).length()-1).equals(spalte.get(i+1).substring(0,spalte.get(i+1)
.length()-1))) {
                spalte.remove(i);
                spalte.remove(i); //zweimal i, da durch das erste Löschen
die hinteren Elemente 1 nach vorne rücken
                i--;
            }
        }

        for (int i = 0; i < spalte.size(); i = i+2) {
            summe += Integer.parseInt(spalte.get(i+1).substring(0,
spalte.get(i+1).length()-1)) - Integer.parseInt(spalte.get(i).substring(0,
spalte.get(i).length()-1)) + 1;
        }
    }

    return summe;
}

/**
 * Selection-Sort, um die Werte pro Spalte nach Zahlenwert zu sortieren.
 */
private ArrayList<String> sortArrayList(ArrayList<String> input) {
    for (int i = 0; i < input.size(); i++) {
        int min = i;
        for (int j = i; j < input.size(); j++) {
            if (Integer.parseInt(input.get(j).substring(0,
input.get(j).length()-1)) < Integer.parseInt(input.get(min).substring(0,
input.get(min).length()-1))) {
                min = j;
            }
        }
        String temp = input.get(min);
        input.set(min, input.get(i));
        input.set(i, temp);
    }

    return input;
```

```
}
```