D.1 Untersuchen in den folgenden Teilaufgaben das Problem MINIMUM INTERVAL PARTITIONING. Die Eingabe ist identisch zu Problem MAXIMUM INTERVAL SCHEDULING aus der Vorlesung, jedoch müssen jetzt *alle* m Intervalle berücksichtigt werden. Dafür stehen bis zu m identische Ressourcen zur Verfügung. Die einer Ressource zugeordneten Intervalle müssen wie gehabt überschneidungsfrei sein.

1. Modellieren Sie das Problem MINIMUM INTERVAL PARTITIONING: Was ist die minimale Anzahl notwendiger Ressourcen, um alle Anfragen zu erfüllen? Eine *zulässige Lösung* ist eine endliche Funktion, die von Intervallen auf Ressourcen abbildet und bestimmte Eigenschaften haben muss (welche?).

Die Intervalle werden von 0 bis $m-1$ nummeriert. Für jede Anfrage i wird eine Startzeit $s_i \in \mathbb{N}$ und eine Endzeit $f_i \in \mathbb{N}$ mit $s_i < f_i$ vorgegeben.

Wir nennen eine Menge von Intervallen kompatibel, falls je zwei Intervalle überschneidungsfrei sind.

Problem: MINIMUM INTERVAL PARTITIONING:

Eingabe: $m \geq 1$ Intervalle $(s_i, f_i) \in \mathbb{N}^2$ mit $s_i < f_i$ für $0 \leq i \leq m-1$

Lösung: Eine endliche, totale Funktion r, die alle gegebenen Intervalle auf Ressourcen abbildet, sodass die Intervalle in allen Ressourcen kompatibel sind

Maß: Kardinalität des Wertebereichs von r

2. Intervalle sind als Liste L vorgegeben. Implementieren Sie eine Funktion `compatible(L,M)`, die `True` zurückliefert falls die Intervallmenge $M \subseteq \{0, \ldots, m-1\}$ kompatibel ist, und `False` sonst.

```
# ist Intervallmenge M kompatibel?
# für jedes i,j mit i!=j in M muss gelten:
# si >= fj  oder  sj >= fi
def compatible(L,M):
    for i in M:
        for j in M:
            if i != j:
                (si, fi) = L[i]
                (sj, fj) = L[j]
                if (not si >= fj) and (not sj >= fi):
                    return False
    return True
```

3. Implementieren Sie eine Funktion `min_intpart_exhaustive`, die MINIMUM INTERVAL PARTITIONING nach dem Entwurfsmuster *Exhaustive Search* optimal löst. Analysieren Sie die Laufzeit.

```python
# zulaessige Loesung:
# - r ist stets total
# - jede Intervallmenge pro Ressource ist kompatibel
def sol_min_intpart(L,r):
    m = len(L)
    R = [set() for _ in range(m)]          # O(m)

    for i in range(m):                     # O(m)
        R[r(i)].add(i)

    for ressource in R:                    # O(m)
        if not compatible(L, ressource):   # O(m^2)
            return False

    return True
```

```python
# Bewertungsfunktion:
# - Kardinalitaet Wertebereich von r
def m_min_intpart(L,r):
    m = len(L)
    R = [set() for _ in range(m)]          # O(m)
    value = 0

    for i in range(m):                     # O(m)
        R[r(i)].add(i)

    for ressource in R:                    # O(m)
        if ressource != set():
            value += 1

    return value
```

```python
# Laufzeit: O(m^(m+2))
def min_intpart_exhaustive(L):
    m = len(L)
    opt = m
    r_opt = None

    # Erstelle alle möglichen Funktionen r und wähle diejenige
    # mit dem kleinsten Wertebereich
    for t in product([i for i in range(m)], repeat=m): #O(m^m)
        r = def_r(t)
        # Falls Lösung zulässig (Intervalle in Ressourcen sind kompatibel)
        if sol_min_intpart(L, r):        # O(m^3)
            deg = m_min_intpart(L,r)     # O(m)
            # Vergleiche Wertebereich von r mit Wertebereich der (bisher) optimalen Lösung
            if deg < opt:
                opt = deg
                r_opt = r                # optimale Funktion

    return opt
```