Day 18: RAM Run

Tag 18 ist ein Paradebeispiel für den Dijkstra Algorithmus! Der Graph ist in dem Fall ein zweidimensionales Array und es gibt pro Zelle/Knoten 4 potenzielle Richtungen, in die man sich bewegen kann.

Teil 1

Vorgehensweise:

- Es bietet sich an, die Breite (=Höhe) des Arrays und die Anzahl der fallenden Bytes als Variablen ganz zu Beginn der Methode festzuhalten, damit man sie schnell ändern kann, je nachdem, ob man gerade das "Example" oder den "realen Input" betrachtet.
- Baue den "Graphen" für den Dijkstra-Algorithmus auf bzw. bereite diesen vor. Wie oben angekündigt, entspricht die Karte einem zweidimensionalen Array, daher macht es absolut Sinn, auch den Graphen in einem zweidimensionalen Array darzustellen. Geklärt werden muss, was zu jedem Knoten (jeder Koordinate) gespeichert werden muss:
    - Als Erinnerung: Beim Dijkstra-Algorithmus muss jeder Knoten seine **Distanz** zum Startknoten speichern, wobei diese für jeden Knoten anfangs auf "unendlich" gesetzt ist (`Integer.MAX_VALUE`). Außerdem muss man für jeden Knoten entscheiden können, ob er bereits entdeckt/besucht wurde - dies lässt sich glücklicherweise über die Distanz lösen: Wenn ein Knoten bereits entdeckt wurde, dann ist seine Distanz kleiner als "unendlich". Wenn er bereits besucht wurde, dann ist seine Distanz bereits "irrelevant klein".
    - Außerdem muss in diesem Fall für jede Koordinate/Knoten gespeichert werden, **ob sie eine "Wand"** bzw. ein Byte ('#') ist.
    - Zuletzt muss jeder Knoten seine **x- und y-Koordinate** kennen, damit man vom aktuellen Knoten zu den direkten Nachbarknoten springen kann.
- Erstelle also zunächst eine separate Klasse, z. B. `D18Node` (D18 für "Day 18"), die einen einzelnen Knoten des Graphen widerspiegelt und die oberen 4 Eigenschaften als Instanzvariablen enthält. Erstelle entsprechende Getter- und Setter-Methoden.
- Erstelle anschließend ein zweidimensionales Array vom Typ `D18Node[][]` und initialisiere in einer doppelten Schleife auch jede Koordinate (gib dem Konstruktor die x- und y-Koordinate als Parameter mit).
- Gehe nun die ersten Zeilen des Inputs durch (beim Example `12`, beim realen Input `1024`) und setze in der Karte den entsprechenden Knoten als "Wand".
- Anschließend musst du dir Gedanken machen für deine Datenstruktur zur Speicherung der bereits entdeckten, nächsten Knoten. Du kannst z. B. eine `ArrayList` nehmen und vor dem Entfernen des nächst-kleinsten Elements jeweils die ArrayList sortieren. Du kannst aber auch eine `PriorityQueue` nutzen. Diese funktioniert so, dass die enthaltenen Elemente immer automatisch sortiert sind und immer direkt das kleinste (oder auch größte) Element entnommen werden kann. Beispiel für die PriorityQueue:
    - Importiere die folgenden packages: `import java.util.Queue; import java.util.PriorityQueue; import java.util.Comparator;`
    - Erstelle einen `Comparator`, der der Queue mitteilt, nach welchen Kritieren überhaupt sortiert werden soll. `p1` und `p2` stehen dabei für zwei zu vergleichende Elemente vom Typ `D18Node`. Die Methodennamen `getDistance()` musst du u. U. an deine Klasse

anpassen: `Comparator<D18Node> comparator = (p1, p2) → Integer.compare(p1.getDistance(), p2.getDistance());`
    - Nun wird die PriorityQueue erstellt: `Queue<D18Node> queue = new PriorityQueue<>(comparator);`
- Jetzt sind die Vorbereitungen abgeschlossen und der Dijkstra-Algorithmus kann starten. Zu Beginn musst du immer die Distanz des Startknotens (hier Koordinate 0,0) auf 0 setzen und diesen Knoten in die Queue einfügen (`queue.offer(map[0][0])`).
- Nun wiederholst du solange, wie es noch Elemente in der Queue gibt (diese also noch nicht leer ist):
    - Hole dir das kleinste Element aus der Queue (`queue.poll()`).
    - Prüfe nun für jede der 4 Nachbar-Richtungen:
        - Darf man sich überhaupt in die Richtung bewegen, oder ist man dann außerhalb der Map?
        - Ist der Nachbarknoten keine Wand?
        - Ist die Distanz des Nachbarknotens größer als die aktuelle Distanz + 1?
    - , Nur wenn alle oberen Fragen mit JA beantwortet wurden, dann darf die Distanz des Nachbarknotens auf die aktuelle Distanz + 1 gesetzt werden. Außerdem muss der Nachbarknoten zur Queue hinzugefügt werden, falls er nicht bereits darin enthalten ist (`if(!queue.contains(…)) {…}`).
- Sobald kein Element mehr in der Queue enthalten ist, hat man alle Knoten besucht, die überhaupt besucht werden können, also auch den Endknoten. Grundsätzlich kann man die große Schleife natürlich auch schon abbrechen, sobald man den Endknoten aus der Queue entnommen hat. Dadurch spart man sich wenige Schleifendurchgänge.

Lösungsvorschlag

```
public void partOne() {
    // Speichere hier die Startwerte, die du je nach "Example" oder "realem
Input" schnell anpassen kannst
    int numBytes = 1024;
    int width = 71;
    int height = 71;

    // Karte mit Knoten
    D18Node[][] map = new D18Node[width][height];

    // Initialisiere alle Knoten
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            map[x][y] = new D18Node(x, y);
        }
    }
    // Markiere für die ersten "numBytes" Koordinaten, dass dort "Wände"
sind.
    for (int i = 0; i < numBytes; i++) {
        int x = Integer.parseInt(inputLines.get(i).split(",")[0]);
        int y = Integer.parseInt(inputLines.get(i).split(",")[1]);
        map[x][y].setWall();
    }
```

```
    // Der Comparator ist nötig, damit die Priority-Queue weiß, nach welchen
Kriterien sie zwei D18Nodes miteinander vergleichen soll
    Comparator<D18Node> comparator = (p1, p2) ->
Integer.compare(p1.getDistance(), p2.getDistance());
    // Diese Queue ist besonders: Sie ist eine PriorityQueue, die
automatisch das kleinste Element nach vorne holt
    Queue<D18Node> queue = new PriorityQueue<>(comparator);

    // Startknoten markieren und zur Queue hinzufügen
    map[0][0].setDistance(0);
    queue.offer(map[0][0]);

    while(!queue.isEmpty()) {
        D18Node p = queue.poll();
        int dis = p.getDistance();
        int x = p.getX();
        int y = p.getY();

        // Abbrechen, wenn Endknoten gefunden
        if (x == width-1 && y == height-1) {
            break;
        }

        // markiere alle Nachbarn
        // links
        if (x > 0 && map[x-1][y].getDistance() > dis+1 &&
!map[x-1][y].isWall()) {
            map[x-1][y].setDistance(dis+1);
            if (!queue.contains(map[x-1][y])) {
                queue.offer(map[x-1][y]);
            }
        }
        // rechts
        if (x < width-1 && map[x+1][y].getDistance() > dis+1 &&
!map[x+1][y].isWall()) {
            map[x+1][y].setDistance(dis+1);
            if (!queue.contains(map[x+1][y])) {
                queue.offer(map[x+1][y]);
            }
        }
        // oben
        if (y > 0 && map[x][y-1].getDistance() > dis+1 &&
!map[x][y-1].isWall()) {
            map[x][y-1].setDistance(dis+1);
            if (!queue.contains(map[x][y-1])) {
                queue.offer(map[x][y-1]);
            }
        }
        // unten
        if (y < height-1 && map[x][y+1].getDistance() > dis+1 &&
```

```
!map[x][y+1].isWall()) {
            map[x][y+1].setDistance(dis+1);
            if (!queue.contains(map[x][y+1])) {
                queue.offer(map[x][y+1]);
            }
        }
    }

    System.out.println(map[width-1][height-1].getDistance());
}
```

Teil 2

Für Teil 2 müssen nun minimale Änderungen durchgeführt werden!

- Der gesamte Code aus Teil 1 muss in eine Schleife gepackt werden.
- Die Variable, die ganz zu Beginn festlegt, wie viele Bytes herunterfallen, muss "variabel" werden. Sie beginnt weiterhin bei 12 bzw. 1024, zählt aber mit jedem Schleifendurchlauf eins hoch.
- Immer, **nachdem** die innere "Queue-Schleife" beendet wurde, muss geprüft werden, ob nun der Endknoten **nicht mehr erreicht** wurde. Dies geht, in man prüft, ob seine Distanz nun der initialen Distanz `Integer.MAX_VALUE` entspricht. Dann wurde sie offensichtlich nie im Schleifendurchlauf verkleinert. In diesem Fall darf das Programm abbrechen, und man kann die Anzahl der gefallenen Bytes zurückgeben.

Lösungsvorschlag

```
public void partTwo() {
    int numBytes = 1024;
    while(true) {
        numBytes++; // erhöhe nun die Anzahl der fallenden Bytes mit jedem
Schleifendurchlauf
        int width = 71;
        int height = 71;

        D18Node[][] map = new D18Node[width][height];

        Comparator<D18Node> comparator = (p1, p2) ->
Integer.compare(p1.getDistance(), p2.getDistance());
        Queue<D18Node> queue = new PriorityQueue<>(comparator);

        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                map[x][y] = new D18Node(x, y);
            }
        }
        for (int i = 0; i < numBytes; i++) {
```

```
            int x = Integer.parseInt(inputLines.get(i).split(",")[0]);
            int y = Integer.parseInt(inputLines.get(i).split(",")[1]);
            map[x][y].setWall();
        }

        map[0][0].setDistance(0);
        queue.offer(map[0][0]);

        while(!queue.isEmpty()) {
            D18Node p = queue.poll();
            int dis = p.getDistance();
            int x = p.getX();
            int y = p.getY();

            // Abbrechen, wenn Endknoten gefunden
            if (x == width-1 && y == height-1) {
                break;
            }

            // markiere alle Nachbarn
            // links
            if (x > 0 && map[x-1][y].getDistance() > dis+1 &&
!map[x-1][y].isWall()) {
                map[x-1][y].setDistance(dis+1);
                if (!queue.contains(map[x-1][y])) {
                    queue.offer(map[x-1][y]);
                }
            }
            // rechts
            if (x < width-1 && map[x+1][y].getDistance() > dis+1 &&
!map[x+1][y].isWall()) {
                map[x+1][y].setDistance(dis+1);
                if (!queue.contains(map[x+1][y])) {
                    queue.offer(map[x+1][y]);
                }
            }
            // oben
            if (y > 0 && map[x][y-1].getDistance() > dis+1 &&
!map[x][y-1].isWall()) {
                map[x][y-1].setDistance(dis+1);
                if (!queue.contains(map[x][y-1])) {
                    queue.offer(map[x][y-1]);
                }
            }
            // unten
            if (y < height-1 && map[x][y+1].getDistance() > dis+1 &&
!map[x][y+1].isWall()) {
                map[x][y+1].setDistance(dis+1);
                if (!queue.contains(map[x][y+1])) {
                    queue.offer(map[x][y+1]);
                }
```

```
                }
            }

            // Endknoten nicht mehr erreichbar? -> Initiale Maximaldistanz ist
unverändert geblieben
            if (map[width-1][height-1].getDistance() == Integer.MAX_VALUE) {
                System.out.println(inputLines.get(numBytes-1));
                return;
            }
        }
}
```