Tag 12 - Hot Springs

Lösungshinweise Teil 1

Teil 1 ist (noch) über verschiedene Wege lösbar. Prinzipiell die einfachste (aber nicht die schnellste) Methode ist, sich rekursiv alle möglichen Eingabestrings zu erstellen.
Sobald der String erstellt ist, ist der Basisfall der Rekursion erreicht und man kann mithilfe eines regulären Ausdrucks prüfen, ob der String den angeforderten Gruppengrößen entspricht. Z. B. `^\\.*\\#{3}\\.+\\#{2}\\.*$` würde prüfen, ob zuerst eine Dreiergruppe an Rauten und dann eine Zweiergruppe an Rauten im String vorkommt.

Alternativ lässt man die Überprüfung mit regulären Ausdrücken weg (diese sind nämlich langsam) und überprüft Buchstaben für Buchstaben, ob dies den angeforderten Gruppen-Größen entspricht.

Lösungshinweis für reguläre Ausdrücke

```
public int partOne() {
    int summe = 0;

    for (String line: inputLines) {
        int[] numbers = strToIntArray(line.split(" ")[1]);

        // Baue einen korrekten regulären Ausdruck (regex)
        String regex = "^\\.*";
        for (int n: numbers) {
            regex += "\\#{" + n + "}\\.+";
        }
        regex = regex.substring(0, regex.length()-1);
        regex += "*$";

        correctArrangementsPerLine = 0;
        // Baue nacheinander alle denkbaren arrangements auf.
        buildArrangements(line.split(" ")[0], "", regex);
        summe += correctArrangementsPerLine;
    }

    return summe;
}

private void buildArrangements(String input, String current, String regex) {
    if (current.length() == input.length()) {
        // Prüfe, ob dieser String erlaubt ist.

        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(current);
        if (matcher.find()) {
```

```
            correctArrangementsPerLine++;
        }

    } else if (input.charAt(current.length()) != '?') {
        buildArrangements(input, current + input.charAt(current.length()),
regex);
    } else {
        buildArrangements(input, current + '.', regex);
        buildArrangements(input, current + '#', regex);
    }
}
```

Teil 2

Teil 2 ist definitiv nicht trivial. Beide einfachen Ansätze aus Teil 1 genügen hier nicht mehr, da mehrere Milliarden Kombinationen durchprobiert werden müssten! Man wird sich wohl z. B. die "dynamische Programmierung" zunutze machen müssen. Damit speichert man für jeden rekursiven Methodenaufruf dessen Ergebnis, bevor es als `return` zurückgegeben wird. Dann kann man zu Beginn jedes Aufrufs dieser Methode bei gegebenen Parametern überprüfen, ob diese Parameter schon einmal aufgerufen wurden und man dazu direkt das Ergebnis liefern kann. *(Mangels Zeit konnte ich leider keine Lösung finden, die mit der dynamischen Programmierung funktioniert.)*