1. (3 Punkte) Das Backtracking-Muster aus dem Unterricht:

```
def back(a):
    if isSolution(a):
        solutions.append(a.copy())
        return
    for cand in candidates(a):
        if isGood(cand, a):
            a.append(cand)
            back(a)
            a.pop()
```

   (a) Warum wird `a.copy()` gespeichert und nicht `a`?

   (b) Wozu dient `a.pop()` nach dem rekursiven Aufruf?

   (c) Beim 8-Damen-Problem wird `back` nur 2057-mal aufgerufen, obwohl es 8^8 Möglichkeiten gibt, in jede Zeile eine Dame zu setzen. Woran liegt das?

Lösung:

(a) Es gibt nur eine einzige Liste `a`, die immer weiter verändert wird. Würde man `a` speichern, stünde in `solutions` mehrmals dieselbe Liste, und am Ende wäre sie leer. Die Kopie hält die Lösung fest.

(b) Damit wird die letzte Entscheidung zurückgenommen. Danach kann der nächste Kandidat an dieselbe Stelle gesetzt werden. Das ist das Zurückgehen beim Backtracking.

(c) `isGood` erkennt eine Sackgasse, sobald eine Dame eine andere bedroht. Dann wird die Teillösung nicht weiter ausgebaut, und alle Stellungen, die damit beginnen würden, werden gar nicht erst erzeugt.

2. (2 Punkte) Beim Backtracking für das 5-Damen-Problem wie im Unterricht wird irgendwann `back([0, 3, 1, 4])` aufgerufen. Was sind die Argumente für die 5 folgenden Aufrufe von `back`?

Lösung:

```
[0, 3, 1, 4, 2]
[0, 4]
[0, 4, 1]
[1]
[1, 3]
```

3. (2 Punkte) Beim Backtracking für das 6-Damen-Problem wie im Unterricht wird irgendwann `back([1, 3, 5, 0, 2])` aufgerufen. Was sind die Argumente für die 6 folgenden Aufrufe von `back`?

Lösung:

```
[1, 3, 5, 0, 2, 4]
[1, 3, 5, 0, 4]
[1, 3, 5, 2]
[1, 3, 5, 2, 4]
[1, 4]
[1, 4, 0]
```

4. (2 Punkte) Beim Backtracking für das Haus vom Nikolaus wie im Unterricht wird irgendwann `back([0, 1, 2, 3, 1, 4, 0, 3])` aufgerufen. Die Kandidaten werden in der Reihenfolge 0, 1, 2, 3, 4 ausprobiert. Was sind die Argumente für die 5 folgenden Aufrufe von `back`?



Lösung:

```
[0, 1, 2, 3, 1, 4, 0, 3, 4]
[0, 1, 2, 3, 1, 4, 3]
[0, 1, 2, 3, 1, 4, 3, 0]
[0, 1, 2, 3, 1, 4, 3, 0, 4]
[0, 1, 2, 3, 4]
```

5. (2 Punkte) Beim Backtracking für das Haus vom Nikolaus wie im Unterricht wird irgendwann `back([0, 4, 1, 3, 2, 1, 0])` aufgerufen. Die Kandidaten werden in der Reihenfolge 0, 1, 2, 3, 4 ausprobiert. Was sind die Argumente für die 5 folgenden Aufrufe von `back`?



Lösung:

```
[0, 4, 1, 3, 2, 1, 0, 3]
[0, 4, 1, 3, 2, 1, 0, 3, 4]
[0, 4, 1, 3, 4]
[0, 4, 3]
[0, 4, 3, 0]
```

6. (3 Punkte) Am ersten Abend sitzen n Jugendliche im Kreis ums Lagerfeuer, der Reihe nach mit 1 bis n nummeriert. Am zweiten Abend soll niemand neben einem seiner Nachbarn vom ersten Abend sitzen. Mit Backtracking werden alle Sitzordnungen gesucht. Eine Teillösung ist die Liste der Personen, die schon sitzen, im Uhrzeigersinn.
   (a) Wann ist ein Kandidat `cand` gut für die Teillösung `a`?
   (b) Welche Kandidaten sind für $n = 5$ und die Teillösung `[1, 3]` gut?
   (c) Für $n = 6$ ist `[1, 3, 5, 2, 4, 6]` keine Lösung, obwohl jeder Kandidat beim Einsetzen gut war. Warum nicht?

Lösung:

(a) Wenn `cand` noch nicht in `a` vorkommt und sein Abstand zum letzten Element von `a` weder 1 noch $n - 1$ ist. Abstand $n - 1$ haben 1 und n, die im Kreis auch Nachbarn waren.

(b) nur 5. Die 1 und die 3 sitzen schon, 2 und 4 waren Nachbarn der 3.

(c) Die Personen sitzen im Kreis, also sind auch die erste und die letzte Person der Liste Nachbarn. 1 und 6 waren am ersten Abend Nachbarn. Das muss `isSolution` zusätzlich prüfen.

7. (3 Punkte) Das 4 × 4-Sudoku wird mit Backtracking gelöst. Die Leerstellen werden zeilenweise von links oben nach rechts unten gefüllt, die Kandidaten sind 1, 2, 3, 4.

```
. . 3 .
4 . . .
. . 1 .
. 2 . .
```

   (a) Welche Kandidaten sind für die erste Leerstelle (links oben) gut?

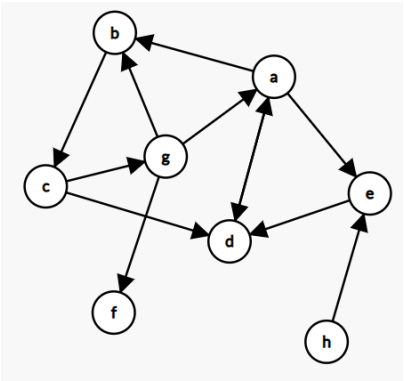
   (b) Gib die Lösung an.

Lösung:

(a) 1, 2. Die 3 steht schon in der Zeile, die 4 in der Spalte und im Quadrat.

(b)
```
2 1 3 4
4 3 2 1
3 4 1 2
1 2 4 3
```

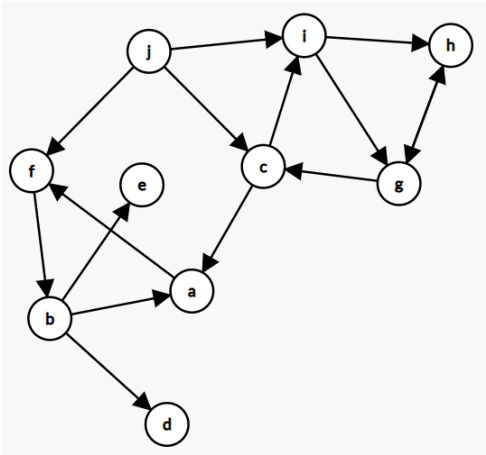
8. (2 Punkte) Notiere für alle einfachen Kreise in dem gerichteten Graphen ihre sortierte Kantenliste. Eine Kante von a nach b schreiben wir als ab.



Lösung:

```
ab bc cd da
ab bc cg ga
ad da
ae da ed
bc cg gb
```

9. (3 Punkte) In dem gerichteten Graphen wird von Knoten g aus ein Backtracking zum Finden aller einfachen Kreise gestartet, so wie im Unterricht. Die Nachfolger eines Knotens werden alphabetisch durchlaufen. Notiere die Argumente der Aufrufe von `back` bis zum Ende des Algorithmus. Wie viele Lösungen werden gefunden?



Lösung:

g
g c
g c a
g c a f
g c a f b
g c a f b d
g c a f b e
g c i
g c i g
g c i h
g c i h g
g h
g h g

Es werden 3 Lösungen gefunden.

10. (5 Punkte) Auf wie viele Arten kann man 10 Cent mit 1-, 2- und 5-Cent-Münzen bezahlen? Es kommt nicht auf die Reihenfolge der Münzen an, 5, 2, 2, 1 und 2, 5, 1, 2 zählen also nur einmal.

    Schreibe für das Backtracking-Muster aus dem Unterricht die Funktionen `isSolution`, `candidates` und `isGood`. Tipp: Notiere die Münzen absteigend.

Lösung:

```
def isSolution(a):
    return sum(a) == betrag

def candidates(a):
    return [5, 2, 1]

def isGood(cand, a):
    if sum(a) + cand > betrag:
        return False
    if len(a) > 0 and cand > a[-1]:       # absteigend, sonst doppelt
        return False
    return True

betrag = 10
solutions = []
back([])
print(len(solutions))
for s in solutions:
    print(*s)
```

Es gibt 10 Möglichkeiten. Weil die Münzen absteigend notiert werden, kommt jede Zusammenstellung nur einmal vor.