Tag 21 - Step Counter

- Variante 1

Teil 1 und 2 unterscheiden sich in ihrer Vorgehensweise erheblich, auch wenn man in Teil 2 Methoden aus Teil 1 benötigt.

Lösungshinweise Teil 1

- Dieser Teil ist eine klassische Breitensuche. Lagere die eigentliche Breitensuche in eine eigene Methode aus (du wirst sie häufiger brauchen). Sorge dafür, dass du drei Listen hast, in denen du die aktuellen Koordinaten, die nächsten zu besuchenden Koordinaten und die zuletzt besuchten Koordinaten speichern kannst.
- Lies die Eingabe in ein char[][]-Array ein und ersetze dabei das S durch einen `.`. Füge die Koordinate von S direkt der Liste der aktuellen Knoten hinzu. Du kannst die Koordinaten in den Listen z. B. in Form von int[]-Arrays speichern (Index 0 ist x, Index 1 ist y).
- Führe nun die Breitensuche in der Methode aus. Wichtig zur Beschleunigung des Algorithmus ist folgende Erkenntnis: Sobald ein Feld erstmalig betreten wurde, wird es danach mit jedem weiteren Schritt wieder abwechselnd verlassen werden und betreten werden. Wir müssen daher jedes Feld nur einmalig betrachten, dafür aber die Anzahl der betretenen Felder bei gerader und ungerader Schrittzahl unterscheiden. Je nach Iterationsschritt werden die neuen Felder also der einen oder anderen Variablen zugefügt.
- Prüfe für jede aktuelle Koordinate, ob alle 4 möglichen Nachbarkoordinaten:
  - eine erlaubte ("legale") Koordinate innerhalb des Spielfeldes ist
  - kein Stein ist
  - die Koordinate bereits besucht wurde → Entweder in den letzten Koordinaten vorkommt ODER bereits in den nächsten Koordinaten vermerkt ist.
- Erhöhe dann die Anzahl der besuchten Felder (= mögliche Schritte) um die Menge der nächsten Koordinaten.
- Aus den aktuellen Koordinaten werden die alten, aus den nächsten die aktuellen.
- Wiederhole dann die letzten Schritte, solange es noch aktuelle Koordinaten gibt.

Lösungsvorschlag Teil 1

```
private int breite = 0;
private int hoehe = 0;

private char[][] map;
private ArrayList<int[]> nextTiles;
private ArrayList<int[]> currentTiles;
private ArrayList<int[]> oldTiles;

private long bfs(int iterations) {
    long stepsGerade = 1;
    long stepsUngerade = 0;

    for (int i = 1; i <= iterations; i++) {
        for(int[] k: currentTiles) {
            // füge die Nachbarn zu den nextTiles hinzu
```

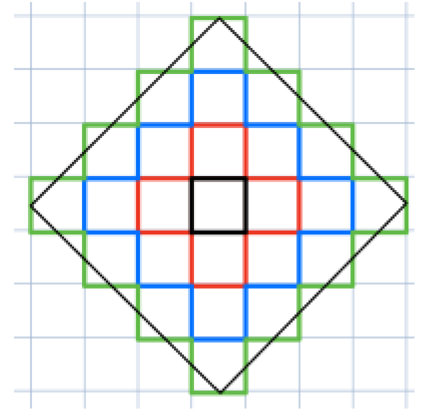
```
            int x = k[0]-1;
            int y = k[1];
            if (x >= 0 && map[x][y] != '#' && !tileVisited(x, y)) {
                nextTiles.add(new int[]{x, y});
            }
            x = k[0]+1;
            if (x < breite && map[x][y] != '#' && !tileVisited(x, y)) {
                nextTiles.add(new int[]{x, y});
            }
            x = k[0];
            y = k[1]-1;
            if (y >= 0 && map[x][y] != '#' && !tileVisited(x, y)) {
                nextTiles.add(new int[]{x, y});
            }
            y = k[1]+1;
            if (y < hoehe && map[x][y] != '#' && !tileVisited(x, y)) {
                nextTiles.add(new int[]{x, y});
            }
        }

        if (i % 2 == 0) {
            stepsGerade += nextTiles.size();
        } else {
            stepsUngerade += nextTiles.size();
        }

        oldTiles = currentTiles;
        currentTiles = nextTiles;
        nextTiles = new ArrayList<int[]>();
    }
    return (iterations % 2 == 0) ? stepsGerade : stepsUngerade;
}

private void resetLists() {
    nextTiles = new ArrayList<int[]>();
    currentTiles = new ArrayList<int[]>();
    oldTiles = new ArrayList<int[]>();
}

public long partOne() {
    breite = inputLines.get(0).length();
    hoehe = inputLines.size();
    map = new char[breite][hoehe];

    resetLists();

    for (int y = 0; y < inputLines.size(); y++) {
        String line = inputLines.get(y);
        for (int x = 0; x < line.length(); x++) {
```

```
            char c = line.charAt(x);
            if (c == 'S') {
                currentTiles.add(new int[]{x, y});
                c = '.';
            }
            map[x][y] = c;
        }
    }

    return bfs(64);
}

private boolean tileVisited(int x, int y) {
    int[] k = new int[]{x, y};
    // die Koordinate darf weder bereits besucht worden sein...
    for (int[] o: oldTiles) {
        if (Arrays.equals(k, o)) {
            return true;
        }
    }
    // noch bereits vermerkt sein besucht zu werden
    for (int[] n: nextTiles) {
        if (Arrays.equals(k, n)) {
            return true;
        }
    }
    return false;
}
```

Lösungshinweise Teil 2

Vorwort: Möglicherweise kann man die hier dargestellte Lösung nochmals erheblich weiter optimieren. Auch bereits so ist sie aber äußerst performant.
Wenn man Teil 2 wie Teil 1 direkt berechnen will, dann wird man sehr lange auf die Lösung warten! Man muss dringend eine optimierte Lösung suchen. Diese findet man, wenn man sich die Eingabedatei genau anschaut und zunächst einigen Grips OHNE einen PC investiert.

- Das S ist exakt zentral auf der Karte platziert und **dieselbe Spalte und Zeile enthält keine #** (keinen Stein)! Das bedeutet, dass man sich horizontal, wie auch vertikal bei jedem Schritt **garantiert** um eine x-/y-Koordinate vom Zentrum entfernen kann.
- Außerdem findet man heraus, dass die Breite wie Höhe 131 beträgt, sowie das S selbst bei (65|65) (0-Index-Zählung) liegt. Damit kann man Vermutungen anstellen, wie weit man z. B. bei horizontaler Bewegung maximal kommen kann: `65 + 131 * n = 26501365`. Wobei n ein ganzes Spielfeld bedeutet. Umgestellt nach n findet man tatsächlich das perfekte Ergebnis (ohne Nachkommazahlen) `n=202300`. Das bedeutet, dass man sich zumindest in horizontaler und vertikaler Richtung um ganze 202300 Spielfelder weit (exklusive des zentralen Feldes) bewegen kann.



- Betrachtet man dies nun zeichnerisch, kann man vereinfacht betrachten, wie weit man kommt (siehe rechtes Bild).
  - Für n=0 kommt man nur bis zum Rand des schwarzen Feldes → insg. 1.
  - Für n=1 kommt man bis zum Rand der roten Felder → insg. 5.
  - Für n=2 kommt man bis zum Rand der blauen Felder → insg. 13.
  - Für n=3 kommt man bis zum Rand der grünen Felder → insg. 25.
- Unter der Annahme, dass n=3 unser zuvor gefundenes Maximum sei, stellt man allerdings fest, dass keines der grünen Felder mehr vollständig durchlaufen werden kann. Vielmehr kommt man überall nur in etwa so weit, wie man am schwarzen Strich erkennen kann.
- Zunächst brauchen wir nun eine Systematik, wie viele Spielfelder für jedes n betreten werden können. Mithilfe der eben bereits gefundenen Werte kann man eine quadratische Funktion aufstellen. Für den hier dargestellten Lösungsweg braucht man dies aber nicht einmal. Vielmehr ist es wichtig die Systematik zu erkennen, dass von einem zum nächsten n insgesamt n*4 Felder hinzukommen.
- Tatsächlich wichtig ist zuletzt noch folgende Erkenntnis: Da die Felder eine ungerade Größe von 131 haben, werden sich benachbarte Felder immer genau unterscheiden, welche Koordinaten (nicht) betreten sind. Im einen Feld ist bei derselben Schritt-Iteration die mittlere Koordinate betreten, aber bei allen direkten Nachbarfeldern nicht. Man kann dies als *gerade* und *ungerade* Felder unterscheiden. Dieses reddit-Bild zeigt das ganz gut.
- Berechne zuerst wie viele Koordinaten betreten sind im geraden und ungeraden Fall.
- Berechne dann, wie viele Felder gerade und ungerade Felder sind.
- Daraus kannst du schon mal für alle inneren/vollständig-betretenen Felder die Anzahl aller betretenen Koordinaten berechnen.
- Nun musst du noch jeweils Breitensuchen starten für die 4 äußeren "Diamant-Spitzen-Felder". Ebenso für die 4 möglichen "1/8"-Felder und für die 4 möglichen "7/8"-Felder.

Lösungsvorschlag Teil 2

```
public long partTwo() {
    breite = inputLines.get(0).length();
    hoehe = inputLines.size();
    map = new char[breite][hoehe];

    resetLists();

    for (int y = 0; y < inputLines.size(); y++) {
        String line = inputLines.get(y);
```

```
        for (int x = 0; x < line.length(); x++) {
            char c = line.charAt(x);
            if (c == 'S') {
                currentTiles.add(new int[]{x, y});
                c = '.';
            }
            map[x][y] = c;
        }
    }

    long oddFull = bfs(129);

    resetLists();
    currentTiles.add(new int[]{65,65});
    long evenFull = bfs(130);

    long amountOfOddGardens = 1;
    long amountOfEvenGardens = 0;
    for (int i = 1; i <= 202299; i++) {
        if (i % 2 != 0) {
            amountOfEvenGardens += 4*i;
        } else {
            amountOfOddGardens += 4*i;
        }
    }

    long total = amountOfEvenGardens * evenFull + amountOfOddGardens *
oddFull;

    // Pointy-Flächen (Diamantspitzen)
    resetLists();
    currentTiles.add(new int[]{0,65});
    total += bfs(130);

    resetLists();
    currentTiles.add(new int[]{65,0});
    total += bfs(130);

    resetLists();
    currentTiles.add(new int[]{130,65});
    total += bfs(130);

    resetLists();
    currentTiles.add(new int[]{65,130});
    total += bfs(130);

    // "7/8"-Flächen
    resetLists();
    currentTiles.add(new int[]{0,130});
    long siebenachtel = bfs(195);
    total += (202300-1) * siebenachtel;
```

```
    resetLists();
    currentTiles.add(new int[]{130,0});
    siebenachtel = bfs(195);
    total += (202300-1) * siebenachtel;

    resetLists();
    currentTiles.add(new int[]{0,0});
    siebenachtel = bfs(195);
    total += (202300-1) * siebenachtel;

    resetLists();
    currentTiles.add(new int[]{130,130});
    siebenachtel = bfs(195);
    total += (202300-1) * siebenachtel;

    // "1/8"-Flächen
    resetLists();
    currentTiles.add(new int[]{0,130});
    long einachtel = bfs(64);
    total += 202300 * einachtel;

    resetLists();
    currentTiles.add(new int[]{130,0});
    einachtel = bfs(64);
    total += 202300 * einachtel;

    resetLists();
    currentTiles.add(new int[]{0,0});
    einachtel = bfs(64);
    total += 202300 * einachtel;

    resetLists();
    currentTiles.add(new int[]{130,130});
    einachtel = bfs(64);
    total += 202300 * einachtel;

    return total;
}
```

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

Permanent link:
https://info-bw.de/faecher:informatik:oberstufe:java:aoc:aco2023:day21:start

Last update: **25.12.2023 12:15**