Julius-Maximilians-
UNIVERSITÄT WÜRZBURG

Lehrstuhl für
INFORMATIK I
Effiziente Algorithmen und wissensbasierte Systeme

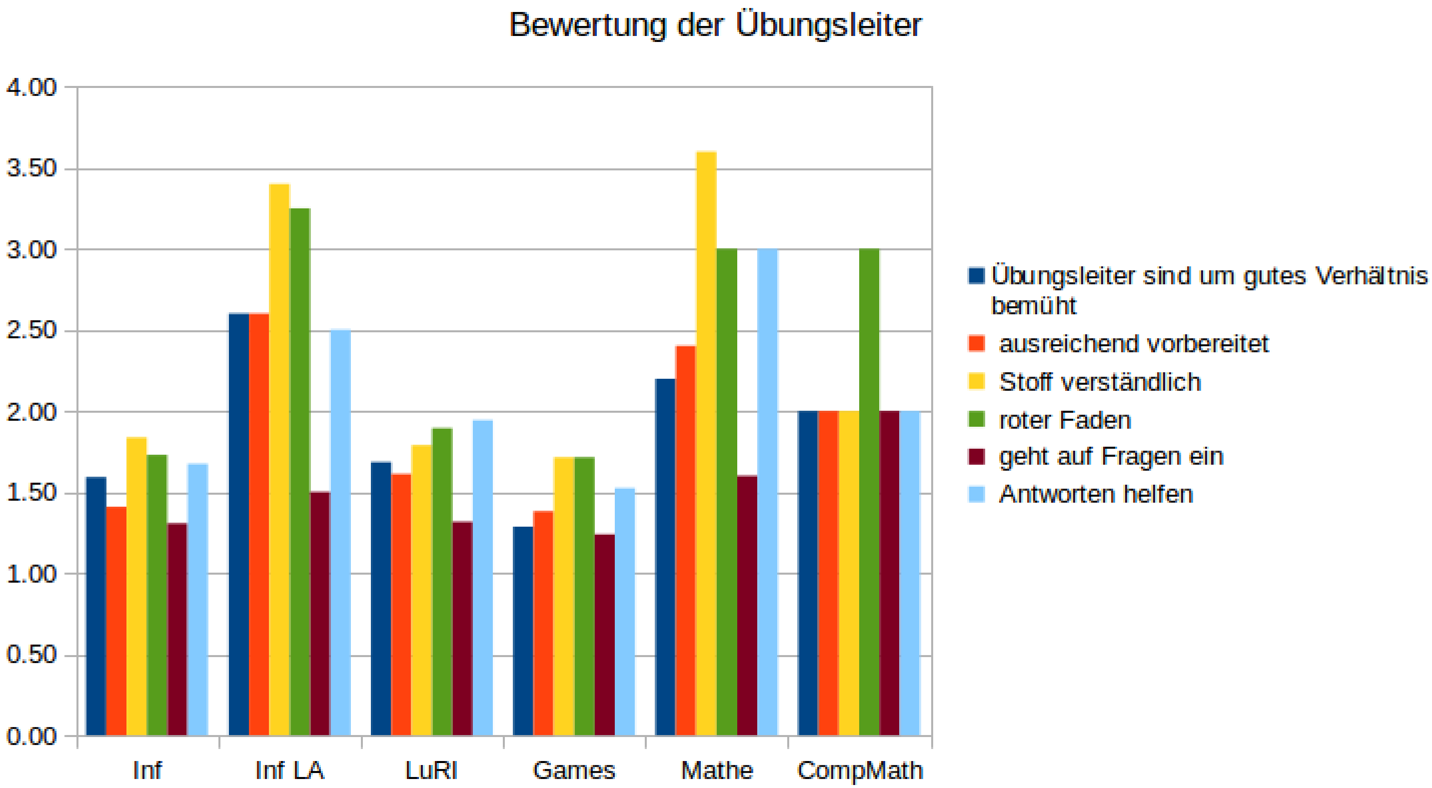
Institut für Informatik

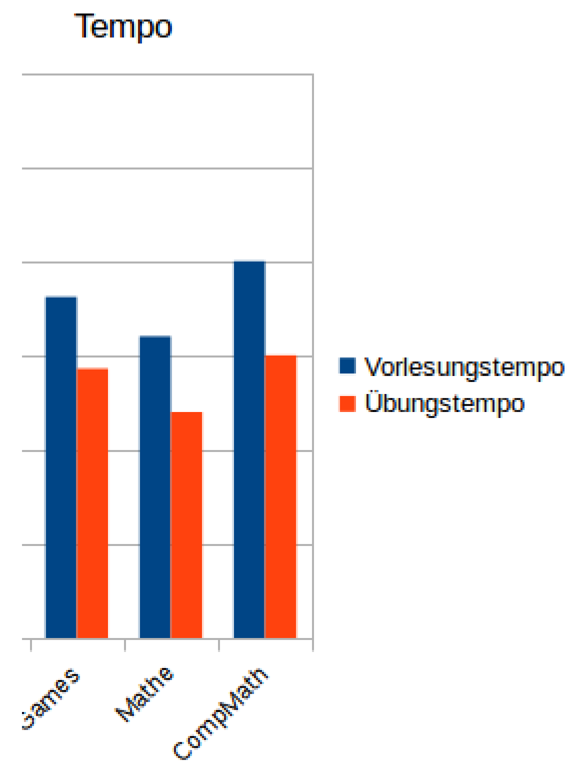
Algorithmen und Datenstrukturen

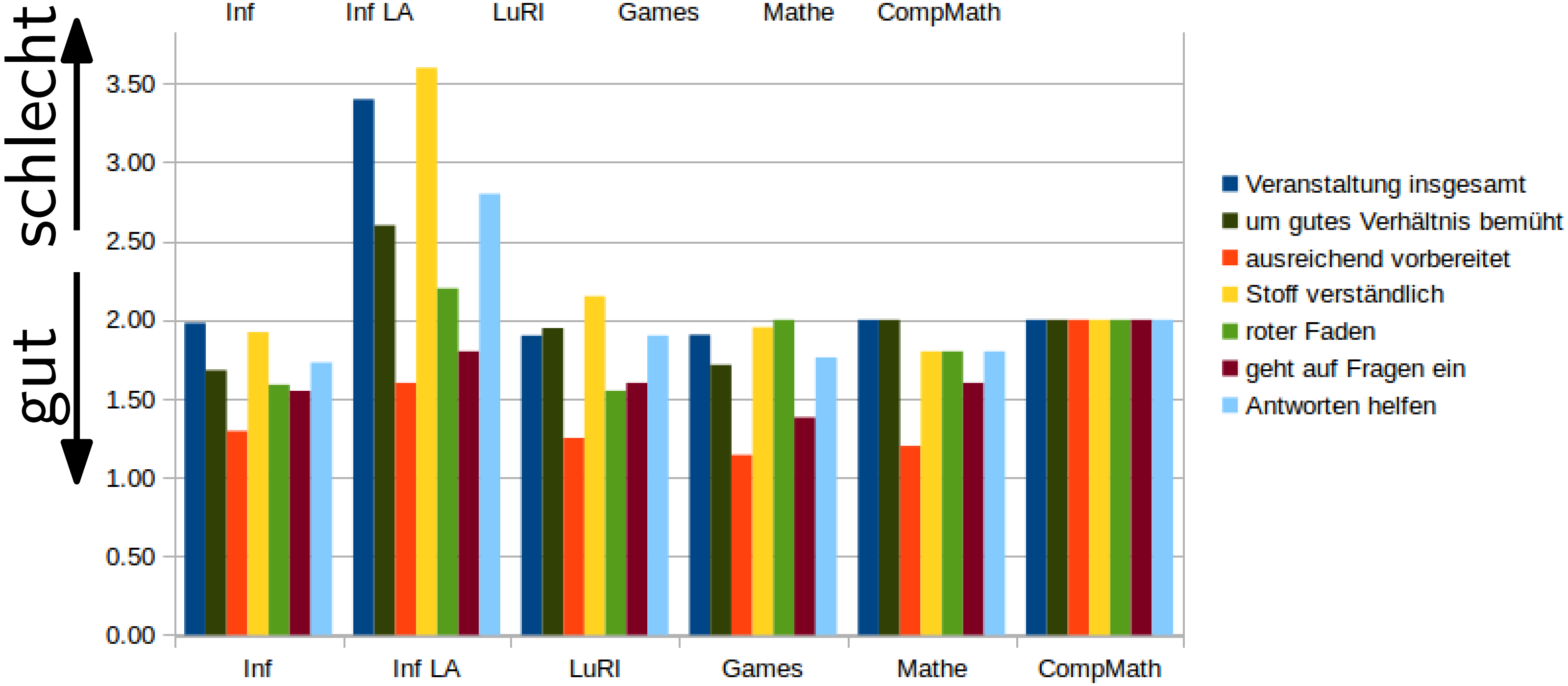
Wintersemester 2018/19
23. Vorlesung

Greedy- und Approximationsalgorithmen

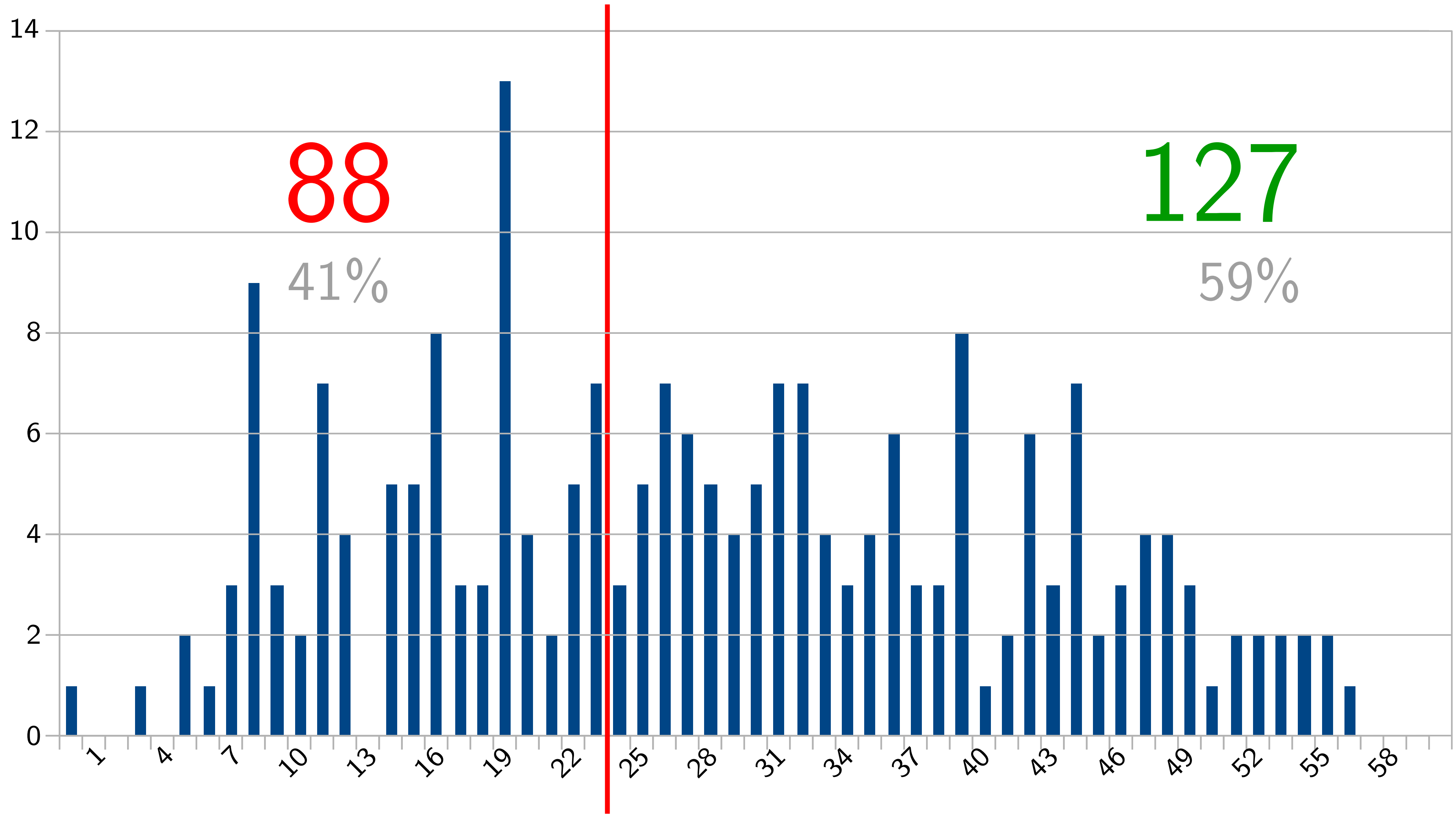
Prof. Dr. Alexander Wolff *Lehrstuhl für Informatik I*


Bewertung der Übungsleiter
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
Inf
Inf LA
LuRI
Games
Mathe
CompMath
Übungsleiter sind um gutes Verhältnis bemüht
ausreichend vorbereitet
Stoff verständlich
roter Faden
geht auf Fragen ein
Antworten helfen


Tempo
Games
Mathe
CompMath
Vorlesungstempo
Übungstempo

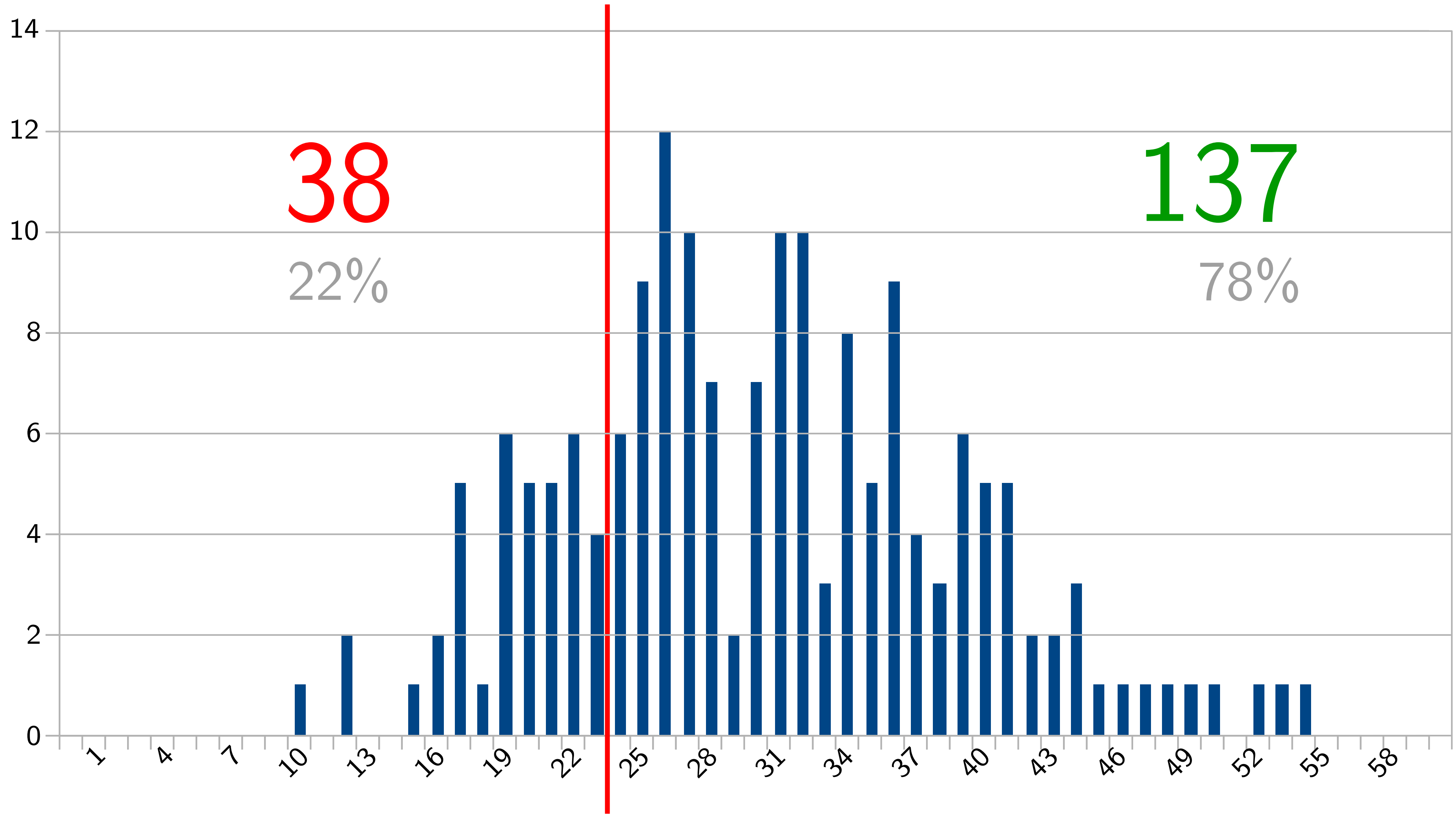

schlecht
gut
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
Inf
Inf LA
LuRI
Games
Mathe
CompMath
Veranstaltung insgesamt
um gutes Verhältnis bemüht
ausreichend vorbereitet
Stoff verständlich
roter Faden
geht auf Fragen ein
Antworten helfen

Ergebnisse des 1. Kurztests



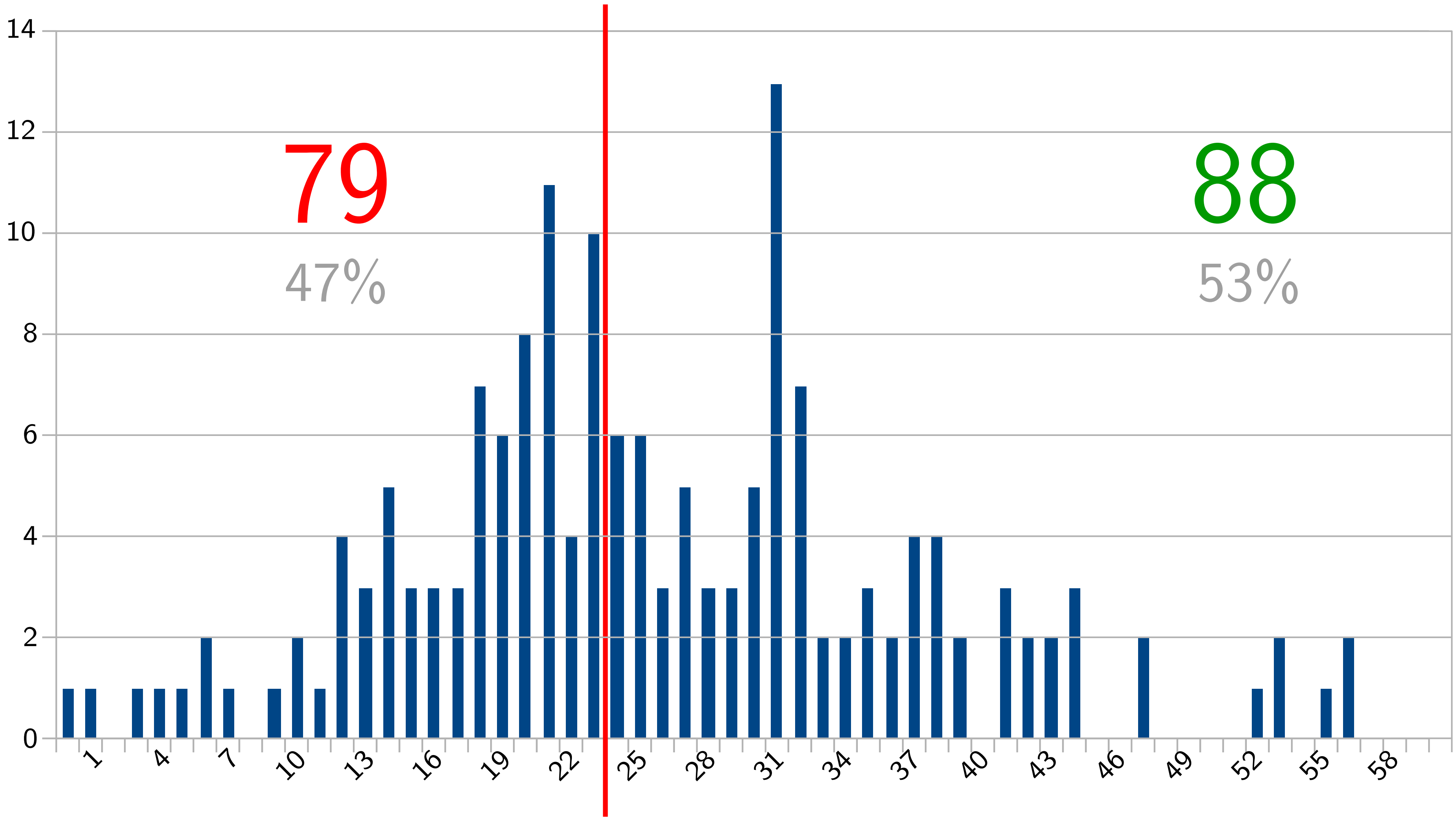
$n = 215$; Durchschnitt 28,3; Median 27

Ergebnisse des 2. Kurztests



$n = 175$; Durchschnitt 30,5; Median 27

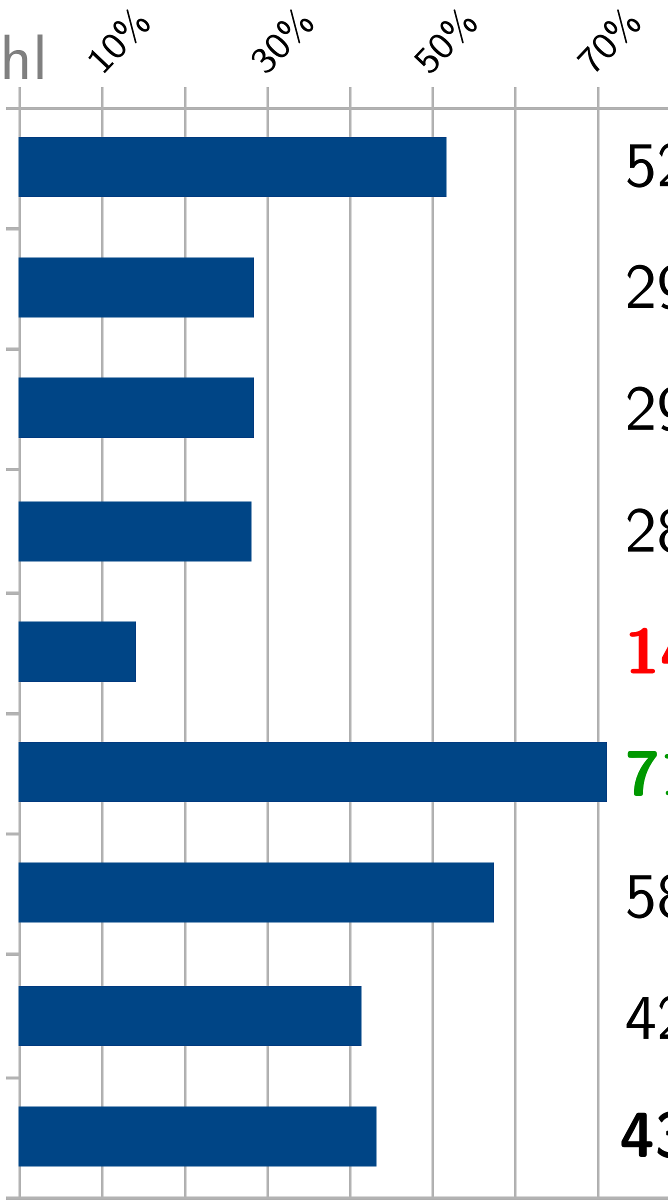
Ergebnisse des 3. Kurztests



$n = 167$; Durchschnitt 26,0; Median 24

Kurztest 3: Abschneiden nach Aufgaben

	max. Punktzahl	
Rot-Schwarz-Bäume	10	52%
Median im Rot-Schwarz-Baum	9	29%
FindK	6	29%
Backup-Schlange	5	28%
Liste mit Varianz	6	**14%**
Graphen & Breitensuche	10	**71%**
Nächstes Paar auf einer Geraden	8	58%
Knoten mit gleichem Abstand	6	42%
	60	**43%**



Vorbereitung auf die Klausur am Di, 12.2.

Tipp: Schreiben Sie sich alle Fragen auf, die Ihnen bei der Vorbereitung in den Kopf kommen!

- Nutzen Sie die Übungen zum Fragen.
- Kommen Sie nach der Vorlesung zu mir.
- Nutzen Sie meine Sprechstunde (Mi, 13–14 Uhr)

Auch in der letzten Vorlesung (Do, 7.2.) können Sie Fragen stellen. Ich bringe alle Folien mit.

Die vorletzte Vorlesung (Di, 5.2.) fällt aus, da beide großen HS für die Klausur GADS für WiInfs & MCS benötigt werden.

In WueStudy bis Di, 31.1., 23:59 Uhr zur Klausur *anmelden!*
Wer sich nicht anmeldet, kann die Klausur NICHT mitschreiben!

ADS-Repetitorium

- Mo., 1.4., bis Fr, 5.4. (5 Tage, SR 10)
- Bereitet auf den zweiten Klausurtermin (Di, 9.4.) vor
- Täglich 9:30–16:00 Uhr: Abwechselnd Vorlesung / Übung
- Leitung: Michael Kreuzer und Diana Sieper
- 2008 haben 15 HörerInnen an der 2. Klausur teilgenommen.

	ADS-Repetitorium	*bestanden*
regelmäßig teilgenommen	7	4
nicht oder unregelmäßig	8	1

Operations Research

Optimierung für Wirtschaftsabläufe:

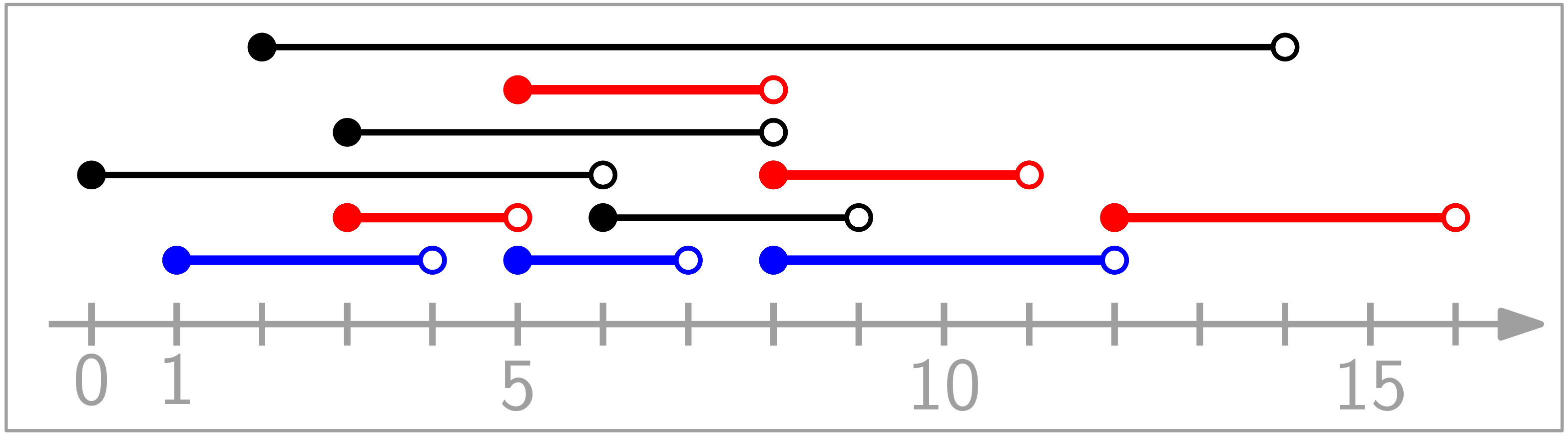
- Standortplanung
- Ablaufplanung
- Flottenmanagement
- Pack- und Zuschnittprobleme
- ...

Werkzeuge:

Statistik, Optimierung, Wahrscheinlichkeitstheorie, Spieltheorie, Graphentheorie, mathematische Programmierung, Simulation...

Ein einfaches Problem der Ablaufplanung

Gegeben: Menge $A = \{a_1, \dots, a_n\}$ von *Aktivitäten*, wobei für $i = 1, \dots, n$ gilt $a_i = [s_i, e_i)$.



a_i und a_j sind *kompatibel*, wenn $a_i \cap a_j = \emptyset$.

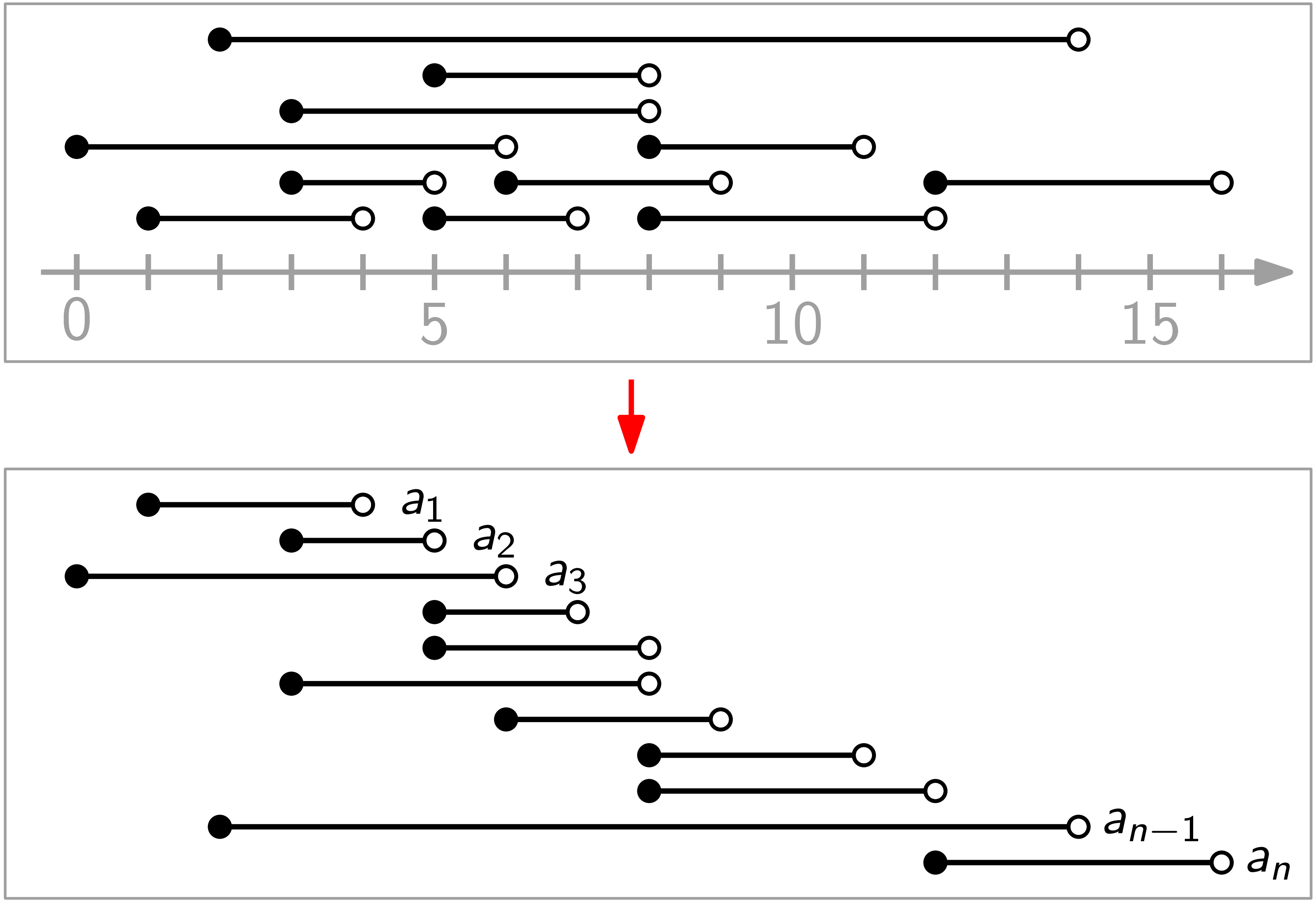
Die Aktivitäten in $A' \subset A$ sind *paarweise kompatibel*, wenn für jedes Paar $a_i, a_j \in A'$ gilt, dass a_i und a_j kompatibel sind.

Gesucht: eine größtmögliche Menge paarweise kompatibler Aktivitäten.

Grund: Aktivitäten (à 1€), die gleiche Ressource benutzen

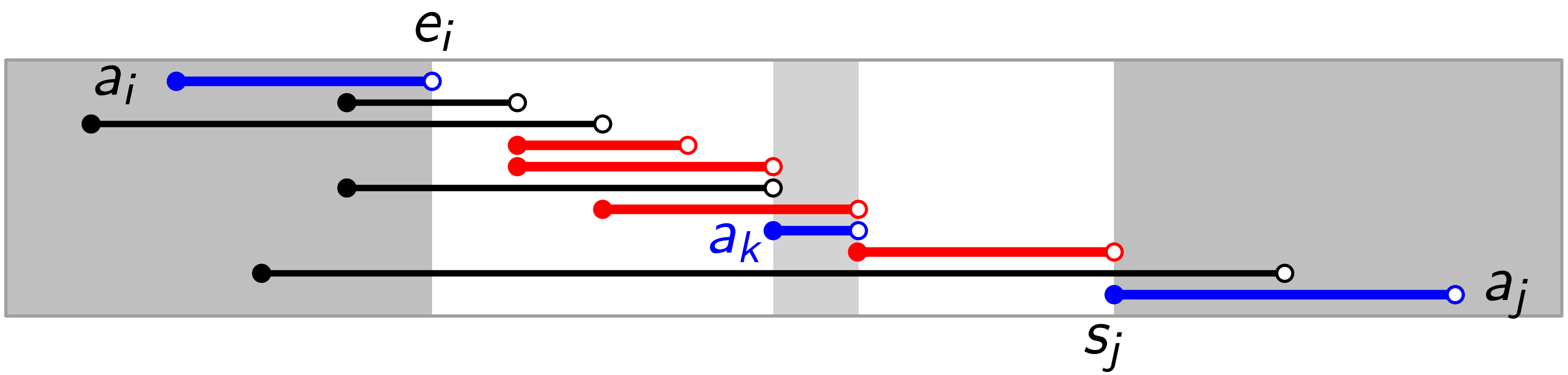
Ein kleiner technischer Trick

Wir nummerieren (für den Rest der Vorlesung) die Aktivitäten so, dass für die Endtermine gilt $e_1 \leq e_2 \leq \cdots \leq e_n$.



Dynamisches Programmieren?

1. Struktur einer optimalen Lösung charakterisieren



Sei $A_{ij} \subset A$ die Menge aller Aktivitäten in $[e_i, s_j)$, d.h. „zwischen“ a_i und a_j.

Ang. a_i und a_j sind in einer opt. Lösung $L \subseteq A$ enthalten, dann ist $L \cap A_{ij}$ eine opt. Lösung für A_{ij}.

$\Rightarrow$ optimale Substruktur!

Beweis? *Austauschargument!*

2. Wert einer optimalen Lösung rekursiv definieren

Sei c_{ij} die Kardinalität einer opt. Lösung für A_{ij}.

Dann gilt: $c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$

Dynamisches Programmieren?

3. Wert einer optimalen Lösung berechnen

Setze $e_0 = -\infty$ und $s_{n+1} = +\infty$. Dann ist $A = A_{0,n+1}$.

Berechne $c_{0,n+1}$, die Kardinalität einer opt. Lösung für A.

(a) *top-down*

TopDownDP(int[ ]s, int[ ]e, int i, int j)

$\to$ liefert c_{ij}

(b) *bottom-up*

BottomUpDP(int[ ]s, int[ ]e)

$\to$ liefert $c_{0,n+1}$

Siehe Folie
„Zurück zum dynamischen Programmieren“

Laufzeit? $O(n^3)$...

2. Wert einer optimalen Lösung rekursiv definieren

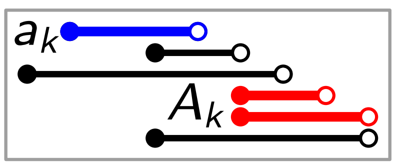
Sei c_{ij} die Kardinalität einer opt. Lösung für A_{ij}.

Dann gilt: $c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$

Darf's auch etwas einfacher sein?

Idee: Sei L opt. Lösung für A. – Welche Aktivität hat gute Chancen die erste („linkeste“) in L zu sein?

Intuition: Die Aktivität a_1 mit frühester Endzeit – weil a_1 die gemeinsame Ressource am wenigsten einschränkt.



Sei $A_k = \{a_i \in A : s_i \geq e_k\}$ die Menge der Aktivitäten, die nach Ablauf von a_k beginnen.

Sei L_k eine optimale Lösung von A_k.

Falls Intuition korrekt, dann ist $\{a_1\} \cup L_1$ optimal.

Satz. Sei $A_k \neq \emptyset$.
Sei a_m Aktivität mit frühester Endzeit in A_k.
$\Rightarrow$ es gibt eine opt. Lösung von A_k, die a_m enthält.

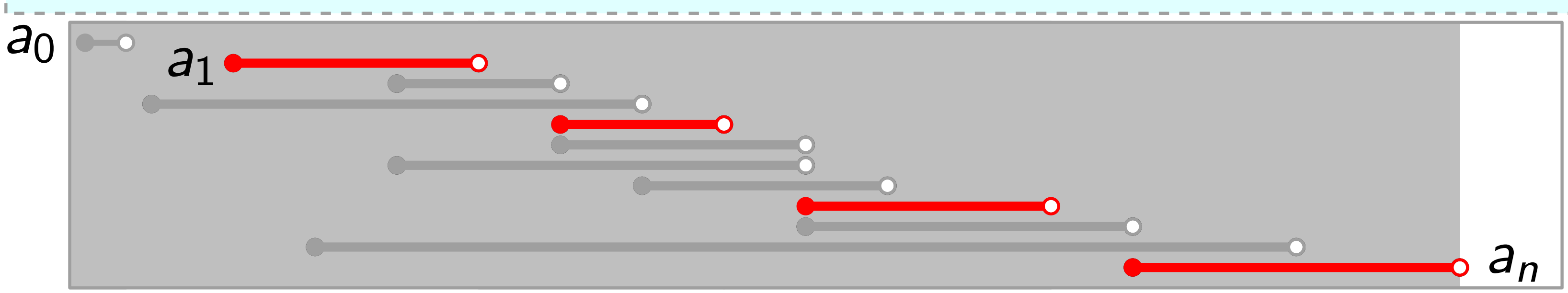
optimale Teilstruktur!

Beweis. *Austauschargument!*

Greedy – rekursiv

GreedyRecursive(int[ ]s, int[ ]e)
  $e_0 = -\infty$ $// \Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[ ]s, int[ ]e, int k) // best. Lsg. für A_k
  $m = k + 1$; $n = s.length$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

GreedyRecursive(int[ ]s, int[ ]e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[ ]s, int[ ]e, int k) // best. Lsg. für A_k
  $m = k + 1$; $n = s.length$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)

Laufzeit? Wie oft wird m inkrementiert?
Insgesamt, über alle rekursiven Aufrufe, n Mal.
D.h. GreedyRecursive läuft (ohne Sortieren) in $\Theta(n)$ Zeit.

Greedy – iterativ

```
GreedyIterative(int[] s, int[] e)
  n = s.length
  if n = 0 then return ∅
  L = {a_1}
  k = 1    // höchster Index in L
  for m = 2 to n do
    if s[m] ≥ e[k] then
      L = L ∪ {a_m}
      k = m
  return L
```

Laufzeit? GreedyIterative läuft ebenfalls in $\Theta(n)$ Zeit.

Bemerkung: GreedyIterative berechnet dieselbe optimale Lösung wie GreedyRecursive – die „linkeste“.

Die Greedy-Strategie

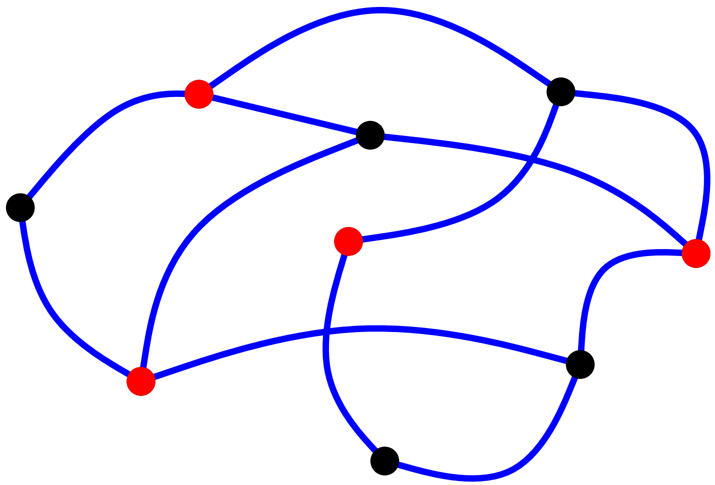
1. Teste, ob das Problem optimale Teilstruktur aufweist.
2. Entwickle eine rekursive Lösung
3. Zeige, dass bei einer Greedy-Entscheidung nur *ein* Teilproblem bleibt
4. Beweise, dass die Greedy-Wahl „sicher“ ist (vgl. Kruskal!)
5. Entwickle einen rekursiven Greedy-Algorithmus
6. Konvertiere den rekursiven in einen iterativen Algorithmus

Food for Thought

1. Welches allgemeinere Ablaufproblem könnte das DP lösen – aber nicht der GA?

   Zur Erinnerung: Das DP berechnet $c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$.

2. Problem *größte unabhängige Menge (guM)* in Graphen:



Finde eine größte Teilmenge U der Knoten, so dass keine zwei Knoten in U benachbart sind.

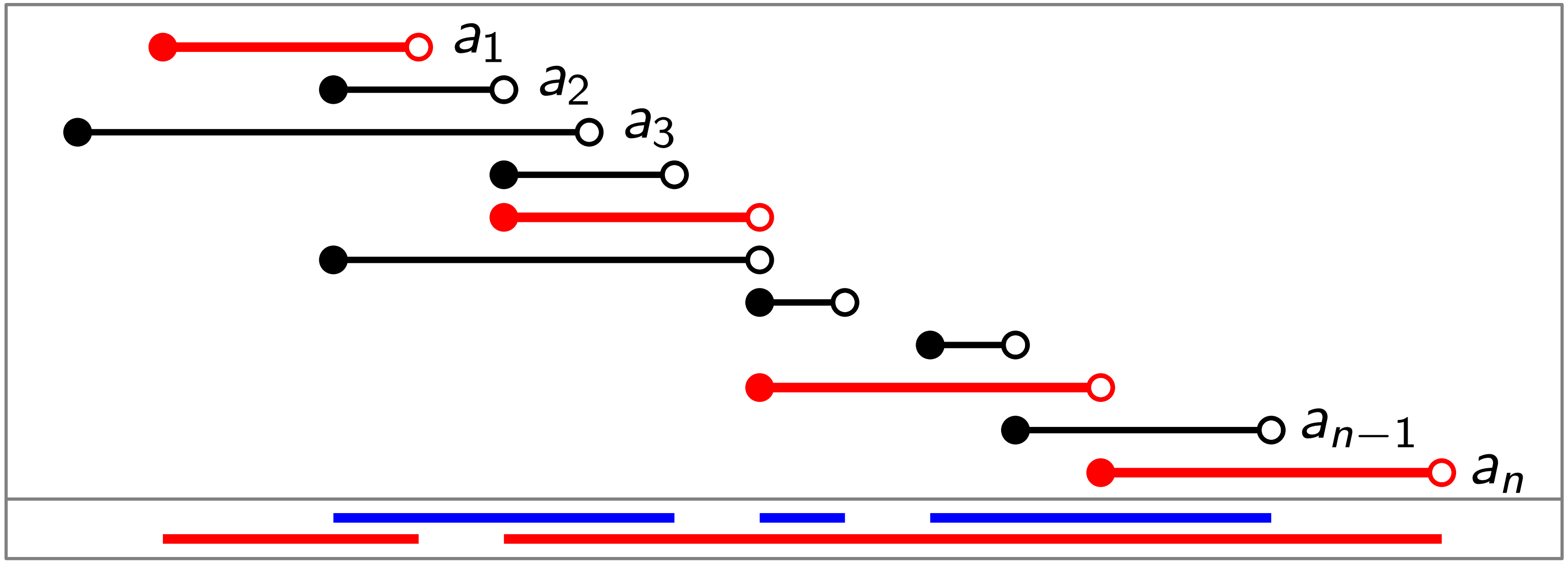
– Was hat guM mit unserem Ablaufplanungsproblem zu tun?

– Kann man guM mit DP oder GA lösen?

Ein ähnliches Problem der Ablaufplanung

Gegeben: Menge $A = \{a_1, \dots, a_n\}$ von halboffenen Intervallen, mit $a_i = [s_i, e_i)$ für $i = 1, \dots, n$.

Für die Endpunkte gelte $e_1 \leq e_2 \leq \dots \leq e_n$.



Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren Gesamtlänge $\ell(A')$ maximal ist.

Grund: Intervalle $\hat{=}$ Prozesse, die die gleiche Ressource nutzen; der Gesamtertrag ist proportional zur Auslastung.

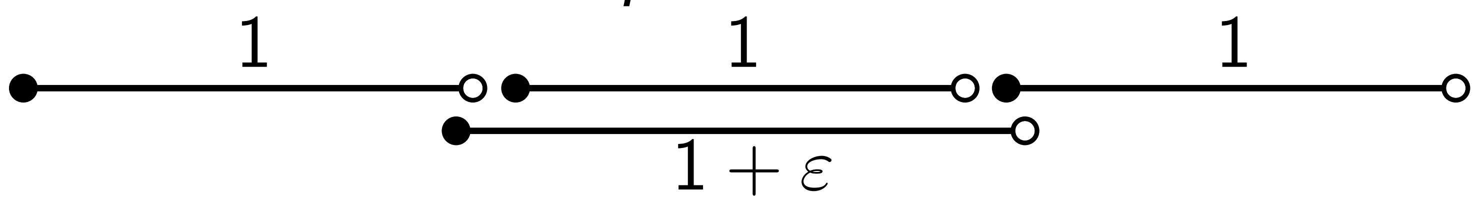
Greedy?

1. Versuch: *Nimm Aktivität mit frühestem Endtermin, streiche dazu inkompatible Aktivitäten und iteriere.*

Gegenbsp.:

2. Versuch: *Nimm längste Aktivität, streiche dazu inkompatible Aktivitäten und iteriere.*

Gegenbsp.:



Aufgabe: Können Sie den 2. GA in $O(n \log n)$ Zeit implementieren?

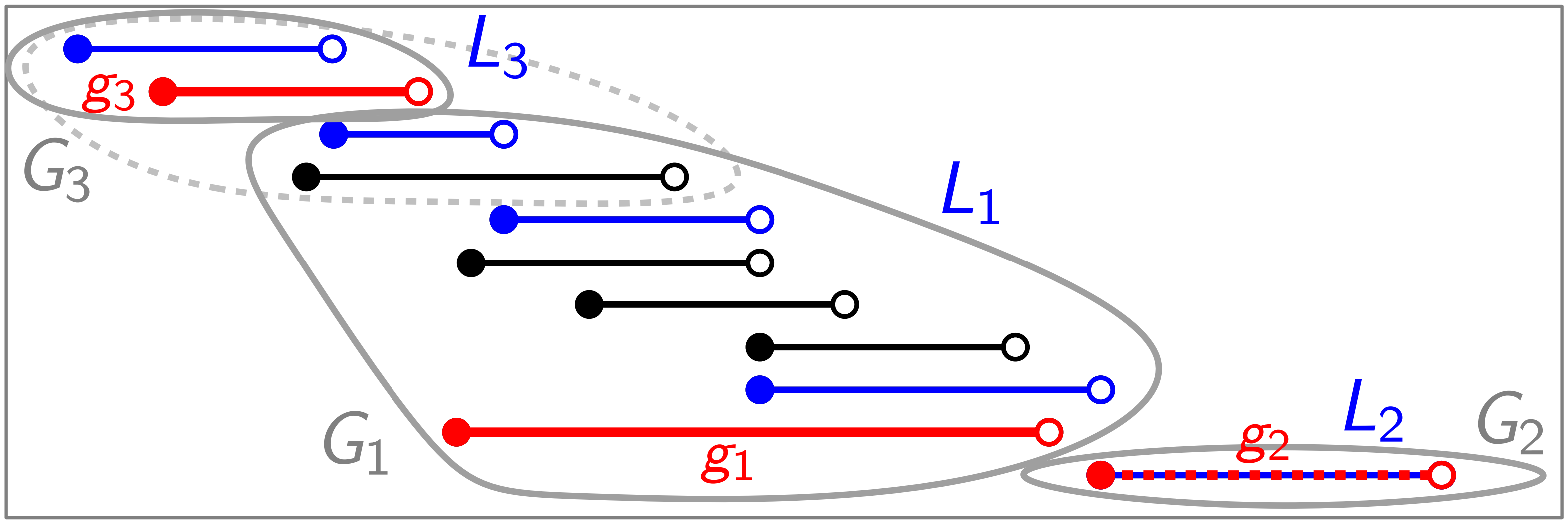
Diskutieren Sie mit Ihrer NachbarIn!

Wie gut/schlecht ist der 2. GA?

Betrachte eine optimale Lösung $L \subseteq A$.

Sei $G = \{g_1, g_2, \ldots, g_k\} \subseteq A$ die Greedy-Lösung *(in dieser Rf.)*.

Für $i = 1, \ldots, k$ sei $G_i = \{a \in A \mid a \cap g_i \neq \emptyset\} \setminus (G_1 \cup \cdots \cup G_{i-1})$



und sei $L_i = L \cap G_i$.

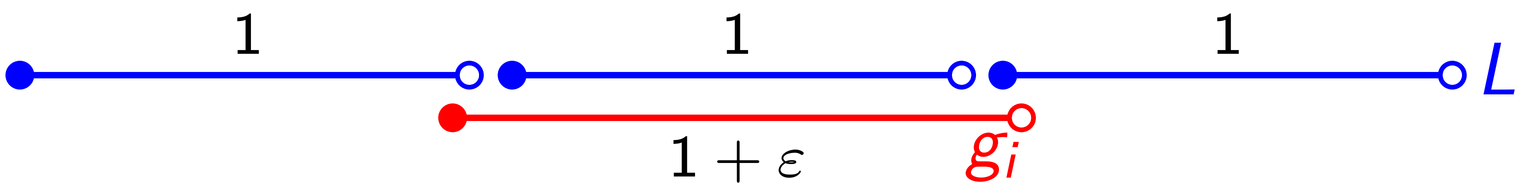
Dann gilt $A = G_1 \dot\cup G_2 \dot\cup \cdots \dot\cup G_k$ und $L = L_1 \dot\cup L_2 \dot\cup \ldots \dot\cup L_k$.

„$\subseteq$“: GA wählt so lange Intervalle aus, bis es keine mehr gibt.

„$\supseteq$“: klar, da $G_1 \subseteq A$, $G_2 \subseteq A$, $\ldots$, $G_k \subseteq A$

Wie gut/schlecht ist der 2. GA?

Behauptung: Für $i = 1, \ldots, k$ gilt $\ell(L_i) < 3\ell(g_i)$.



Beweis. (a) g_i ist nach Wahl ein längstes Intervall in G_i

(b) jedes $a \in L_i$ schneidet g_i

(c) Intervalle in L_i sind paarweise disjunkt

$$\Rightarrow \text{OPT} = \ell(L) = \sum_{i=1}^{k} \ell(L_i) < 3 \sum_{i=1}^{k} \ell(g_i) = 3\ell(G)$$

$$\Rightarrow \ell(G) > \text{OPT}/3$$

$\Rightarrow$ 2. GA liefert *immer* mind. 1/3 der maximalen Gesamtlänge.

Also ist der 2. GA ein *Faktor-(1/3)-Approximationsalgorithmus*.

Approxi... *hä?*

„All exact science is dominated by the idea of approximation."
Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*. z.B. Ablaufplanung

Sei ζ die *Zielfunktion* von Π: Lösung $\mapsto \mathbb{Q}_{\geq 0}$. $\zeta = \ell$

Sei γ eine Zahl ≤ 1. $\gamma = 1/3$

Ein Algorithmus $\mathcal{A}$ heißt γ-*Approximation*, wenn

- $\mathcal{A}$ für jede Instanz I von Π eine Lösung $\mathcal{A}(I)$ berechnet, so dass

$$\frac{\zeta(\mathcal{A}(I))}{\mathrm{OPT}(I)} \geq \gamma$$

ζ(optimale Lösung)

- die Laufzeit von $\mathcal{A}$ polynomiell in $|I|$ ist.

Größe der Instanz I

1/3-Approx. liefert Menge von Aktivitäten, deren Gesamtlänge mindestens 1/3 der maximal möglichen Länge ist.

$O(n \log n)$

Zurück zum dynamischen Programmieren

BottomUpDPWeighted(int[ ]s, int[ ]e)
  $n = s.\mathit{length}$
  $c =$ new int$[0..n][1..n+1]$ // c_{ij} = Wert einer opt. Lsg. für A_{ij}
  for $d = 1$ **to** $n - 1$ **do** // d = „Distanz“ zwischen j und i
    for $i = 0$ **to** $(n+1) - d$ **do**
      $j = i + d$
      if a_i und a_j kompatibel **then** // falls $a_i \cap a_j = \emptyset$
        $c = 0$
        for $k = i + 1$ **to** $j - 1$ **do**
          $c' = c[i][k] + \ell(a_k) + c[k][j]$
          if $c' > c$ **then** $c = c'$
        $c[i][j] = c$
      else $c[i][j] = 0$
  return $c[0, n+1]$

NEU! Im ungewichteten Fall stand hier eine Eins.

Laufzeit? $O(n^3)$

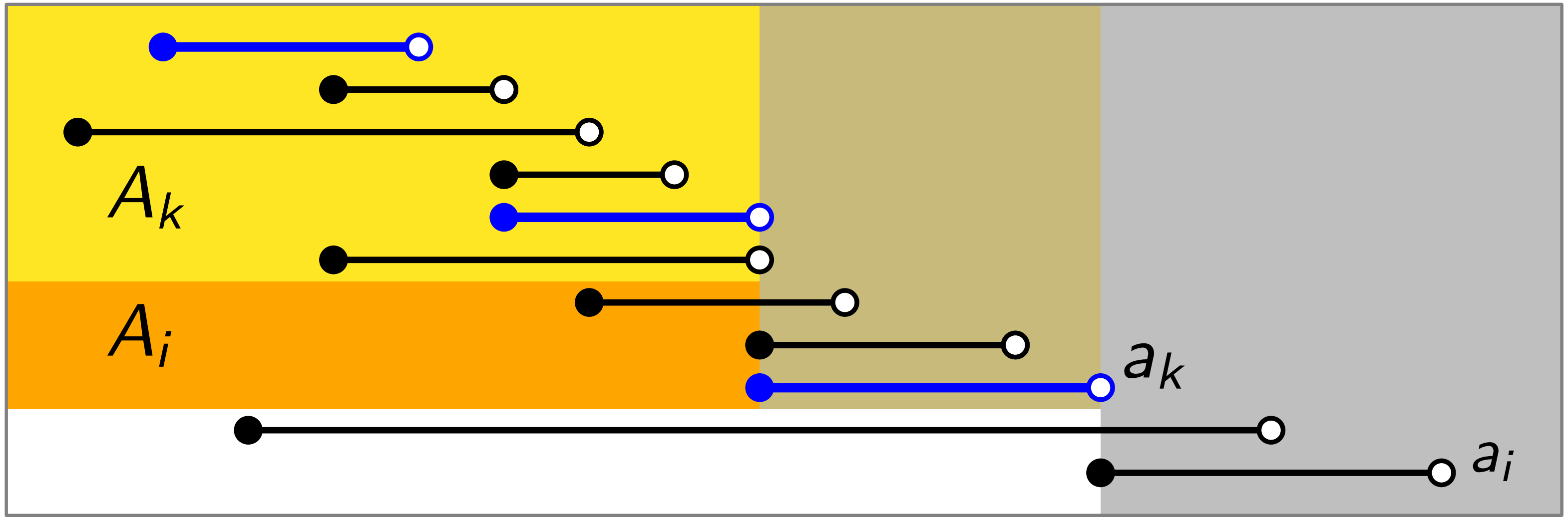
Aber warum verzweigen wir hier zweimal?

2. Wert einer optimalen Lösung rekursiv definieren

$$c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$$

Dynamisches Programmieren, aber einfacher

Für $i = 1, \ldots, n$ sei $A_i = \{a_j \in A \mid e_j \leq s_i\}$ die Menge aller Intervalle in A, die enden, bevor a_i beginnt. (Setze $A_{n+1} = A$.)



$$\left.\begin{array}{l}\text{Eine optimale Lösung für } A_i \text{ besteht aus:}\\ \text{– einem letzten Intervall } a_k \text{ und}\\ \text{– einer optimalen Lösung für } A_k.\end{array}\right\} \begin{array}{l}\text{optimale}\\ \text{Teilstruktur!}\end{array}$$

Also gilt für den Wert c_i einer optimalen Lösung für A_i:

$$c_i \;=\; \max_{a_k \in A_i} \; c_k + \ell(a_k)$$

Dynamisches Programmieren, aber einfacher

Also gilt für den Wert c_i einer optimalen Lösung für A_i:

$$c_i \;=\; \max_{a_k \in A_i} \; c_k + \ell(a_k)$$

Erinnern wir uns...
c_{n+1} ist der Wert der optimalen Lösung für $A_{n+1} = A$.

Also genügt es $c_1, \ldots, c_{n+1}$ zu berechnen.

Laufzeit? $O(n^2)$ **Work out the details!**

Resultate:

- Der 2. Greedy-Alg. findet in $O(n \log n)$ Zeit eine Lösung, die *mindestens 1/3 des maximalen Ertrags* garantiert.
- Unser neues DP findet in $O(n^2)$ Zeit eine Lösung mit *maximalem* Ertrag.

Trade-Off zwischen Zeit und Qualität!