# Tag 12 - Hot Springs

- Variante 1

## 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

*Mit 3 Tagen Verspätung nun auch endlich eine ausformulierte Lösung für Teil 2... (danke an Frank Schiebel für den Impuls )*
Teil 2 ist definitiv nicht trivial. Beide einfachen Ansätze aus Teil 1 genügen hier nicht mehr, da (je nach Eingabedatei) etwa 1,6 Billionen (!) Kombinationen durchprobiert werden müssten! Man muss sich daher dringend die "dynamische Programmierung" zunutze machen. 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.

- Rufe pro Eingabezeile eine rekursive Methode `getMatches()` auf, die zwei Parameter entgegennimmt. Erstens die 5-fach hintereinander gehängten linken Strings. Zweitens die 5-fach hintereinander gehängten Zahlen als int-Array.
- Die Methode `getMatches()` soll nun für den jeweils übergebenen (noch vorhandenen) Eingabestring und die verbleibenden Zahlen zurückgeben, wie viele Kombinationen möglich sind. Die Grundidee ist, den Eingabestring vorne immer weiter zu kürzen (ebenso die Zahlen) und rekursiv dieselbe Methode wieder mit verkürztem String aufzurufen. Dabei wird immer nur das vorderste Zeichen überprüft und beim Vorkommnis eines ? kann es passieren, dass die Methode sich rekursiv aufsplittet in zwei Methoden - dadurch kommen die gigantisch großen Kombinationsmöglichkeiten zustande. Wie funktioniert die Methode nun im Detail?
    - Der ganz entscheidende Punkt ist es, direkt zu Beginn innerhalb der Methode zu prüfen, ob die rekursive Methode bereits mit denselben Parametern aufgerufen wurde. Man speichert sich für jeden Parameter unmittelbar vor dem `return` den Rückgabewert und kann damit später direkt zum Einstieg in die Methode prüfen, ob nun wieder rekursiv zig Unter-Varianten geprüft werden müssen, oder ob direkt das bereits zuvor berechnete Ergebnis zurückgegeben werden kann. Dies nennt sich dynamische Programmierung, ein unheimlich mächtiges Programmierverfahren.
    - Eine gute Datenstruktur dazu ist die `HashMap` in Java, welche über `import java.util.HashMap;` eingebunden werden muss. Dies ist eine **key-value-Datenstruktur**: Das bedeutet, dass man über einen (Such-)Key einen (Daten-)Wert finden kann. Prinzipiell ist es ein wenig mit einem Array bzw. einer ArrayList vergleichbar: dort findet man "für" einen Index einen Wert. Allerdings kann der Key (≈ Index) bei HashMaps gänzlich flexibel sein und muss nicht den aufsteigenden Indizes genügen. Vielmehr bastelt man sich direkt zu Beginn des Methodenaufrufs den Key direkt als String aus den beiden Methodenparametern zusammen.
    - Wurde der Key bisher noch nicht in die HashMap eingetragen, dann muss man sich zuerst

um die Abbruchbedingungen kümmern. Davon gibt es mehrere, u. a.:
    - Genügt die Anzahl an String-Buchstaben um alle Zahlen-Vorkommnisse im besten Falle unterzubringen?
    - Ist der String empty?
    - Gibt es keine Zahlen mehr, sodass keine "Matches" mehr möglich sind ?
  - Anschließend muss in Abhängigkeit des ersten Characters vom String vielfach unterschieden werden. Je nachdem, ob der erste Buchstabe ein `.`, ein ? oder ein # ist, muss anders verfahren werden:
    - Bei einem `.` kann einfach mit dem restlichen Substring exklusive des ersten Zeichens die Methode wieder aufgerufen werden.
    - Bei einem # müssen zahlreiche Fälle unterschieden werden (teste es mit Stift und Papier, welche Fälle alle auftreten können, auch bei verschiedenen Zahlen-Matches).
    - Bei einem ? musst du die Rekursion aufsplitten in die Fälle, dass das ? durch ein `.` oder durch ein # ersetzt wird.
  - Immer bei einer Abbruchbedingung gibt es entweder den Fall, dass dies zu einem oder zu keinem Match führt. In der Summe führt das insgesamt zu Billionen (!) Fällen, die dank der dynamischen Programmierung in einigen Millisekunden (!) berechnet werden können.

From:
https://info-bw.de/ -

Permanent link:
**https://info-bw.de/faecher:informatik:oberstufe:java:aoc:aco2023:day12:start?rev=1702665683**

Last update: **15.12.2023 18:41**