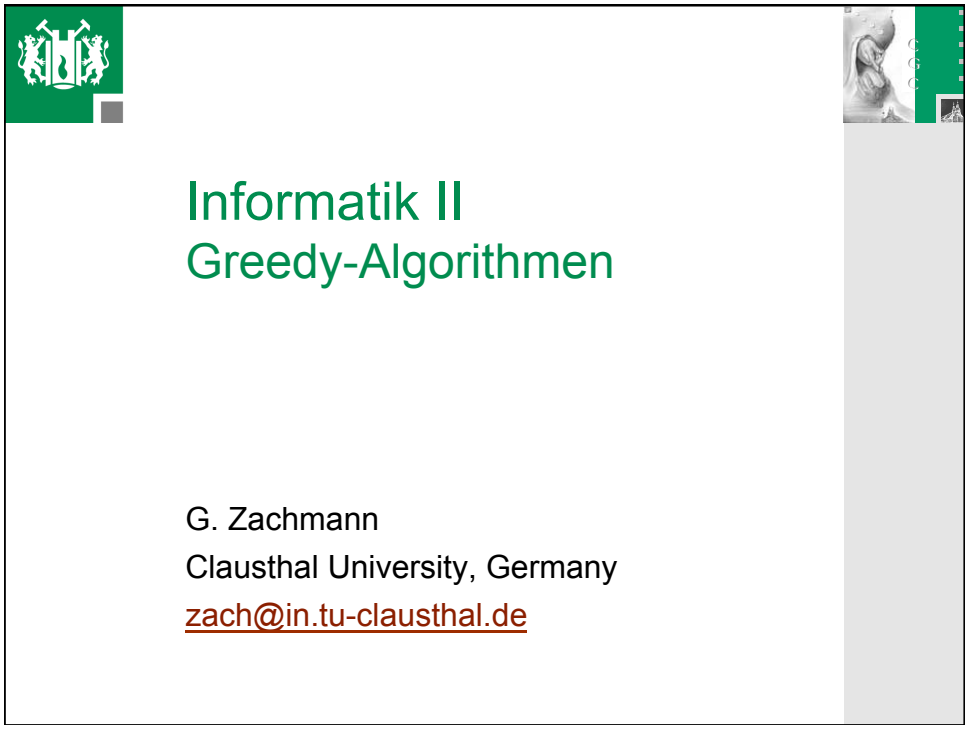
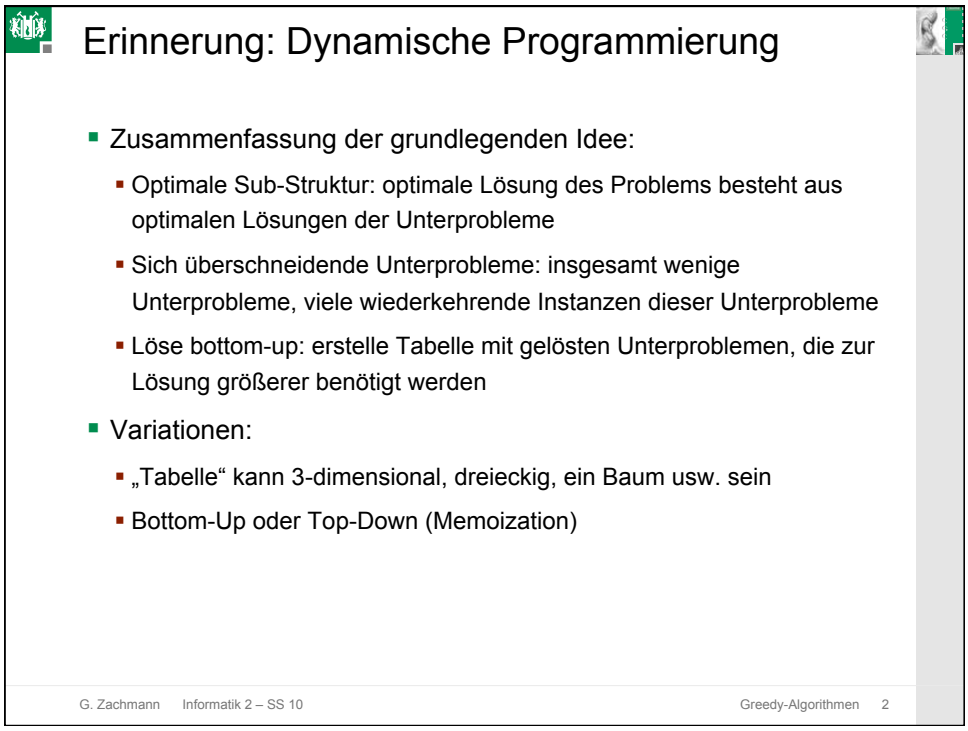


Informatik II
Greedy-Algorithmen

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de



Erinnerung: Dynamische Programmierung

- Zusammenfassung der grundlegenden Idee:
  - Optimale Sub-Struktur: optimale Lösung des Problems besteht aus optimalen Lösungen der Unterprobleme
  - Sich überschneidende Unterprobleme: insgesamt wenige Unterprobleme, viele wiederkehrende Instanzen dieser Unterprobleme
  - Löse bottom-up: erstelle Tabelle mit gelösten Unterproblemen, die zur Lösung größerer benötigt werden
- Variationen:
  - „Tabelle“ kann 3-dimensional, dreieckig, ein Baum usw. sein
  - Bottom-Up oder Top-Down (Memoization)

Greedy-Algorithmen

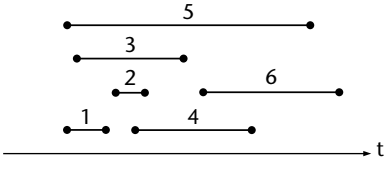
- "Greedy" = gierig
- Grundidee:
  - Konstruiere Lösung iterativ (keine Rekursion)
  - Wähle in jedem Schritt die am besten erscheinende Alternative
  - Eine Entscheidung wird nie revidiert ("blicke nicht zurück")
  - Die Hoffnung: eine lokal optimale Lösung führt zu einer global optimalen Lösung
- Dynamische Programmierung kann "Overkill" sein, Greedy-Algorithmen sind oft einfacher zu implementieren
- Greedy-Algorithmen sind nicht immer korrekt / optimal / gut

Das Activity-Selection-Problem (ASP)

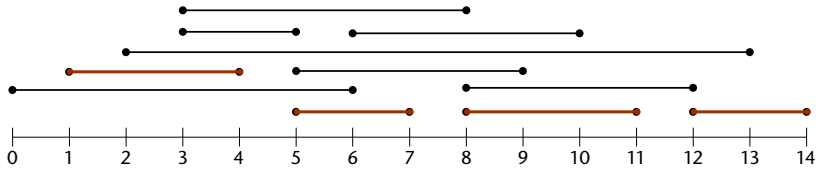
- Beispiel: in einem Vergnügungspark das "Meiste mitnehmen"
  - Eine Karte ermöglicht das Benutzen aller Attraktionen
  - Attraktionen starten und enden zu unterschiedlichen Zeiten
  - Ziel: so viele Attraktionen wie möglich besuchen (ein anderes Ziel wäre, so viel Zeit wie möglich in Attraktionen zu verbringen)

⇒Activity-Selection-Problem

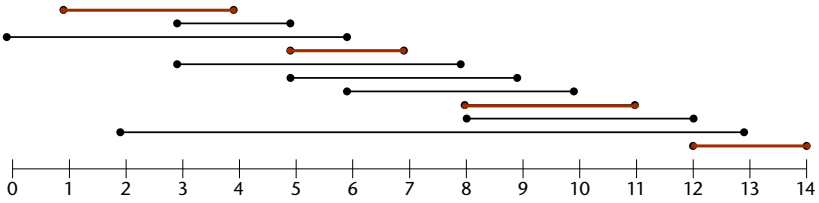
- Formal: sei $S = \{ (s_i, f_i) \mid i = 1 \ldots n \}$ eine Menge von n Aktivitäten
  - s_i / f_i = Startzeit / Endzeit von Aktivität i
  - Aufgabe: finde größte Menge $A \subseteq S$ mit kompatiblen Aktivitäten
  - OBdA sei $f_1 \leq f_2 \leq \ldots \leq f_n$
    - Falls nicht: sortiere Aktivitäten in $O(n \log n)$ oder $O(n)$ gemäß f_i



Beispiel



Wieviel Aktivitäten können ausgeführt werden?



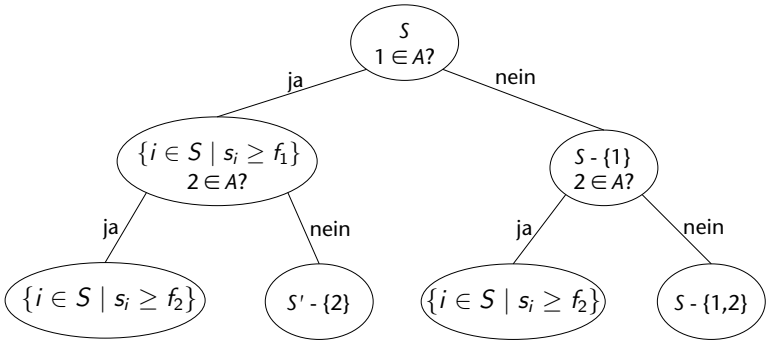
Wie lässt sich die Korrektheit beweisen?

Die optimale Unterstruktur

- Behauptung:
  Sei A eine optimale Lösung und k die minimale Aktivität in A (d.h. genau die mit kleinstem f_k), dann ist $A \setminus \{k\}$ eine maximale Lösung für $S' = \{i \in S \mid s_i \geq f_k\}$
- In Worten: wenn Aktivität $a_1=(s_k,f_k) \in S$ (richtig) gewählt ist, reduziert sich das Problem darauf, die maximale Lösung für diejenigen Aktivitäten $S' \subseteq S$ zu finden, die zu Aktivität a_1 "passen"
- Beweis:
  - Ann.: wir finden B = maximale Lösung zu S' mit $|B| > |A \setminus \{k\}|$
  - Dann passt Aktivität k auch zu B, und $B \cup \{k\}$ ist maximale Lösung zu S
  - $|B \cup \{k\}| > |A|$
  - A war nicht maximale Lösung

Wiederkehrende Unterprobleme

- Betrachte den rekursiven Algorithmus, der alle verträglichen Untermengen untersucht, um die maximale Menge zu finden:



- Beachte die sich wiederholenden Unterprobleme:
- Dynamische Programmierung? Memoisierung? Ja, aber …

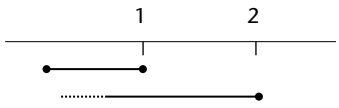
Die Greedy-Choice-Eigenschaft

- Das Actitity-Selection-Problem zeigt auch die Greedy-Choice-Eigenschaft:

  Lokal optimale Auswahl ⇒ global optimale Lösung

➔Lemma:
wenn S ein, nach der Endzeit sortiertes, Activity-Selection-Problem ist, dann ex. eine optimale Lösung $A \subseteq S$, so daß $\{ (s_1,f_1) \} \in A$

- Beweisskizze:
  - Wenn eine optimale Lösung B existiert, die $\{ (s_1,f_1) \}$ nicht enthält, kann die erste Aktivität in B (z.B. (s_2,f_2)) immer durch (s_1,f_1) ersetzt werden
  - Gleiche Anzahl an Aktivitäten, also immer noch optimal



Ein greedy Algorithmus für das ASP

- Eigentlicher Algorithmus ist einfach:
  1. Sortiere Aktivitäten nach ihrer Endzeit f_i
  2. Lege die erste Aktivität fest (also (s_1,f_1), d.h. diejenige, die am frühesten wieder endet)
  3. Entferne alle Aktivitäten aus der sortierten Liste, die vor der Endzeit von f_1 starten (also nicht kompatibel sind)
  4. Wiederhole, bis keine Aktivitäten mehr übrig sind
- Intuition ist noch einfacher: nimm immer diejenige Aktivität mit der geringsten Dauer, die gerade als nächstes beginnt

- Der Algorithmus als Python-Programm:

```
sortiere die Arrays s & f bzgl f[i]
A = []   # solution array
i = 0    # = last finished activity
for k in range( 1, n ):
    if s[k] >= f[i]:
        # found next compatibel act.
        A.append( (s[k],f[k]) )
        i = k
return A
```