C.2:

Entwerfen und implementieren Sie verschiedene Algorithmen für MAX KS. Bleiben Sie jeweils möglichst nah am vorgegeben Entwurfsmuster und geben Sie zusätzlich eine optimale Lösung über **`print`** aus.

1. Implementieren Sie einen Algorithmus `max_ks_exhaustive` nach dem Muster *Exhaustive Search*. Wählen Sie einen geeigneten Iterator aus und geben Sie den optimalen Wert zurück. Analysieren Sie die Laufzeit.

```python
# zulaessige Loesung, auch fuer Teilloesungen
def sol_max_ks(s,v,S,t):
    size = 0
    for i in range(len(t)):             # Größen der Gegenstände addieren
        if t[i] == 1:
            size += s[i]

    if size <= S:           # Überprüfen ob Maximalgröße überschritten
        return True

    return False
```

```python
# Bewertungsfunktion, auch fuer Teilloesungen
def m_max_ks(s,v,S,t):
    value = 0
    for i in range(len(t)):             # Werte der Gegenstände addieren
        if t[i] == 1:
            value += v[i]

    return value

```

```python
# Entwurfsmuster Exhaustive Search
from itertools import product
def max_ks_exhaustive(s,v,S):
    opt = -1
    for t in product((0,1), repeat=len(s)):
        if sol_max_ks(s,v,S,t):                 # überprüfe ob t zulässige Lösung ist
            value = m_max_ks(s,v,S,t)
            if value > opt:                     # überprüfe ob t neue beste Lösung ist
                opt = value

    return opt
```

2. Wenden Sie das Entwurfsmuster *Backtracking* an und implementieren Sie einen Algorithmus `max_ks_backtracking`, der den optimalen Wert zurückgibt. Wählen Sie als vereinfachtes Backtracking-Kriterium die Einhaltung der Rucksackgröße durch die Teillösung.

```python
# vereinfachtes Backtracking-Kriterium:
# Teilloesung noch zulaessig?
def K_max_ks(s,v,S,t):
    size = 0
    for i in range(len(t)):         # Größen der Gegenstände addieren
        if t[i] == 1:
            size += s[i]

    if size <= S:                   # Überprüfen ob Maximalgröße überschritten
        return True

    return False
```

```python
# Entwurfsmuster Backtracking
def max_ks_backtracking(s,v,S):
    opt = -1                        # optimaler Wert
    m = len(s)                      # Anzahl der Gegenstände
    M = {()}                        # Menge der aktiven Knoten
                                    # () = Wurzel
    while M:
        t_prev = M.pop()            # beliebigen Knoten entnehmen
        for a in range(2):          # nächsten Gegenstand mitnehmen oder nicht
            t = t_prev + (a,)       # neues Tupel
            if len(t) == m:
                if sol_max_ks(s,v,S,t):         # überprüfe ob t zulässige Lösung ist
                    value = m_max_ks(s,v,S,t)
                    if value > opt:             # überprüfe ob t neue beste Lösung ist
                        opt = value

            else:
                if K_max_ks(s,v,S,t):
                    M.add(t)
                else:
                    pass
    return opt
```

3. Erweitern Sie ihren *Backtracking*-Algorithmus zu einem Algorithmus `max_ks_bab` nach dem Muster *Branch and Bound*. Verwenden Sie die Schrankenfunktion aus der Vorlesung.

```python
# obere Schranke:
# Bewertung der Teilloesung t + Werte aller restlichen Gegenstaende
def o_max_ks(s,v,S,t):
    max_value = m_max_ks(s,v,S,t)       # alle bisherigen Gegenstaende einpacken

    for i in range(len(t), len(s)):     # alle verbleibende Gegenstaende einpacken
        max_value += v[i]

    return max_value
```

```python
# Entwurfsmuster Branch and Bound
def max_ks_bab(s,v,S):
    opt = -1                        # optimaler Wert
    m = len(s)                      # Anzahl der Gegenstände
    M = deque({()})                 # Menge der aktiven Knoten
                                    # () = Wurzel
    while M:
        t_prev = M.pop()                # beliebigen Knoten entnehmen
        if o_max_ks(s,v,S,t_prev) > opt:
            for a in range(2):          # nächsten Gegenstand mitnehmen oder nicht
                t = t_prev + (a,)       # neues Tupel
                if len(t) == m:
                    if sol_max_ks(s,v,S,t):         # überprüfe ob t zulässige Lösung ist
                        value = m_max_ks(s,v,S,t)
                        if value > opt:             # überprüfe ob t neue beste Lösung ist
                            opt = value

                else:
                    if K_max_ks(s,v,S,t):
                        M.appendleft(t)
                    else:
                        pass
    return opt
```

4. Bessere obere Schranken sind *kleinere* obere Schranken, weil so mehr Teilbäume weggelassen werden können. Wie könnte die obere Schranke für MAX KS verbessert werden?

```
# Verbesserung der oberen Schranke:

# Problem: wir beachten das Gewicht in Schrankenfunktion garnicht
#   -> somit kann Schranke Wert annehmen, bei dem die Gegenstaende
#   weit über der Gewichtsgrenze liegen
#   -> gueltige Loesungen koennen diesen Wert nie erreichen
#   -> somit ist Schranke viel zu groß gewaehlt

# 1. Idee zur Schrankenverbesserung
#   -> so lange beliebige Einträge streichen, bis Gewichtgrenze nicht mehr
#    ueberschritten wird
#   -> funktioniert nicht, weil so mögliche Lösungen gestrichen werden könnten

# 2. Idee zur Schrankenverbesserung
#   -> nur "die besten Gegenstände" mitnehmen, sodass man unter Gewichtsgrenze
#   bleibt
#   -> damit machen wir das, was unser Algorithmus eigentlich machen soll

# 3. Idee zur Schrankenverbesserung
#   -> Kosten-Nutzen-Funktion erstellen, die zu jedem Zeitpunkt bewertet
#      wie die Kosten-Nutzen im Vergleich zur besten Lösung aussehen
```