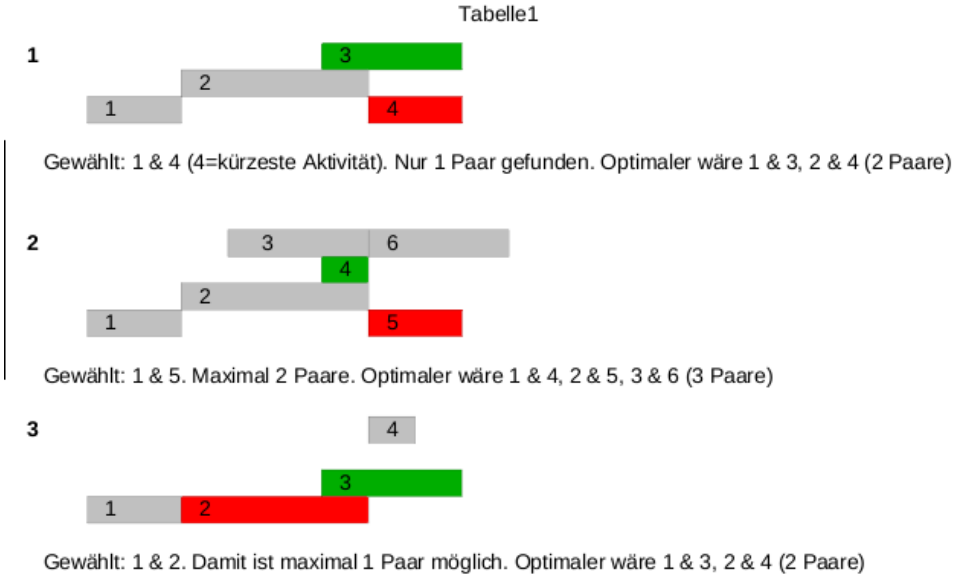
Theoretische Aufgaben

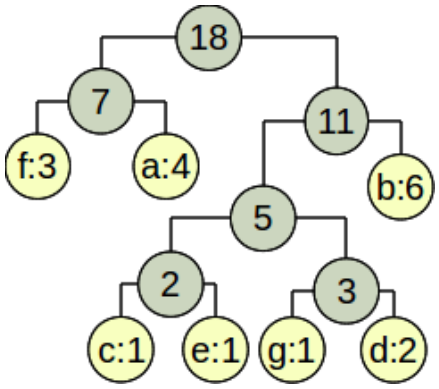
Aufgabe 1

Gesucht: *Maximale Menge paarweise zueinander kompatible Aktivitäten.*
Unterste Zeile: Ganze Auswahl. Rot: (nicht optimale) Auswahl der nächsten Aktivität. Grün: optimale Auswahl.



Aufgabe 2

Häufigkeit: `a:4, b:6, c:1, d:2, e:1, f:3, g:1`.
Sortiert: `c:1, e:1, g:1, d:2, f:3, a:4, b:6`.



Aufgabe 3

- a) Greedy-Algorithmus: Nimm die grösstmögliche Münze, die kleiner oder gleich gross ist wie der Gesamtbetrag.

Listing 1: Aufgabe 3

```
int[] chooseCoins(int n)
int[] coins = {25, 10, 5, 1}
int[] coinsChosen = {}
int moneyLeft = n

while moneyLeft > 0
   for i = 0 to coins.size
      if (coins[i] >= moneyLeft)
         coinsChosen.add(coins[i])
         moneyLeft -= coins[i]
         break

return coinsChosen
```

Optimale Teilstruktur
Das Wechselgeld-Problem hat in dieser Form eine optimale Teilstruktur. Wird von einem anfänglichen Betrag n der bestmögliche Betrag abgezogen, dann bleibt ein optimaler Teilbetrag, von dem wieder ein bester Betrag abgezogen werden kann. Die Teilbeträge sind alle für sich optimal, da mit dem Greedy-Algorithmus der beste Betrag gewählt wird.

Gierige-Auswahl-Eigenschaft
Für jeden Betrag gibt es eine optimale lösung, welche die gierige Auswahl enthält:
z.B. für Beträge ab 25 Rappen muss immer 25 Rappen enthalten sein. Wenn z.B. 40 Rappen in {10, 10, 10, 10} aufgeteilt wird, kann man die Beträge ersetzen mit {25, 10, 5}, um eine verbesserte Lösung zu enthalten.
Beispiel für Nennwert 10: 12 Rappen kann in {5, 5, 1, 1} unterteilt werden. {5, 5} kann jedoch durch {10} ersetzt werden (die optimale Lösung).
Beispiel für Nennwert 5: 8 Rappen kann man in {1, 1, 1, 1, 1, 1, 1, 1} aufteilen, aber au mit der gierigen Auswahl in {5, 1, 1, 1}.
Beispiel für Nennwer 1: 2 Rappen können nur in {1, 1} aufgeteilt werden, welches bereits die optimale Lösung ist und auch die gierige Auswahl enthält.

- b) Eine Menge mit {25, 10, 1} Rappen kann mit dem Greedy-Algorithmus zu einer suboptimalen Lösung führen. z.B. 30 Rappen wird in {25, 1, 1, 1, 1, 1} aufgeteilt, statt in optimal 3 Münzstücke {10, 10, 10}.

Listing 2: Aufgabe 3

```
int[] chooseCoinsDP(int n, int[] k)
int[] denom

c[0] = 0

//loop to n (possible money values)
for j = 1 to n

   //loop through DP-table
   for i = 0 to j

      //loop through coins available
      for i2 = 0 to k.size
        if c[i]+k[i2] == j
```

```
28                denom.add(k[i])
29                break
30
31    return denom
```

Aufruf, um Münzwert zu finden:

Listing 3: Aufgabe 3

```
32  //call, denom was calculated before
33  int[] getCoins(int n)
34  int[] coinsChosen
35  moneyLeft = n
36
37  while moneyLeft > 0
38      coinsChosen.add(denom[n])
39      moneyLeft - denom[n]
40
41  return coinsChosen
```

Praktische Aufgaben

Aufgabe 1, 2 3

Siehe Anhang `output.txt`, `HuffmannCode.java` und `Node.java`.