Aufgabe 1
„Die Kunst der Fuge“- Dokumentation

36. Bundeswettbewerb Informatik 2017/18 - 2. Runde

Lukas Rost

Teilnahme-ID: 44137

9. April 2018

Inhaltsverzeichnis

1 Lösungsidee

1.1 Mögliche Maximalhöhe der Mauer

Da in der Aufgabe explizit nach der Höhe der Mauer für verschiedene n gefragt ist, lohnt es sich, diese zunächst einmal mathematisch zu berechnen. Damit lässt sich der Suchraum effektiv eingrenzen, da ab dieser Maximalhöhe die Forderungen der Aufgabenstellung nicht mehr erfüllt werden können. Pro Reihe der Mauer gibt es $n-1$ verschiedene Fugen. Diese Zahl ergibt sich daraus, dass hinter jedem Klötzchen genau eine Fuge folgt. Nur beim letzten Klötzchen der Reihe zählt diese Position nicht, da sich dahinter kein weiteres Klötzchen befindet.

Fugen sind also per definitionem nur die Stellen zwischen den Klötzchen, da die Position 0 sowie die Position $\frac{n\cdot(n+1)}{2}$ am Ende der Reihe in jeder Reihe belegt werden. Nun interessiert noch die Zahl der insgesamt belegbaren Fugenpositionen. Dies sind alle Positionen von 1 bis zur Länge der Mauer minus 1, da, wie bereits erwähnt, die Position am Ende nicht als Fuge zählt. Die Länge der Mauer ist die Summe aller natürlichen Zahlen bis n, also nach der Gaußschen Summenformel $\frac{n\cdot(n+1)}{2}$.

Teilt man nun die Anzahl der insgesamt verfügbaren Fugenpositionen durch die Anzahl der Fugen pro Reihe, erhält man die mögliche Anzahl an Reihen, also die Höhe.

$$H(n) = \frac{\frac{n\cdot(n+1)}{2} - 1}{n-1} = \frac{(n+2)\cdot(n-1)}{2\cdot(n-1)} = \lfloor\frac{n+2}{2}\rfloor \tag{1}$$

Die Abrundungsfunktion ist an dieser Stelle nötig, da sonst für ungerade n das Resultat keine natürliche Zahl ist. Halbe Reihen jedoch ergeben wenig Sinn. Die angegebene Funktion $H(n)$ ist für alle natürlichen Zahlen $n \geq 2$ definiert.

Der Fall $n = 1$ ist damit nicht berechenbar, da in der Originalgleichung durch 0 geteilt würde. Dieser muss auch ausgeschlossen werden, da es dabei keine Fugen im Sinne von Stellen zwischen den Klötzen gibt. Somit ist nicht eindeutig definiert, ob in diesem Fall nur eine Reihe (mit einem Klotz der Länge 1) oder unendlich viele Reihen erlaubt sind.

1.2 Lösungsansatz: Backtracking

1.3 Laufzeit und NP-Äquivalenz

2 Umsetzung

3 Beispiele

```
Lösung für n = 2:
| 1 | 2 |
| 2 | 1 |

Lösung für n = 3:
| 1 | 2 | 3 |
| 2 | 3 | 1 |
```

```
Lösung für n = 4:
| 1 | 2 | 3 | 4 |
| 2 | 3 | 4 | 1 |
| 4 | 3 | 1 | 2 |
```

```
Lösung für n = 5:
| 1 | 2 | 3 | 4 | 5 |
| 2 | 3 | 4 | 5 | 1 |
| 4 | 3 | 5 | 1 | 2 |
```

```
Lösung für n = 6:
| 1 | 2 | 3 | 4 | 5 | 6 |
| 5 | 6 | 3 | 2 | 4 | 1 |
| 4 | 3 | 6 | 5 | 1 | 2 |
| 2 | 6 | 1 | 3 | 5 | 4 |
```

4 Quellcode

```java
    private static int[][] getSolution(int n){
        int[][] solution = new int[(int) Math.floor((n+2)/2)][n];
        for (int z = 1; z <= n; z++){
            solution[0][z-1] = z;
            if (n != 6 && n != 10) {
                solution[1][z - 1] = z + 1;
            }
        }
        solution[1][n-1] = 1;
        HashSet<Integer> usedFugen = new HashSet<>();
        int fuge = 0;
        for (int z = 1; z < n; z++){
            fuge += z;
            usedFugen.add(fuge);
        }
        if (n != 6 && n!= 10) {
            int fuge2 = 0;
            for (int z = 2; z <= n; z++) {
                fuge2 += z;
                usedFugen.add(fuge2);
            }
        }
        LinkedList<Integer> usableKloetzeNextEbene = new LinkedList<>();
        for (int k = 1; k <= n; k++){
            usableKloetzeNextEbene.add(k);
        }
        int t = (n == 6 || n== 10) ? 1 : 2;
        solution = backTrack(n,
        ↪  solution,t,0,usableKloetzeNextEbene,usedFugen);
        return solution;
    }

    private static int[][] backTrack(int n, int[][] mauerBefore, int reihe,
    ↪  int klotzInReihe, LinkedList<Integer> usableKloetze, HashSet<Integer>
    ↪  usedFugen){
        if (reihe > (int) (Math.floor(((n+2)/2)-1))){
            return mauerBefore;
        }
        int summe = 0;
        for (int i = 0; i < klotzInReihe; i++){
            summe += mauerBefore[reihe][i];
        }
        LinkedList<Integer> usableKloetzeNextEbene = new LinkedList<>();
```

```
41          usableKloetzeNextEbene.addAll(usableKloetze);
42          for (Integer klotz : usableKloetzeNextEbene) {
43              if (usedFugen.contains(summe + klotz) && ((summe + klotz) != ((n *
                ↪  (n + 1)) / 2))) {
44                  usableKloetze.remove(klotz);
45              }
46          }
47          while (!(usableKloetze.isEmpty())){
48              int uk = usableKloetze.get(usableKloetze.size()-1);
49              mauerBefore[reihe][klotzInReihe] = uk;
50              usableKloetzeNextEbene.remove(Integer.valueOf(uk));
51              usableKloetze.remove(usableKloetze.size()-1);
52              if (summe + uk != ((n)*(n+1))/2) {
53                  usedFugen.add(summe + uk);
54              }
55              if (reihe == (int) (Math.floor(((n+2)/2)-1)) && klotzInReihe ==
                ↪  n-1){
56                  return mauerBefore;
57              }
58              int reiheneu = reihe;
59              int klotzInReiheNeu = klotzInReihe + 1;
60              LinkedList<Integer> usableKloetzeNextEbene2 = new LinkedList<>();
61              if (klotzInReihe == n-1){
62                  reiheneu = reihe + 1;
63                  klotzInReiheNeu = 0;
64                  for (int k = 1; k <= n; k++){
65                      usableKloetzeNextEbene2.add(k);
66                  }
67              } else {
68                  usableKloetzeNextEbene2.addAll(usableKloetzeNextEbene);
69              }
70              int[][] mauerDanach = backTrack(n,mauerBefore,reiheneu⌋
                ↪  ,klotzInReiheNeu,usableKloetzeNextEbene2,usedFugen);
71              if (mauerDanach[0][0] == -1){
72                  usedFugen.remove(summe + uk);
73                  usableKloetzeNextEbene.add(uk);
74              } else {
75                  return  mauerDanach;
76              }
77          }
78          return new int[][]{{-1}};
79      }
80  }
```

Quellcode 1: Der Backtracking-Algorithmus