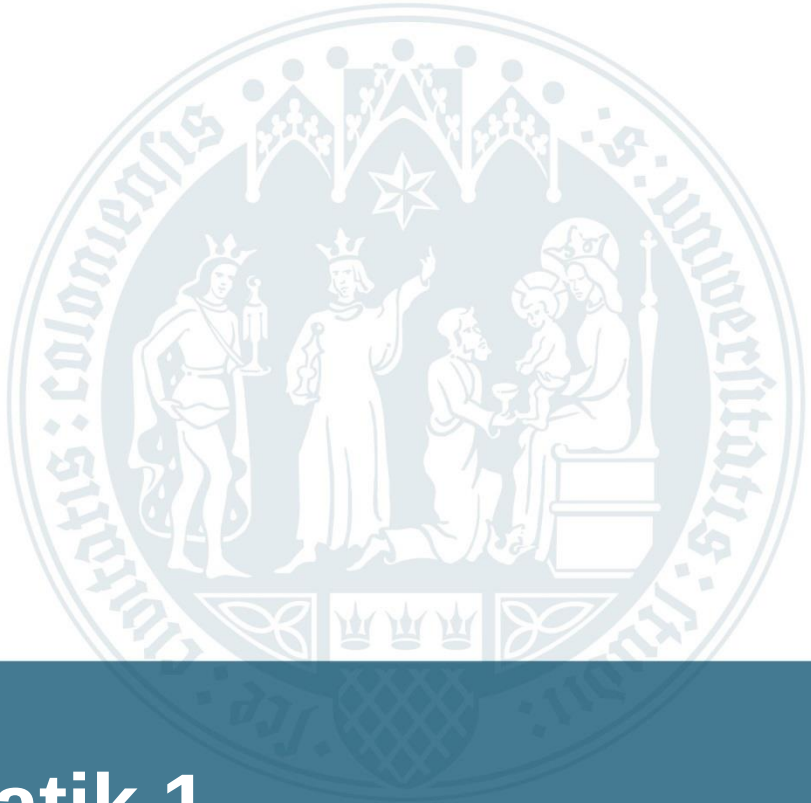
Universität zu Köln



Grundzüge der Informatik 1

Vorlesung 14

Dynamische Programmierung

Überblick

- Wiederholung
  - Entwurfsprinzip „gierige Algorithmen“
  - Beispiel: Zeitplanerstellung
  - Optimale Lösung durch gierigen Algorithmus

- Zeitplanerstellung – Lateness Scheduling
  - Problemdefinition
  - Diskussion unterschiedlicher Strategien
  - Korrektheit der optimalen Strategie

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

Beobachtungen

- 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

Wie können wir das erste Intervall wählen?

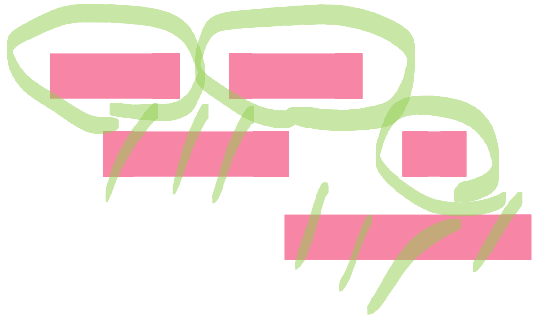
- Idee: Wir müssen die Ressource möglichst bald wieder freigeben
- Nimm das Intervall mit dem frühesten Endzeitpunkt
  (und entferne dann alle nicht kompatiblen Intervalle)

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 r-ten Intervalls zur aktuellen Lösung beider Algorithmen

Vergleichkriterium

- Maximaler Endzeitpunkt der bisher ausgewählten Intervalle

Gierige Algorithmen

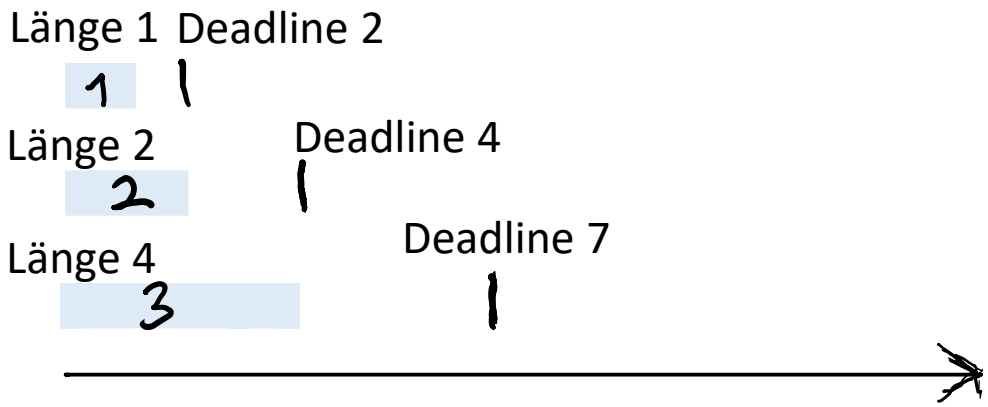
Satz 13.3

- 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

Zeitplanerstellung mit Deadlines

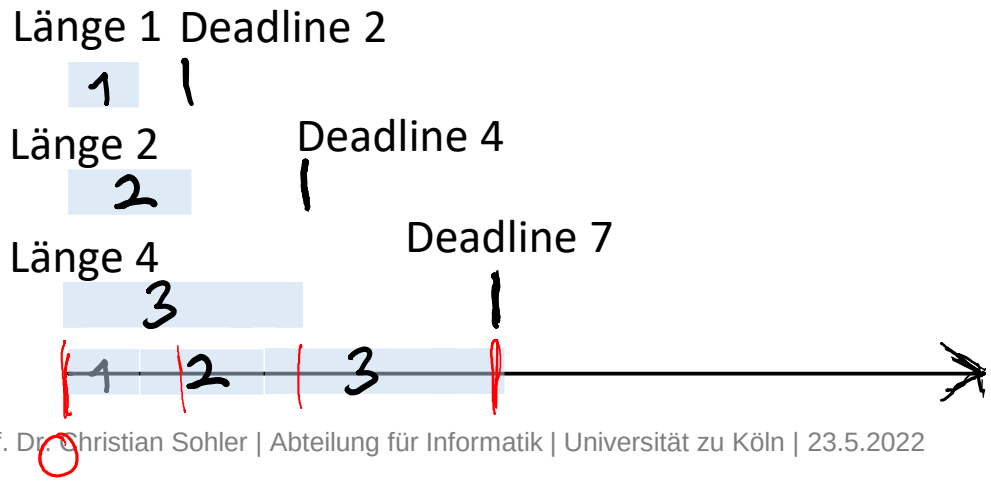
- Ressource (Maschine, Hörsaal, Parallelrechner, etc.) soll möglichst gut genutzt werden
- Auf der Ressource sollen Aufgaben durchgeführt werden
- Jede Aufgabe nimmt für eine bestimmte Dauer die Ressource in Anspruch
- Jede Aufgabe hat einen Zeitpunkt, an dem die Aufgabe bearbeitet sein soll (Deadline)



Gierige Algorithmen

Zeitplanerstellung mit Deadlines

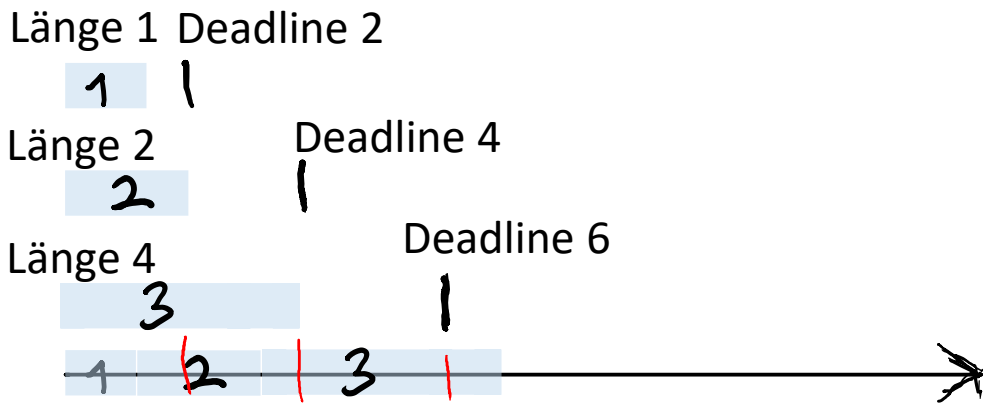
- Ressource (Maschine, Hörsaal, Parallelrechner, etc.) soll möglichst gut genutzt werden
- Auf der Ressource sollen Aufgaben durchgeführt werden
- Jede Aufgabe nimmt für eine bestimmte Dauer die Ressource in Anspruch
- Jede Aufgabe hat einen Zeitpunkt, an dem die Aufgabe bearbeitet sein soll (Deadline)



Gierige Algorithmen

Zeitplanerstellung mit Deadlines

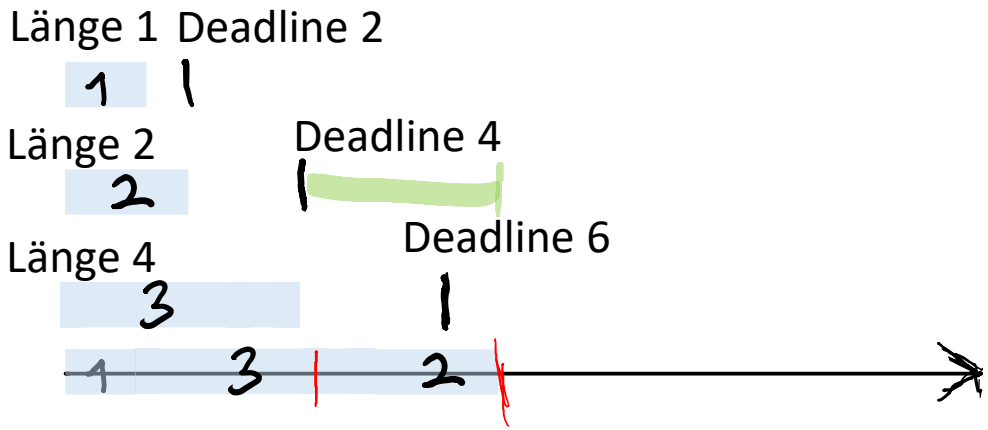
- Verzögerung:
  - Wird eine Aufgabe erst z Zeiteinheiten nach ihrer Deadline bearbeitet, so hat sie eine Verzögerung von z.
  - Wird eine Aufgabe innerhalb ihrer Deadline fertig, so hat sie keine Verzögerung 0.
- Optimierungsziel: Minimiere die maximale Verzögerung



Gierige Algorithmen

Zeitplanerstellung mit Deadlines

- Verzögerung:
  - Wird eine Aufgabe erst z Zeiteinheiten nach ihrer Deadline bearbeitet, so hat sie eine Verzögerung von z.
  - Wird eine Aufgabe innerhalb ihrer Deadline fertig, so hat sie Verzögerung 0.
- Optimierungsziel: Minimiere die maximale Verzögerung



Gierige Algorithmen

Zeitplanerstellung mit Deadlines

- Eingabe:
  - Anzahl Aufgaben n
  - Felder t und d
  - t[i] enthält Dauer der i-ten Aufgabe
  - d[i] enthält Deadline der i-ten Aufgabe
- Ausgabe:
  - Startzeitpunkte der Aufgaben, so dass die maximale Verzögerung minimiert wird

Gierige Algorithmen

Aufgabe

- Welche der folgenden Strategien ist optimal?

  A) Bearbeite die Aufgaben in der Reihenfolge ihrer Deadlines

  B) Bearbeite die Aufgaben in der Reihenfolge ihrer Dauer

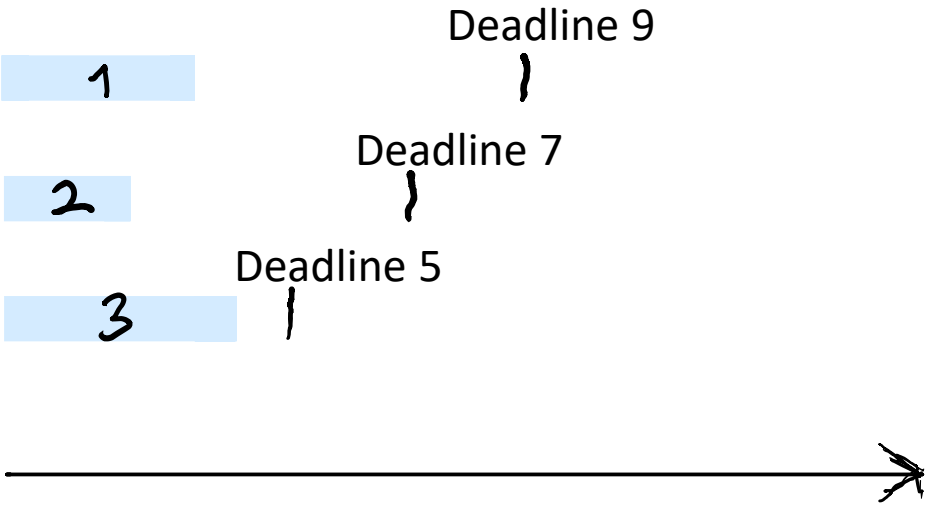
  C) Bearbeite die Aufgaben in der Reihenfolge ihrer Spielräume d[i]-t[i]

  D) Keine

Gierige Algorithmen

Strategie 2

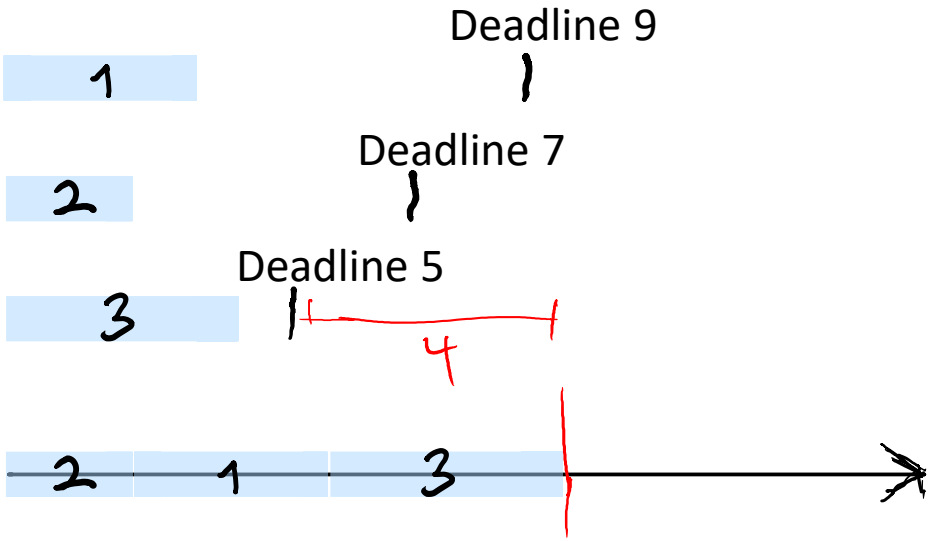
- Bearbeite Aufgaben in der Reihenfolge ihrer Dauer



Gierige Algorithmen

Strategie 2

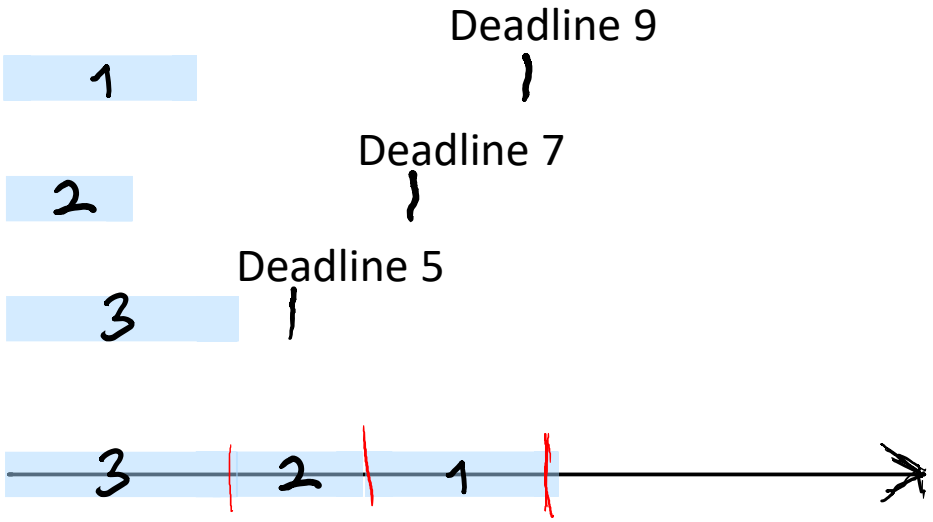
- Bearbeite Aufgaben in der Reihenfolge ihrer Dauer



Gierige Algorithmen

Strategie 2

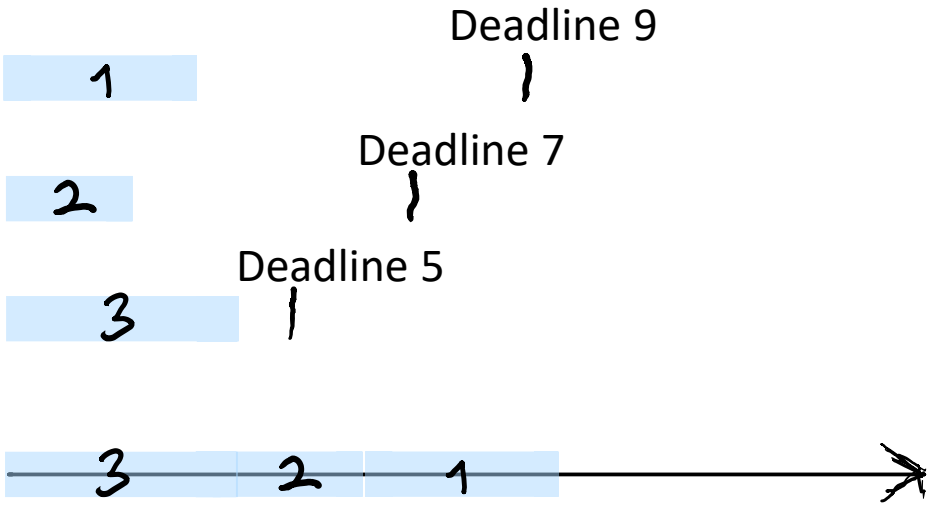
- Bearbeite Aufgaben in der Reihenfolge ihrer Dauer
- Nicht optimal!



Gierige Algorithmen

Strategie 2

- Bearbeite Aufgaben in der Reihenfolge ihrer Dauer
- Nicht optimal!

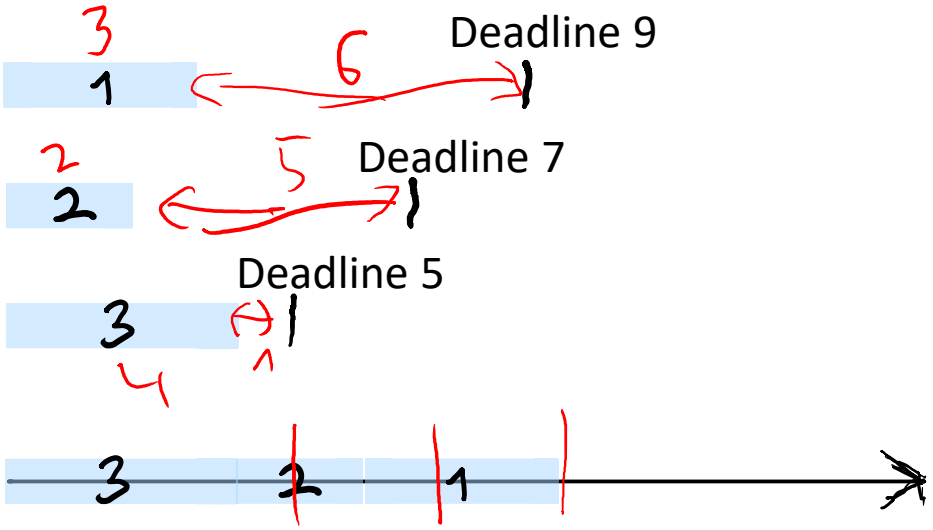


- Problem: Strategie ignoriert Deadlines völlig

Gierige Algorithmen

Strategie 3

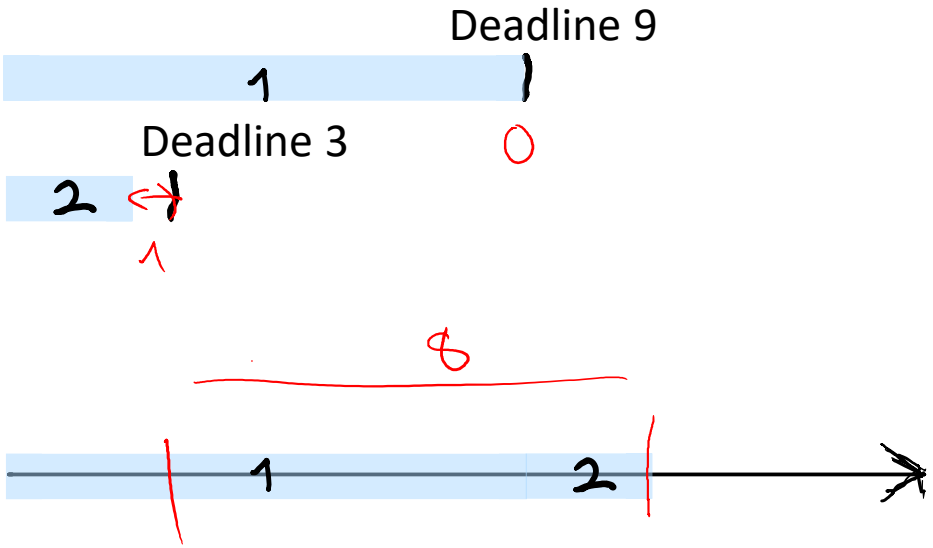
- Bearbeite Aufgaben in der Reihenfolge ihres Spielraums d[i]-t[i]



Gierige Algorithmen

Strategie 3

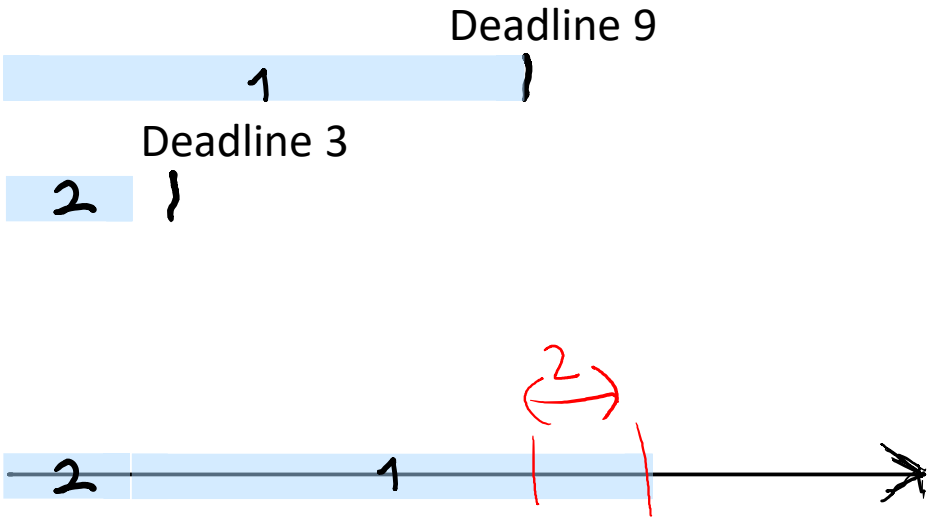
- Bearbeite Aufgaben in der Reihenfolge ihres Spielraums d[i]-t[i]



Gierige Algorithmen

Strategie 3

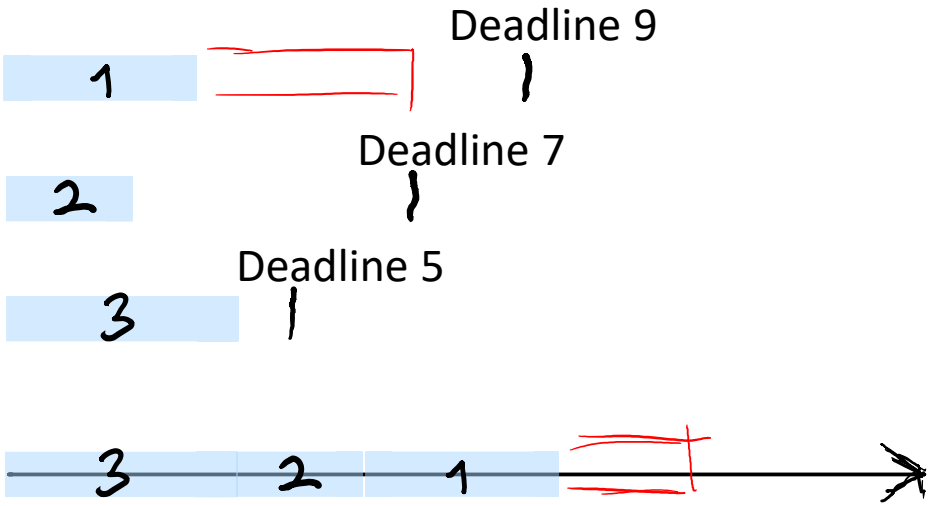
- Bearbeite Aufgaben in der Reihenfolge ihres Spielraums d[i]-t[i]



Gierige Algorithmen

Strategie 1

- Bearbeite Aufgaben in der Reihenfolge ihrer Deadlines
- Strategie ist optimal!

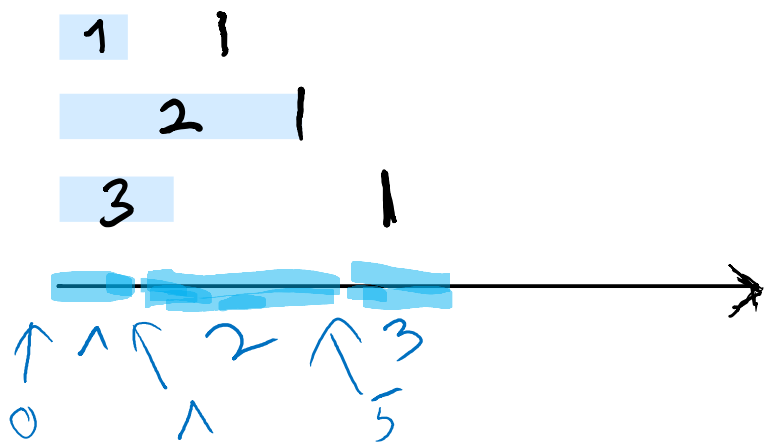


Gierige Algorithmen

LatenessScheduling(t,d,n)

1. A = new array [1..n]
2. z = 0
3. **for** i=1 **to** n **do**
4. A[i] = z
5. z = z + t[i]
6. **return** A

t	1	4	2
d	3	4	6

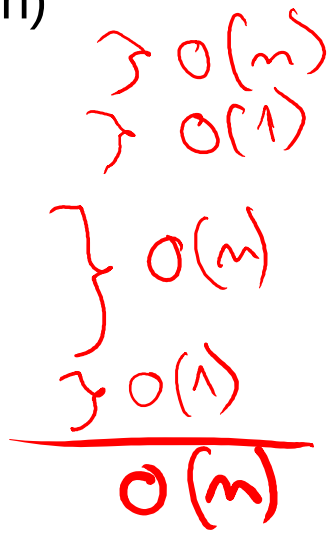


Annahme:

- Die Aufgaben sind nach aufsteigender Deadline sortiert

Gierige Algorithmen

```
LatenessScheduling(t,d,n)
1.  A = new array [1..n]
2.  z = 0
3.  for i=1 to n do
4.      A[i] = z
5.      z = z + t[i]
6.  return A
```



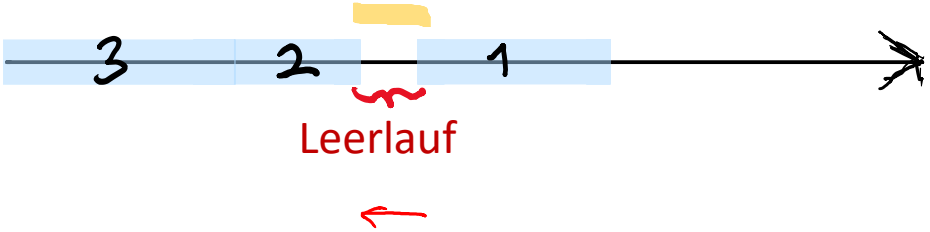
Laufzeitanalyse

- Die Laufzeit des Algorithmus ist O(n)

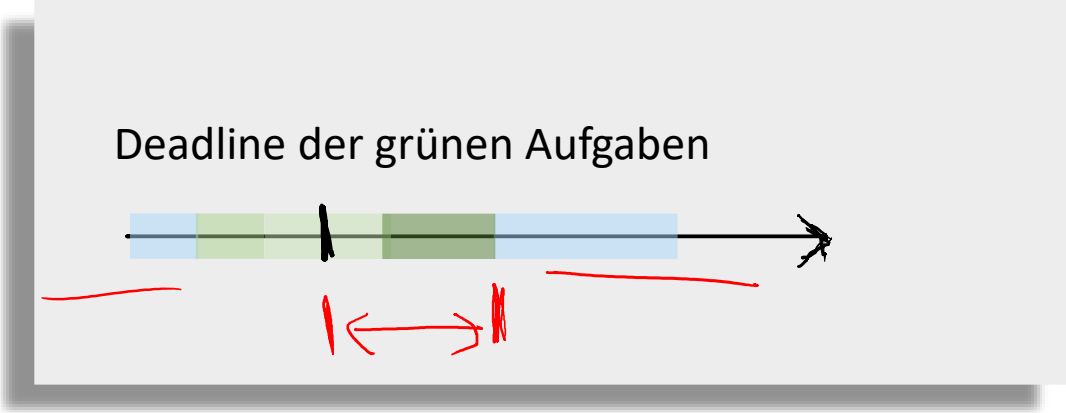
Gierige Algorithmen

Beobachtung

- Es gibt eine optimale Lösung ohne Leerlaufzeiten.



Gierige Algorithmen



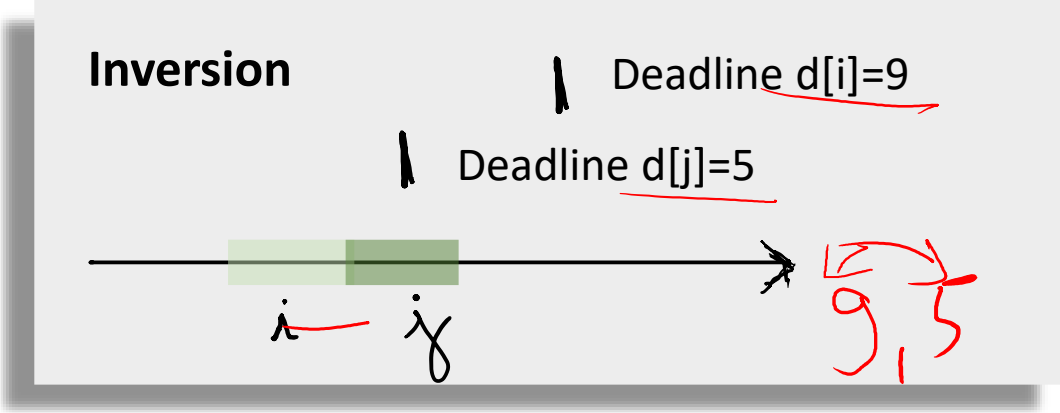
Lemma 14.1

- Alle Lösungen ohne Leerlauf, bei denen die Aufgaben nicht-absteigend nach Deadline sortiert sind, haben dieselbe maximale Verzögerung.

Beweis

- Die Lösungen können sich nur in der Reihenfolge von Aufgaben mit identischer Deadline unterscheiden
- In allen nicht-absteigend sortierten Lösungen folgen Aufgaben mit derselben Deadline direkt aufeinander
- Die maximale Verzögerung ist durch die letzte dieser aufeinanderfolgenden Aufgaben bestimmt
- Damit hängt die maximale Verzögerung nicht von der Reihenfolge der Aufgaben mit derselben Deadline ab, was das Lemma beweist

Gierige Algorithmen



Definition 14.2

- Eine Reihenfolge von Aufgaben hat eine Inversion (i,j), wenn Aufgabe i vor Aufgabe j in der Reihenfolge auftritt, aber die Deadline d[i] von Aufgabe i größer ist als die Deadline d[j] von Aufgabe j.

Beobachtung

- Eine Reihenfolge ohne Inversionen ist nicht-absteigend sortiert.

Gierige Algorithmen

Beweisskizze

- Wir wollen argumentieren, dass man mit einer beliebigen Lösung ohne Leerlauf starten und dann die Aufgaben bzgl. Deadlines sortieren kann, und dabei die Lösung nicht verschlechtern
- Daraus folgt dann, dass es eine nicht-absteigend sortierte, optimale Lösung gibt
- Da alle nicht-absteigend sortierten Lösungen ohne Leerlauf die gleiche maximale Verzögerung haben, sind alle optimal
- Unser Algorithmus berechnet aber eine solche Lösung

Gierige Algorithmen

Sortieren

- Wenn es eine Inversion gibt, so gibt es auch eine Inversion benachbarter Aufgaben
- Man kann eine Inversion benachbarter Aufgaben auflösen, ohne die Lösung zu verschlechtern
- Man kann durch Vertauschen von Inversionen benachbarter Elemente die Aufgaben sortieren

Gierige Algorithmen

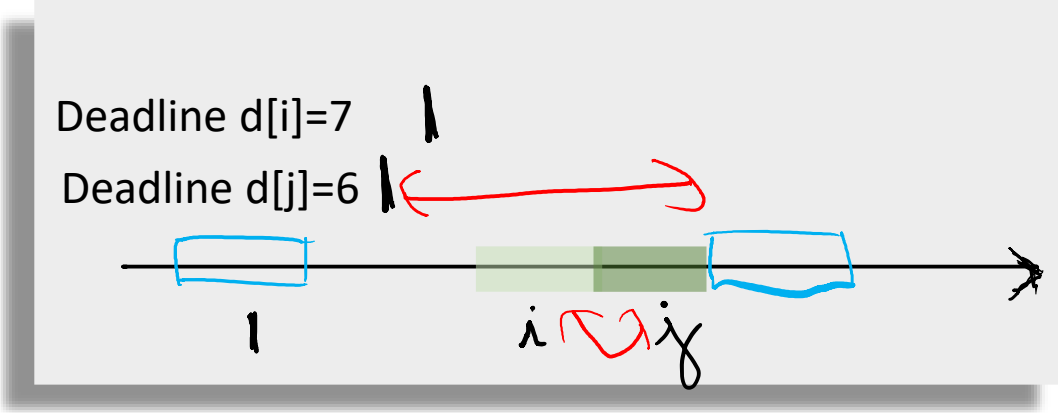
Lemma 14.3

- Gibt es in einer Reihenfolge von Aufgaben eine Inversion (i,j), dann gibt es auch eine Inversion zweier in der Reihenfolge benachbarter Aufgaben.

Beweis

- Sei (i,j) eine Inversion: Es gilt i ist vor j in der Reihenfolge und d[i]>d[j]
- Wir betrachten die Aufgaben beginnend mit Aufgabe i in der vorgegebenen Reihenfolge bis Aufgabe j
- Wenn i und j benachbart sind, so sind wir fertig
- Wenn Aufgabe i und ihr Nachfolger k eine Inversion bilden, sind wir fertig
- Wenn Aufgabe i und ihr Nachfolger k keine Inversion bilden, so gilt d[k] ≥ d[i]
- Somit bilden k und j eine Inversion und wir können unsere Argumentation mit i:=k und j wiederholen
- Nach endlich vielen Schritten sind i und j benachbart

Gierige Algorithmen



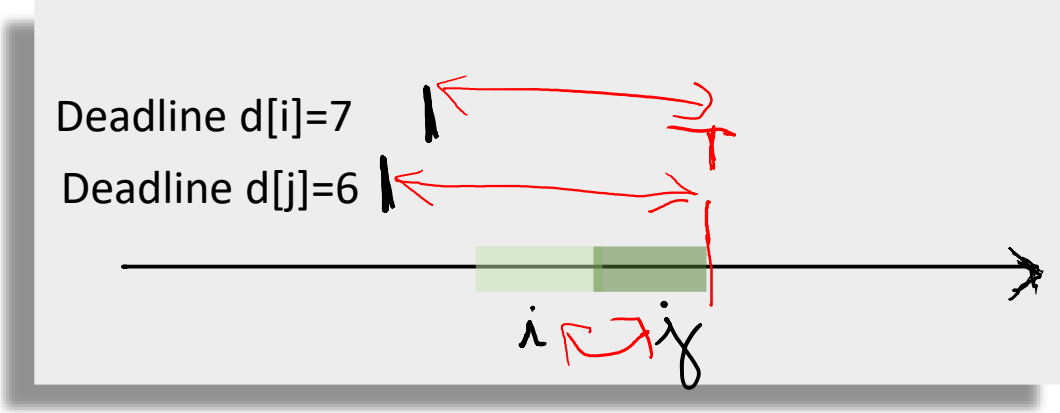
Lemma 14.4

- Gibt es in einer Reihenfolge von Aufgaben eine Inversion (i,j) von zwei in der Reihenfolge benachbarten Aufgaben i und j, dann kann man Aufgabe i und j vertauschen, ohne die Lösung zu verschlechtern.

Beweis

- Betrachte zwei benachbarte Aufgaben i und j, so dass i vor j auftritt und d[i]> d[j] ist ((i,j) bildet eine Inversion)
- Das Tauschen von i und j hat keinen Einfluss auf die Abarbeitungszeitpunkte der anderen Aufgaben
- Da Aufgabe j in der neuen Reihenfolge eher abgearbeitet wird, kann sich ihre Verzögerung nur verringern oder gleich bleiben

Gierige Algorithmen



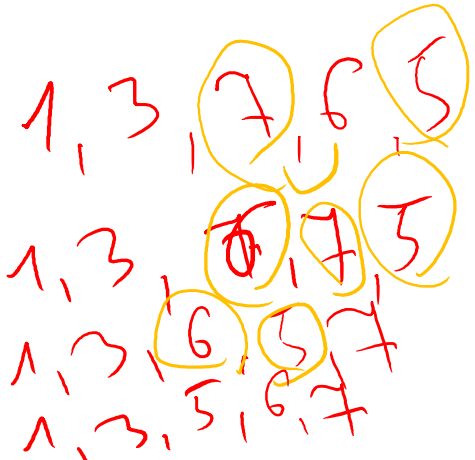
Lemma 14.4

- Gibt es in einer Reihenfolge von Aufgaben eine Inversion (i,j) von zwei in der Reihenfolge benachbarten Aufgaben i und j, dann kann man Aufgabe i und j vertauschen, ohne die Lösung zu verschlechtern.

Beweis

- Sei T der Zeitpunkt, an dem Aufgabe j vor dem Vertauschen abgearbeitet wurde
- Dann wird Aufgabe i nach dem Vertauschen auch zum Zeitpunkt T abgearbeitet
- Da d[i]>d[j] ist T-d[i] < T-d[j]
- Somit erhöht sich die Verzögerung durch das Vertauschen nicht, weil die Verzögerung von Aufgabe j vor dem Vertauschen mind. so groß ist wie die Verzögerung von Aufgabe i nach dem Vertauschen

Gierige Algorithmen



Lemma 14.5

- Es gibt eine optimale Lösung ohne Leerlauf, bei der die Aufgaben nicht-absteigend nach Deadline sortiert sind.

Beweis

- Betrachte eine optimale Lösung ohne Leerlauf
- Ist die Lösung nicht-absteigend nach Deadlines sortiert, so sind wir fertig
- Ansonsten gibt es eine Inversion (i,j) und nach Lemma 14.3 auch eine Inversion benachbarter Aufgaben
- Durch Vertauschen der benachbarten Aufgaben wird die Lösung nicht schlechter (Lemma 14.4) und es wird eine Inversion entfernt
- Es gibt maximal n^2 Inversionen. Wir wiederholen den Prozess, bis keine Inversionen mehr vorhanden sind
- Damit folgt das Lemma

Gierige Algorithmen

Satz 14.6

- Algorithmus LatenessScheduling berechnet eine optimale Lösung in O(n) Laufzeit, wenn die Aufgaben nicht-absteigend nach Deadline sortiert sind.

Beweis

- Es gibt eine optimale Reihenfolge ohne Leerlauf, die nicht-absteigend sortiert ist (Lemma 14.5)
- Jede nicht-absteigend sortierte Reihenfolge ohne Leerlauf hat dieselben Kosten (Lemma 14.1) und ist damit optimal
- Unser Algorithmus berechnet eine nicht-absteigend sortierte Reihenfolge ohne Leerlauf
- Die Laufzeitanalyse haben wir bereits durchgeführt

Gierige Algorithmen

Zusammenfassung

- Zeitplanerstellung – Lateness Scheduling
  - Problemdefinition
  - Diskussion unterschiedlicher Strategien
  - Korrektheit der optimalen Strategie

Algorithmische Entwurfsmethoden

- Teile und Herrsche Prinzip
  - Aufteilen der Daten und rekursives Lösen des Problems
  - Laufzeitanalyse durch Auflösen von Rekursionsgleichungen
  - Geeignet z.B. für Felder und geometrische Daten
- Dynamische Programmierung
  - Problem auf optimale Teillösungen zurückführen (Rekursion)
  - Finden des Lösungswertes durch Ausfüllen einer Tabelle
  - Konstruktion der Lösung mit Hilfe der Tabelle
- Gierige Algorithmen
  - Verfolgen immer eine einfache lokale Strategie
  - Dadurch schnell, aber es werden nicht alle Lösungen betrachtet
  - Berechnen unter Umständen nicht die optimale Lösung

Referenzen

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