

Aufgabe:

Implementieren Sie eine Funktion "merge", die eine Liste von Intervallen entgegen nimmt und als Ergebnis wiederum eine Liste von Intervallen zusammenfaltet.

|| merge (intervals : Interval[]) : Interval[]

Im Ergebnis sollen alle sich überlappenden Intervalle zusammengeführt sein. Alle nicht überlappenden Intervalle bleiben unberücksichtigt.

Überlappung:

Zerlegung des Problems in Teilprobleme.

Problem I: gegeben zwei Intervalle A und B.
Überlappen diese Intervalle?

|| areOverlapping (a : Interval, b : Interval) : bool

Problem II: gegeben zwei überlappende
Intervalle A und B, wie mergen wir diese?

|| mergeTwo (a : Interval, b : Interval) : Interval

Problem III: gegeben N Intervalle, wie
vergleiche und n möglichst effizient alle
Intervalle so, dass keiner ausgelassen wird?

Offizieller Gedanke: gegeben N Intervalle $I_1 \dots I_n$,

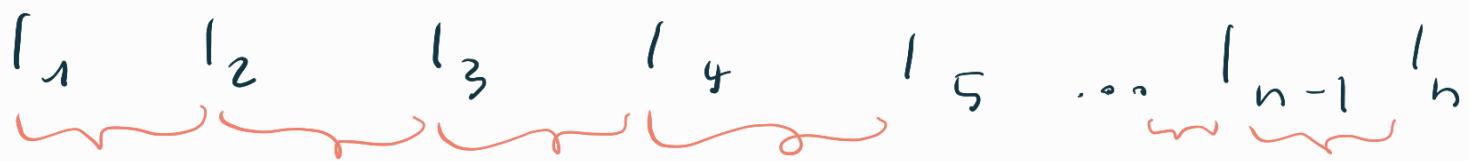
vergleichen und mergen wir paarweise

|| for i in $[1 \dots n]$:

|| if areOverlapping (I_i, I_{i+1}) :

|| mergeTwo (I_i, I_{i+1})

ffterher speichern wir aber nicht die
generierten Intervalle, sondern vergl. & wenden
die nur pauschal:



Zuletzt merken wir generelles:

merged = []

for i in [1, n-1]:

if areOverlapping(I_i, I_{i+1}):

merged.push(mergeTwo(I_i, I_{i+1}))

Zuletzt haben wir nur einig merges
gefunden, allerdings vergleichen wir wieder um
Nachbarn. Das ändern wir, steigen damit die
Komplexität auf $O(n^2)$:

merged = []

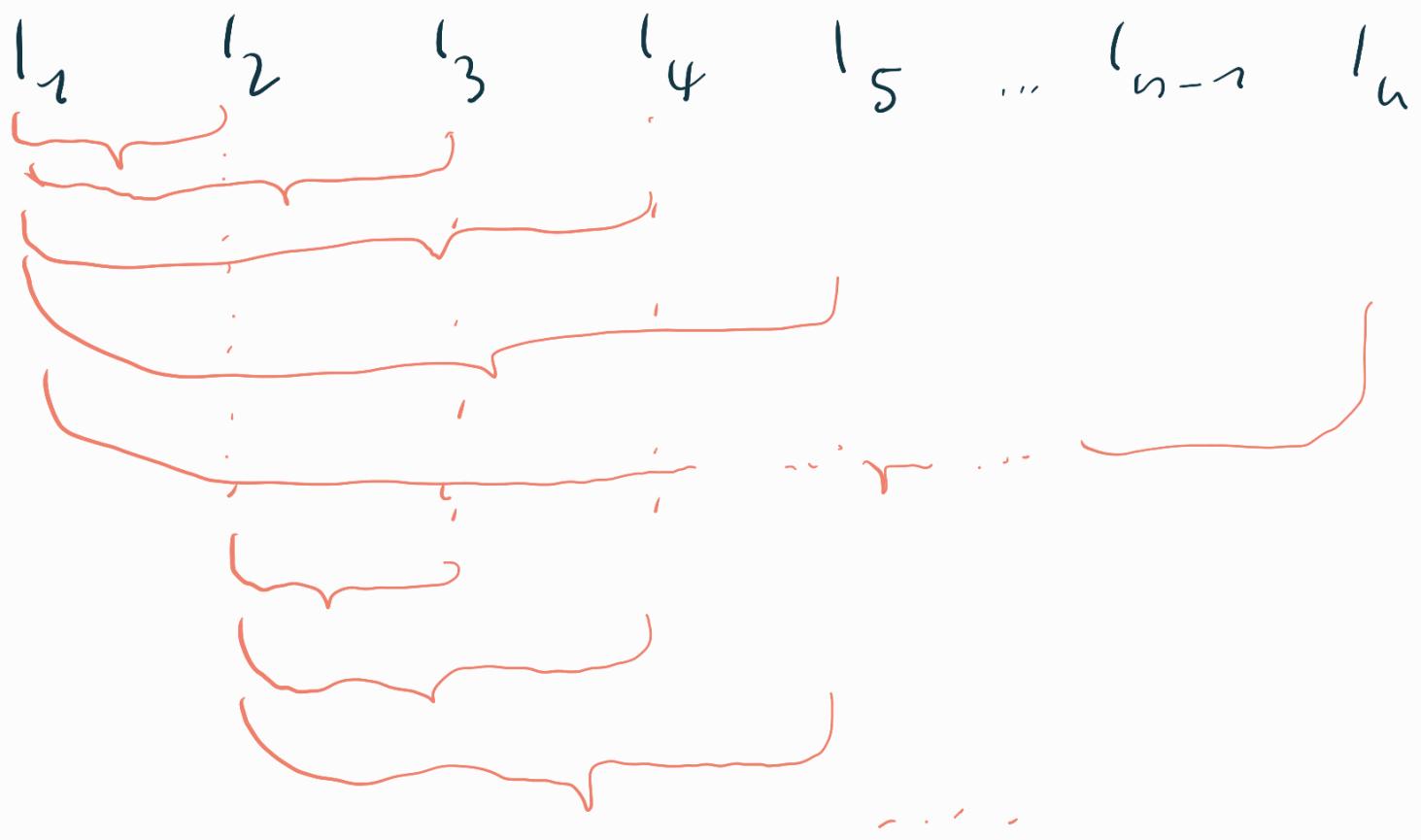
for i in [1, n-1]:

for j in [i, n]:

if areOverlapping(I_i, I_{j+1}):

merged.push(mergeTwo(I_i, I_{j+1}))

Jetzt vergleichen wir wie folgt:



Wir vergleichen alle mit allen.

Allerdings bedarf das arl., dass wir bereit genugte Intervalle einer vergleichen.

Idee: Sortierung könnte helfen

\Rightarrow Später ausdrucken.

Nun wollen wir bereit genugte Intervalle auf entfernen:

```

merged = []
for i in [1, n-1]:
    for j in [i, n]:
        if areOverlapping(l_i, l_{j+1}):
            merged.push(mergeTwo(l_i, l_{j+1}))

```

\downarrow Startt generisch zu speidern und
aus $[$ zu lösen, übersehen wir
einfach:

```

merged = []
for i in [1, n-1]:
    for j in [i, n]:
        if areOverlapping(l_i, l_{j+1}):
            l_i = mergeTwo(l_i, l_{j+1})
            l_i.drop(j+1)

```

Berechnet:

0. $[1, 5] [13, 18] [4, 8] [3, 15] [23, 80]$
- I. $\cancel{[1, 15]} \cancel{[13, 18]} \cancel{[23, 80]}$
- II. $[1, 18] [23, 80]$
- III. $[1, 18] [23, 80] \Rightarrow$ terminiert wenn kein merge

Jel ee: Wir können uns die innere for-Schleife sparen, wenn wir eine Invariante A einführen.

→ Die zu mergenden Intervalle sind nach dem ersten Element aufsteigend sortiert. Dies gewährleistet, dass sobald das zu einem Intervall $I(s_i, e_i) = [s_i, e_i]$ liegende Intervall $J(s_j, e_j) = [s_j, e_j]$ nicht mit I generiert wurde, die auf J folgende Intervalle und nicht mit I generiert werden können, da bereits für J gelten muss:

$$s_j > e_i$$

und durch die Sortierung gilt:

$$s_j \leq s_k, s_l, s_m \dots$$

Wir sortieren also vor dem mergen und passen die merge-Schleife entsprechend an:

$I = \text{Sort Ascending By Start Elements}(I)$

$\text{merged} = [I.\text{first}]$

for i in $[2, n-1]$:

$\text{lastInterval} = \text{merged}[\text{merged.length}]$

if oneOverlapping($\text{lastInterval}, I_i$):

$m = \text{mergeTwo}(\text{lastInterval}, I_i)$

$\text{merged}[\text{merged.length}] = m$

else:

$\text{merged.push}(I_i)$

return merged

Wir vergleichen hier in der merge-Schleife stets das letzte Element der bereits geworfenen Elemente mit den vergleichenden Elementen. Sobald ein Intervall I_i nicht mit dem zuletzt geworfenen Intervall gemerged werden kann, fügen wir I_i an die Liste der geworfenen Intervalle an. Invariante A gewährleistet nun, dass wir kein mergebares Intervall auslassen.