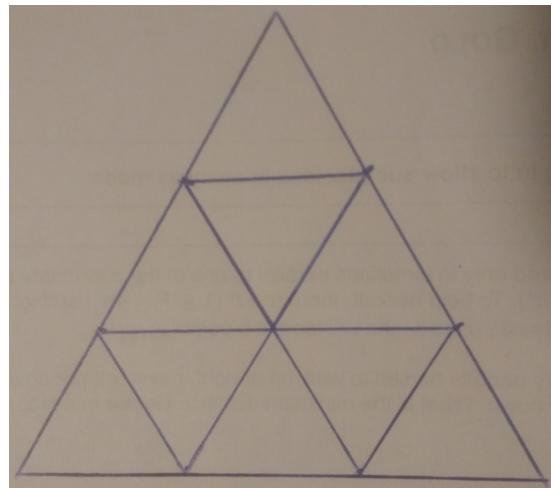


Aufgabe 2: Dreieckspuzzle

00759 - TranshumanTechnocrator - Erik Klein

1 Aufgabe

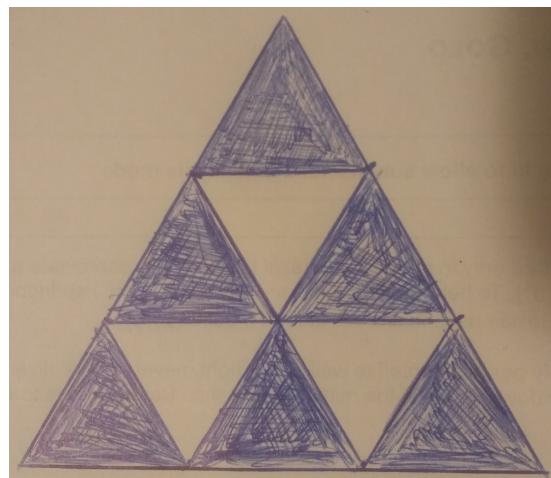
Gegeben sind 9 Dreiecke, die wie folgt zu einem großen Dreieck angeordnet werden müssen:



Jedes kleine Dreieck hat an jeder Seite einen Teil einer Figur, entweder einen oberen, oder einen unteren Teil. Es gibt verschiedene Arten von Figuren. Im großen Dreieck müssen alle Figuren zusammenpassen, d.h. für alle angrenzenden Kanten müssen die dortigen Figuren die gleiche Art haben, und eine davon ein oberer, die andere ein unterer Teil sein.

2 Lösungsidee

Die Aufgabe wurde mit Brutaler Kraft, d.h. Complete Search gelöst. Kanten werden als Ganzzahlen angegeben, wobei ihr Betrag ihre Art angibt, und das Vorzeichen, ob es ein oberer Teil oder ein unterer ist. Damit es weniger Möglichkeiten gibt, die man durchgeht, betrachten wir dieses Dreieck:



Die Idee ist, nur die 6 schwarzen/blauen Dreiecke durchzugehen. Wie die weißen sein müssten, ist eindeutig bestimmbar, da alle ihre Kanten an die Kante eines schwarzen Dreiecks angrenzen. Man invertiert die

schwarzen¹ und erhält die weißen Kanten. Wenn die schwarzen Kanten -1, 2 und 3 sind, muss das davon umgebene weiße Dreieck die Kanten -1, 2 und -3 haben.²

Man wählt also, welche 6 der 9 Dreiecke die schwarzen sind. Dafür gibt es $\binom{9}{6}$ Möglichkeiten. Dann wählt man, welches schwarze Dreieck an welche Position kommt. ($9!/3! = 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4$ Möglichkeiten), und schließlich wählt man, welches schwarze Dreieck wie gedreht ist (3^6 Möglichkeiten). Insgesamt sind das ca. $4.41 \cdot 10^7$ Möglichkeiten. Da man pro Sekunde ca. 10^8 Möglichkeiten durchgehen kann (Competitive Programming Faustregel), dauert es weniger als 0.5s.

Für jede Möglichkeit wird geprüft, ob sie valide ist, also ob die 3 übrigen Dreiecke für die wie oben ausgerechneten Dreiecke passen. Wenn ja, kann die Lösung ausgegeben und die gesamte Suche vorzeitig abgebrochen werden. Wenn keine Möglichkeit valide ist, ist das Puzzle unlösbar.

3 Umsetzung

Da ich zu wenig Zeit vor Einsendeschluss hatte, musste ich die Optimierung bei der Implementierung leider weglassen. Die Idee zählt ;). Ich habe also reines Brute-Force angewendet. Damit dies trotzdem schnell genug läuft, habe ich C++ verwendet. Im schlimmsten Fall, wenn das Puzzle unlösbar ist, dauert dies nur ca. 35s. Die Lösung erachte ich deshalb als effizient genug. Das allein zu implementieren, war schon schwer genug! Bitte hängt mir das nicht an, ich liebe den BWINF, will zum Jugendforum und zur IOI 2022 fahren! Dazu muss ich weiter kommen xD

Die Ausgabe wird in der Methode *ausgabe()* getätigt. In der Hauptmethode werden alle Permutationen der Dreiecke durchgegangen, mithilfe *next_permutation()*, für jede Permutation werden in 9 verschachtelten Schleifen alle Drehungen durchgegangen, wobei jede Möglichkeit geprüft wird. Um zu prüfen, ob eine Möglichkeit valide ist, arbeite ich mit sog. Constraints, die ich als *struct* definiert habe. Die Constraints speichern, welche Kante welches Dreiecks an welche Kante welches anderen Dreiecks grenzt. Für diese benachbarten Kanten wird in der Methode *check()* geprüft, ob die Art der einen das Negativkomplement der Art der anderen ist. In einer Reihung *con* speichere ich alle 9 Constraints des Dreieckspuzzles. Mithilfe von *all_of()* wird für jeden Constraint *check()* aufgerufen und nur wenn alle Constraints zu treffen, wurde eine valide Lösung gefunden. In dem Fall wird sie ausgegeben und die Suche mit *exit(0)* abgebrochen.

4 Quellcode

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 int rot[9];
4 int fig[9][3]; // art der figuren mit vorzeichen
5 vector<int> ord {0,1,2,3,4,5,6,7,8}; // reihenfolge
6
7 struct constraint { int idx1, edg1, idx2, edg2; } // bedingungen
8 con[] = {
9     {0,1,2,0},{1,2,2,1},{3,0,2,2}, // erstes weisses dreieck
10    {1,1,5,0},{4,2,5,1},{6,0,5,2}, // zweites ""
11    {3,1,7,0},{6,2,7,1},{8,0,7,2} // drittes ""
12 };
13
14 bool check(constraint c) {
15     return fig[ord[c.idx1]][(rot[c.idx1]+c.edg1)%3]
16         == -fig[ord[c.idx2]][(rot[c.idx2]+c.edg2)%3];
17 }
18
19 signed main()
20 {
21     // einlesen
22     int N, M; cin >> M >> N;
23     for(int j = 0; j < 9; j++)

```

¹Aus oben wird unten und andersrum

²Die schwarzen Dreiecke sind unabhängig voneinander auswählbar, da sie keine Kanten teilen und sich nicht gegenseitig blockieren.

```

25     cin >> fig[j][0] >> fig[j][1] >> fig[j][2];
26     // alle reihenfolgen der dreiecke durchgehen
27     do {
28         // alle drehungen durchgehen
29         for(rot[0] = 0; rot[0] < 3; rot[0]++)
30             for(rot[1] = 0; rot[1] < 3; rot[1]++)
31                 for(rot[2] = 0; rot[2] < 3; rot[2]++)
32                     for(rot[3] = 0; rot[3] < 3; rot[3]++)
33                         for(rot[4] = 0; rot[4] < 3; rot[4]++)
34                             for(rot[5] = 0; rot[5] < 3; rot[5]++)
35                             for(rot[6] = 0; rot[6] < 3; rot[6]++)
36                             for(rot[7] = 0; rot[7] < 3; rot[7]++)
37                             for(rot[8] = 0; rot[8] < 3; rot[8]++)
38                                 if(all_of(con,con+9,check)) {
39                                     // loesung gefunden
40                                     ausgabe();
41                                     exit(0);
42                                 }
43     } while(next_permutation(ord.begin(),ord.end()));
44     cout << "unloesbar\n";
45     return 0;
46 }
```

5 Beispiele

Hier zuerst die Ausgaben für die Beispiellösungen aus der Aufgabenstellung:
So sieht die Ausgabe meines Programms aus (in der Konsole sieht es besser aus):

Dreieck 1

/\
-1/ \1
-2-

Dreieck 2

/\
-1/ \1
-3-

Dreieck 3

2
-1\/-2
\\

Dreieck 4

/\
2 / \ -1
-1-

Dreieck 5

/\
2 / \ -3
-1-

Dreieck 6

-3
3 \/\ 2
\\

Dreieck 7

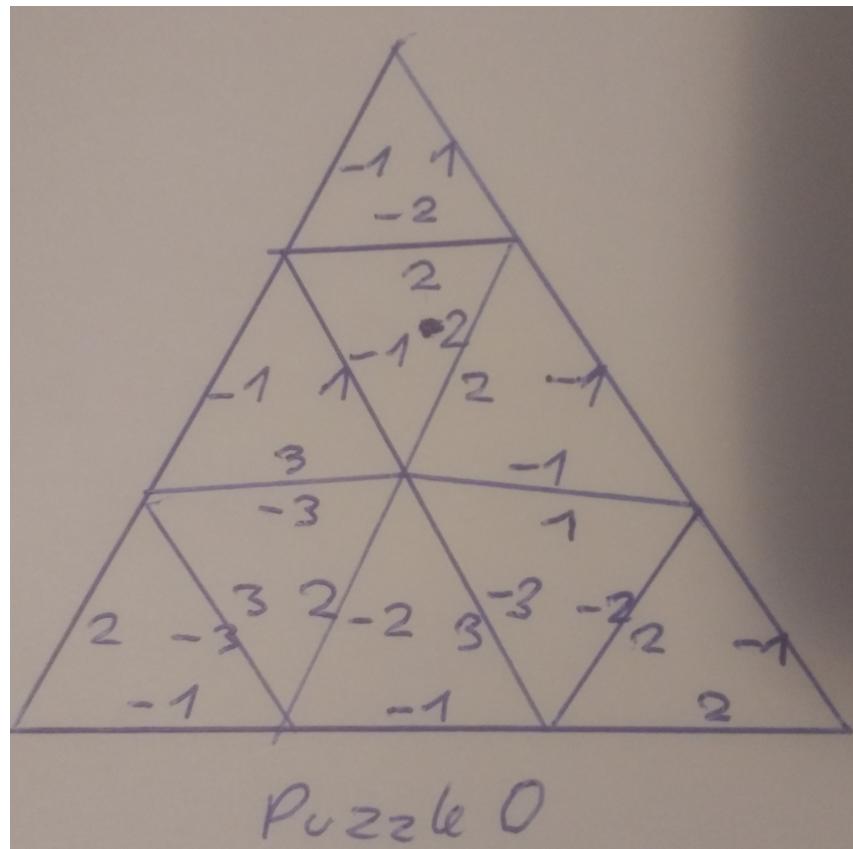
/\
-2 / \ 3
-1-

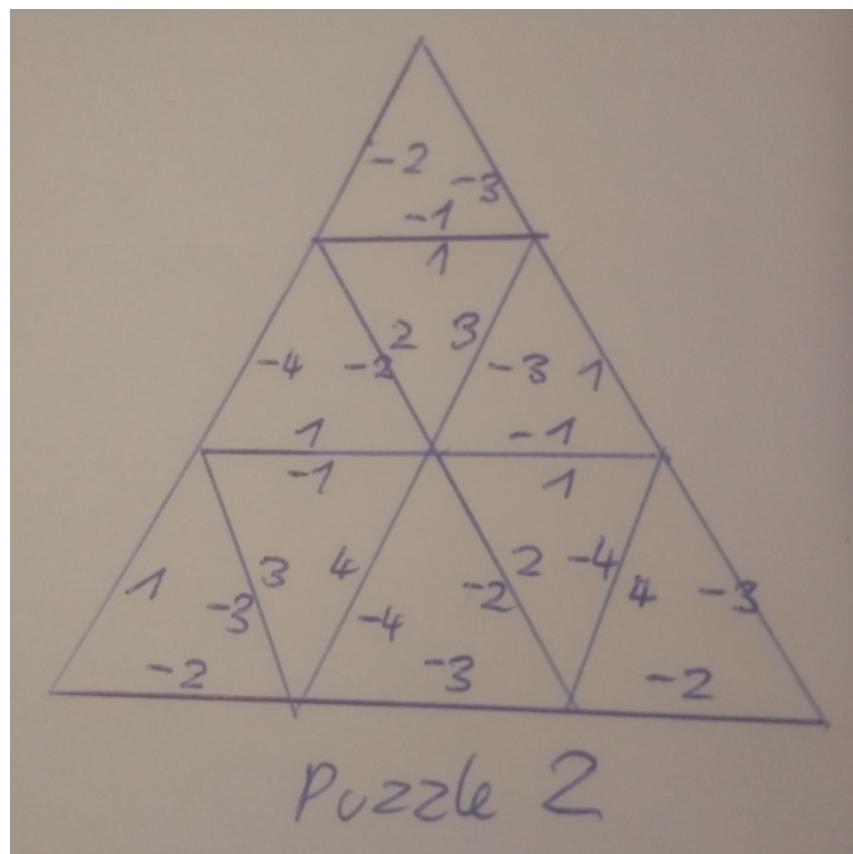
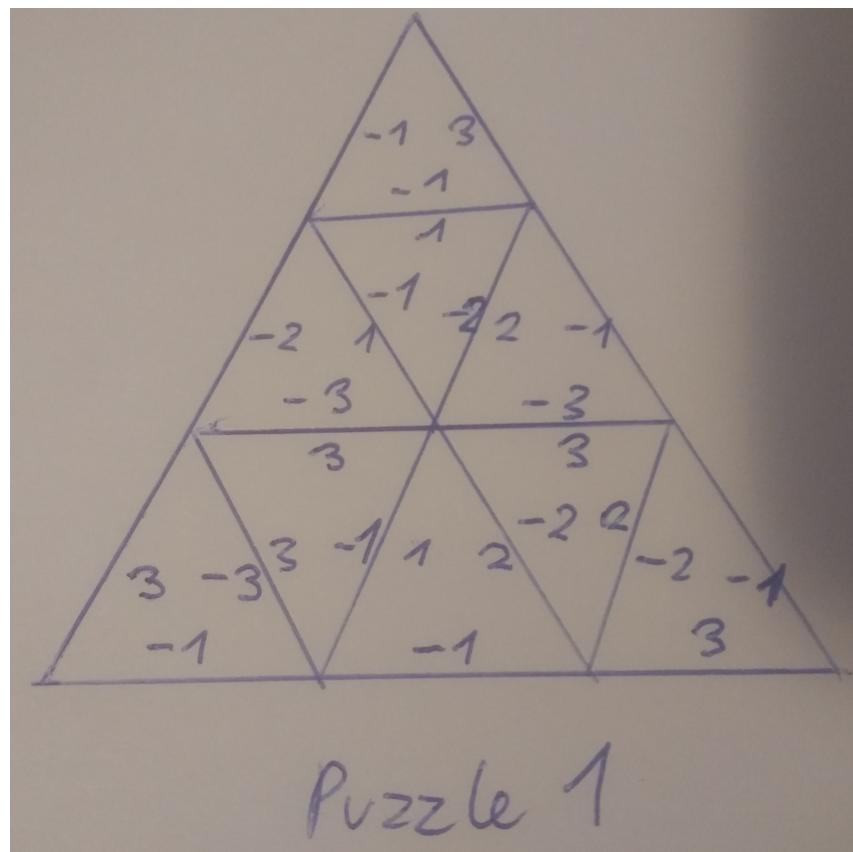
Dreieck 8

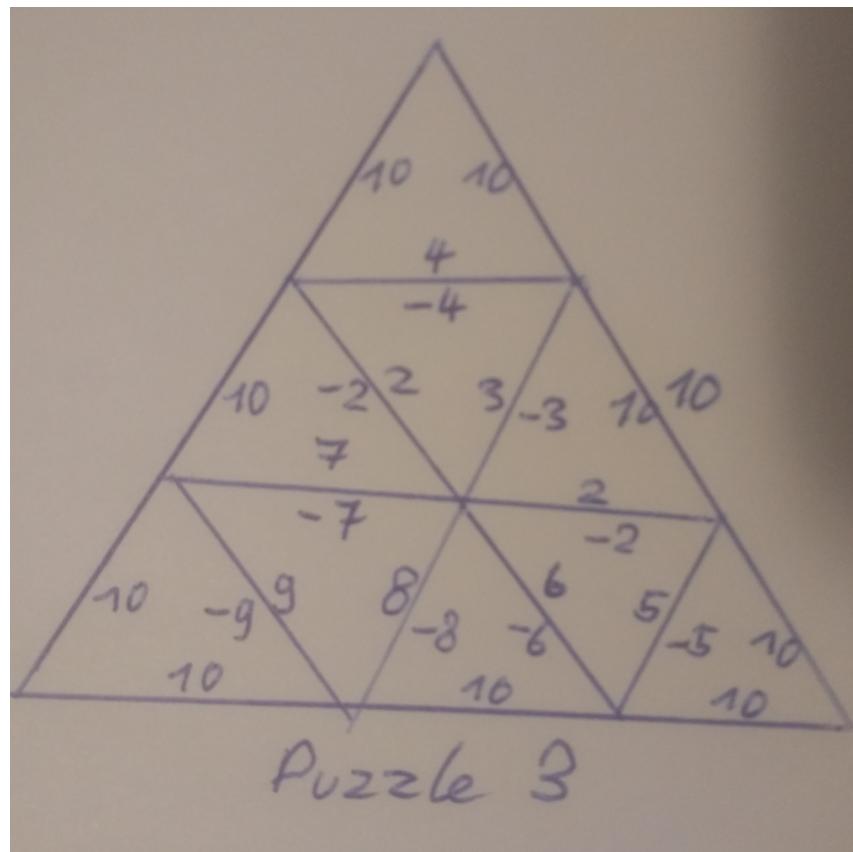
1
-3 \/\ -2

\begin{array}{c} \backslash \\ \text{Dreieck } 9 \\ / \quad \backslash \\ 2 \quad -1 \\ -2- \end{array}

Hier noch die Visualisierungen der Ausgaben:







Und noch ein eigenes unlösbares Beispiel:

1 9
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1

Ausgabe:
unlösbar