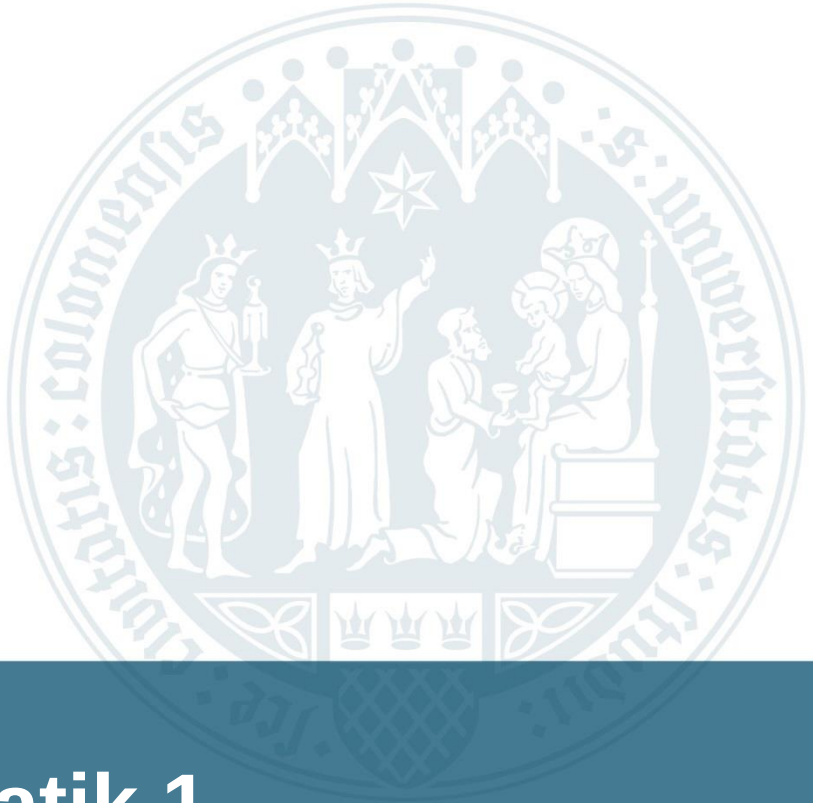
Universität zu Köln

Universität zu Köln



# Grundzüge der Informatik 1

Vorlesung 13

# Dynamische Programmierung

## Überblick

- Gierige Algorithmen
  - Entwurfsprinzip „gierige Algorithmen“
  - Beispiel: Zeitplanerstellung
  - Diskussion unterschiedlicher Strategien
  - Optimale Lösung durch gierigen Algorithmus

# Gierige Algorithmen

**Entwurfsprinzip „Gierige Algorithmen“**

- Ziel: Lösung eines Optimierungsproblems
- Herangehensweise: Konstruiere Lösung Schritt für Schritt, indem immer ein einfaches „lokales“ Kriterium optimiert wird
- Vorteil: Typischerweise einfache, schnelle und leicht zu implementierende Algorithmen
- Schwierigkeit: Löst der Algorithmus tatsächlich das Optimierungsproblem optimal?

# Gierige Algorithmen

## Einfaches Beispiel: Wechselgeldproblem

- Eingabe: Betrag in Cent zwischen 1 und 99
- Ausgabe: Minimale Anzahl Münzen (der Einfachheit halber aus: 50, 10, 5, 1), die benötigt werden, um Betrag darzustellen

## Gierige Strategie

- Für Betrag B wähle die größte Münze M mit $M \leq B$
- Zahle M aus und wende die gierige Strategie auf Betrag B-M an

# Gierige Algorithmen

## Beobachtung

- Der gierige Algorithmus löst das Wechselgeldproblem korrekt

## Beweisskizze

- Wir beobachten zunächst, dass der gierige Algorithmus für jeden Betrag eine Menge von Münzen findet, die diesem Betrag entspricht
- Die Münzen 50 und 5 kommen maximal einmal in einer optimalen Lösung vor
- Die Münzen 10 und 1 kommen maximal viermal in einer optimalen Lösung vor
- Damit kann mit den Münzen 10, 5 und 1 in einer opt. Lösung nur ein Wert bis zu 49 dargestellt werden
- Somit muss für B≥50 die Münze 50 in einer optimale Lösung genutzt werden

# Gierige Algorithmen

## Beobachtung

- Der gierige Algorithmus löst das Wechselgeldproblem korrekt

## Beweisskizze

- Mit den Münzen 5 und 1 kann in einer opt. Lösung nur ein Wert bis 9 darstellt werden und mit 1 nur bis zu 4
- Damit muss für $49 \geq B \geq 10$ und $9 \geq B \geq 5$ jeweils 10 und 5 in der optimalen Lösung sein
- Somit ist der Algorithmus korrekt

# Gierige Algorithmen

## Zweites Beispiel: Wechselgeldproblem mit 7-Cent Münze

- Eingabe: Betrag in Cent zwischen 1 und 99
- Ausgabe: Minimale Anzahl Münzen (aus: 50, 10, 7, 5, 1), die benötigt werden, um Betrag darzustellen

## Gierige Strategie funktioniert hier nicht!

- Für Betrag B=14 liefert die gierige Strategie 10+1+1+1+1
- Mann kann aber B=7+7 darstellen

# Gierige Algorithmen

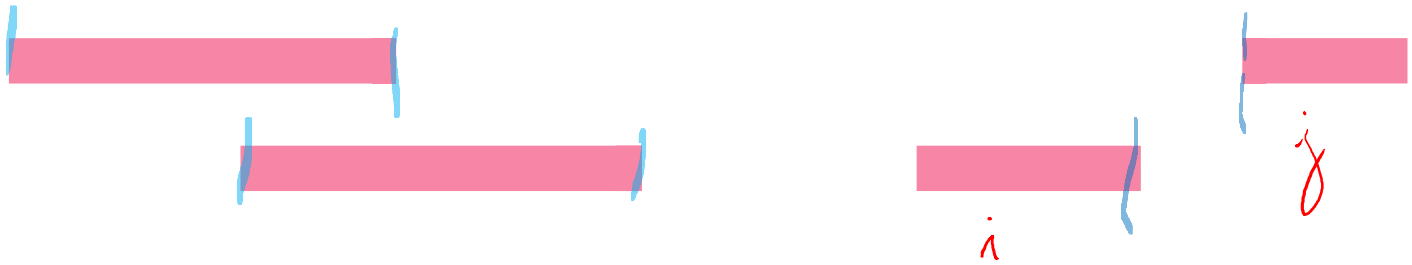
## Erstes Fazit

- Gierige Algorithmen optimieren einfaches lokales Kriterium
- Dadurch werden nicht alle möglichen Lösungen betrachtet
- Dies macht die Algorithmen oft schnell
- Je nach Problem und Algorithmus kann die optimale Lösung übersehen werden

# Gierige Algorithmen

## Intervall Zeitplanerstellung (Scheduling)

- Motivation: Ressource (Maschine, Hörsaal, Parallelrechner, etc.) soll möglichst gut genutzt werden
- Eingabe: Anzahl Intervalle n, Felder A und E, so dass A[i] den Anfangszeitpunkt des i-ten Intervalls und E[i] seinen bezeichnet ($1 \leq i \leq n$)
- Ausgabe: Menge $S \subseteq \{1,..,n\}$ von Intervallen, so dass $|S|$ maximiert wird unter der Bedingung, dass für alle $i,j \in S$, $i \neq j$, $E[i] \leq A[j]$ oder $E[j] \leq A[i]$ gilt (die Intervalle überlappen nicht)



# Gierige Algorithmen

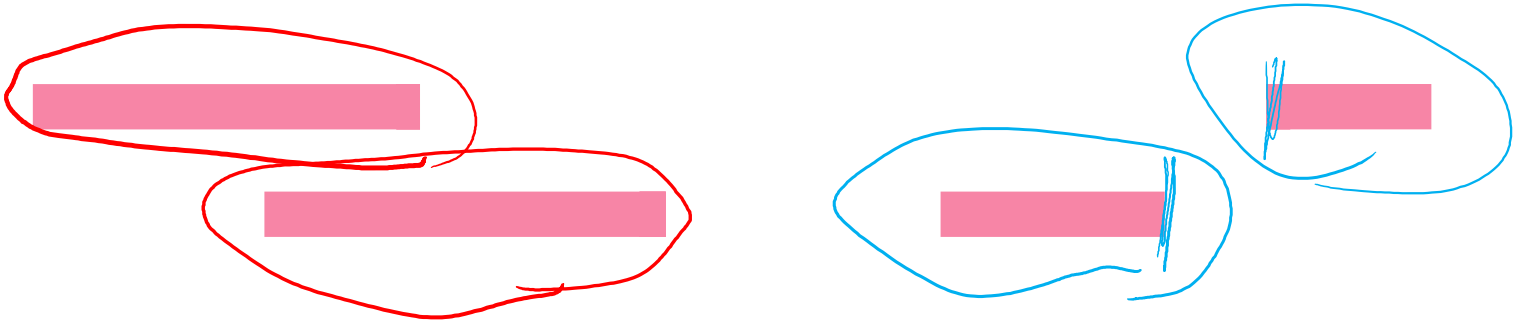
## Intervall Zeitplanerstellung (Scheduling)

- Motivation: Ressource (Maschine, Hörsaal, Parallelrechner, etc.) soll möglichst gut genutzt werden
- Eingabe: Anzahl Intervalle n, Felder A und E, so dass A[i] den Anfangszeitpunkt des i-ten Intervalls und E[i] seinen bezeichnet ($1 \leq i \leq n$)
- Ausgabe: Menge $S \subseteq \{1,..,n\}$ von Intervallen, so dass $|S|$ maximiert wird unter der Bedingung, dass für alle $i,j \in S$, $i \neq j$, $E[i] \leq A[j]$ oder $E[j] \leq A[i]$ gilt (die Intervalle überlappen nicht)

# Gierige Algorithmen

**Definition**

- Zwei Intervalle i,j, i≠j, heißen kompatibel wenn A[i]≥E[j] oder A[j]≥E[i] gilt.



# Gierige Algorithmen

## Generelle Überlegung (Gierige Algorithmen)

- j=1
- Wiederhole bis Intervalle mehr vorhanden sind:
  - Wähle Intervall $i_j$ geschickt und füge es in S ein * **Auswahlschritt**
  - Entferne alle Intervalle, die nicht mit $i_j$ kompatibel sind
  - Erhöhe j um 1

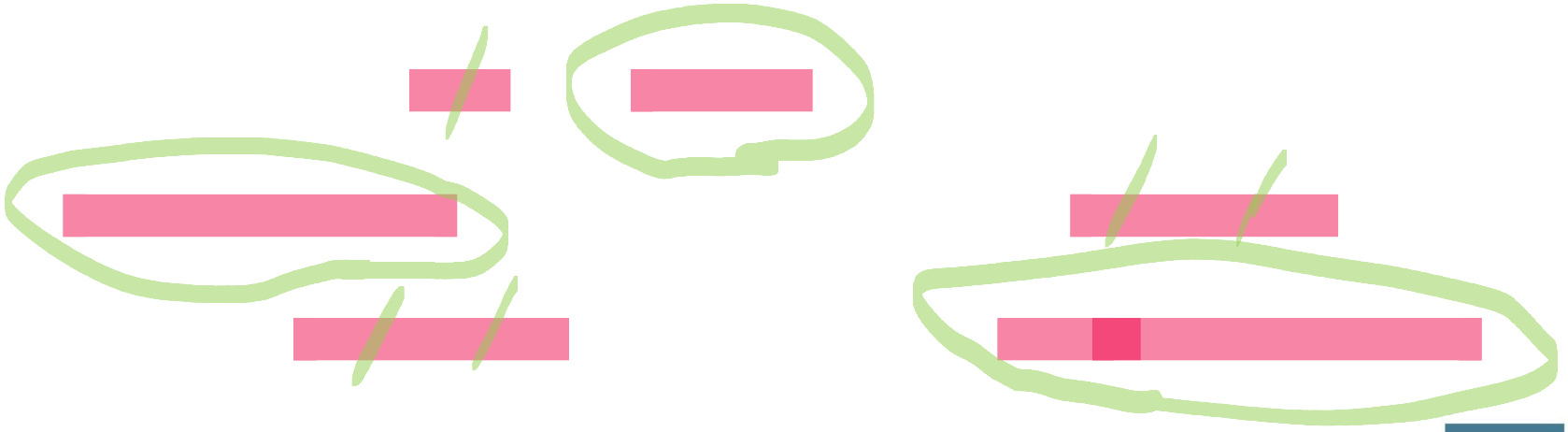
## Herausforderung

- Wie wähle ich das nächste Intervall

# Gierige Algorithmen

## Aufgabe 1

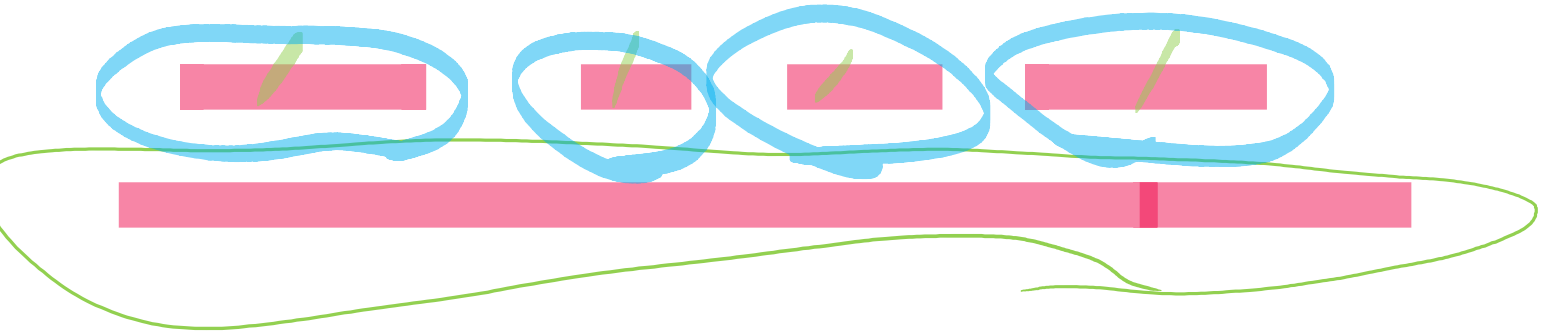
- Betrachten Sie folgende gierige Strategie:
- Wähle immer das Intervall, was am frühesten startet (wir wollen die Ressource möglichst früh auslasten)
- Frage: Liefert diese Strategie eine optimale Lösung?



# Gierige Algorithmen

**Strategie 1 liefert keine optimale Lösung**

- Wähle immer das Intervall, was am frühesten startet (wir wollen die Ressource möglichst früh auslasten)



# Gierige Algorithmen

## Aufgabe 2

- Betrachten Sie folgende gierige Strategie:
- Wähle immer das kürzeste Intervall
  (wir wollen die Ressource möglichst wenig nutzen)
- Frage: Liefert diese Strategie eine optimale Lösung?

# Gierige Algorithmen

**Strategie 2 liefert keine optimale Lösung**

- Wähle immer das kürzeste Intervall
  (wir wollen die Ressource möglichst wenig nutzen)

# Gierige Algorithmen

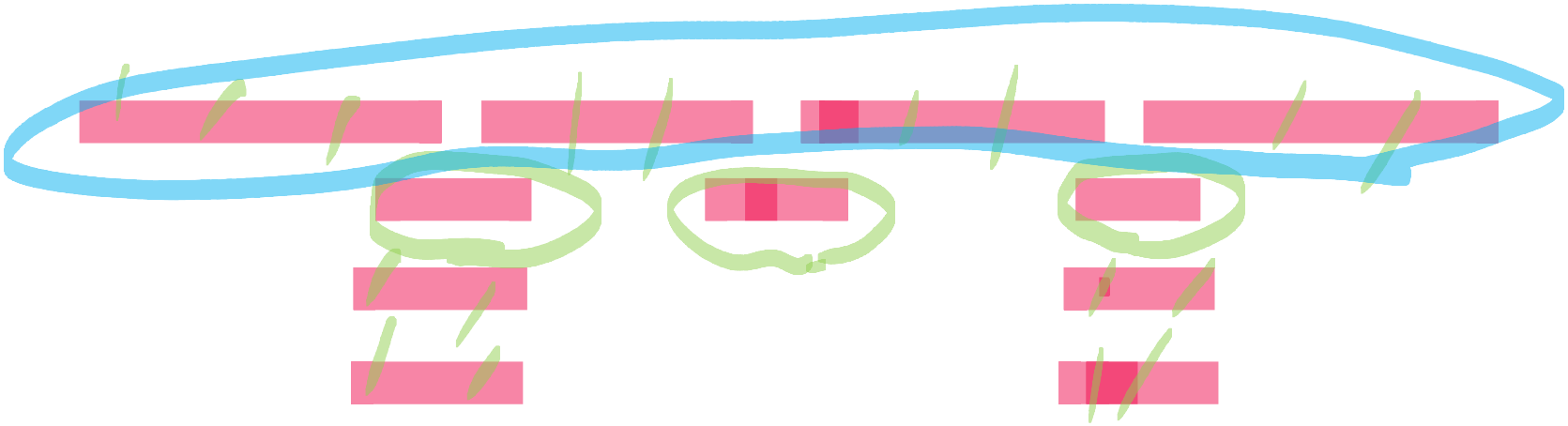
## Aufgabe 3

- Betrachten Sie folgende gierige Strategie:
- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen; bei Gleichheit wähle das kürzeste Intervall
  (wir wollen die Ressource möglichst wenig nutzen)
- Frage: Liefert diese Strategie eine optimale Lösung?

# Gierige Algorithmen

**Strategie 3 liefert keine optimale Lösung**

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen; bei Gleichheit wähle das kürzeste Intervall
  (wir wollen die Ressource möglichst wenig nutzen)



# Gierige Algorithmen

## Zwischenfazit

- Die Wahl einer geeigneten Strategie ist nicht einfach
- Auf den ersten Blick plausibel erscheinende Strategien sind u.U. nicht sinnvoll

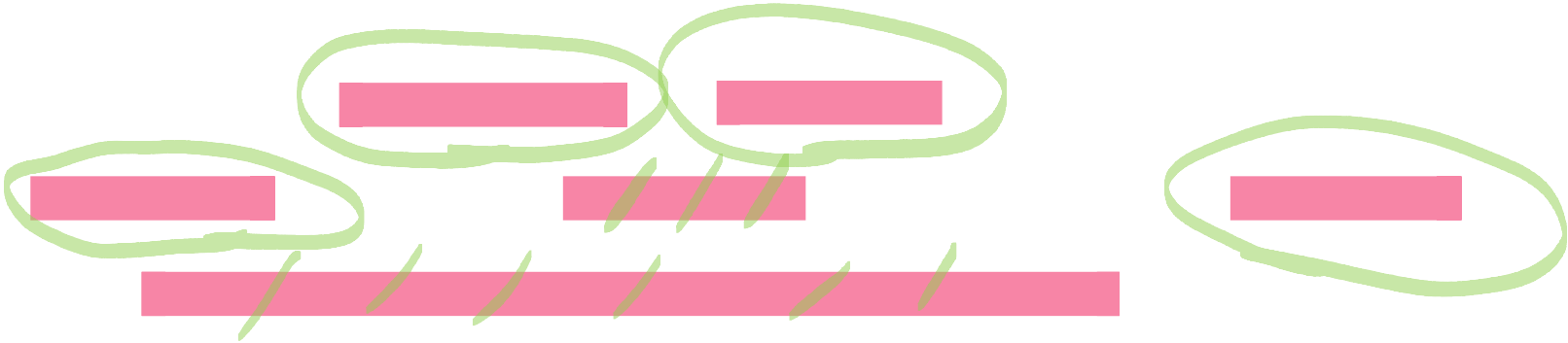
## Gierige Algorithmen

- Einfache Algorithmen
- Leicht zu implementieren

# Gierige Algorithmen

**Wie können wir das erste Intervall wählen?**

- Idee: Wir müssen die Ressource möglichst bald wieder freigeben
- Nimm das Intervall mit dem frühesten Endzeitpunkt
  (und entferne dann alle nicht kompatiblen Intervalle)

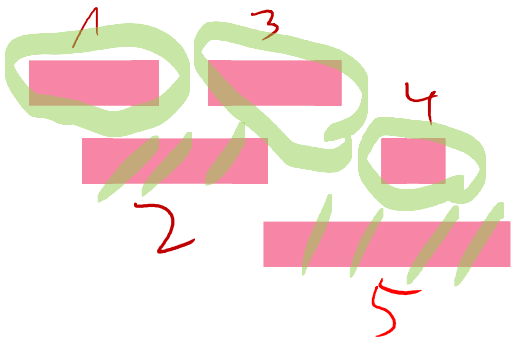


# Gierige Algorithmen

IntervalScheduling(A,E,n)

1. S={1}
2. j=1
3. **for** i=2 **to** n **do**
4. **if** A[i] ≥ E[j] **then**
5. S=S∪{i}
6. j=i
7. **return** S

| A | 1 | 2 | 4 | 7 | 5 |
|---|---|---|---|---|---|
| E | 3 | 5 | 6 | 8 | 9 |




**Annahme:**

- Intervalle nach Endzeitpunkt sortiert

# Gierige Algorithmen

**Beweisidee: Der gierige Algorithmus „liegt vorn“**

- Wir vergleichen eine optimale Lösung mit der Lösung des gierigen Algorithmus zu verschiedenen Zeitpunkten
- Wir zeigen: Die Lösung des gierigen Algorithmus is bzgl. eines bestimmten Kriteriums mindestens genauso gut wie die optimale Lösung

**Vergleichszeitpunkte**

- Nach Hinzufügen des i-ten Intervalls zur aktuellen Lösung beider Algorithmen

**Vergleichkriterium**

- Maximaler Endzeitpunkt der bisher ausgewählten Intervalle

# Gierige Algorithmen

## Beobachtung

- S ist eine Menge von kompatiblen Intervallen.

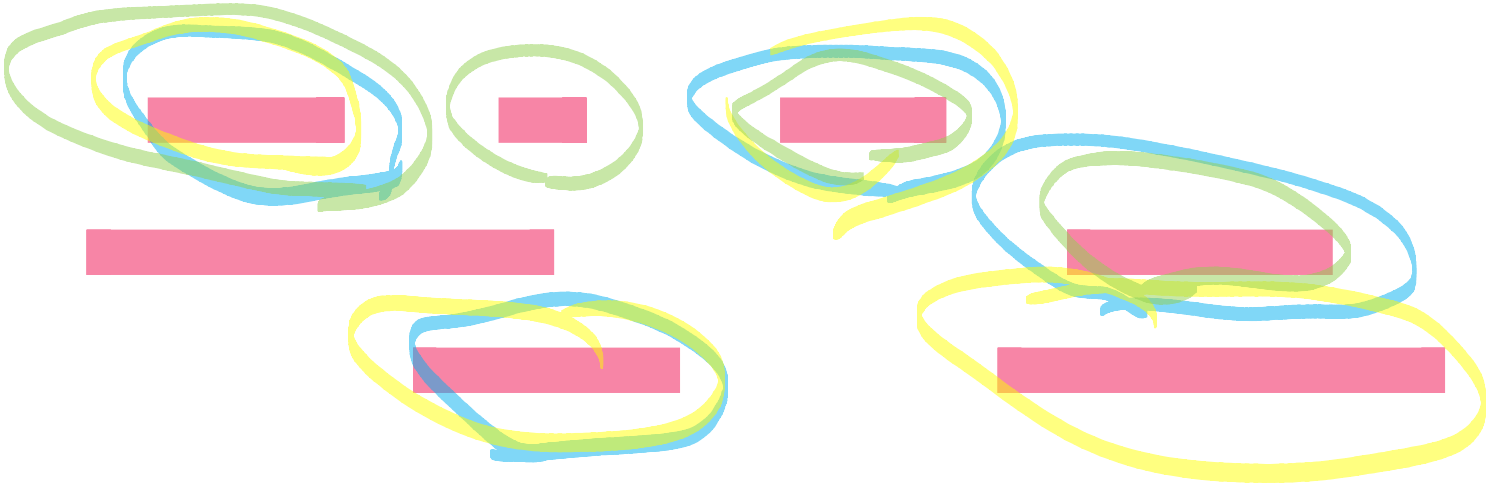
## Beweis

- Der Algorithmus erhält die Invariante aufrecht, dass j das Intervall aus S ist, was den größten Endzeitpunkt hat
- Ein Intervall wird nur zu S hinzugefügt, wenn sein Startzeitpunkt mindestens so groß ist wie der Endzeitpunkt von Intervall j
- Damit ist das zugefügte Intervall kompatibel zu allen Intervallen in S

# Gierige Algorithmen

**Wie können wir Optimalität zeigen?**

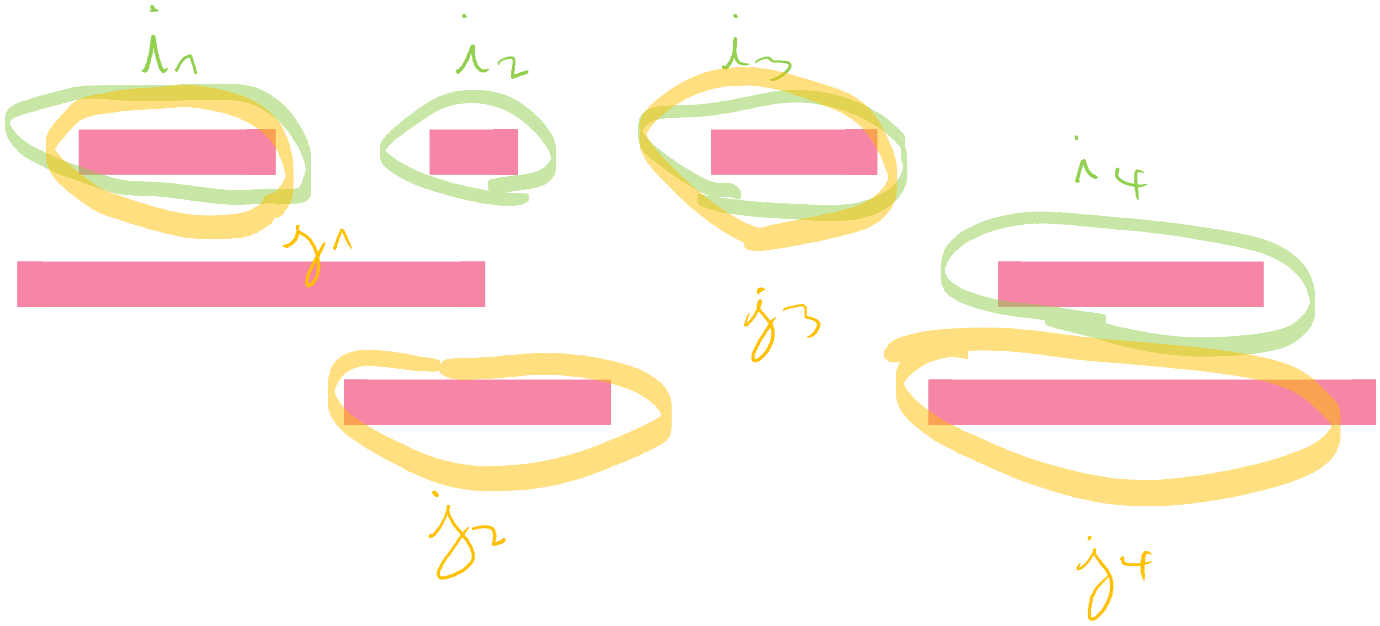
- Sei O eine optimale Menge von Intervallen
- U.U. viele unterschiedliche optimale Lösungen
- Wir wollen zeigen: |S|=|O|



# Gierige Algorithmen

**Notation**

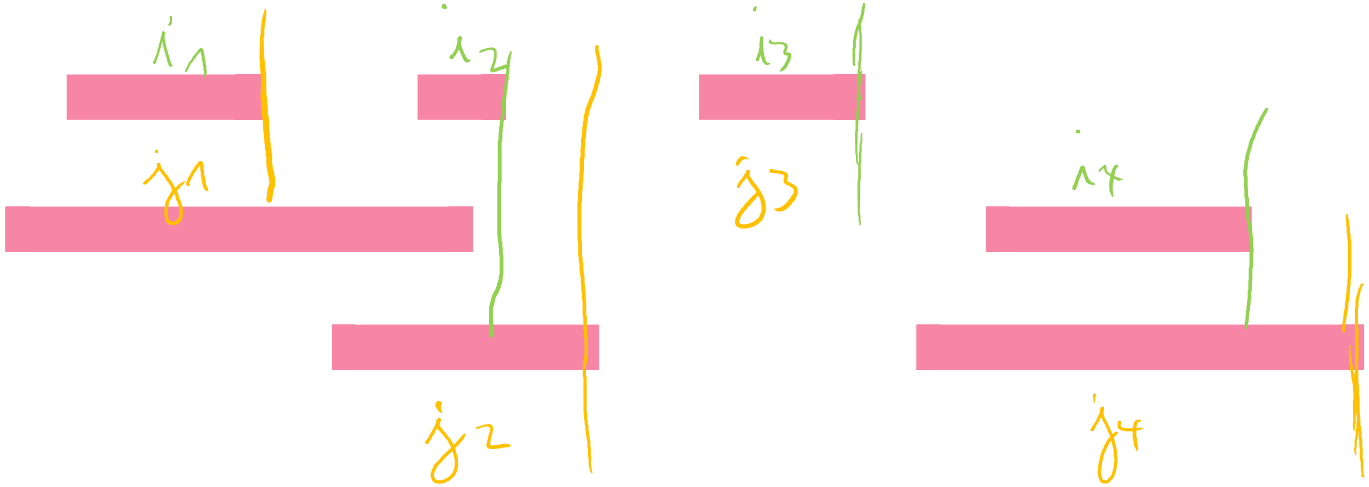
- $i_1,\ldots,i_k$: Intervalle von S in Ordnung des Hinzufügens
- $j_1,\ldots,j_m$: Intervalle von O sortiert nach Endpunkt
- Zu zeigen: k=m



# Gierige Algorithmen

**Der gierige Algorithmus liegt vorn**

- Idee des Algorithmus: Die Ressource soll so früh wie möglich wieder frei werden
- Dies ist wahr für das erste Intervall: $f[i_1] \leq f[j_1]$
- Zu zeigen: Gilt für alle Intervalle 1,…,k



# Gierige Algorithmen

**Lemma 13.1**

- Für alle $r \leq k$ gilt $f[i_r] \leq f[j_r]$.

**Beweis** (Induktion über r)

- Induktionsanfang: Für r=1 ist die Aussage offensichtlich korrekt
- Induktionsannahme: Die Aussage gelte für r-1
- Induktionsschluss: Nach Induktionsannahme gilt $f[i_{r-1}] \leq f[j_{r-1}]$
- Da die Intervalle in O kompatibel sind, gilt $f[j_{r-1}] \leq s[j_r]$ und somit auch $f[i_{r-1}] \leq s[j_r]$
- Damit ist $j_r$ in der Menge der Intervalle, die mit den ersten r-1 Intervallen kompatibel sind, die IntervalScheduling ausgewählt hat
- Da der Algorithmus das Interval mit kleinstem f-Wert auswählt, gilt $f[i_r] \leq f[j_r]$

# Gierige Algorithmen

## Satz 13.2

- Die von Algorithmus IntervalSchedule berechnete Lösung S ist optimal.

## Beweis durch Widerspruch

- Annahme: S ist keine optimale Lösung
- Ist S nicht optimal, so hat O mehr Intervalle, d.h. es gilt m>k. Nach unserem Lemma mit r=k gilt $f[i_k] \leq f[j_k]$
- Da m>k gibt es ein Intervall $j_{k+1}$ in O, das startet, nachdem $j_k$ und somit auch $i_k$ endet, d.h. $s[j_{k+1}] \geq f[i_k]$. Außerdem gilt natürlich $f[j_{k+1}] > s[j_{k+1}] \geq f[i_k]$
- Betrachten wir nun den Zeitpunkt, zu dem IntervalScheduling Intervall $i_k$ in S aufnimmt. Da die Intervalle nach Endzeitpunkten sortiert sind, wurde $j_{k+1}$ noch nicht betrachtet.

# Gierige Algorithmen

**Satz 13.2**

- Die von Algorithmus IntervalSchedule berechnete Lösung S ist optimal.

**Beweis durch Widerspruch**

- Da kein weiteres Intervall in S aufgenommen wird, muss für alle noch nicht betrachteten Intervalle der Startzeitpunkt vor $f[i_k]$ liegen
- Widerspruch, denn wir haben bereits gezeigt, dass $s[j_{k+1}] \geq f[i_k]$ gilt

# Gierige Algorithmen

IntervalScheduling(A,E,n)

1. S={1}
2. j=1
3. **for** i=2 **to** n **do**
4. **if** A[i] ≥ E[j] **then**
5. S=S∪{i}
6. j=i
7. **return** S


**Annahme:**

- Intervalle nach Endzeitpunkt sortiert

# Gierige Algorithmen

**Satz 13.3**

- Algorithmus IntervalSchedule berechnet in O(n) Zeit eine optimale Lösung, wenn die Eingabe nach Endzeit der Intervalle (rechter Endpunkt) sortiert ist. Die Sortierung kann in O(n log n) Zeit berechnet werden.

# Gierige Algorithmen

## Zusammenfassung

- Entwurfprinzip “gierige Algorithmen”
  - Löse globales Optimierungsproblem mit Hilfe von einfachen, lokalen Optimierungsstrategien
- Gierige Algorithmen sind einfach und leicht zu implementieren
- Es ist oft schwierig einen optimalen gierigen Algorithmus zu finden
- Korrektheitsbeweis erfolgt häufig mit Hilfe eines Widerspruchsbeweises
- Beispiel Zeitplanerstellung:
  - Drei verschiedenen plausible, aber nicht optimale Strategien
  - Vierte Strategie hat funktioniert

# Referenzen


- J. Kleinberg, E. Tardos. Algorithm Design. Pearson, 2006.