

Day 20: Race Condition

Das Rätsel lässt sich recht einfach z. B. mit einer **Tiefensuche** lösen. Da es nur genau einen möglichen Pfad gibt, benötigt man keinen komplizierten Algorithmus wie etwa *Dijkstra* - es gibt nur genau einen kürzesten Weg.

Teil 1

Vorgehensweise:

- Speichere den Input in einem zweidimensionalen `int[][]` Array! Setze jede Pfad-Koordinate auf `Integer.MAX_VALUE` (ebenso die Start- und Zielkoordinate) und jede Wand ('#') auf -1. Merke dir außerdem in Variablen die Koordinaten vom Start und Ziel.
- Führe eine Tiefensuche durch, um vom Start zum Ziel zu gelangen. Nutze dazu einen von dir verwalteten Stack und lege den Startknoten auf den Stack. Dazu bietet es sich an, eine **separate Klasse** zu besitzen, um sowohl die nächste zu betrachtende Koordinate als auch deren Distanz ("aktuelleDistanz") auf den Stack legen zu können. Für jede Koordinate, die man dann vom Stack entnimmt (`while(stackIstNichtLeer) {...}`), macht man folgendes:
 - Wenn an der Koordinate der Wert -1 ist (eine Wand), dann überspringe diese Koordinate mit `continue;`
 - Wenn der Wert an der Koordinate `> -1` ist (keine Wand), aber gleichzeitig `< aktuelleDistanz`, dann hat man sich offenbar beim vorherigen "Abbiegen" rückwärts bewegt → ebenfalls mit `continue;` überpringen.
 - Wenn der Wert an der Koordinate auf `Integer.MAX_VALUE` liegt, dann kann er durch die aktuelle Distanz ersetzt werden.
 - Wenn das Ende erreicht wurde (Koordinaten überprüfen), dann kann man die while-Schleife mit `break;` abbrechen.
 - Zum Abschluss folgt noch die "Rekursion" in alle 4 möglichen Richtungen: Man legt die vier benachbarten Koordinaten auf den Stack.

Lösungsvorschlag "separate Knotenklasse"

```
<code java> public class D20Node {
```

```
    private int[] coord;  
    private int distance;
```

```
    /**  
     * Konstruktor für Objekte der Klasse D20Node  
     */  
    public D20Node(int[] coord, int distance)  
    {  
        this.coord = coord;  
        this.distance = distance;  
    }  
    public int[] getCoord() {  
        return this.coord;  
    }
```

```
}  
public int getDistance() {  
    return this.distance;  
}
```

} <code>

Lösungsvorschlag Teil 1:

<code java> public void partOne() {

```
int width = inputLines.get(0).length();  
int height = inputLines.size();  
int[] start = new int[2];  
int[] end = new int[2];
```

```
// Übertrage input in int[][] array  
int[][] map = new int[width][height];  
for (int y = 0; y < height; y++) {  
    String line = inputLines.get(y);  
    for (int x = 0; x < width; x++) {  
        char c = line.charAt(x);  
        if (c == 'S') {  
            start[0] = x;  
            start[1] = y;  
        } else if (c == 'E') {  
            end[0] = x;  
            end[1] = y;  
        }  
    }  
}
```

```
        if (c == 'S' || c == 'E' || c == '.') {  
            map[x][y] = Integer.MAX_VALUE;  
        } else {  
            map[x][y] = -1;  
        }  
    }  
}
```

```
// Tiefensuche, um den ganzen Weg mit Distanzen zu markieren  
Stack<D20Node> nodes = new Stack();  
nodes.add(new D20Node(start, 0));
```

```
while (nodes.size() > 0) {  
    D20Node n = nodes.pop();  
    int distance = n.getDistance();  
    int[] coord = n.getCoord();
```

```
    // Abbruchbedingung: in Wall
```

```
    if (map[coord[0]][coord[1]] == -1) {  
        continue;  
    }
```

```
    // Abbruchbedingung: Weg rückwärts gegangen!  
    if (map[coord[0]][coord[1]] > -1 && map[coord[0]][coord[1]] <  
distance) {  
        continue;  
    }
```

```
    // Weg markieren  
    if (map[coord[0]][coord[1]] == Integer.MAX_VALUE) {  
        map[coord[0]][coord[1]] = distance;  
    }
```

```
    // Abbruchbedingung: Ende erreicht  
    if (coord[0] == end[0] && coord[1] == end[1]) {  
        break;  
    }
```

```
    // In alle 4 Richtungen gehen  
    nodes.add(new D20Node(new int[]{coord[0]+1,coord[1]}, distance+1));  
    nodes.add(new D20Node(new int[]{coord[0]-1,coord[1]}, distance+1));  
    nodes.add(new D20Node(new int[]{coord[0],coord[1]+1}, distance+1));  
    nodes.add(new D20Node(new int[]{coord[0],coord[1]-1}, distance+1));  
}
```

```
int cheats = 0;  
// prüfe für jedes #, ob links/rechts oder oben/unten eine Distanz > 100  
vorhanden ist -> ein weiterer Cheat!  
for (int y = 1; y < height-1; y++) {  
    for (int x = 1; x < width-1; x++) {  
        if (map[x][y] != -1) {  
            continue;  
        }  
        if (map[x-1][y] != -1 && map[x+1][y] != -1 &&  
Math.abs(map[x-1][y]-map[x+1][y]) >= 100 + 2) {  
            cheats++;  
        } else if (map[x][y-1] != -1 && map[x][y+1] != -1 &&  
Math.abs(map[x][y-1]-map[x][y+1]) >= 100 + 2) {  
            cheats++;  
        }  
    }  
}  
System.out.println(cheats);
```

```
} <code>
```

Teil 2

Der Beginn ist absolut unverändert. Man berechnet genauso einmal den Weg bis zum Ziel. Es gibt bloß folgende Änderungen:

- Alle Koordinaten, die Teil des Pfades vom Start zum Ziel sind, werden in einer ArrayList gespeichert.
- Alle darin gespeicherten Koordinaten werden anschließend paarchenweise betrachtet (jede paarchenweise Kombination!).
 - Wenn beide Koordinaten nahe genug beieinander liegen (<20 picoseconds), wenn also überhaupt ein Cheat möglich wäre...
 - UND wenn die Pfad-Distanz an beiden Koordinaten größer ist als 100 (also wenn die Abkürzung überhaupt eine Relevanz hat / "gut genug ist"), wobei dazu noch die Distanz der x- und y-Koordinaten kommt, da der Cheat-Weg auch noch zurückgelegt werden muss...
 - DANN hat man einen **weiteren Cheat** gefunden.

Lösungsvorschlag

```
<code java> public void partTwo() {
```

```
    int width = inputLines.get(0).length();
    int height = inputLines.size();
    int[] start = new int[2];
    int[] end = new int[2];
```

```
    int[][] map = new int[width][height];
    for (int y = 0; y < height; y++) {
        String line = inputLines.get(y);
        for (int x = 0; x < width; x++) {
            char c = line.charAt(x);
            if (c == 'S') {
                start[0] = x;
                start[1] = y;
            } else if (c == 'E') {
                end[0] = x;
                end[1] = y;
            }
        }
    }
```

```
        if (c == 'S' || c == 'E' || c == '.') {
            map[x][y] = Integer.MAX_VALUE;
        } else {
            map[x][y] = -1;
        }
    }
}
```

```
// Tiefensuche, um den ganzen Weg mit Distanzen zu markieren
```

```
Stack<D20Node> nodes = new Stack();
nodes.add(new D20Node(start, 0));
// Neu für Teil 2: Eine ArrayList, die jede Koordinate des Pfades
speichert.
ArrayList<int[]> path = new ArrayList();
```

```
while (nodes.size() > 0) {
    D20Node n = nodes.pop();
    int distance = n.getDistance();
    int[] coord = n.getCoord();
```

```
    // Abbruchbedingung: außerhalb der Map
    if (coord[0] < 0 || coord[0] >= width || coord[1] < 0 || coord[1] >=
height) {
        continue;
    }
```

```
    // Abbruchbedingung: in Wall
    if (map[coord[0]][coord[1]] == -1) {
        continue;
    }
```

```
    // Abbruchbedingung: Weg rückwärts gegangen!
    if (map[coord[0]][coord[1]] > -1 && map[coord[0]][coord[1]] <
distance) {
        continue;
    }
```

```
    // Weg markieren
    if (map[coord[0]][coord[1]] == Integer.MAX_VALUE) {
        map[coord[0]][coord[1]] = distance;
        path.add(coord); // Neu für Teil 2
    }
```

```
    // Abbruchbedingung: Ende erreicht
    if (coord[0] == end[0] && coord[1] == end[1]) {
        break;
    }
```

```
    // In alle 4 Richtungen gehen
    nodes.add(new D20Node(new int[]{coord[0]+1,coord[1]}, distance+1));
    nodes.add(new D20Node(new int[]{coord[0]-1,coord[1]}, distance+1));
    nodes.add(new D20Node(new int[]{coord[0],coord[1]+1}, distance+1));
    nodes.add(new D20Node(new int[]{coord[0],coord[1]-1}, distance+1));
}
```

```
// Neu für Teil 2
// Betrachte ALLE Koordinaten-Päärchen innerhalb des Pfades!
int cheats = 0;
for (int i = 0; i < path.size()-1; i++) {
```

```
for (int j = i+1; j < path.size(); j++) {  
    int[] from = path.get(i);  
    int[] to = path.get(j);
```

```
    // Wenn die Koordinaten beider betrachteten Pfad-Koordinaten nahe  
    genug beieinander liegen (<=20 picoseconds)  
    if (Math.abs(to[0]-from[0])+Math.abs(to[1]-from[1]) <= 20) {  
        // Und wenn die Pfad-Distanz zwischen beiden Koordinaten  
        größer ist als 100 zuzüglich der räumlichen Distanz der beiden Koordinaten  
        if (map[to[0]][to[1]] - map[from[0]][from[1]] >= 100 +  
        Math.abs(to[0]-from[0]) + Math.abs(to[1]-from[1])) {  
            // Dann hat man einen Cheat gefunden  
            cheats++;  
        }  
    }  
}  
System.out.println(cheats);
```

} <code>

From:
<https://www.info-bw.de/> -

Permanent link:
<https://www.info-bw.de/faecher:informatik:oberstufe:java:aoc:aoc2024:day20:start?rev=1734696094>

Last update: **20.12.2024 12:01**

