Das Programm wird in Python implementiert. Die Darstellung der Streichholzanordnungen könnte wie folgt verarbeitet werden:

- Jedes Streichholz in vorherigen Anordnung wird nummeriert von 0.
- Jedes Streichholz in nachherigen Anordnung wird mit Buchstaben markiert von "a".

Die Voraussetzung der Aufgabe, dass die Streichhölzer nur einen Winkel bilden dürfen, der ein Vielfaches von 30° ist, kann ausgenutzt werden. Indem man sich eine Wanduhr vorstellt: an einem Drehpunkt soll der Zeiger auf eine konkrete Ziffer auf der Uhr zeigen. Er hat dafür 12 mögliche Rotationen, die Streichhölzer ebenso.

Jetzt nehmen wir an, dass die Abbildungen in dem GIT-repository idealerweise so vorliegen, dass wir mit der Uhrmethode diesen ablesen können (also dass kein Streichholz mit der X-Achse einen 10° Winkel bildet usw)

- Nun wird jeder Punkt durch ein Dictionary dargestellt: die Keys sind die markierten Streichhölzer, die diesen Punkt formen und deren Values die Richtung.
- Es liegt interessante Eigenschaften vor: wenn ein Streichholz zweimal auftaucht, muss die Differenz dessen Values sich 6 ergeben; die parallel legende Streichhölzer haben entweder gleiche Values oder die Differenz deren Values ergibt sich 6; usw
- Die Abbildung ist nunmal eine Liste der Dictionaries.

Die vorliegende Methode basiert darauf, dass aus der Abbildungen(vorher und nachher) die gleiche Anzahl von Streichhölzer (wird nun "lines" genannt) weggenommen werden. Der Rest der Abbildung (nenne ich “figures”) wird in eine riesige Liste hinzugefügt. Wenn ein figure in vorherigen Liste nun in nachherigen Liste auftaucht, ist die Aufgabe erledigt. Welche lines aus beiden Abbildungen weggezogen wurden, sind die, die umgelegt worden sind.

Methoden

- Z. 21 take_lines() nimmt eine Liste der dictionaries als Übergabe, z:B. `[{1:3, 0:12}, {0:6, 3:3, 4:1}, {4:7, 6:12, 5:5}, {5:11, 3:9, 2:6}, {2:12, 1:9}]` und gibt eine Liste der keys zurück, also z.B. `[0,1,2,3,4,5,6]`

- Z.25 remove_lines() nimmt folgende als Übergabe:
    - figure: `[{1:3, 0:12}, {0:6, 3:3, 4:1}, {4:7, 6:12, 5:5}, {5:11, 3:9, 2:6}, {2:12, 1:9}]`

- lines: `[0,1,2,3,4,5,6]`
- num_of_remove_lines: die Anzahl der Streichhölzer, die weggenommen werden sollen.

Z.27: itertools.combinations(lines, num_of_removed_lines) gibt wieder, auf wie viele verschiedene Arten man `num_of_removed_lines` bestimmte Objekte aus einer Menge von `lines` verschiedenen Objekten auswählen kann, also der Binomialkoeffizient dessen. Die Anzahl der Elemente in dieser Liste ist “len(lines) über len(num_of_removed_lines)”.
z.B. itertools.combinations(`[0,1,2,3,4,5,6], 3`)

```
>> 0,1,2   0,1,3   0,1,4   0,1,5…
```

Diese Liste ist sozusagen die Liste der weggenommenen Streichhölzer.

Z. 30: als manche Streichhölzer herausgezogen worden sind, dict1 stellt die, was übrig geblieben sind, dar. Danach wird diese dictionary sortiert, nämlich mit der Hoffnung, dass der Vergleich nachher zwischen mehreren Elementen vereinfacht wird. Die dictionaries werden dann in eine Liste hinzugefügt.

- Z. 39 compare_dict: vergleicht 2 dictionaries anhand ihren values und gibt einen boolean Wert wieder. Deswegen sollen die dictionaries vorher sortiert worden sein.

- Z.45 compare_list_dict: vergleicht dann 2 Listen von dictionaries. Jede dictionary von diesen 2 Listen werden mit Hilfe der oberen Methode vergleicht. Es ist möglich, eine leere Liste zu bekommen [ ], welche nicht hilfreich ist, von daher soll nur True wiedergegeben werden, wenn es keine leere Liste gibt.

- Z. 55 compare_figures: ist die Hauptmethode von dieser Aufgabe, zwar es wird hier nachgeprüft, ob zwei Abbildungen mit einer beliebigen Anzahl von verschobenen Streichhölzer zusammenpassen können.
    - Zuerst wird die Methode remove_lines auf beiden Abbildungen aufgerufen

      z.B. die einzelnen Elemente
      remove_lines(before0, take_lines(before0), 2) wären:

```
[{}, {4: 1, 3: 3}, {5: 5, 4: 7, 6: 12}, {2: 6, 3: 9, 5: 11},
{2: 12}]
[{1: 3}, {4: 1, 3: 3}, {5: 5, 4: 7, 6: 12}, {3: 9, 5: 11}, {1:
9}]
```

```
[{1: 3}, {4: 1}, {5: 5, 4: 7, 6: 12}, {2: 6, 5: 11}, {1: 9, 2:
12}]
[{1: 3}, {3: 3}, {5: 5, 6: 12}, {2: 6, 3: 9, 5: 11}, {1: 9, 2:
12}]
[{1: 3}, {4: 1, 3: 3}, {4: 7, 6: 12}, {2: 6, 3: 9}, {1: 9, 2:
12}]
[{1: 3}, {4: 1, 3: 3}, {5: 5, 4: 7}, {2: 6, 3: 9, 5: 11}, {1:
9, 2: 12}]
[{0: 12}, {4: 1, 3: 3, 0: 6}, {5: 5, 4: 7, 6: 12}, {3: 9, 5:
11}, {}]
[{0: 12}, {3: 3, 0: 6}, {5: 5, 6: 12}, {2: 6, 3: 9, 5: 11},
{2: 12}]
[{0: 12}, {4: 1, 3: 3, 0: 6}, {4: 7, 6: 12}, {2: 6, 3: 9}, {2:
12}]
[{1: 3, 0: 12}, {3: 3, 0: 6}, {5: 5, 6: 12}, {3: 9, 5: 11},
{1: 9}]
[{1: 3, 0: 12}, {4: 1, 3: 3, 0: 6}, {4: 7, 6: 12}, {3: 9}, {1:
9}]
[{1: 3, 0: 12}, {4: 1, 0: 6}, {4: 7, 6: 12}, {2: 6}, {1: 9, 2:
12}]
[{1: 3, 0: 12}, {4: 1, 0: 6}, {5: 5, 4: 7}, {2: 6, 5: 11}, {1:
9, 2: 12}]
[{1: 3, 0: 12}, {3: 3, 0: 6}, {6: 12}, {2: 6, 3: 9}, {1: 9, 2:
12}]
[{1: 3, 0: 12}, {3: 3, 0: 6}, {5: 5}, {2: 6, 3: 9, 5: 11}, {1:
9, 2: 12}]
[{1: 3, 0: 12}, {4: 1, 3: 3, 0: 6}, {4: 7}, {2: 6, 3: 9}, {1:
9, 2: 12}]
```

- In den nächsten beiden List-comprehensions beziehen sich a und b auf die dictionaries in den Listen in den beiden Abbildungen. Sie werden miteinander verglichen und die passenden Konfiguration werden in 2 Listen dann eingeordnet matched1 und matched2. Sobald eine der beiden Listen nicht leer ist, wird aus jeder Liste jeweils das erste Element zurückgegeben. Falls beide Liste doch leer ist, ist der Puzzle unmöglich zu lösen.

z.B. compare_figures(before0, after0, 3) würde geben:

```
([{0: 12}, {4: 1, 0: 6}, {5: 5, 4: 7}, {2: 6, 5: 11}, {2:12}],
[{'f': 12}, {'a': 1, 'f': 6}, {'b': 5, 'a': 7}, {'c': 6,
'b':11}, {'c': 12}, {}])
```

- Z. 67: move() nimmt folgendes als Übergabe:
    - figures bezieht sich auf das Ergebnis von der Funktion remove_lines
    - Die Abbildungen vor- und nachher.

  `take_lines(figures[0])` sowie `take_lines(figure_before)` geben die Liste der Streichhölzer wieder. `move_lines` stellt nun die Differenz zwischen diesen beiden Listen, oder mit anderen Worten, die umgelegenen Streichhölzer. Die gleiche Vorgehensweise wird bei der nachherigen Abbildung durchgeführt. Die Methode gibt dann die umgelegenen Streichhölzer wieder.

<u>Anmerkung</u>
Die Methode hat ein wesentliches Ausführungszeitproblem. Bei den ersten beiden Beispieldaten und dem letzten ist es möglich, die Lösung zu bekommen, bei den anderen hat die Methode 10+ Minuten gebraucht.