

# Algorithmen und Datenstrukturen

## Vorlesung 23: Greedy- und Approximationsalgorithmen

# Operations Research

## Optimierung für Wirtschaftsabläufe:

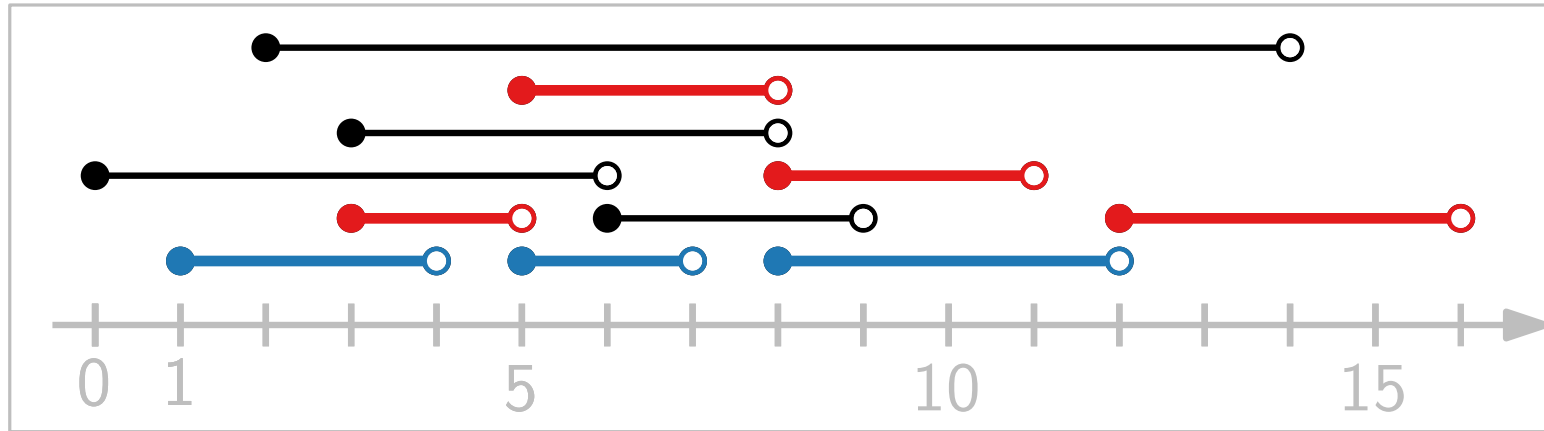
- Standortplanung
- Ablaufplanung
- Flottenmanagement
- Pack- und Zuschnittprobleme
- ...

## Werkzeuge:

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

# Ein einfaches Problem der Ablaufplanung

**Gegeben:** Menge  $A = \{a_1, \dots, a_n\}$  von *Aktivitäten* mit  $a_1 = [s_1, e_1), \dots, a_n = [s_n, e_n)$ .



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

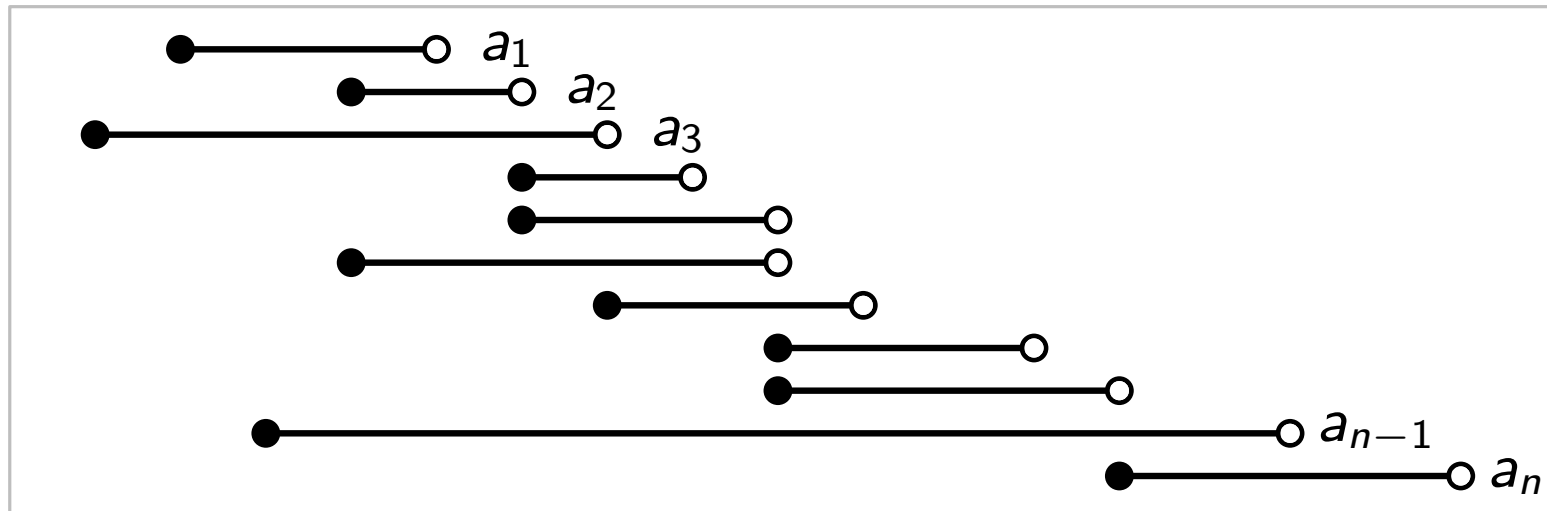
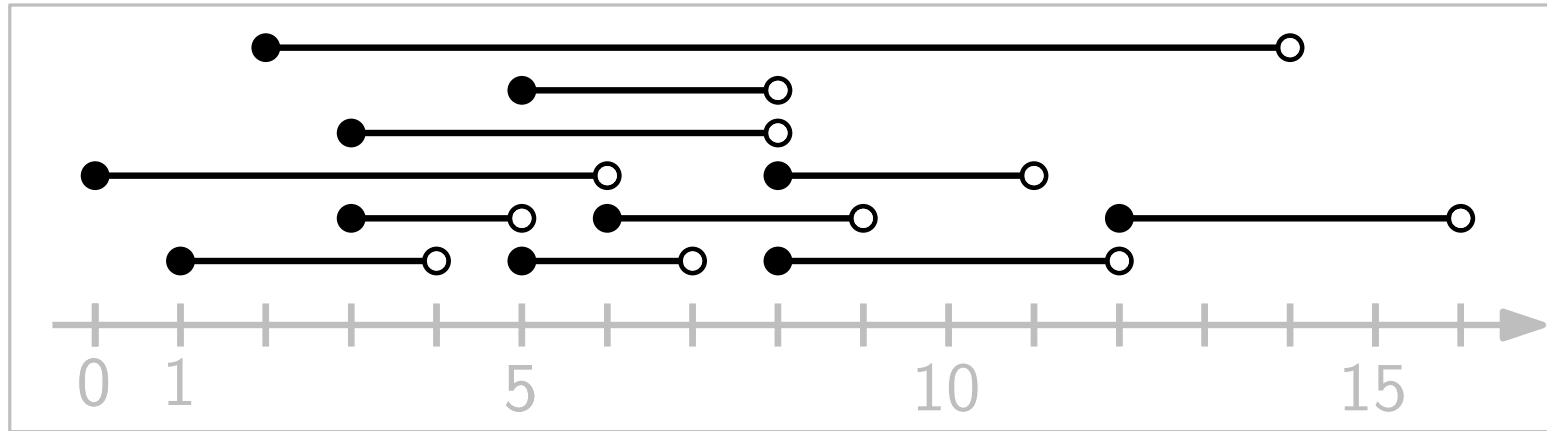
Die Aktivitäten in  $A' \subset A$  sind *paarweise kompatibel*, wenn für jedes Paar  $a_i, a_j \in A'$  gilt, dass  $a_i$  und  $a_j$  kompatibel sind.

**Gesucht:** eine größtmögliche Menge paarweise kompatibler Aktivitäten.

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

# Ein kleiner technischer Trick

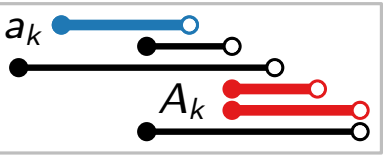
Wir nummerieren (für den Rest der Vorlesung) die Aktivitäten so, dass für die Endtermine gilt  $e_1 \leq e_2 \leq \dots \leq e_n$ .



# Charakterisierung optimaler Lösungen

**Idee:** Sei  $L$  optimale 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] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
```

```
  // Hier: falls nötig, sortiere Aktivitäten nach Endzeiten.
```

```
  return GreedyRecursiveMain(s, e, 0)
```

```
GreedyRecursiveMain(int[] s, int[] e, int k)    // best. Lsg. für  $A_k$ 
```

```
  m = k + 1; n = s.length
```

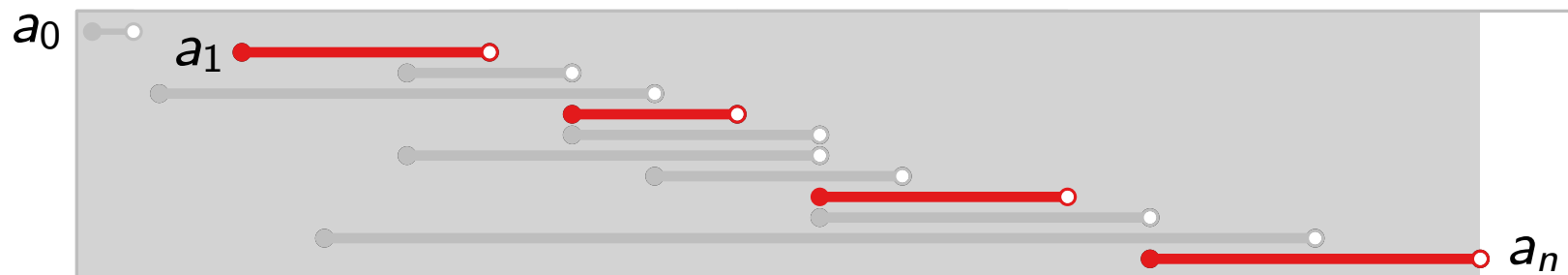
```
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
```

```
  while m  $\leq$  n and s[m] < e[k] do
```

```
     $\perp$  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] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
    // Hier: falls nötig, sortiere Aktivitäten nach Endzeiten.
    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 ≤ n and s[m] < e[k] do
        m = m + 1
    if m > n then return  $\emptyset$ 
    else return  $\{a_m\} \cup \text{GreedyRecursiveMain}(s, e, m)$ 
```

## 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  $\emptyset$ 
  L = {a1}
  k = 1 // höchster Index in L
  for m = 2 to n do
    if s[m] ≥ e[k] then
      L = L ∪ {am}
      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.

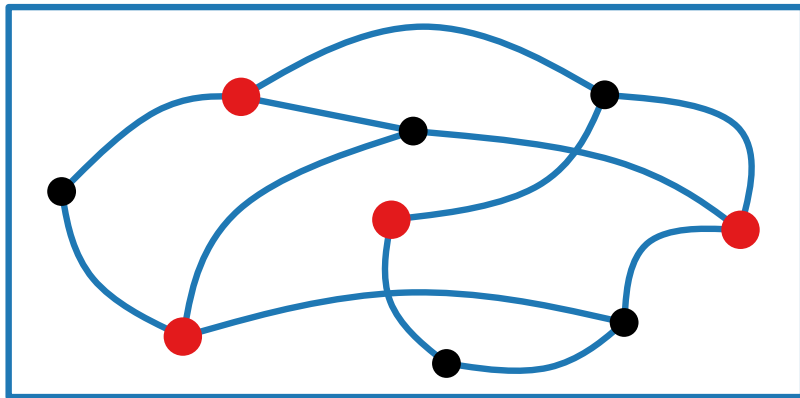
# Food for Thought

1. Welches allgemeinere Ablaufproblem kann der Greedy-Algorithmus (GA) nicht lösen?

Wenn jede Aktivität  $a \in A$  ihren eigenen Ertrag  $w(a)$  erbringt:

Finde  $L \subseteq A$  mit  $L$  kompatibel und  $w(L) := \sum_{a \in L} w(a)$  max.

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



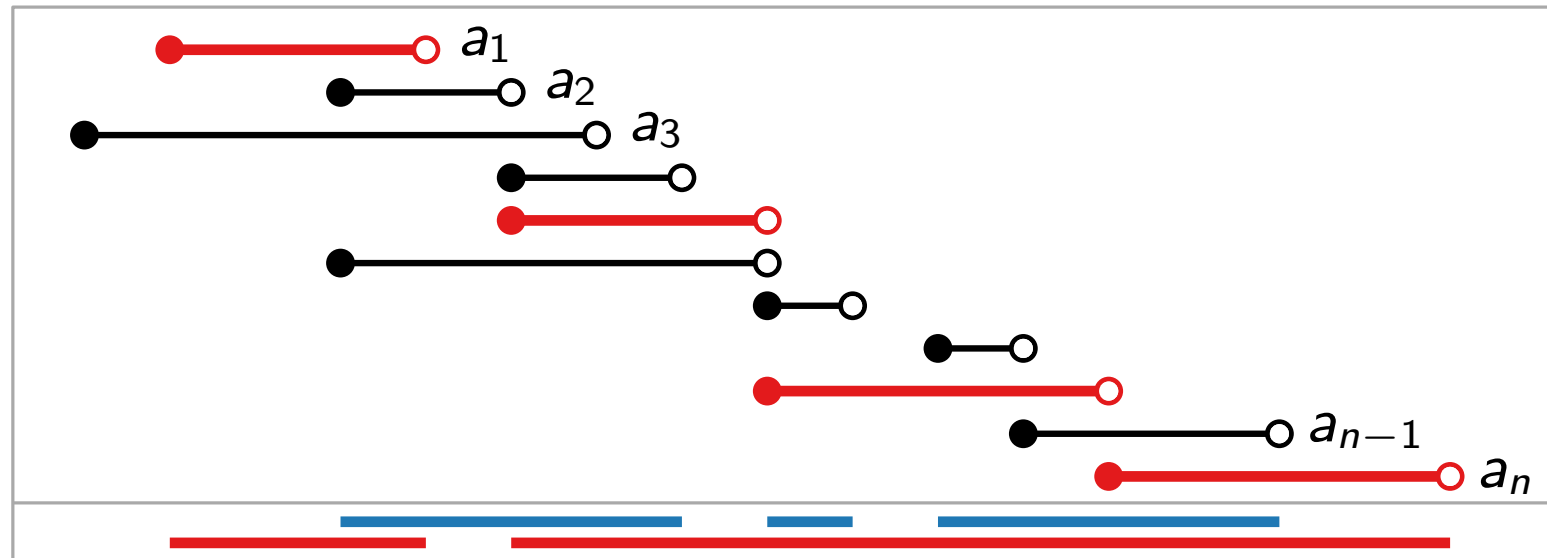
Finde eine größte Teilmenge  $U$  der Knoten, so dass keine zwei Knoten in  $U$  benachbart sind.

- Was hat guM mit unserem Ablaufplanungsproblem zu tun?
- Welche Graphen kommen bei der Ablaufplanung nicht vor?
- Kann man guM mittels dynamischer Programmierung oder Greedy-Alg. 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.

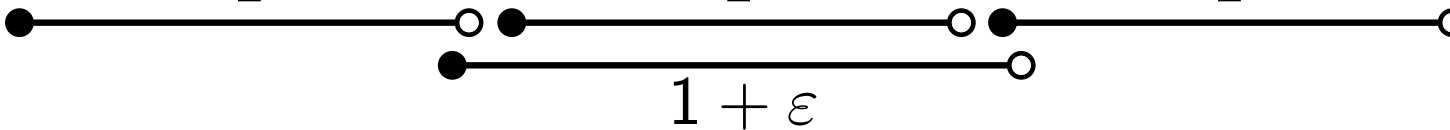
**Grund:** Intervalle  $\hat{=}$  Prozesse, die die gleiche Ressource nutzen; der Gesamtertrag ist proportional zur Auslastung.

# 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.:* 

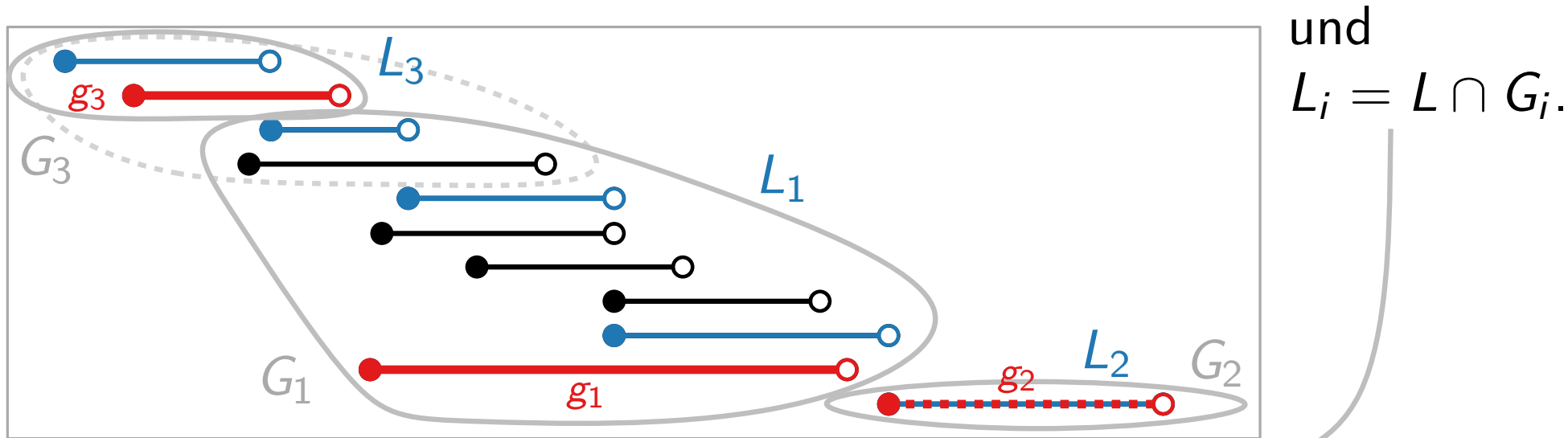
**Aufgabe:** Können Sie den 2. GA in  $O(n \log n)$  Zeit implementieren?  
*Tipp: Gehen Sie so ähnlich wie Kruskal vor!*

# Wie gut/schlecht ist der 2. GA?

Betrachte eine optimale Lösung  $L \subseteq A$ .

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

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



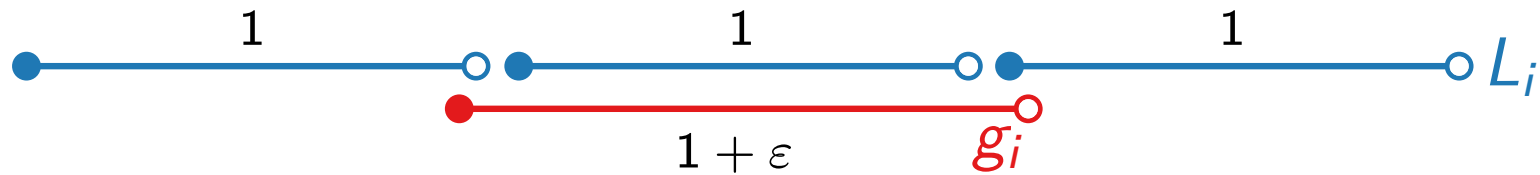
Dann gilt  $A = G_1 \dot{\cup} G_2 \dot{\cup} \dots \dot{\cup} G_k$  und  $L = L_1 \dot{\cup} L_2 \dot{\cup} \dots \dot{\cup} L_k$ .

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

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

# Wie gut/schlecht ist der 2. GA?

**Behauptung:** Für  $i = 1, \dots, k$  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.**

# Approxim. . . hä?

„All exact science is dominated by the idea of approximation.“

Sei  $\Pi$  ein *Maximierungsproblem*.

Sei  $\zeta$  die *Zielfunktion* von  $\Pi$ : Lösung  $\mapsto \mathbb{Q}_{\geq 0}$ .

Sei  $\gamma$  eine Zahl  $\leq 1$ .

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

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

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

$\zeta(\text{optimale Lösung})$   $\swarrow$   $\text{OPT}(I)$

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

z.B. Ablaufplanung

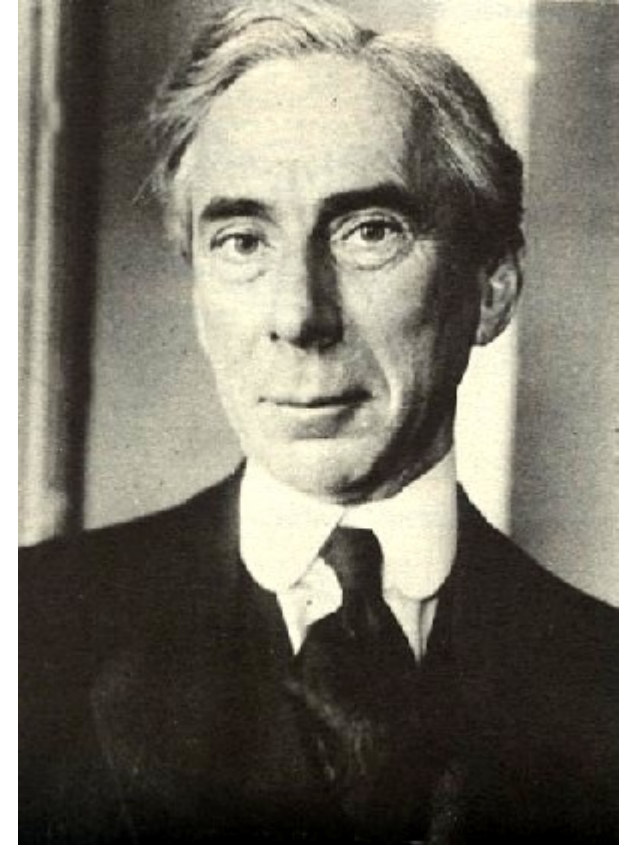
$$\zeta = \ell$$

$$\gamma = 1/3$$

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

Größe der Instanz  $I$

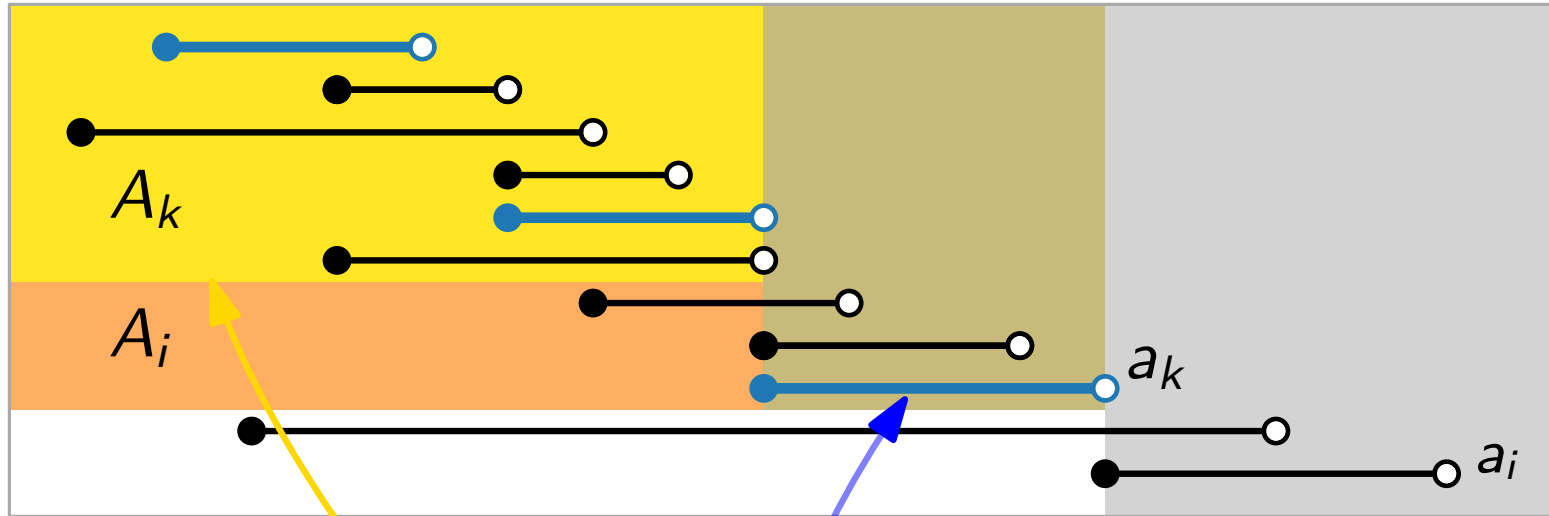
$$O(n \log n)$$



Bertrand Russell  
(1872–1970)

# Ein exakter Algorithmus...

Für  $i = 1, \dots, n$  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  $A_{n+1} = A$ .)



Eine optimale Lösung für  $A_i$  besteht aus:

- *einem* letzten Intervall  $a_k$  und
- einer optimalen Lösung für  $A_k$ .

} optimale Teilstruktur!

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

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



# ... ein Dynamisches Programm!

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

$$c_i = \max_{a_k \in A_i} c_k + \ell(a_k) \quad \left. \begin{array}{l} \text{BERECHNUNG EINES} \\ \text{TABELLENEINTRAGS} \end{array} \right\} \text{ in je } O(n) \text{ Zeit}$$

Erinnern wir uns...

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

Also genügt es **TABELLE**  
 $c_1, \dots, c_{n+1}$  zu berechnen, wobei  $c_1 = 0$ .  
Größe  $O(n)$

**Laufzeit?**  $O(n^2)$

**Schreiben Sie den Pseudocode!**

## Resultate:

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

*Trade-Off zwischen Zeit und Qualität!*