

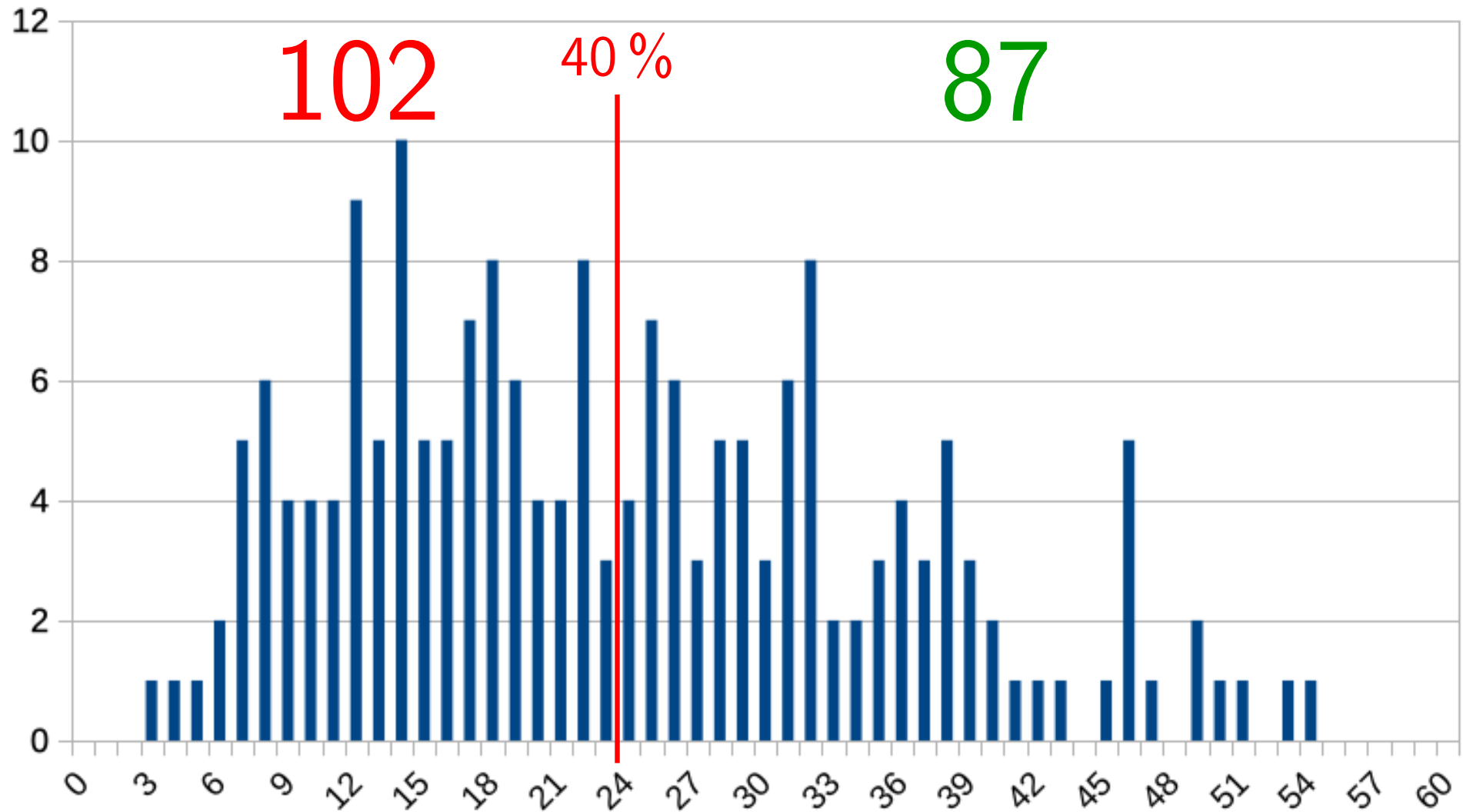
Algorithmen und Datenstrukturen

Wintersemester 2021/22

23. Vorlesung

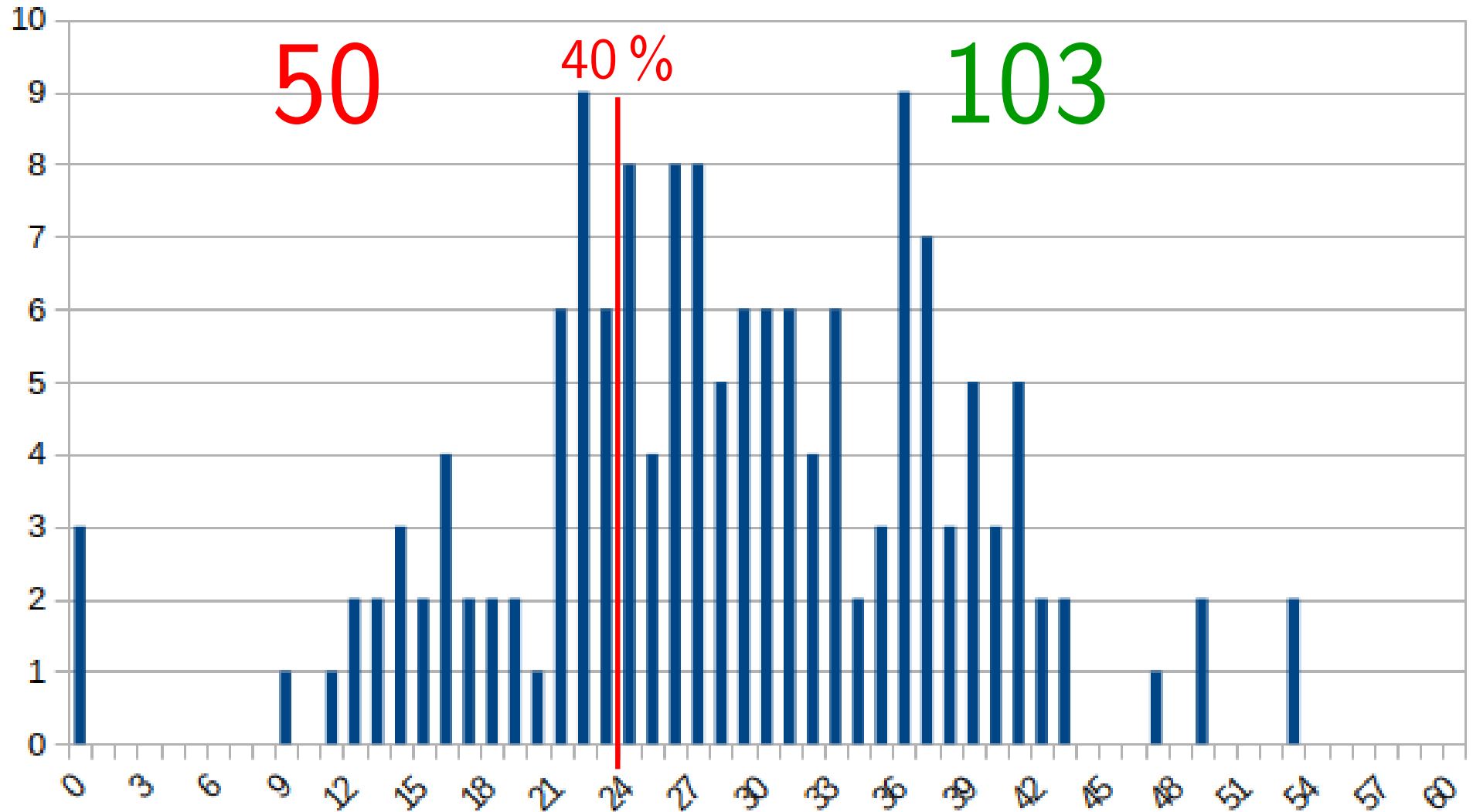
Greedy- und Approximationsalgorithmen

Zwischentest I



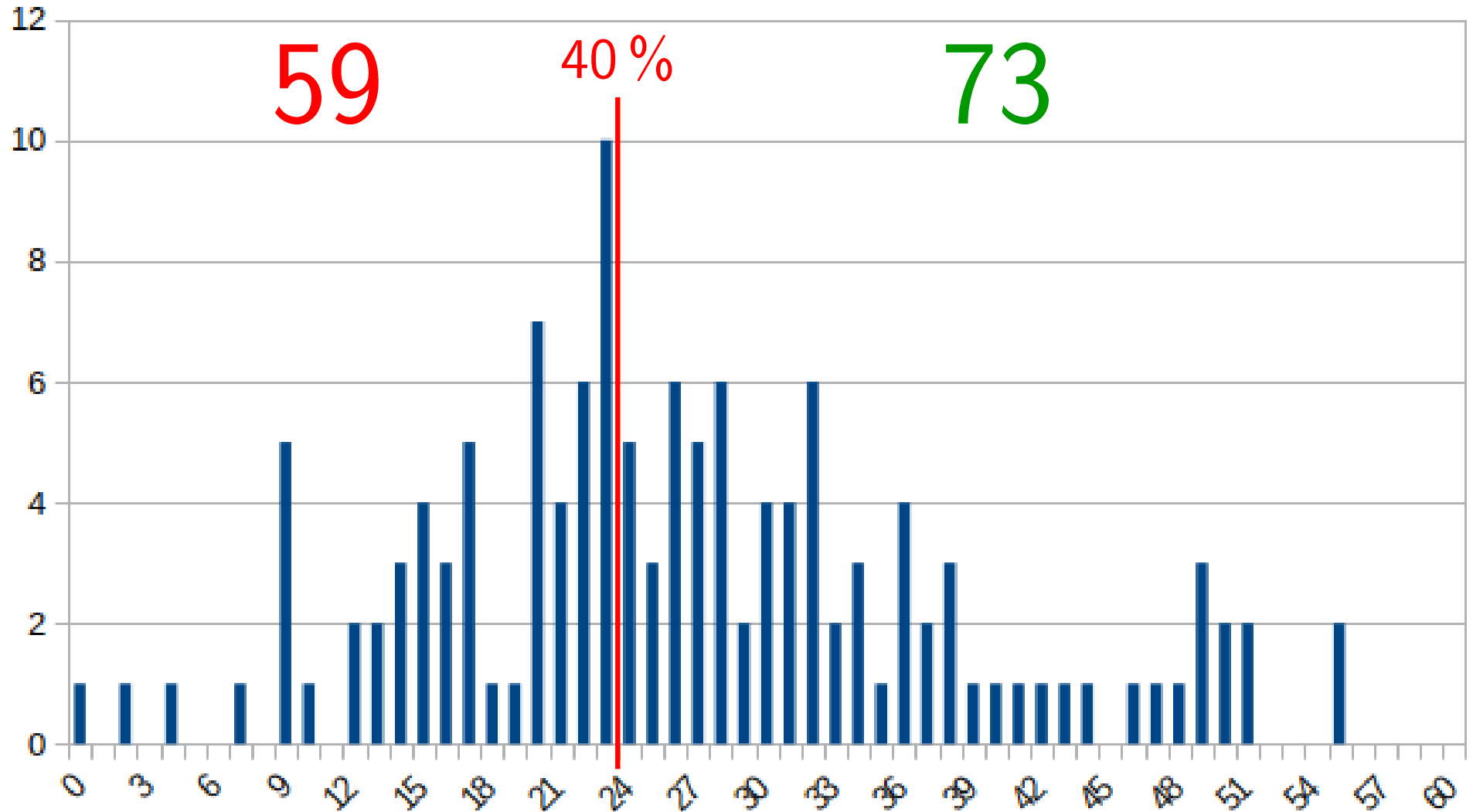
$n = 189$; Median= 21,5; Mittelwert= 23,3

Zwischentest II



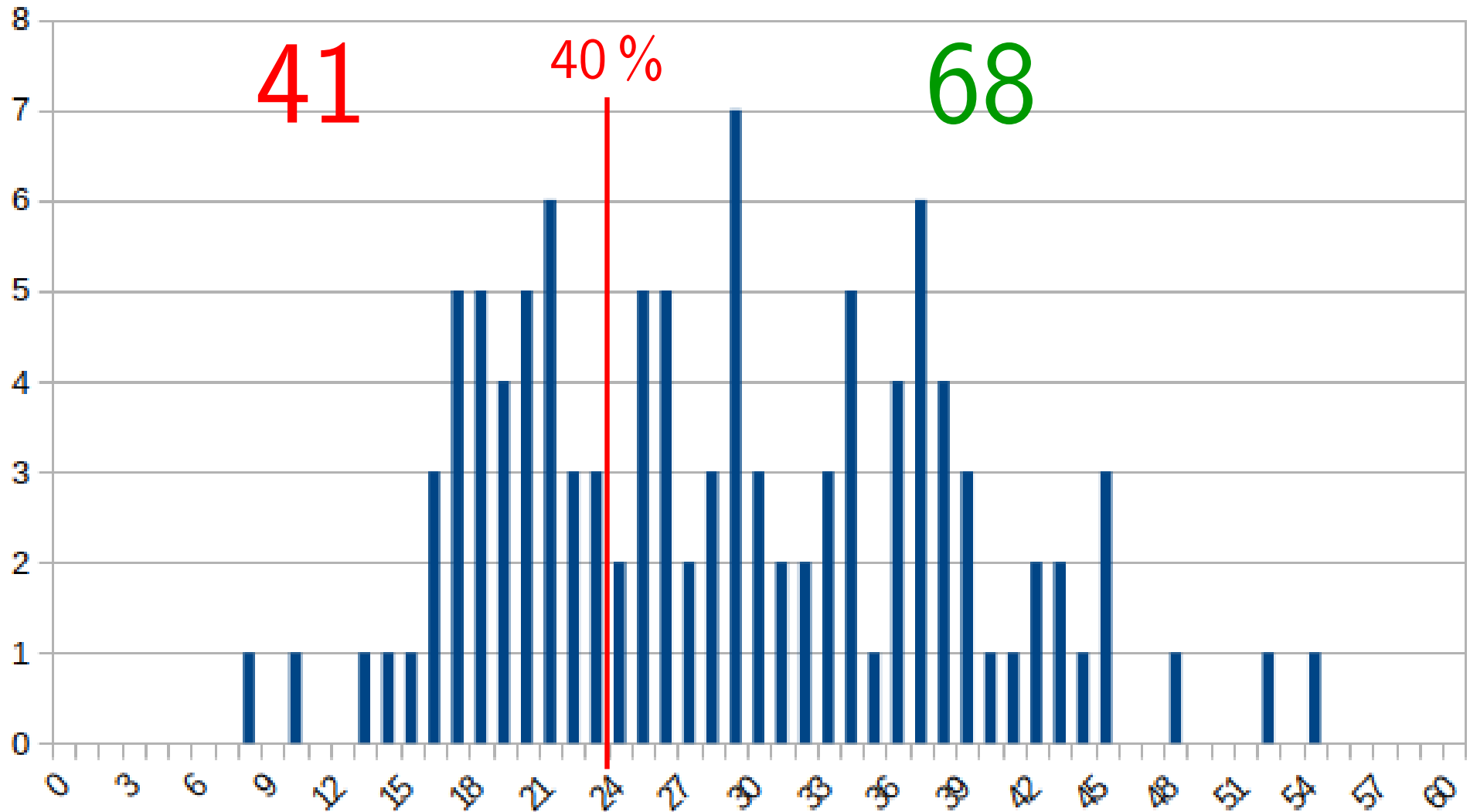
$n = 153$; Median= 27,5; Mittelwert= 28,1

Zwischentest III



$n = 132$; Median= 25,3; Mittelwert= 26,5

Zwischentests gesamt



$n = 109$; Median= 27,7; Mittelwert= 28,1

Operations Research

Optimierung für Wirtschaftsabläufe:

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

Operations Research

Optimierung für Wirtschaftsabläufe:

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

Werkzeuge:

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

Operations Research

Optimierung für Wirtschaftsabläufe:

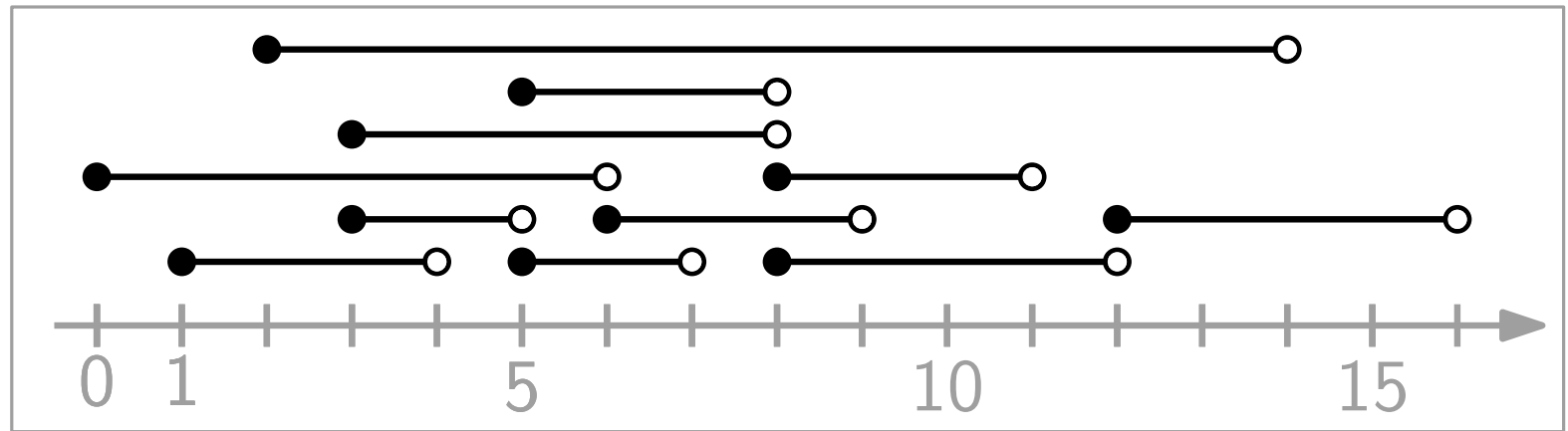
- Standortplanung
- Ablaufplanung
- Flottenmanagement
- Pack- und Zuschnittprobleme
- ...

Werkzeuge:

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

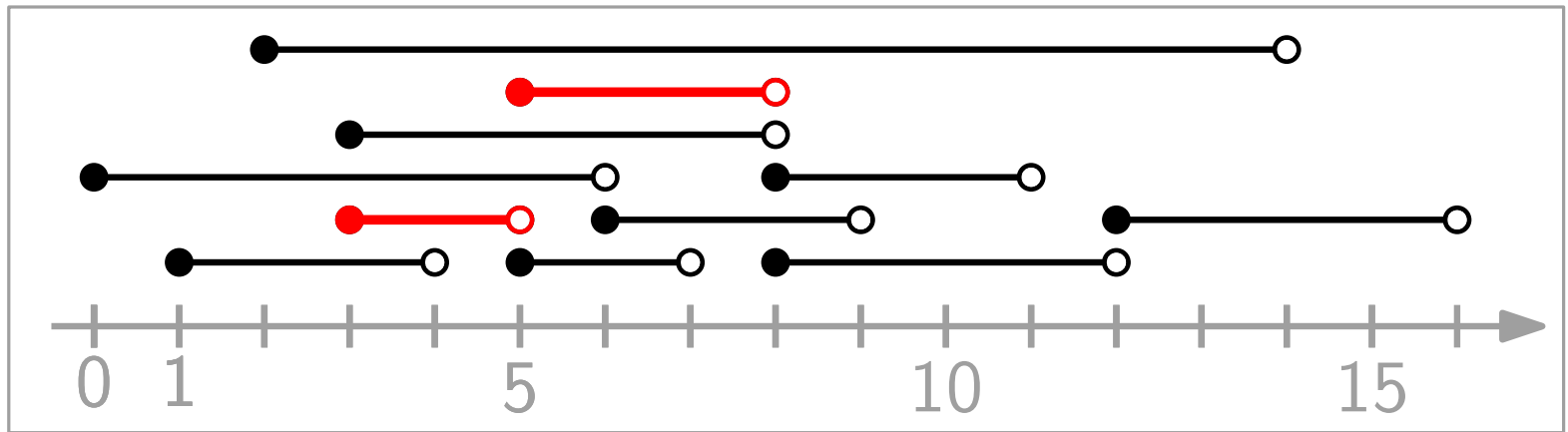
Ein einfaches Problem der Ablaufplanung

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



Ein einfaches Problem der Ablaufplanung

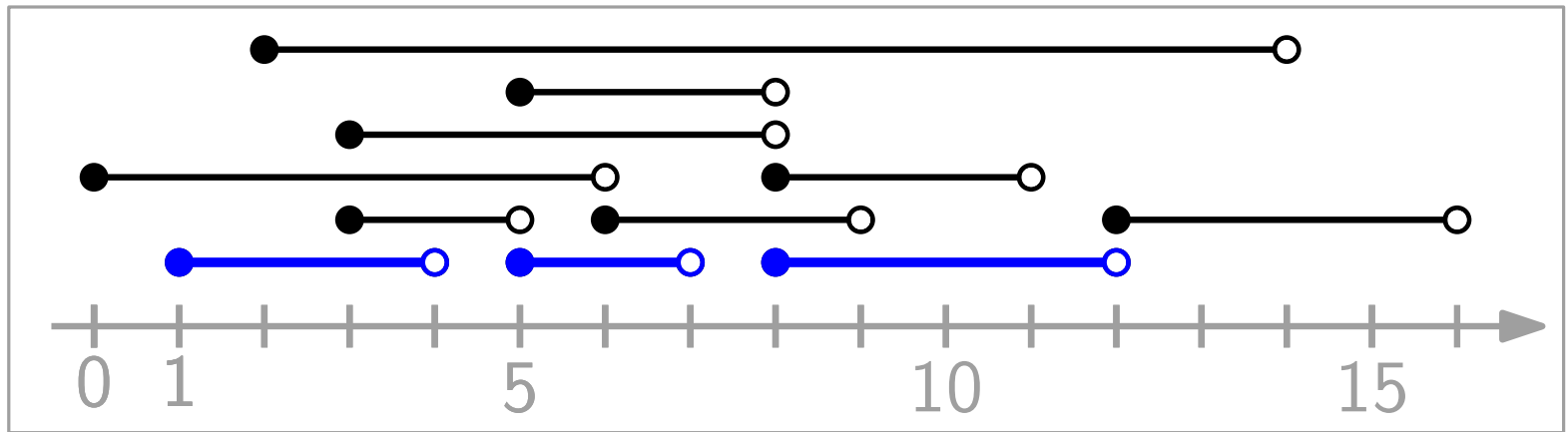
Gegeben: Menge $A = \{a_1, \dots, a_n\}$ von *Aktivitäten*,
wobei für $i = 1, \dots, n$ gilt $a_i = [s_i, e_i)$.



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

Ein einfaches Problem der Ablaufplanung

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

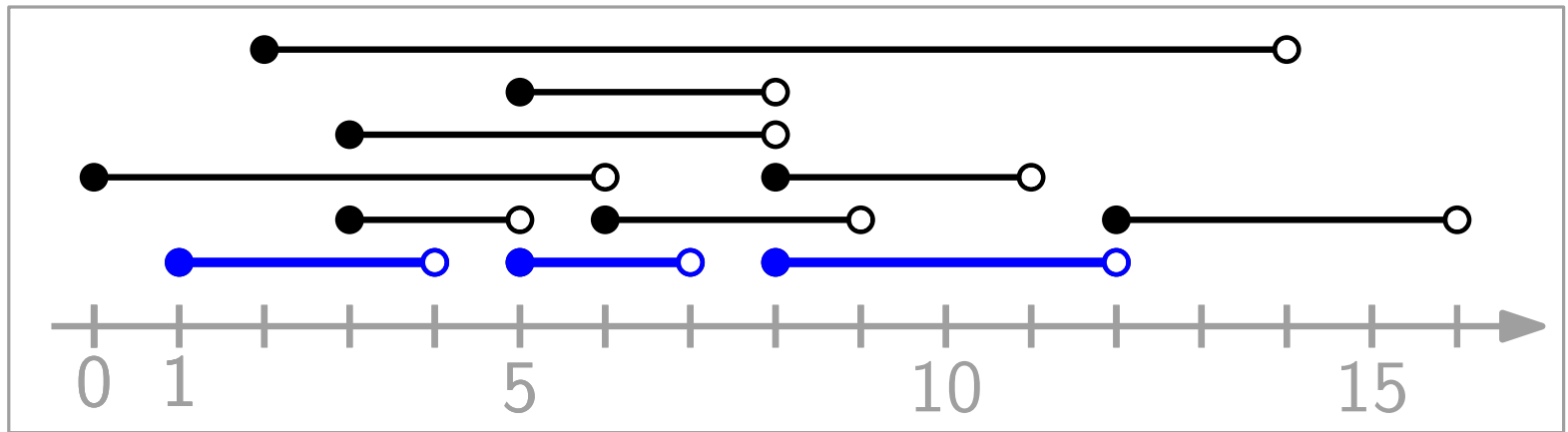


a_i und a_j sind *kompatibel*, wenn $a_i \cap a_j = \emptyset$.

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.

Ein einfaches Problem der Ablaufplanung

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



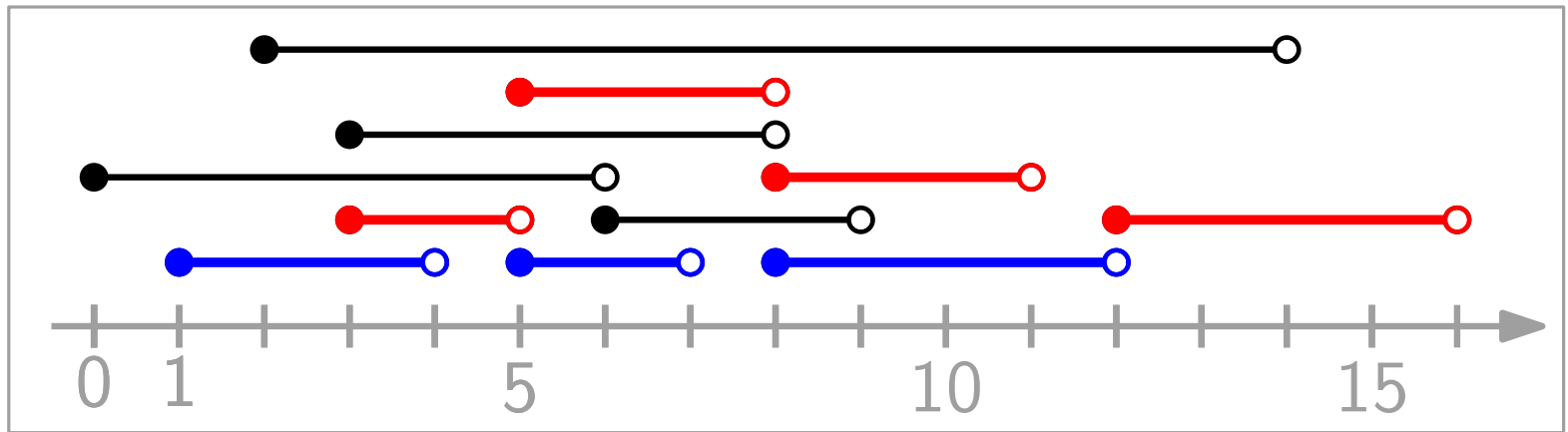
a_i und a_j sind *kompatibel*, wenn $a_i \cap a_j = \emptyset$.

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.

Ein einfaches Problem der Ablaufplanung

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



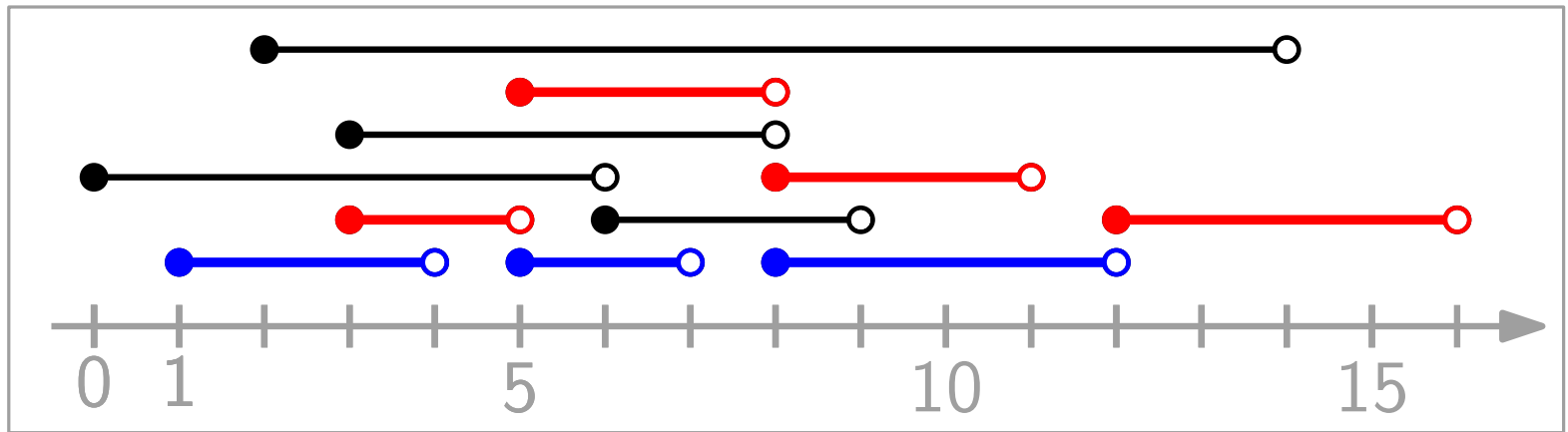
a_i und a_j sind *kompatibel*, wenn $a_i \cap a_j = \emptyset$.

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.

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 $a_i \cap a_j = \emptyset$.

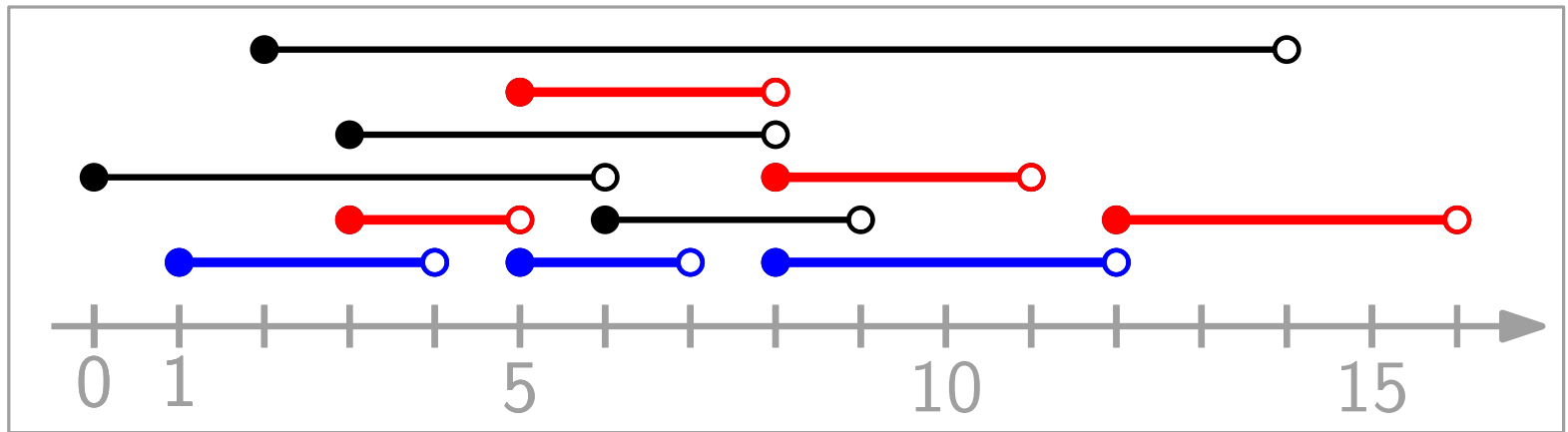
Die Aktivitäten in $A' \subset A$ sind *paarweise kompatibel*, wenn für jedes Paar $a_i, a_j \in A'$ gilt, dass a_i und a_j kompatibel sind.

Gesucht: eine größtmögliche Menge paarweise kompatibler Aktivitäten.

Grund:

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 $a_i \cap a_j = \emptyset$.

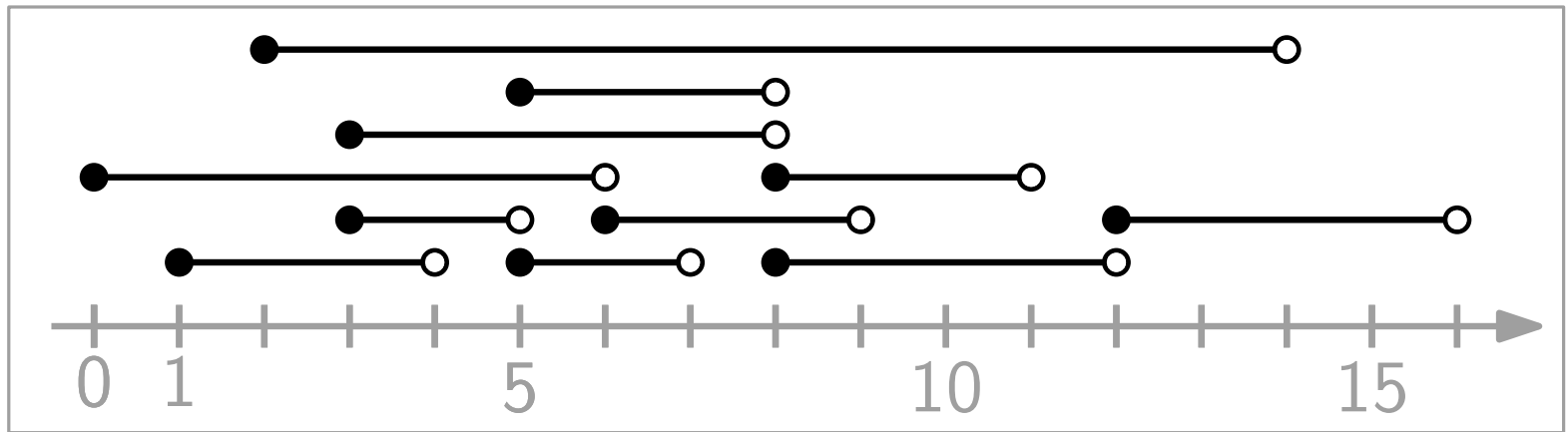
Die Aktivitäten in $A' \subset A$ sind *paarweise kompatibel*, wenn für jedes Paar $a_i, a_j \in A'$ gilt, dass a_i und a_j kompatibel sind.

Gesucht: eine größtmögliche Menge paarweise kompatibler Aktivitäten.

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

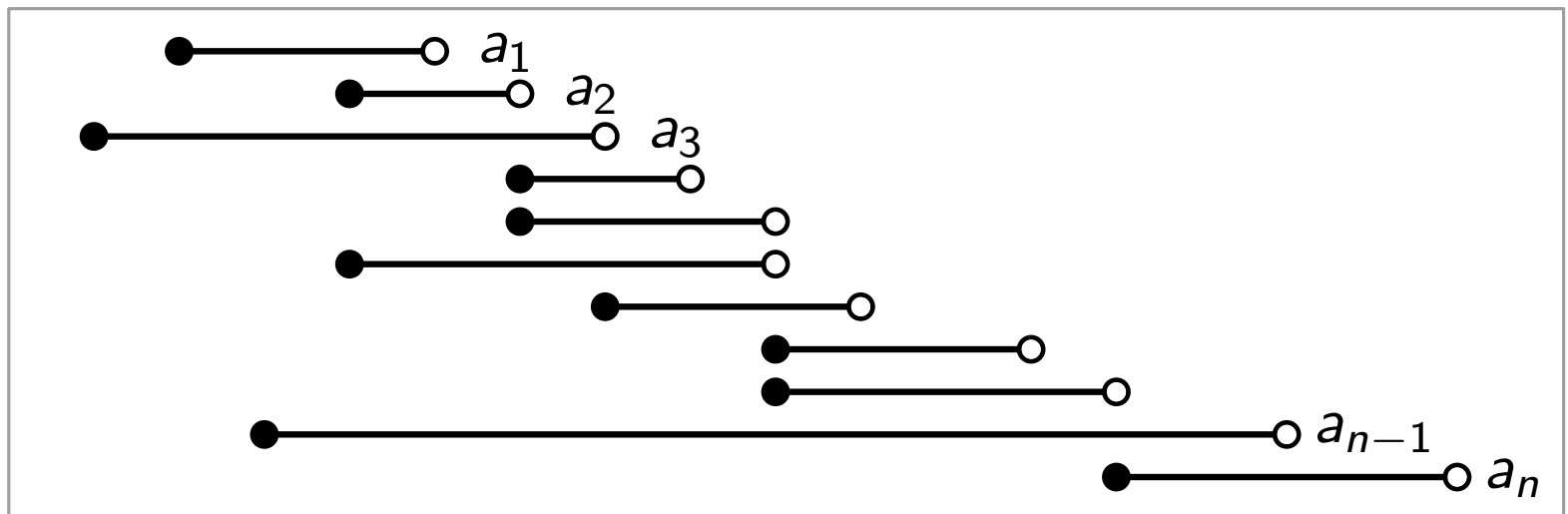
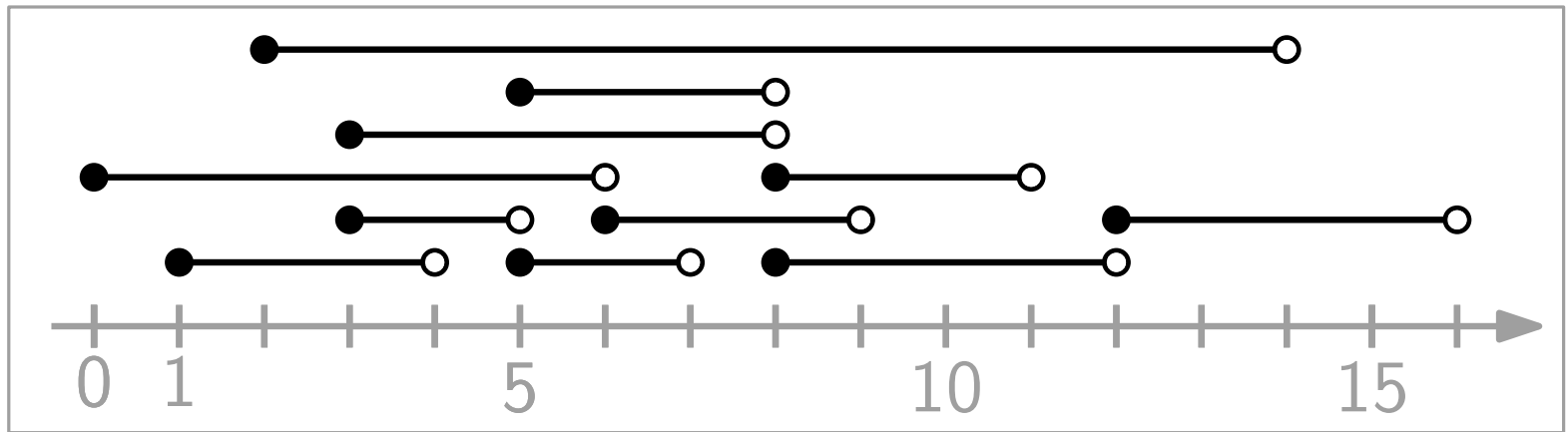
Ein kleiner technischer Trick

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



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

Charakterisierung optimaler Lösungen

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

Charakterisierung optimaler Lösungen

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

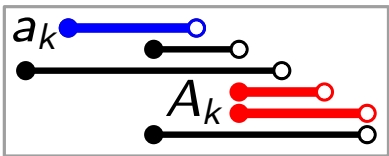
Charakterisierung optimaler Lösungen

- 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.

Charakterisierung optimaler Lösungen

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.

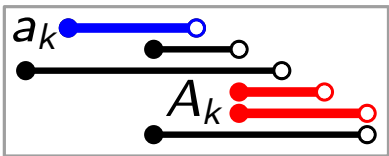


Sei $A_k = \{a_i \in A : s_i \geq e_k\}$ die Menge der Aktivitäten, die nach Ablauf von a_k beginnen.

Charakterisierung optimaler Lösungen

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.



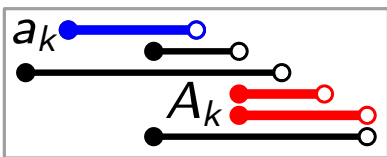
Sei $A_k = \{a_i \in A : s_i \geq e_k\}$ die Menge der Aktivitäten, die nach Ablauf von a_k beginnen.

Sei L_k eine optimale Lösung von A_k .

Charakterisierung optimaler Lösungen

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.

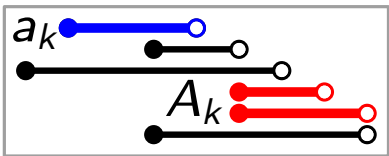
Sei L_k eine optimale Lösung von A_k .

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

Charakterisierung optimaler Lösungen

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.

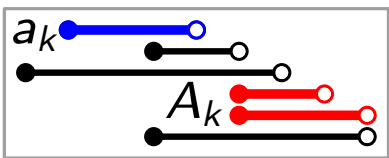
Satz. Sei $A_k \neq \emptyset$.

Sei a_m Aktivität mit frühester Endzeit in A_k .

Charakterisierung optimaler Lösungen

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 $A_k \neq \emptyset$.

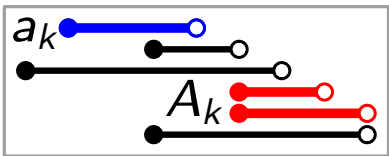
Sei a_m Aktivität mit frühester Endzeit in A_k .

$\Rightarrow$ es gibt eine opt. Lösung von A_k , die a_m enthält.

Charakterisierung optimaler Lösungen

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 .

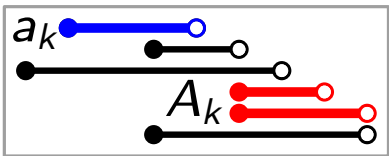
$\Rightarrow$ es gibt eine opt. Lösung von A_k , die a_m enthält.

Beweis.

Charakterisierung optimaler Lösungen

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.

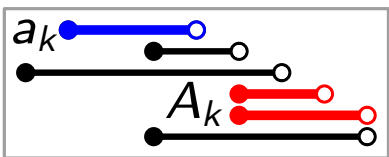
Satz. Sei $A_k \neq \emptyset$.
 Sei a_m Aktivität mit frühester Endzeit in A_k .
 $\Rightarrow$ es gibt eine opt. Lösung von A_k , die a_m enthält.

Beweis. *Austauschargument!*

Charakterisierung optimaler Lösungen

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 .

optimale Teilstruktur! $\Rightarrow$ es gibt eine opt. Lösung von A_k , die a_m enthält.

Beweis. *Austauschargument!*

Greedy – rekursiv

Satz.

Sei $A_k \neq \emptyset$.

optimale

Teilstruktur!

Sei a_m Aktivität mit frühester Endzeit in A_k .

$\Rightarrow$ es gibt eine opt. Lösung von A_k , die a_m enthält.

Greedy – rekursiv

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

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

Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

```
    e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
```

```
    // Aktivitäten nach Endzeiten sortieren, falls nötig
```

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

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

```
    return  $\{a_m\} \cup \text{GreedyRecursiveMain}(s, e, m)$ 
```

$\Rightarrow$ es gibt eine opt. Lösung von A_k , die a_m enthält.

Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

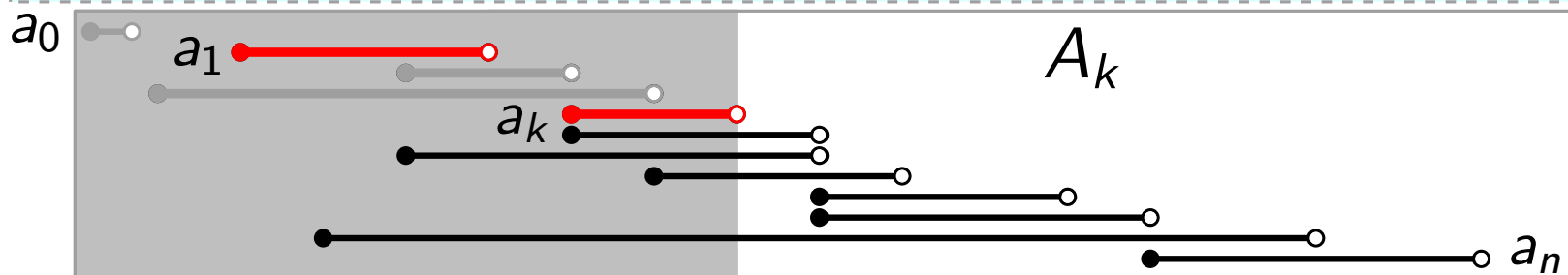
```
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
```

```
  // Aktivitäten nach Endzeiten sortieren, falls nötig
```

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

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

```
  return  $\{a_m\} \cup \text{GreedyRecursiveMain}(s, e, m)$ 
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

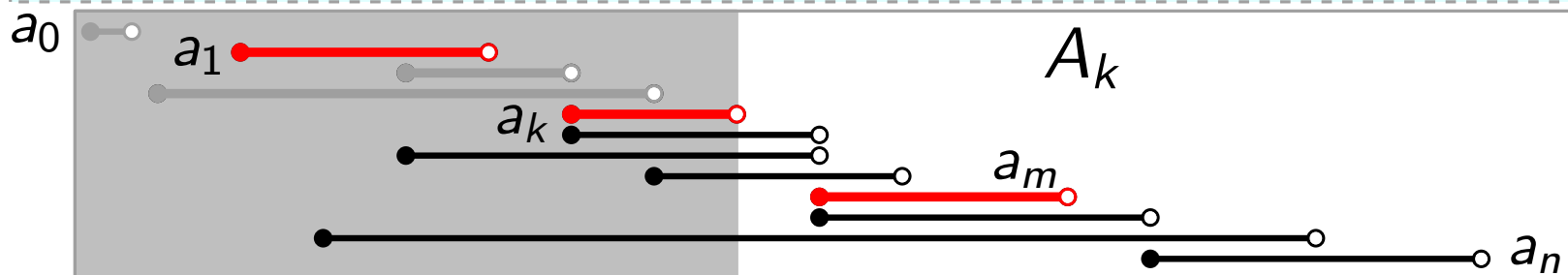
```
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
```

```
  // Aktivitäten nach Endzeiten sortieren, falls nötig
```

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

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

```
  return  $\{a_m\} \cup \text{GreedyRecursiveMain}(s, e, m)$ 
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

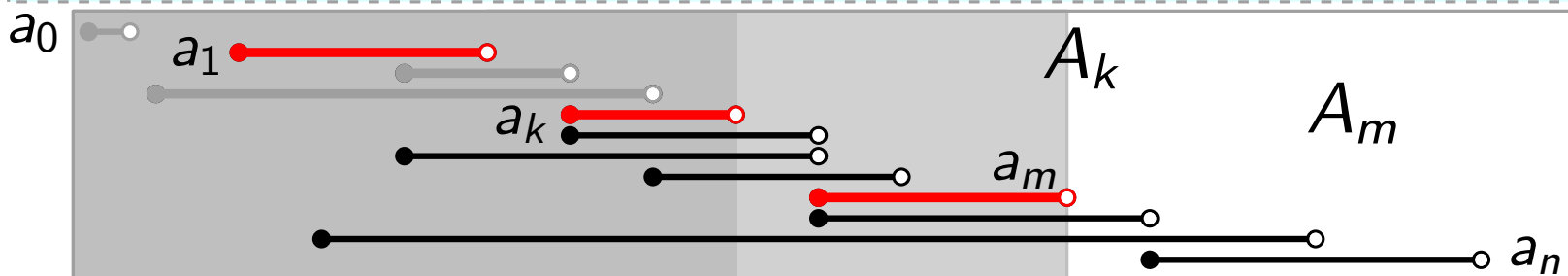
```
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
```

```
  // Aktivitäten nach Endzeiten sortieren, falls nötig
```

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

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

```
  return  $\{a_m\} \cup \text{GreedyRecursiveMain}(s, e, m)$ 
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

```
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
```

```
  // Aktivitäten nach Endzeiten sortieren, falls nötig
```

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

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

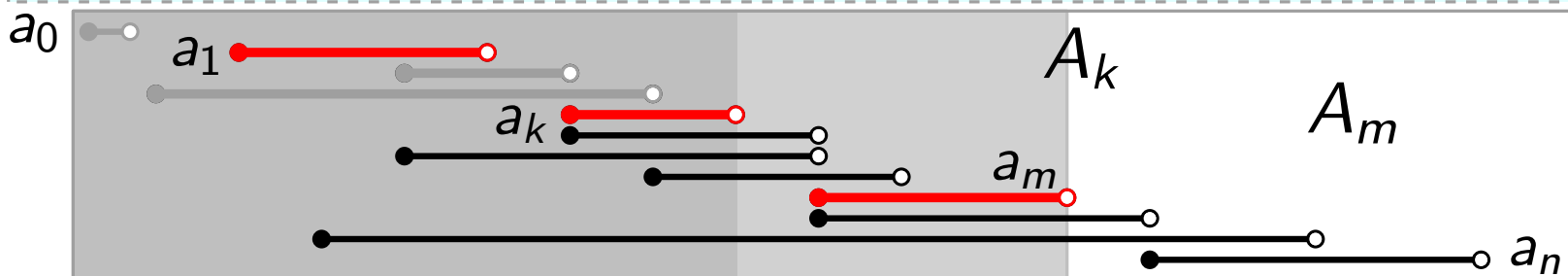
```
  m = k + 1; n = s.length
```

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

```
  while                                do
```

```
    └
```

```
  return  $\{a_m\} \cup \text{GreedyRecursiveMain}(s, e, m)$ 
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

```
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
```

```
  // Aktivitäten nach Endzeiten sortieren, falls nötig
```

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

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

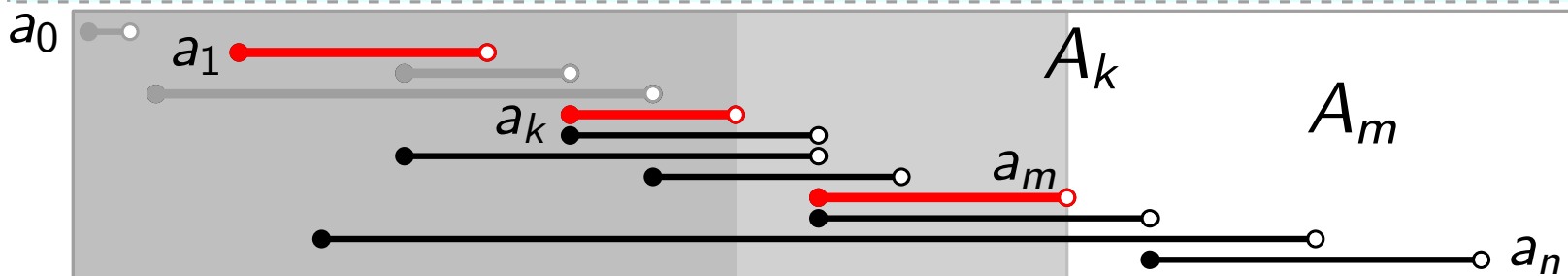
```
  m = k + 1; n = s.length
```

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

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

```
     $\perp$  m = m + 1
```

```
  return  $\{a_m\} \cup$  GreedyRecursiveMain(s, e, m)
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

```
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
```

```
  // 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
```

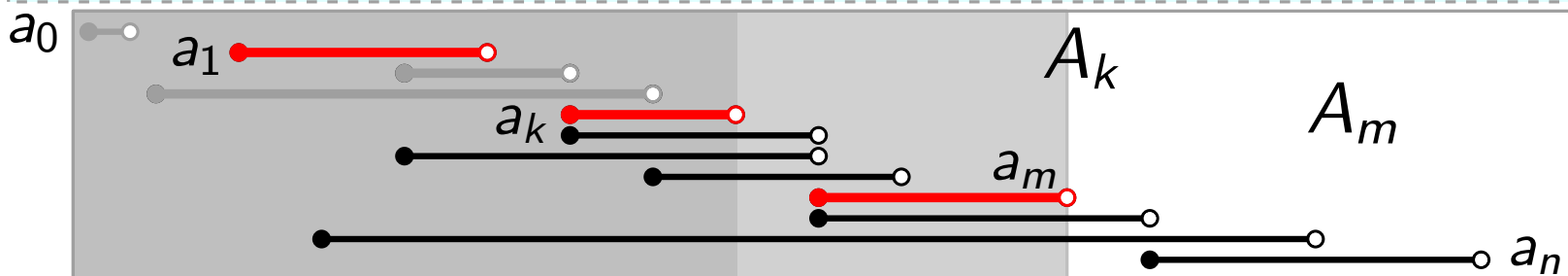
```
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
```

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

```
     $\perp$  m = m + 1
```

```
  if m > n then return  $\emptyset$ 
```

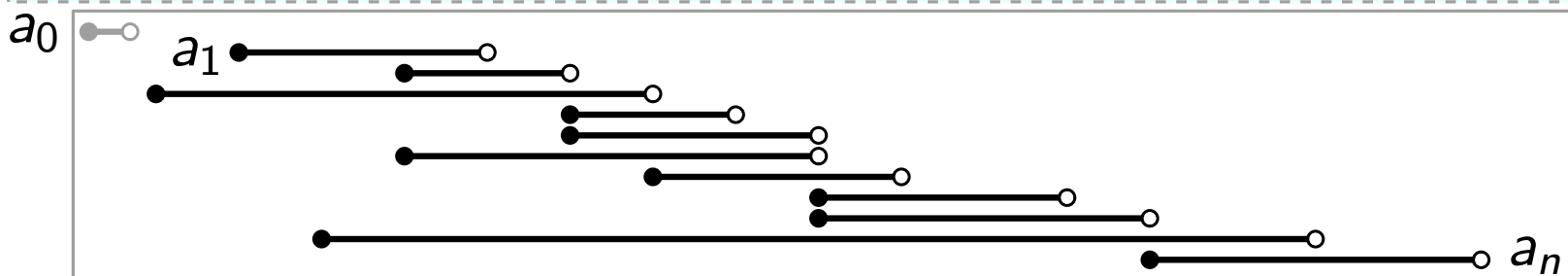
```
  else return  $\{a_m\} \cup$  GreedyRecursiveMain(s, e, m)
```



Greedy – rekursiv

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

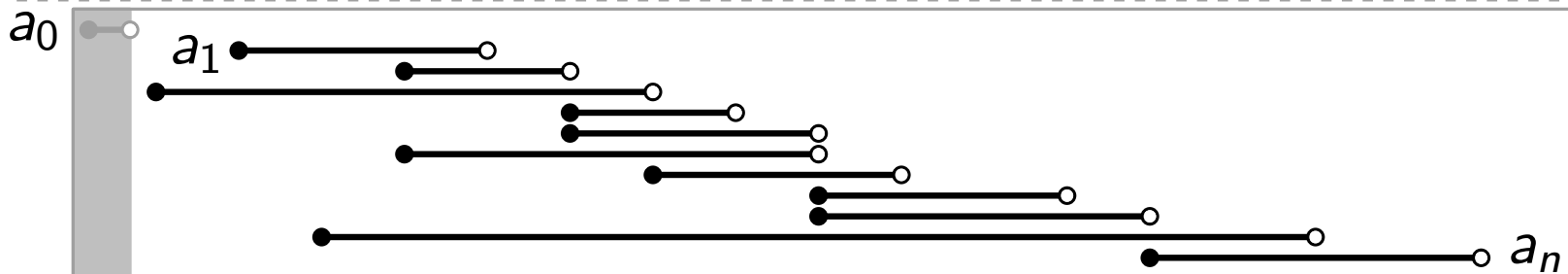
```
GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für  $A_k$ 
  m = k + 1; n = s.length
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
  while m ≤ n and s[m] < e[k] do
    m = m + 1
  if m > n then return  $\emptyset$ 
  else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)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



Greedy – rekursiv

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

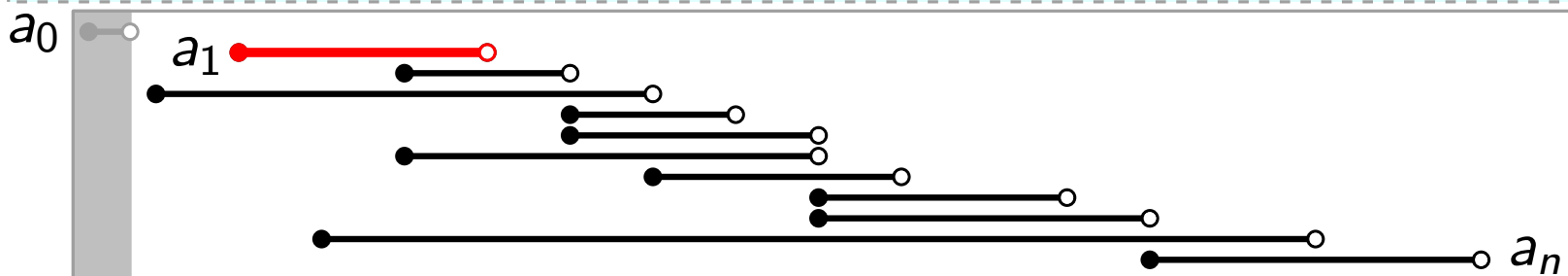
```
GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für  $A_k$ 
    m = k + 1; n = s.length
    // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
    while m ≤ n and s[m] < e[k] do
        m = m + 1
    if m > n then return  $\emptyset$ 
    else return { $a_m$ } ∪ GreedyRecursiveMain(s,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



Greedy – rekursiv

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

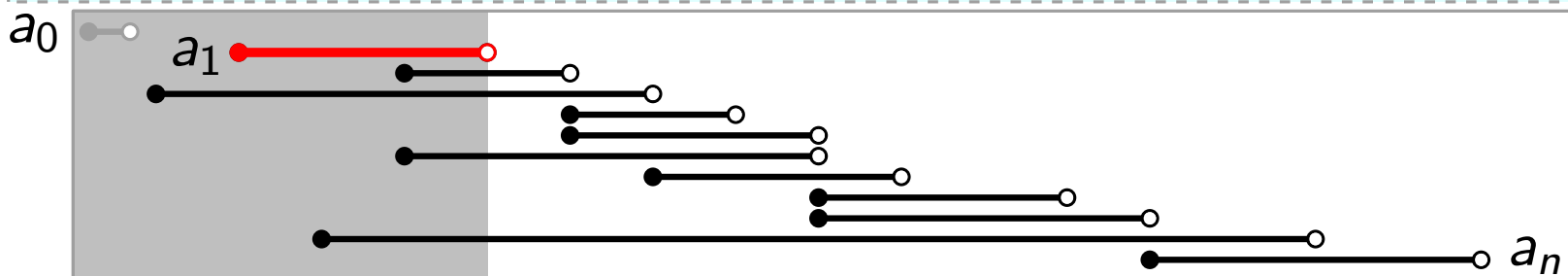
```
GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für  $A_k$ 
  m = k + 1; n = s.length
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
  while m ≤ n and s[m] < e[k] do
    m = m + 1
  if m > n then return  $\emptyset$ 
  else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)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



Greedy – rekursiv

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

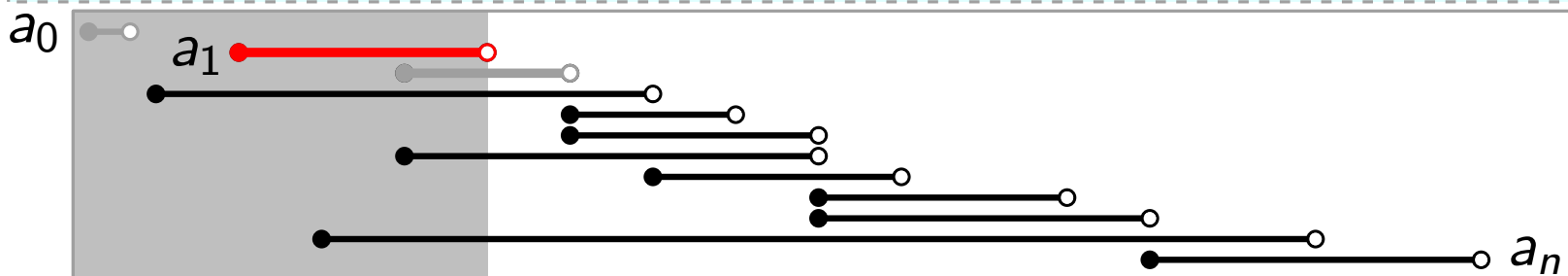
```
GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für  $A_k$ 
  m = k + 1; n = s.length
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
  while m ≤ n and s[m] < e[k] do
    m = m + 1
  if m > n then return  $\emptyset$ 
  else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)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



Greedy – rekursiv

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

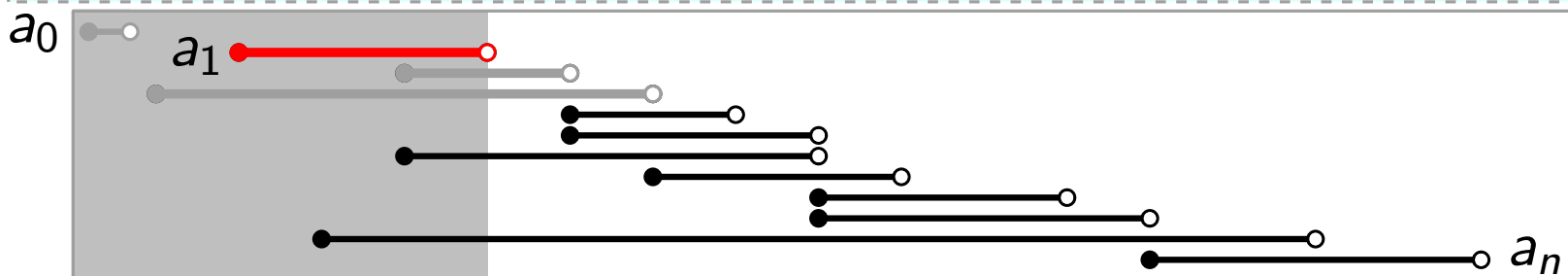
```
GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für  $A_k$ 
  m = k + 1; n = s.length
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
  while m ≤ n and s[m] < e[k] do
    m = m + 1
  if m > n then return  $\emptyset$ 
  else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)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



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
    e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
    // 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 ≤ n and s[m] < e[k] do
        m = m + 1
    if m > n then return  $\emptyset$ 
    else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

```
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
```

```
  // 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
```

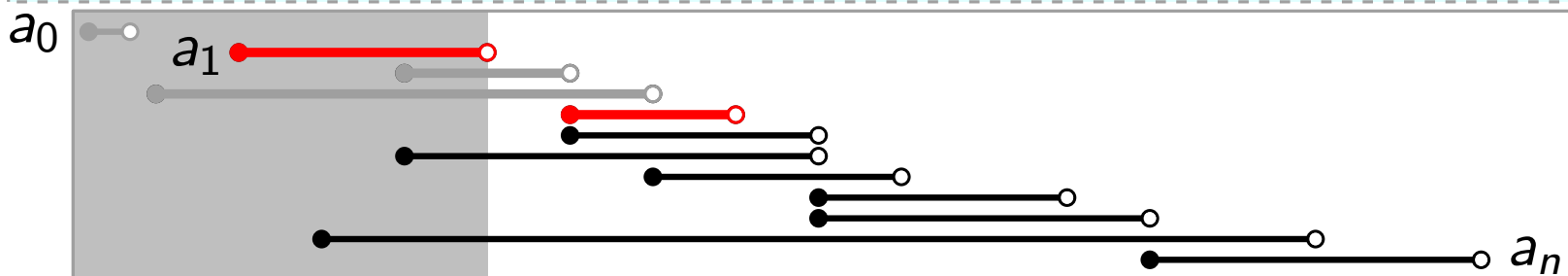
```
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
```

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

```
     $\perp$  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  $\Rightarrow A_0 = A$ 
```

```
  // 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
```

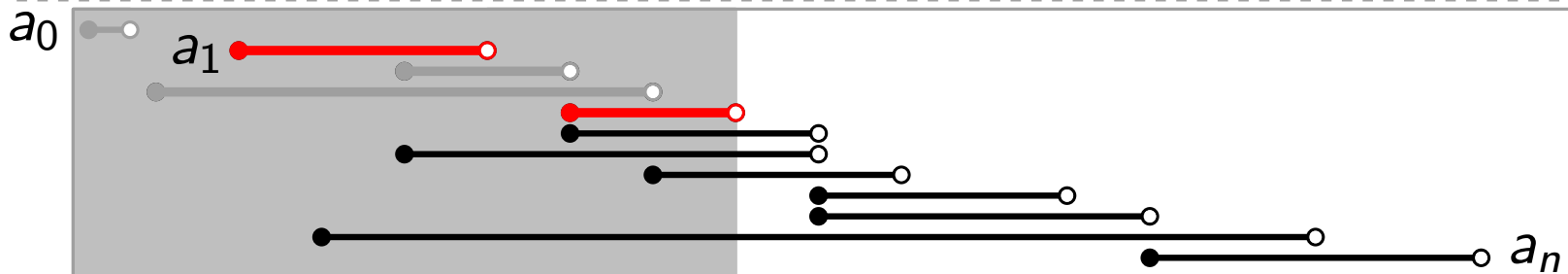
```
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
```

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

```
     $\perp$  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  $\Rightarrow A_0 = A$ 
```

```
  // 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
```

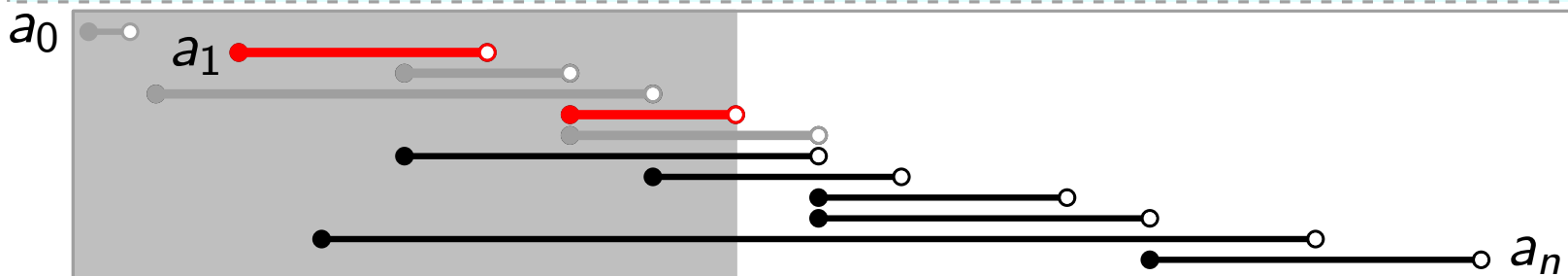
```
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
```

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

```
     $\perp$  m = m + 1
```

```
  if m > n then return  $\emptyset$ 
```

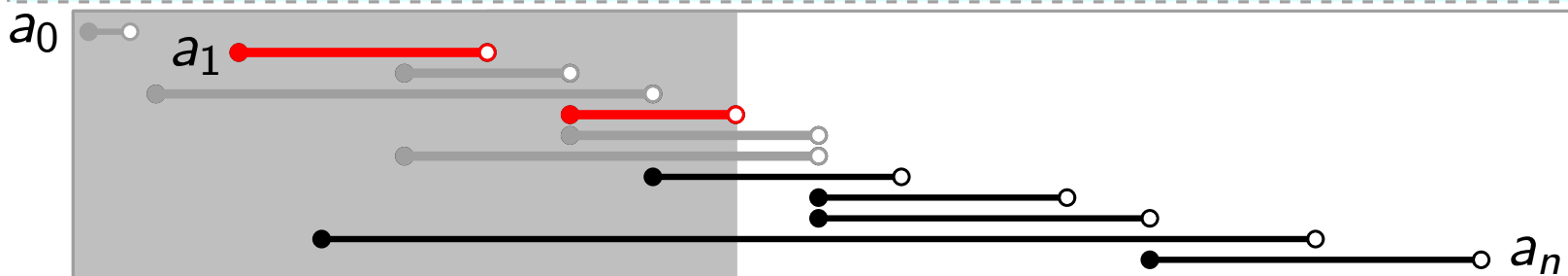
```
  else return  $\{a_m\} \cup$  GreedyRecursiveMain(s, e, m)
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
  // 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 ≤ n and s[m] < e[k] do
    m = m + 1
  if m > n then return  $\emptyset$ 
  else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

```
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
```

```
  // 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
```

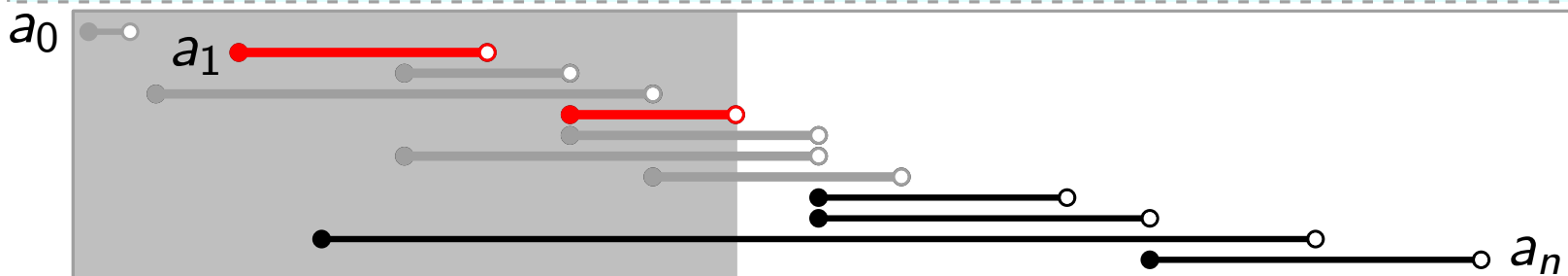
```
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
```

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

```
     $\perp$  m = m + 1
```

```
  if m > n then return  $\emptyset$ 
```

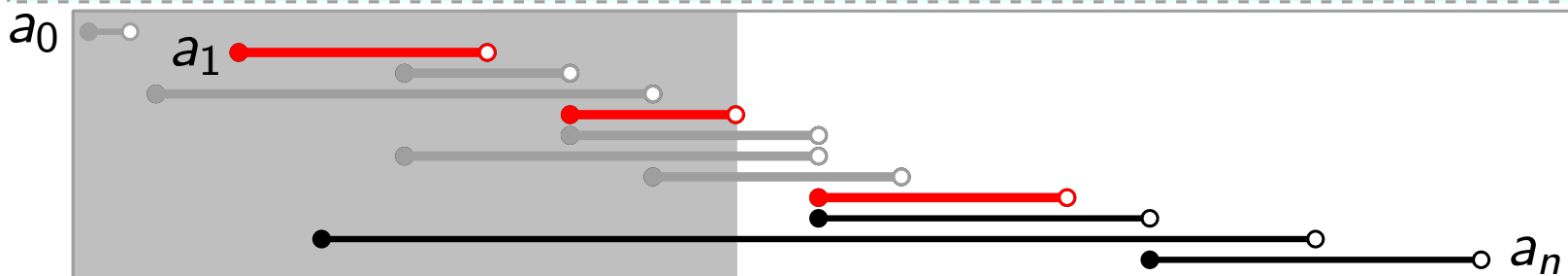
```
  else return  $\{a_m\} \cup$  GreedyRecursiveMain(s, e, m)
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
  // 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 ≤ n and s[m] < e[k] do
    m = m + 1
  if m > n then return  $\emptyset$ 
  else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

```
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
```

```
  // 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
```

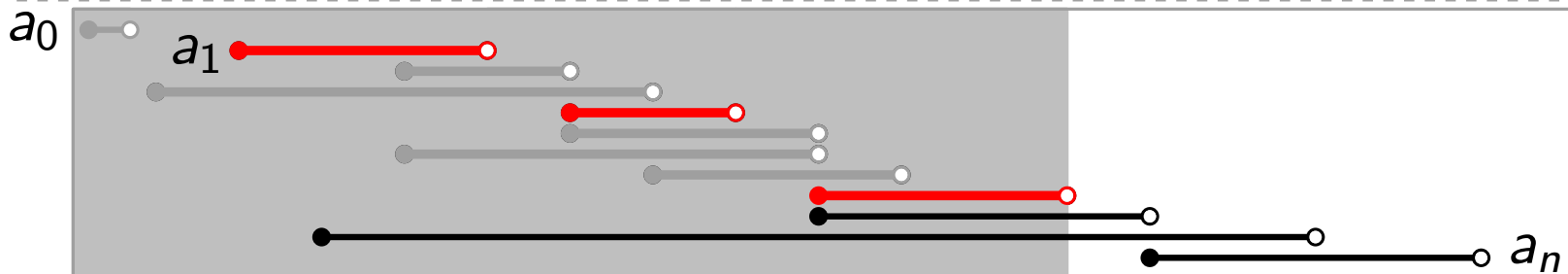
```
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
```

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

```
     $\perp$  m = m + 1
```

```
  if m > n then return  $\emptyset$ 
```

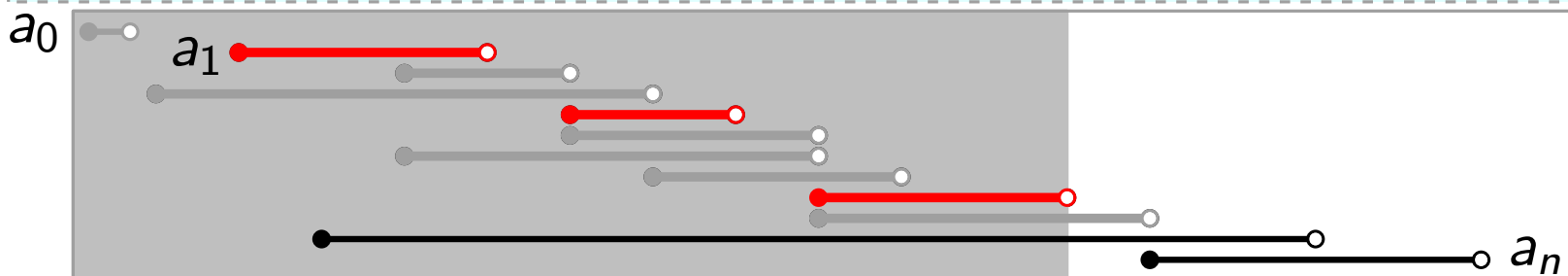
```
  else return  $\{a_m\} \cup$  GreedyRecursiveMain(s, e, m)
```



Greedy – rekursiv

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

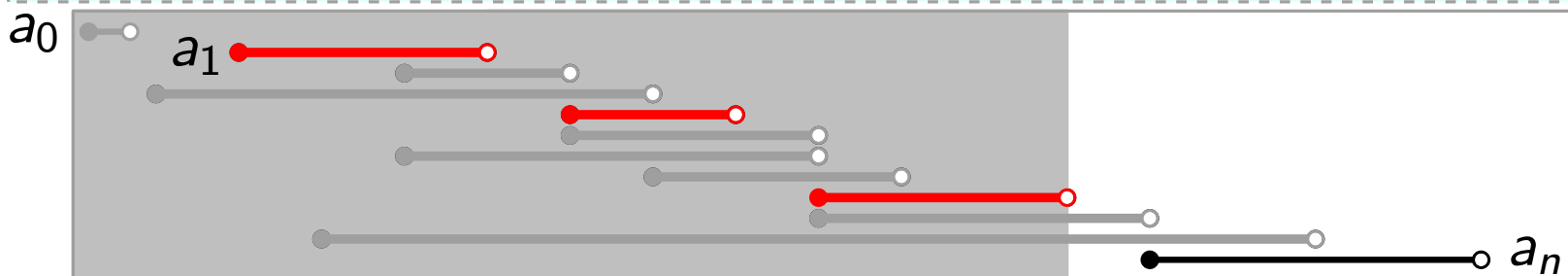
```
GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für  $A_k$ 
  m = k + 1; n = s.length
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
  while m ≤ n and s[m] < e[k] do
    m = m + 1
  if m > n then return  $\emptyset$ 
  else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)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



Greedy – rekursiv

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

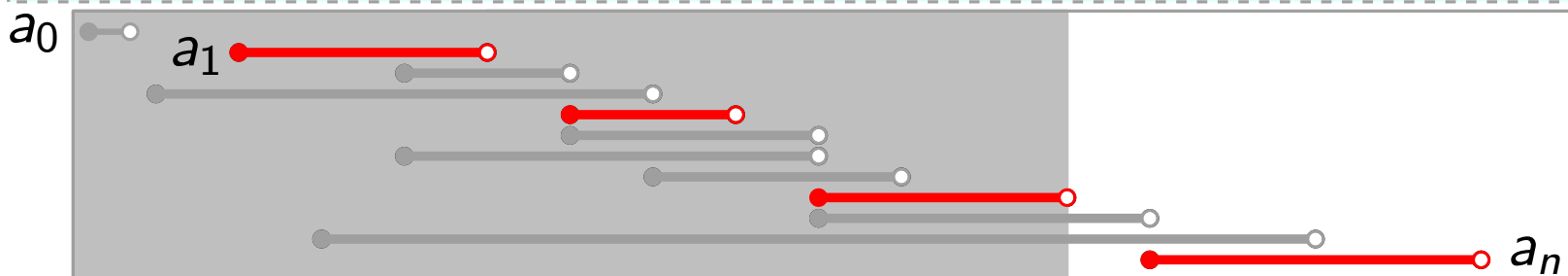
```
GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für  $A_k$ 
  m = k + 1; n = s.length
  // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
  while m ≤ n and s[m] < e[k] do
    m = m + 1
  if m > n then return  $\emptyset$ 
  else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)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



Greedy – rekursiv

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

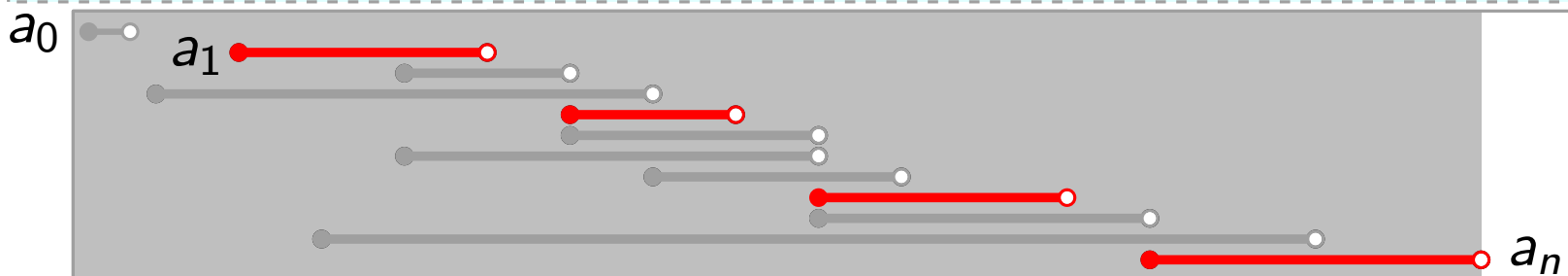
```
GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für  $A_k$ 
    m = k + 1; n = s.length
    // Finde Aktivität  $a_m$  mit kleinster Endzeit in  $A_k$ 
    while m ≤ n and s[m] < e[k] do
        m = m + 1
    if m > n then return  $\emptyset$ 
    else return { $a_m$ } ∪ GreedyRecursiveMain(s,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



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
  e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
  // 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 ≤ n and s[m] < e[k] do
    m = m + 1
  if m > n then return  $\emptyset$ 
  else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
    e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
    // 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  $\{a_m\} \cup \text{GreedyRecursiveMain}(s, e, m)$ 
```

Laufzeit?

Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
    e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
    // 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 ≤ n and s[m] < e[k] do
        └ m = m + 1
    if m > n then return  $\emptyset$ 
    else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)
```

Laufzeit? Wie oft wird m inkrementiert?

Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
    e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
    // 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 ≤ n and s[m] < e[k] do
        └ m = m + 1
    if m > n then return  $\emptyset$ 
    else return { $a_m$ } ∪ GreedyRecursiveMain(s, e, m)
```

Laufzeit?

Wie oft wird m inkrementiert?

Insgesamt, über alle rekursiven Aufrufe, n Mal.

Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
    e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
    // 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 ≤ n and s[m] < e[k] do
        └ m = m + 1
    if m > n then return  $\emptyset$ 
    else return { $a_m$ } ∪ 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 – ~~rekursiv~~

```
GreedyRecursive(int[] s, int[] e)
    e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
    // 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 ≤ n and s[m] < e[k] do
        └ m = m + 1
    if m > n then return  $\emptyset$ 
    else return { $a_m$ } ∪ 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 – ~~rekursiv~~ *iterativ!*

```
GreedyRecursive(int[] s, int[] e)
    e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
    // 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 ≤ n and s[m] < e[k] do
        └ m = m + 1
    if m > n then return  $\emptyset$ 
    else return { $a_m$ } ∪ 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 – ~~rekursiv~~ *iterativ!*

Schreiben Sie
GreedyIterative(int[] s, int[] e)!

```
GreedyRecursive(int[] s, int[] e)
    e[0] =  $-\infty$     // technischer Kniff  $\Rightarrow A_0 = A$ 
    // 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 ≤ 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
    |
  return L
```

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
```


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?

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.

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

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

Die Greedy-Strategie

1. Teste, ob das Problem optimale Teilstruktur aufweist.

Die Greedy-Strategie

1. Teste, ob das Problem optimale Teilstruktur aufweist.
2. Entwickle eine rekursive Lösung

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

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!)

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

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?

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:

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.

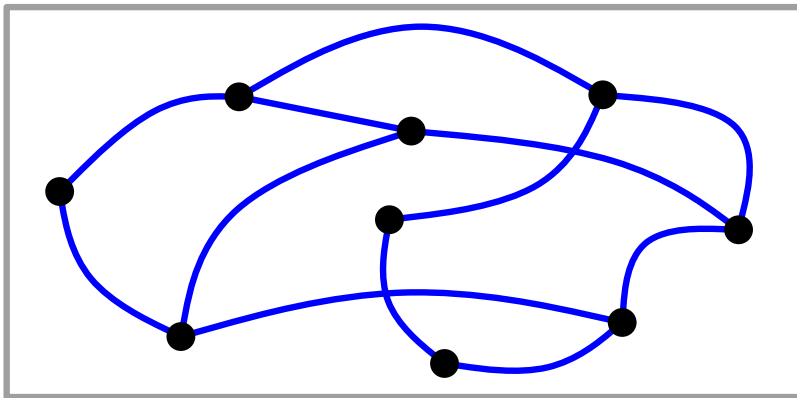
Food for Thought

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:

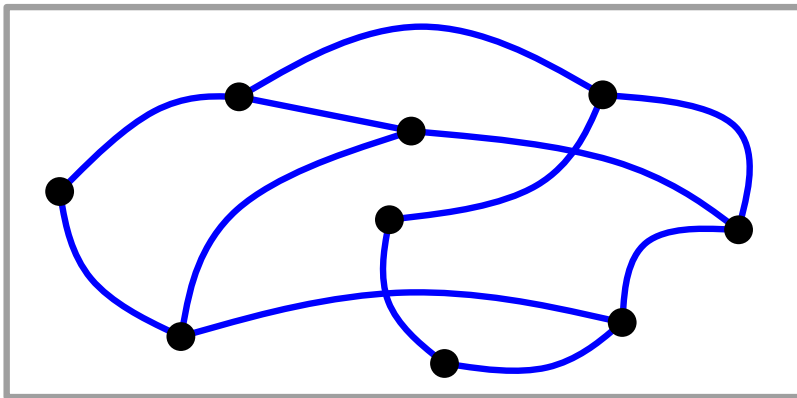


Food for Thought

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:



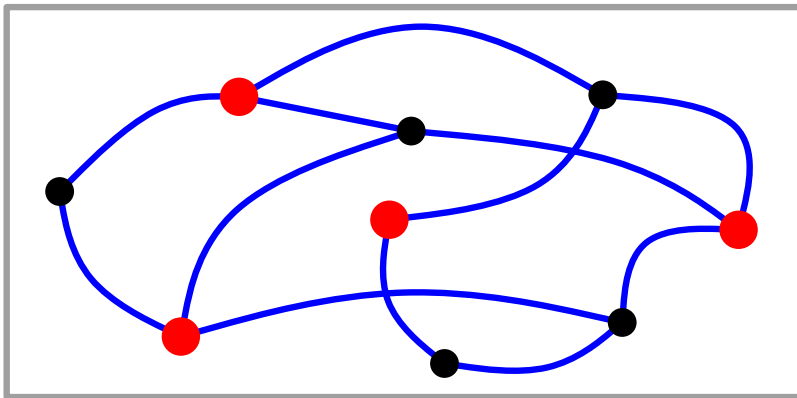
Finde eine größte Teilmenge U der Knoten, so dass keine zwei Knoten in U benachbart sind.

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:



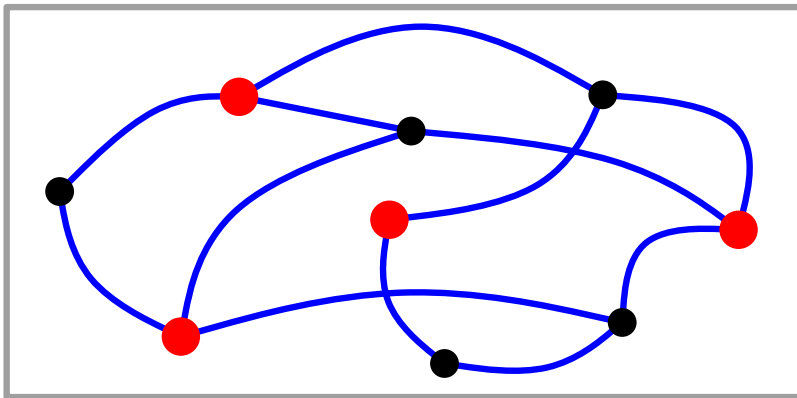
Finde eine größte Teilmenge U der Knoten, so dass keine zwei Knoten in U benachbart sind.

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.

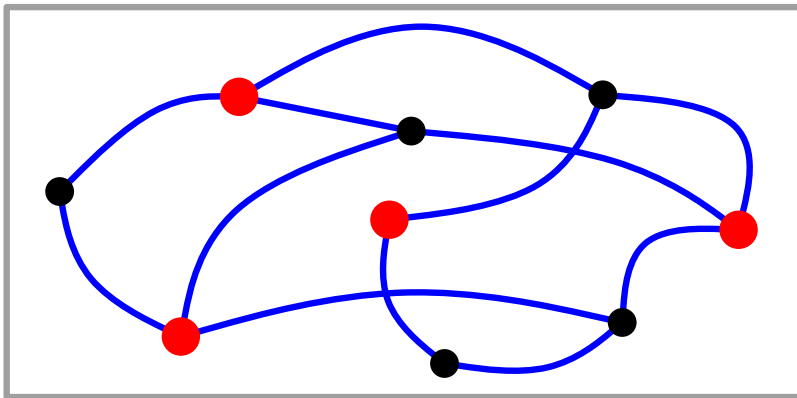
- Was hat guM mit unserem Ablaufplanungsproblem zu tun?

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.

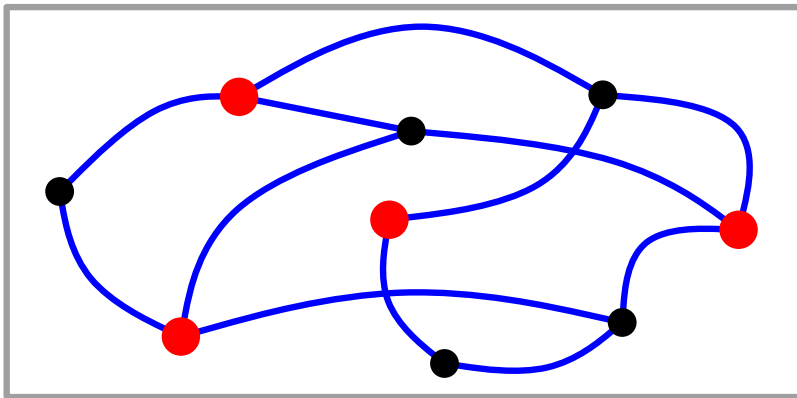
- Was hat guM mit unserem Ablaufplanungsproblem zu tun?
- Welche Graphen kommen bei der Ablaufplanung nicht vor?

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:



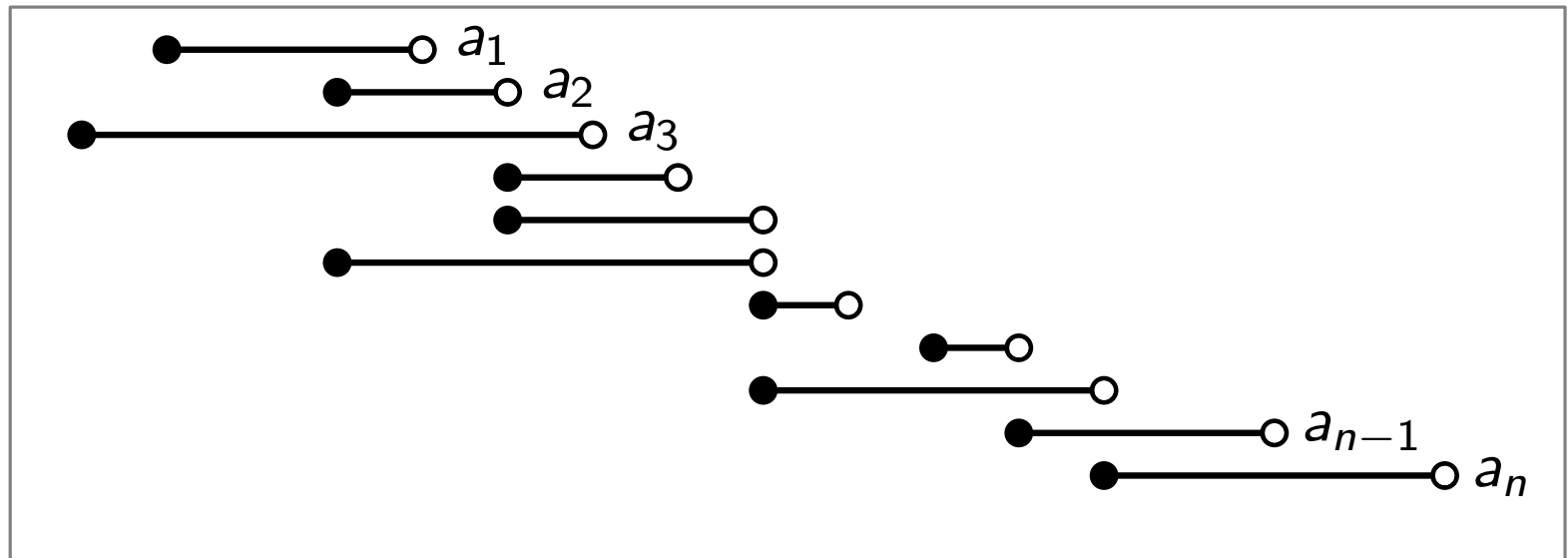
Finde eine größte Teilmenge U der Knoten, so dass keine zwei Knoten in U benachbart sind.

- Was hat guM mit unserem Ablaufplanungsproblem zu tun?
- Welche Graphen kommen bei der Ablaufplanung nicht vor?
- Kann man guM mit Dynam. Progr. (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 $i = 1, \dots, n$.

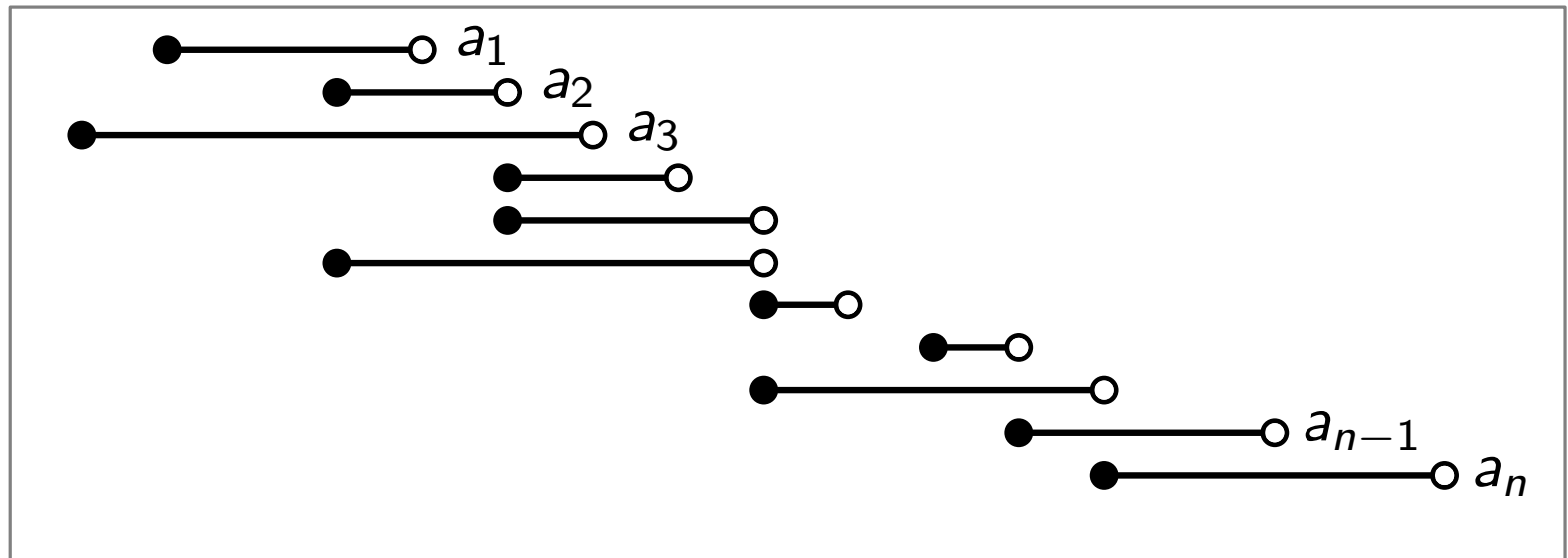
Für die Endpunkte gelte $e_1 \leq e_2 \leq \dots \leq e_n$.



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 $e_1 \leq e_2 \leq \dots \leq e_n$.

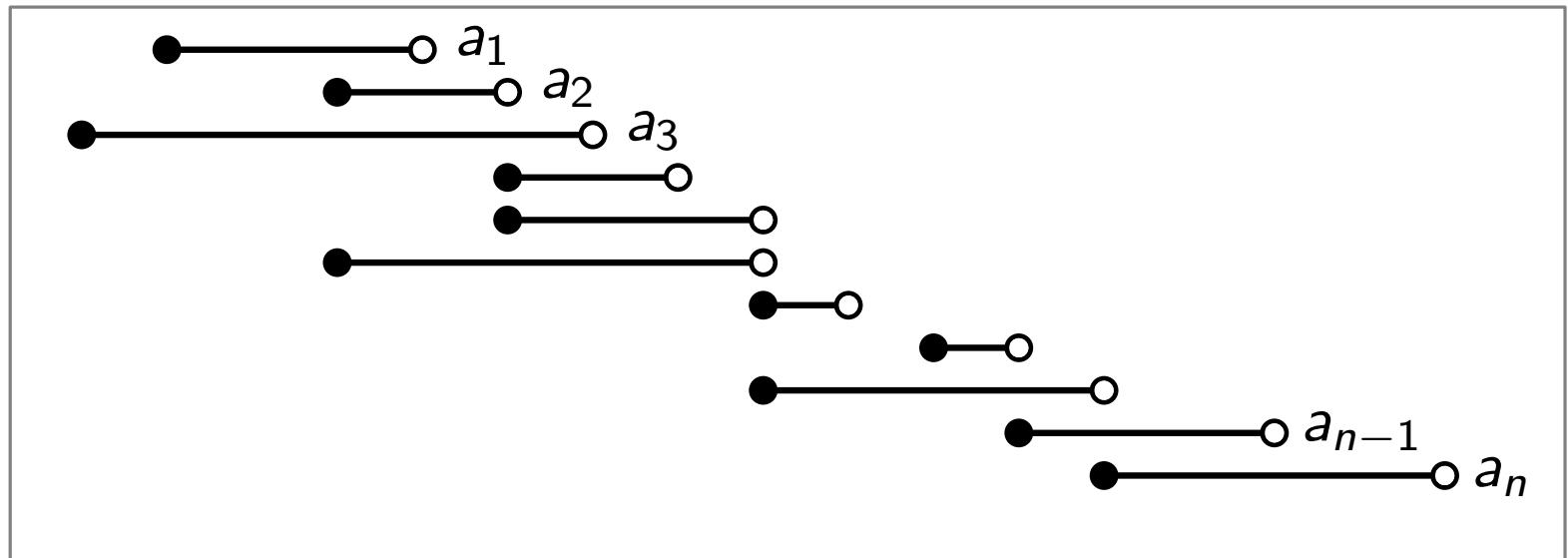


Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren **Gesamtlänge $\ell(A')$ maximal** ist.

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 $e_1 \leq e_2 \leq \dots \leq e_n$.



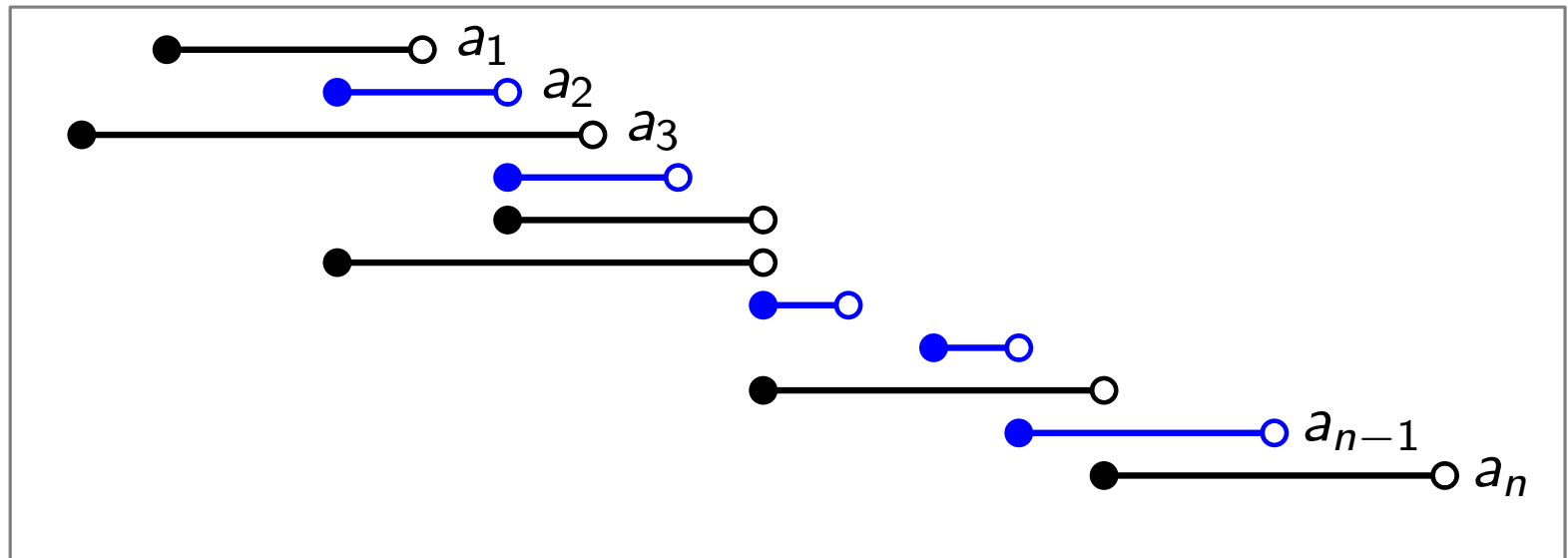
Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren **Gesamtlänge $\ell(A')$ maximal** ist.

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

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 $e_1 \leq e_2 \leq \dots \leq e_n$.



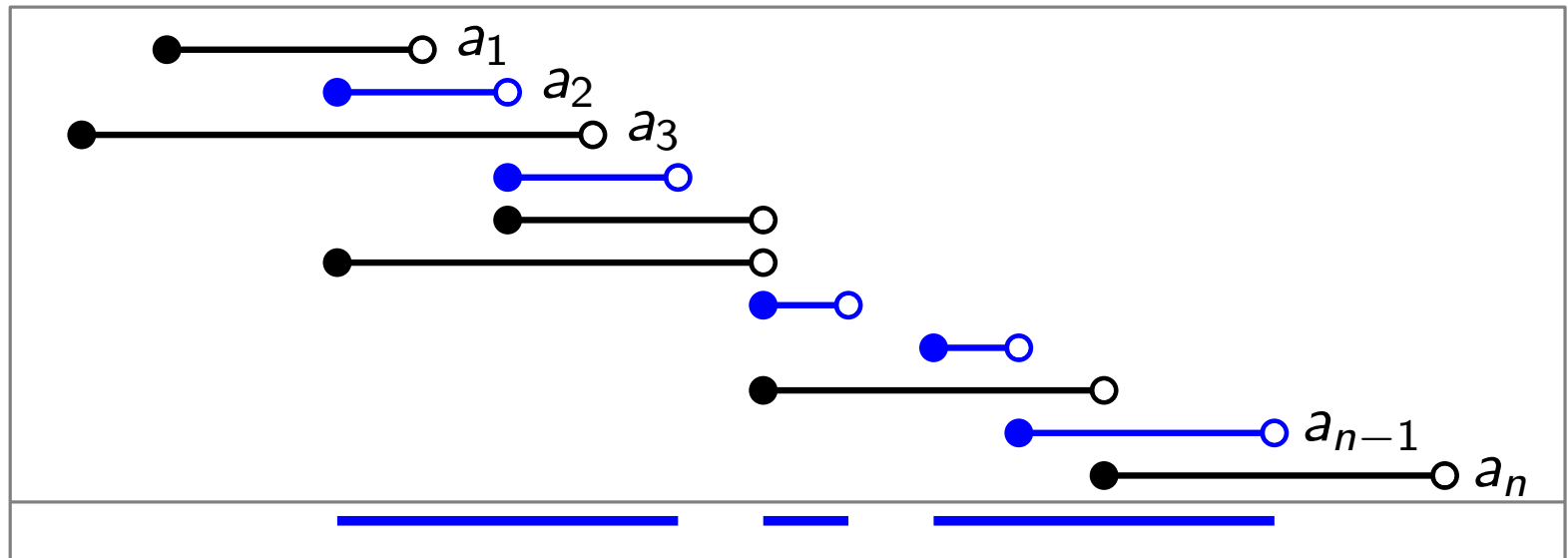
Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren **Gesamtlänge $\ell(A')$ maximal** ist.

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

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 $e_1 \leq e_2 \leq \dots \leq e_n$.



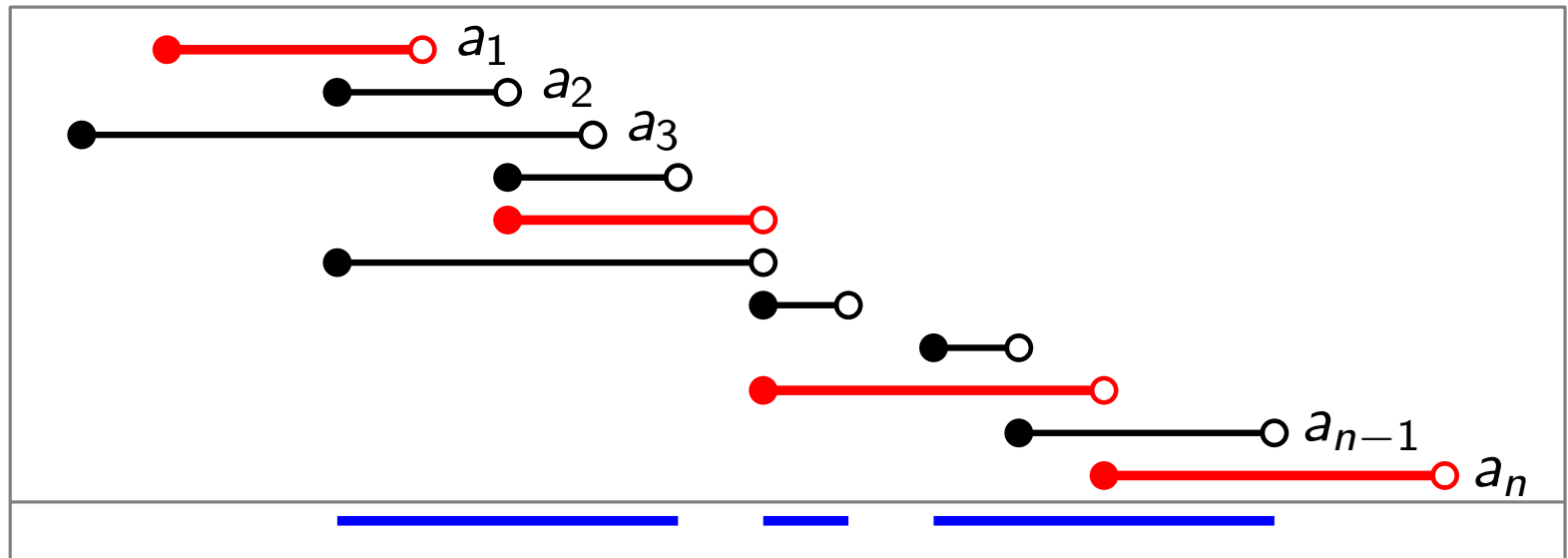
Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren **Gesamtlänge $\ell(A')$ maximal** ist.

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

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 $e_1 \leq e_2 \leq \dots \leq e_n$.



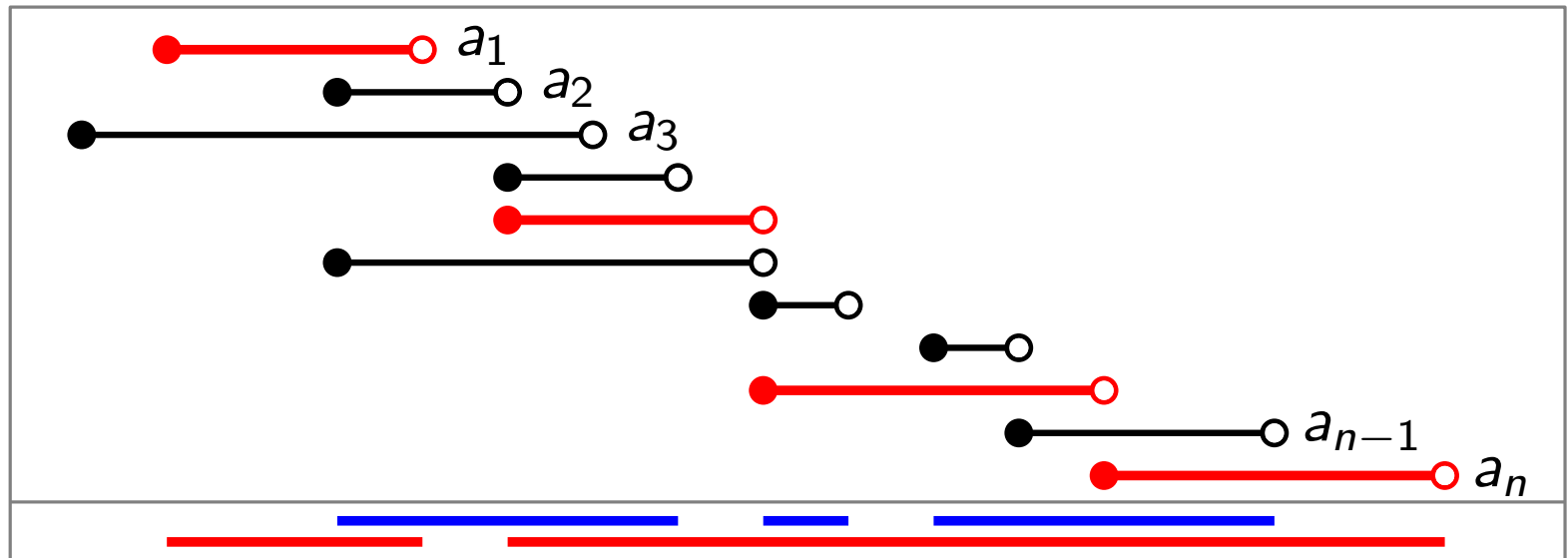
Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren **Gesamtlänge $\ell(A')$ maximal** ist.

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

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?

Greedy?

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

Greedy?

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

Gegenbsp.:

Greedy?

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

Gegenbsp.:



Greedy?

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

Gegenbsp.:



2. Versuch:

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.*

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.*

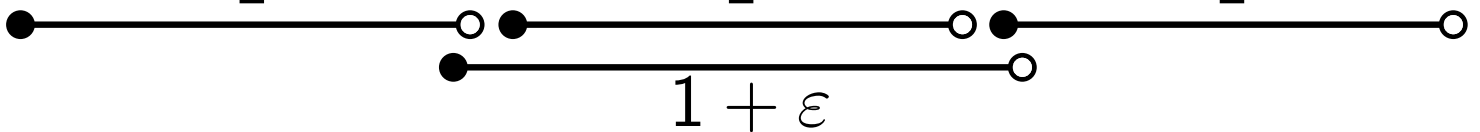
Gegenbsp.:

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.*

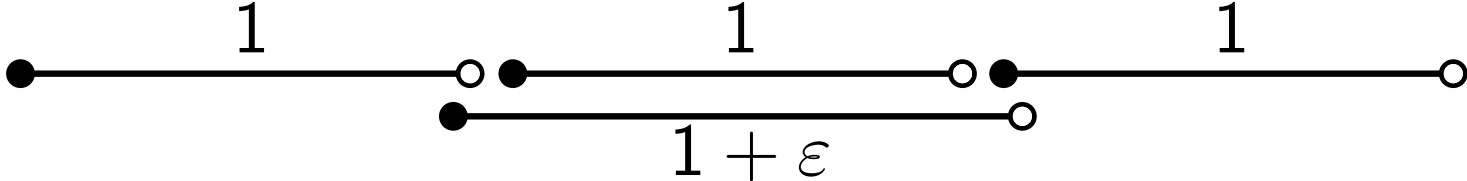
Gegenbsp.: 

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.: 

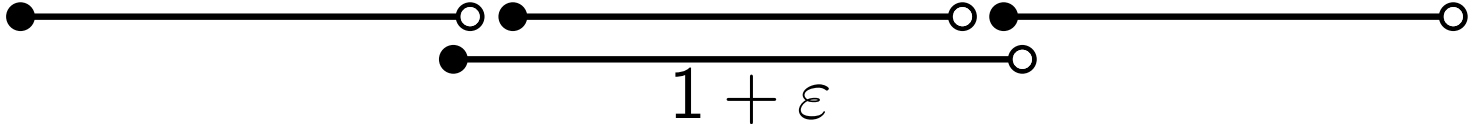
Aufgabe: Können Sie den 2. GA in $O(n \log n)$ Zeit implementieren?

Greedy?

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

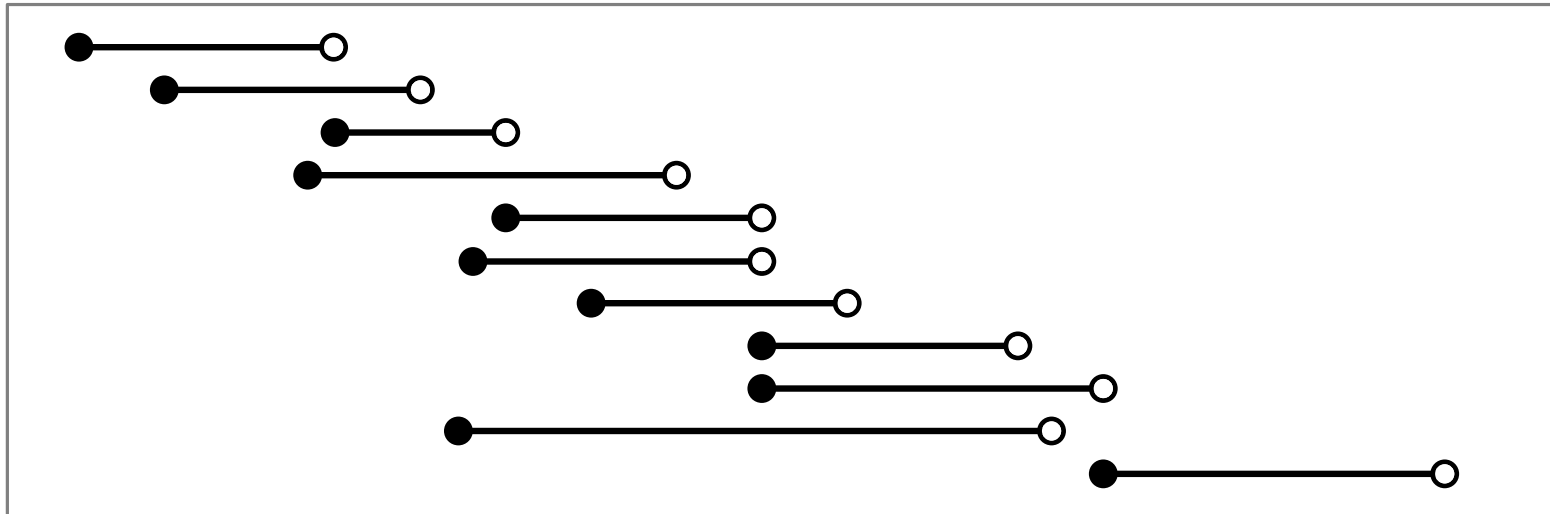
Gegenbsp.: 

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

Gegenbsp.: 

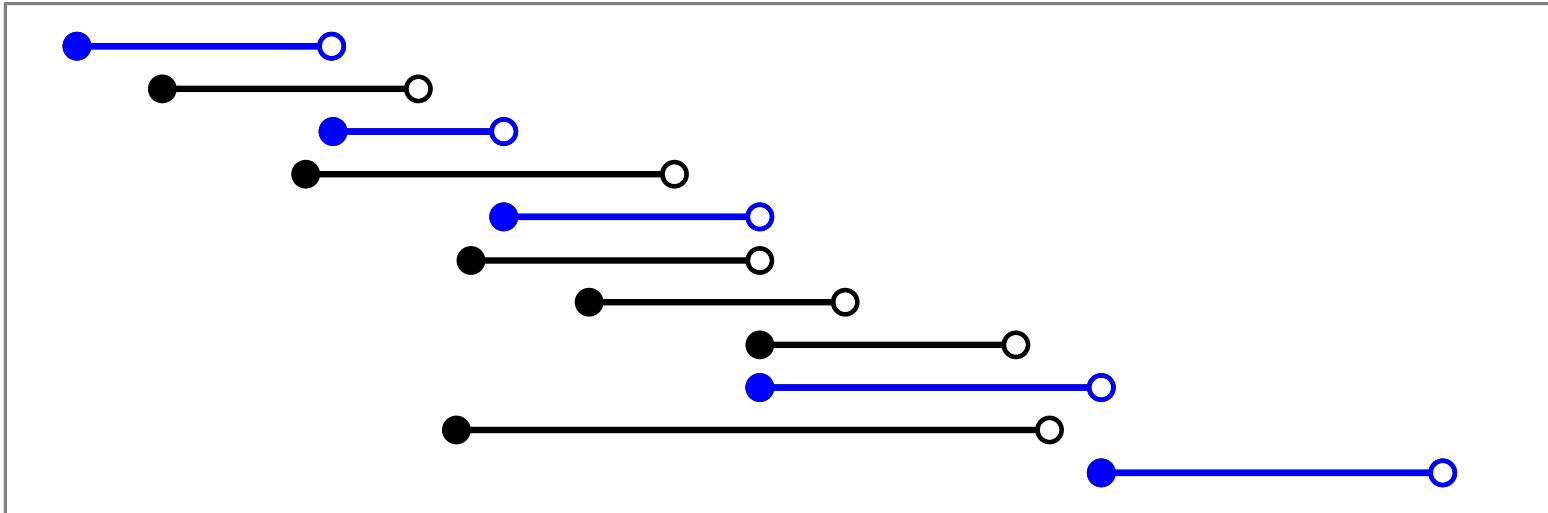
Aufgabe: Können Sie den 2. GA in $O(n \log n)$ Zeit implementieren?

Tipp: Gehen Sie so ähnlich wie Kruskal vor!



Wie gut/schlecht ist der 2. GA?

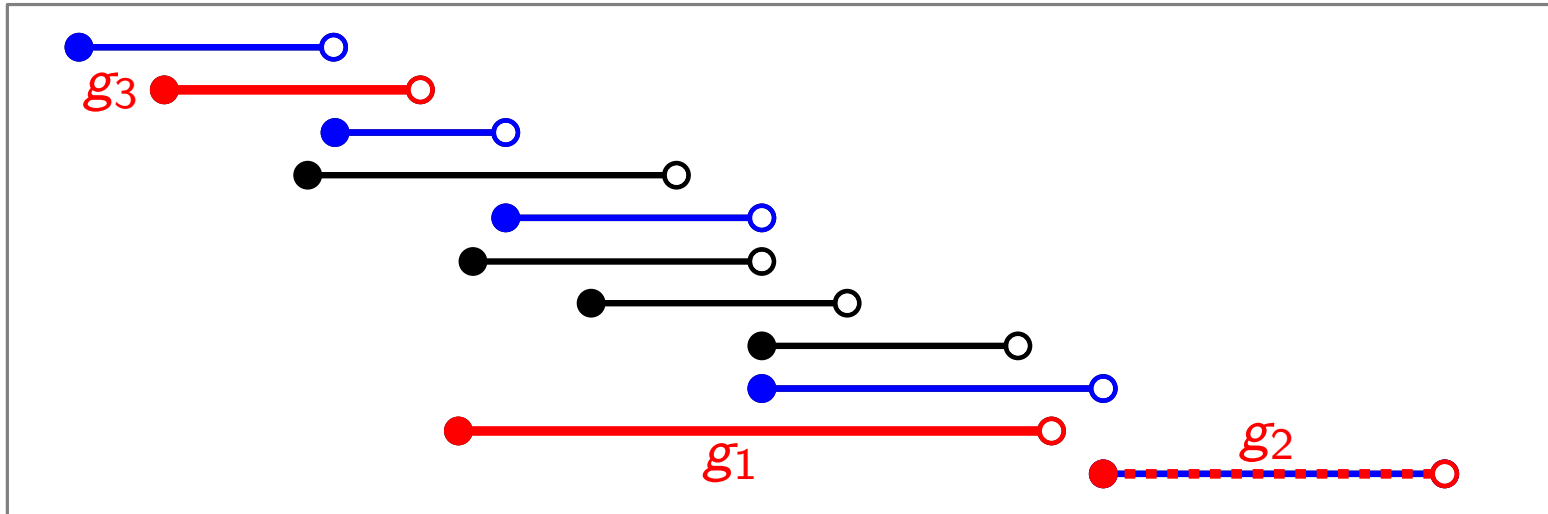
Betrachte eine optimale Lösung $L \subseteq A$.



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.*).

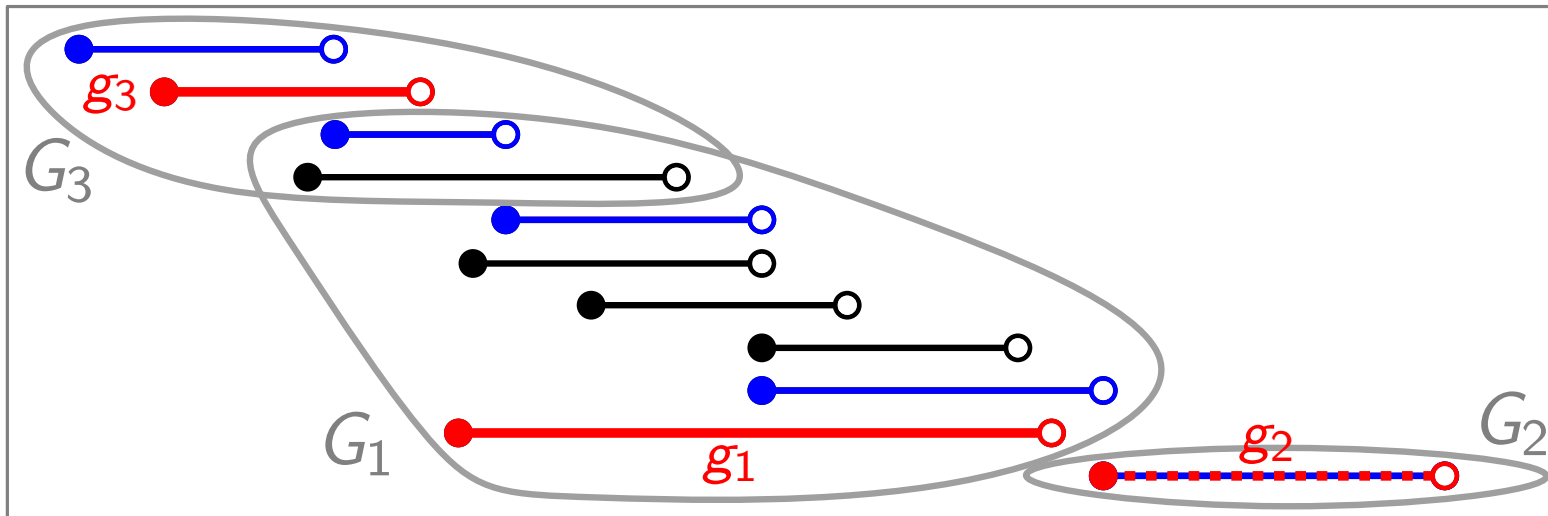


Wie gut/schlecht ist der 2. GA?

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 $G_i = \{a \in A \mid a \cap g_i \neq \emptyset\}$

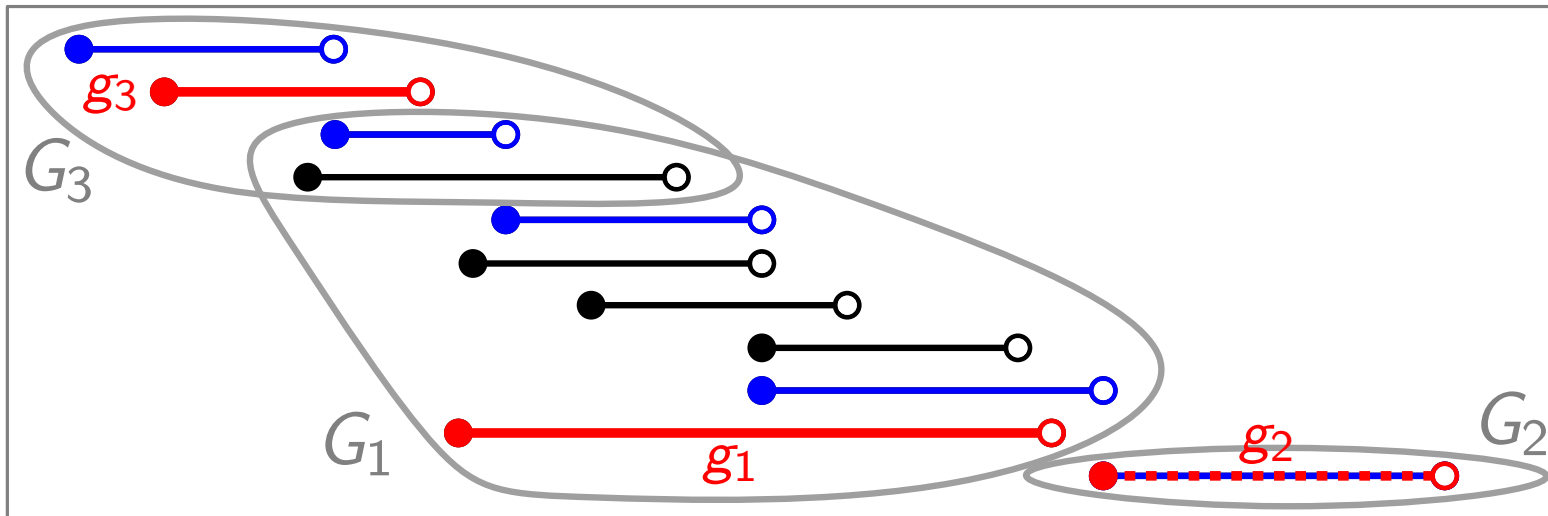


Wie gut/schlecht ist der 2. GA?

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

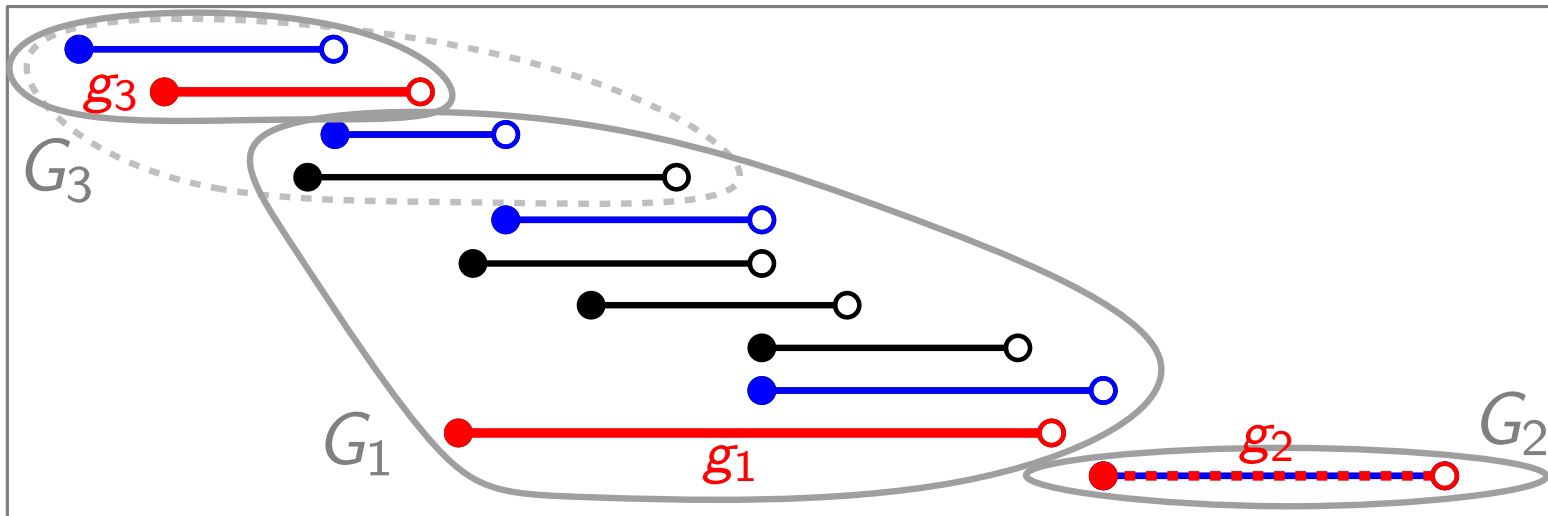


Wie gut/schlecht ist der 2. GA?

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



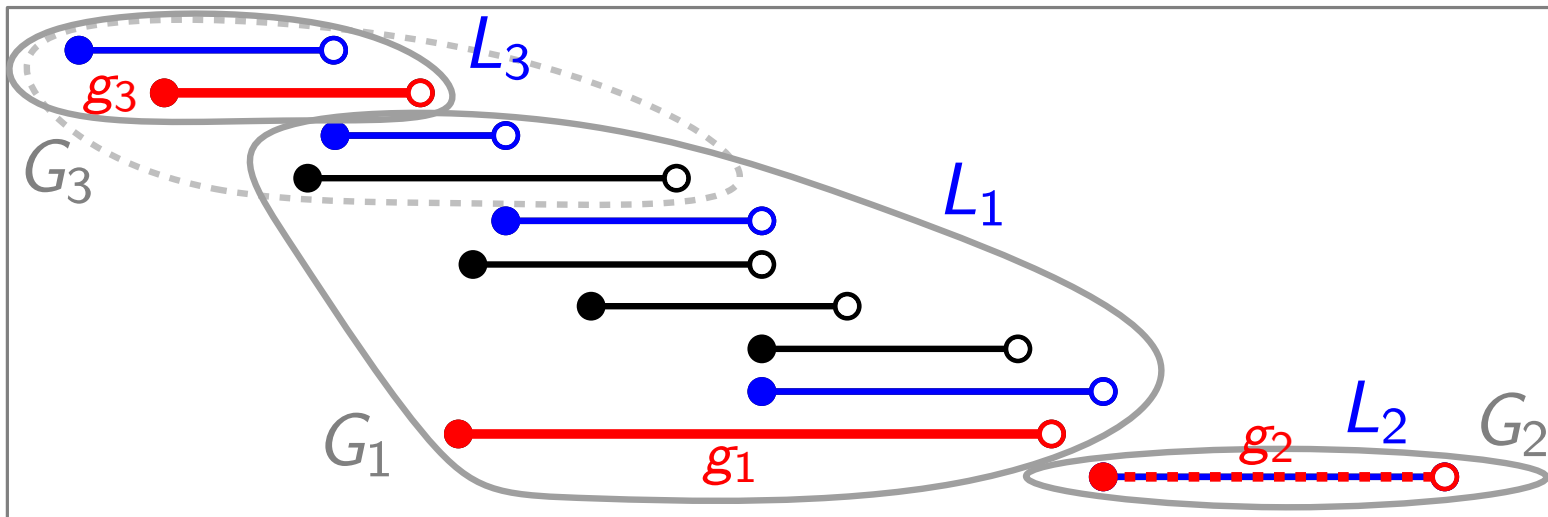
Wie gut/schlecht ist der 2. GA?

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

und
 $L_i = L \cap G_i$.

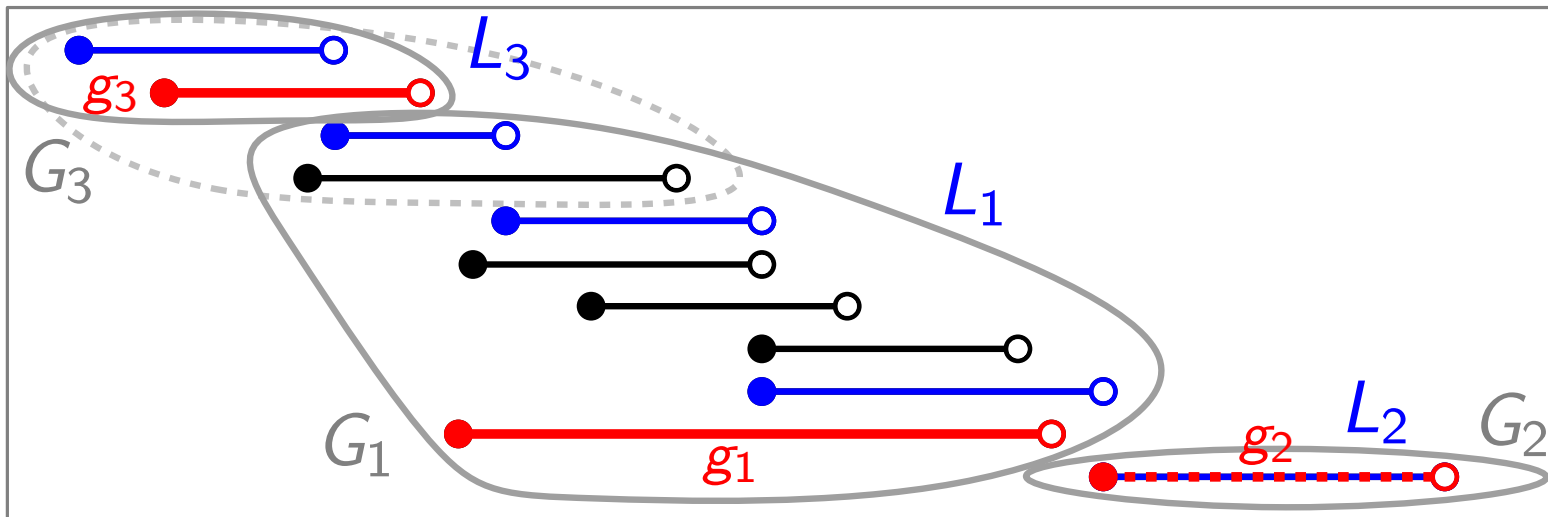


Wie gut/schlecht ist der 2. GA?

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



und
 $L_i = L \cap G_i$.

Dann gilt $A = G_1 \dot{\cup} G_2 \dot{\cup} \dots \dot{\cup} G_k$

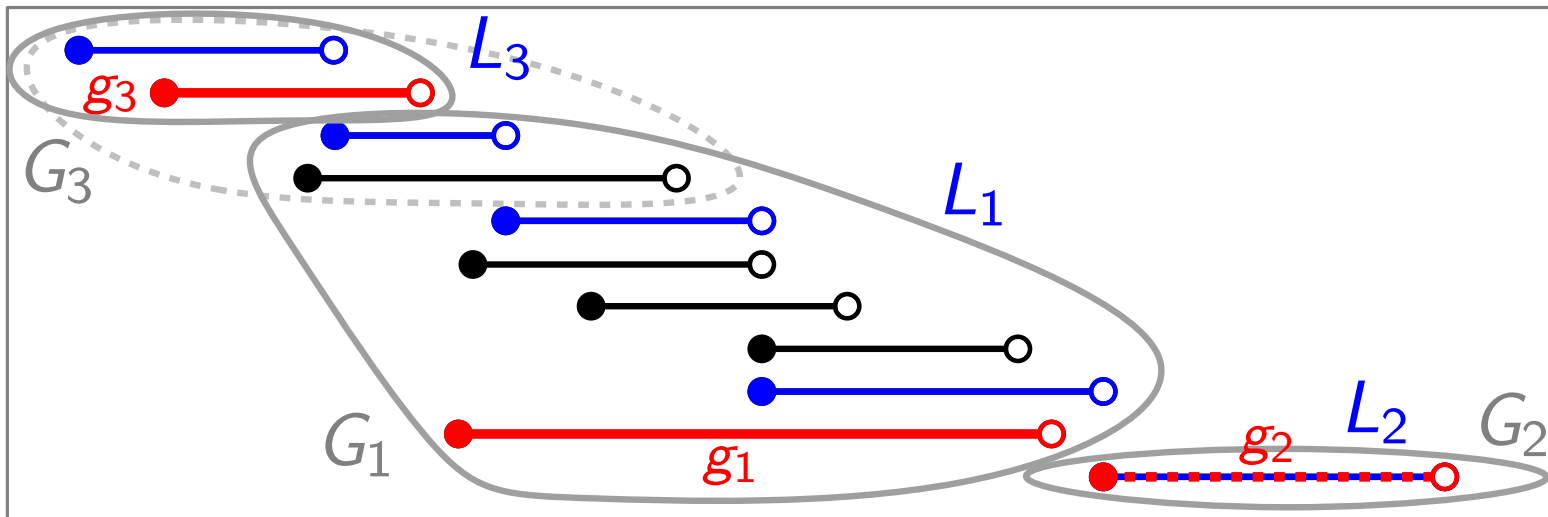
Wie gut/schlecht ist der 2. GA?

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

und
 $L_i = L \cap G_i$.



Dann gilt $A = G_1 \dot{\cup} G_2 \dot{\cup} \dots \dot{\cup} G_k$

„ $\subseteq$ “:

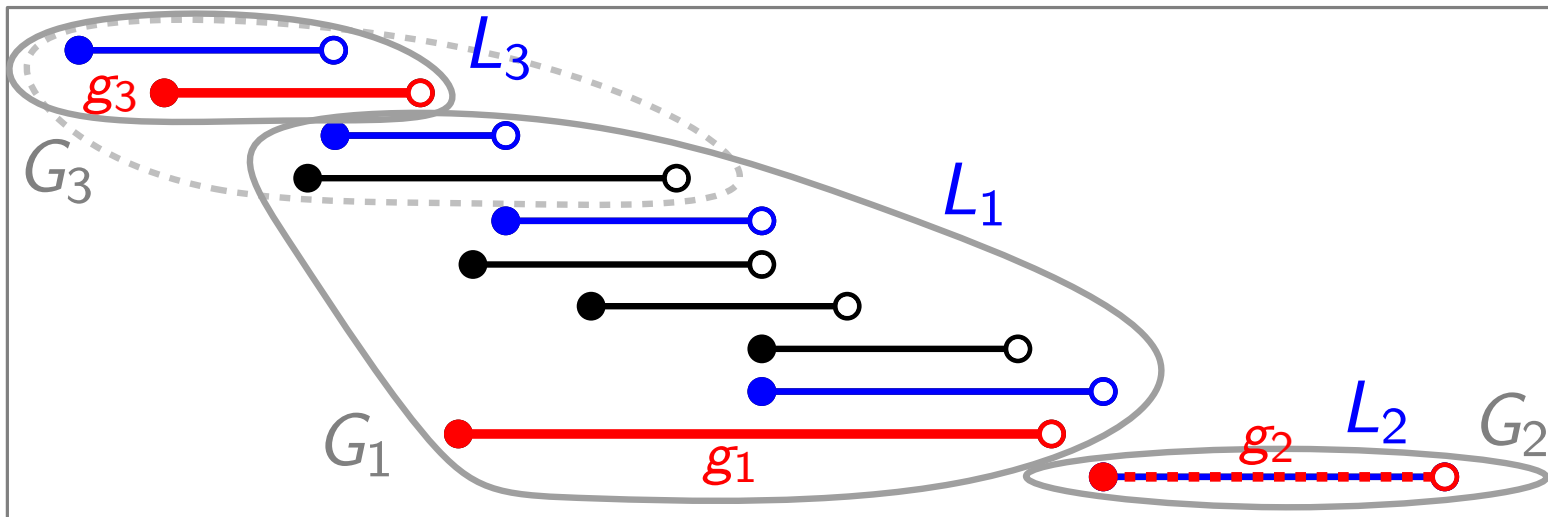
Wie gut/schlecht ist der 2. GA?

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

und
 $L_i = L \cap G_i$.



Dann gilt $A = G_1 \dot{\cup} G_2 \dot{\cup} \dots \dot{\cup} G_k$

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

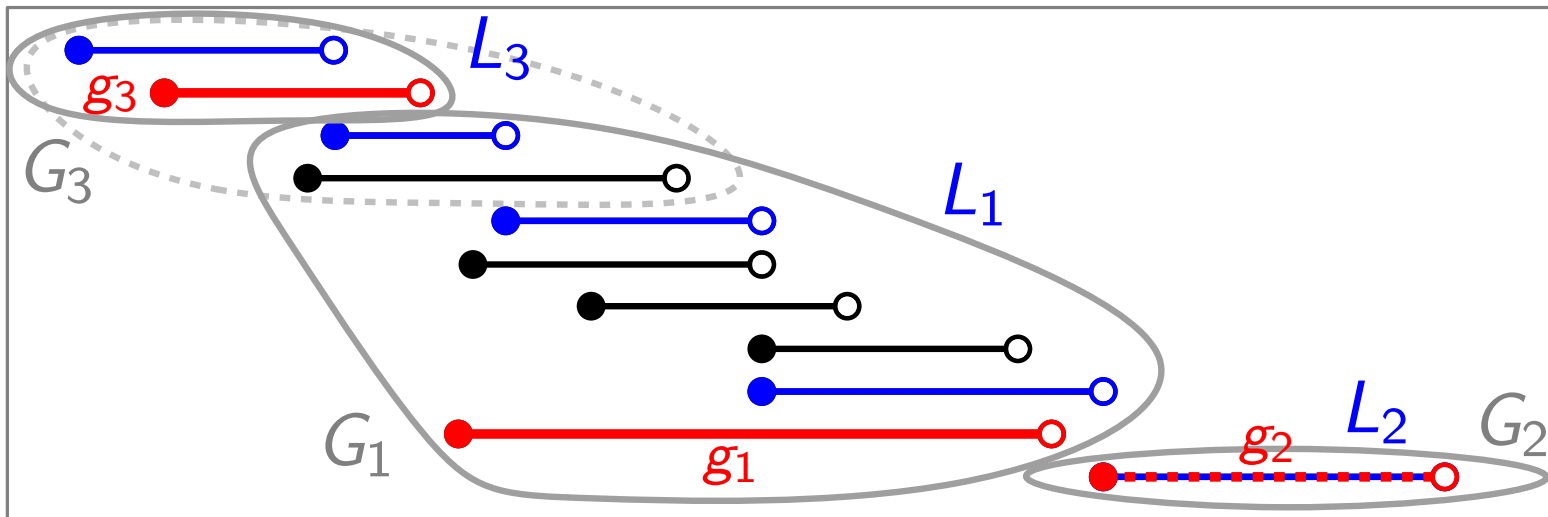
Wie gut/schlecht ist der 2. GA?

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

und
 $L_i = L \cap G_i$.



Dann gilt $A = G_1 \dot{\cup} G_2 \dot{\cup} \dots \dot{\cup} G_k$

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

„ $\supseteq$ “:

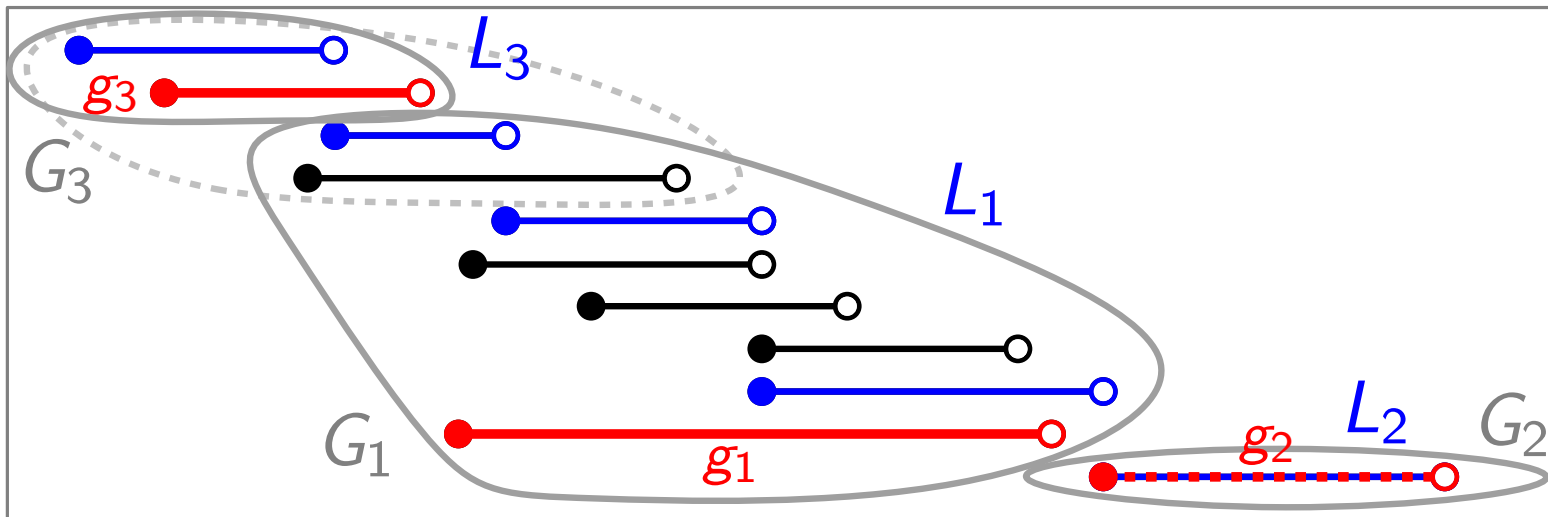
Wie gut/schlecht ist der 2. GA?

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

und
 $L_i = L \cap G_i$.



Dann gilt $A = G_1 \dot{\cup} G_2 \dot{\cup} \dots \dot{\cup} G_k$

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

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

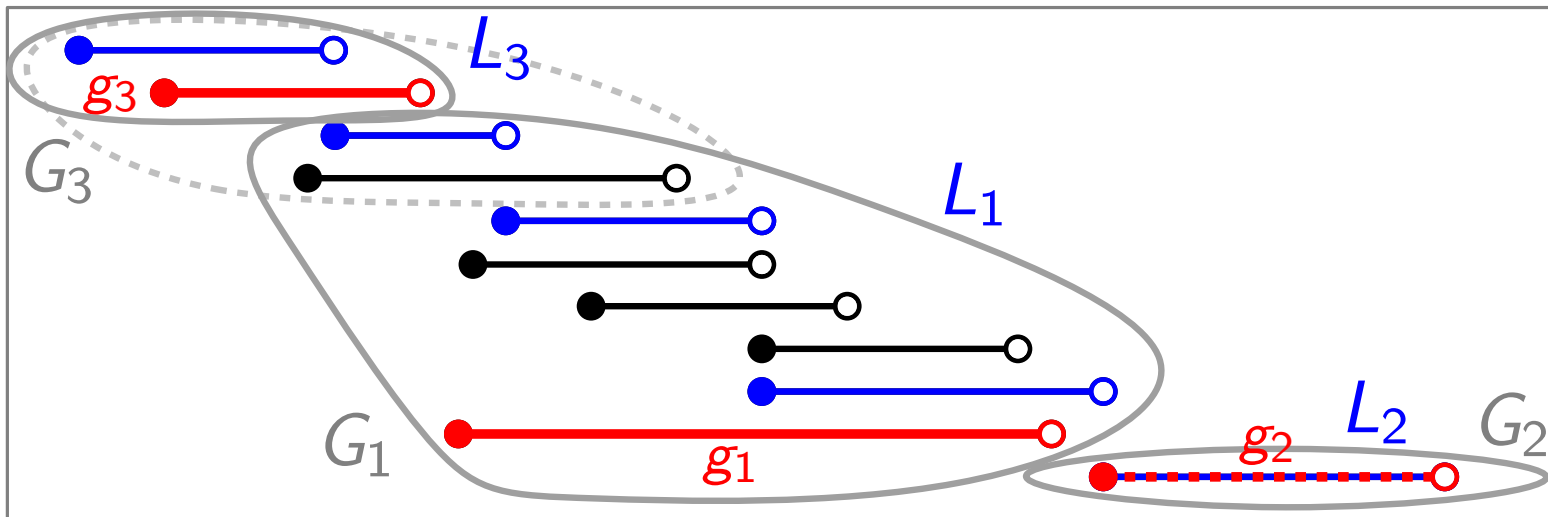
Wie gut/schlecht ist der 2. GA?

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

und
 $L_i = L \cap G_i$.



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.

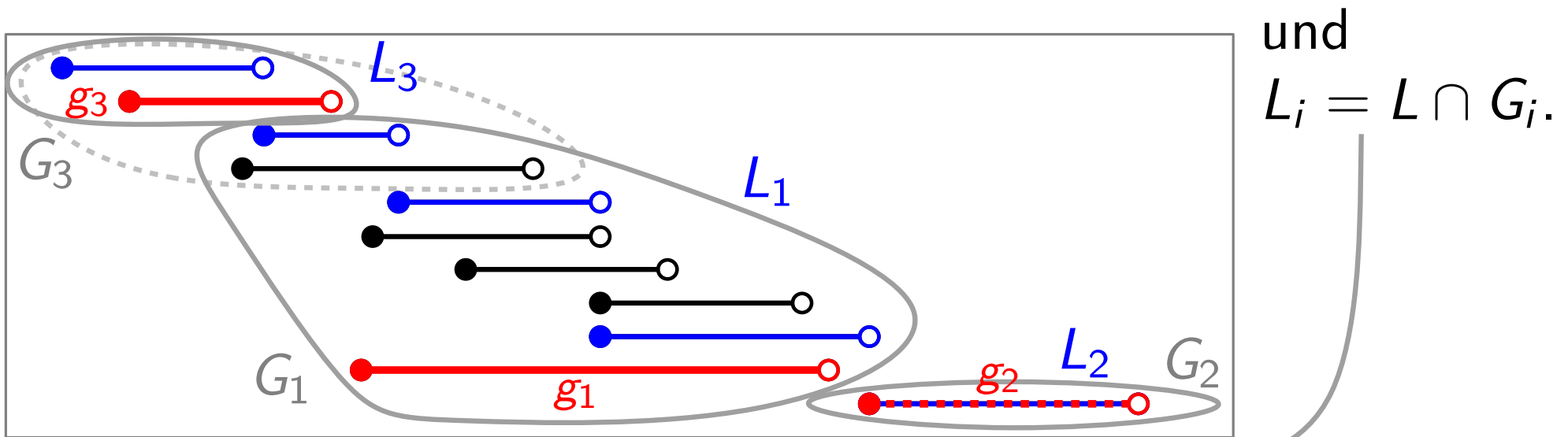
„ $\supseteq$ “: klar, da $G_1 \subseteq A$, $G_2 \subseteq A$, $\dots$, $G_k \subseteq A$

Wie gut/schlecht ist der 2. GA?

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



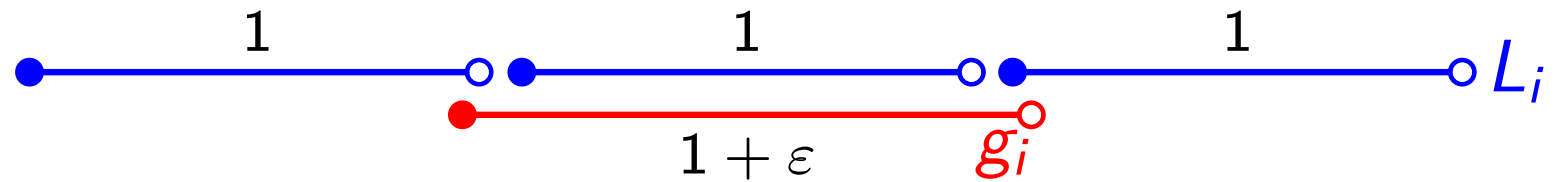
Dann gilt $A = G_1 \dot{\cup} G_2 \dot{\cup} \dots \dot{\cup} G_k$ und $L = L_1 \dot{\cup} L_2 \dot{\cup} \dots \dot{\cup} L_k$.

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

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

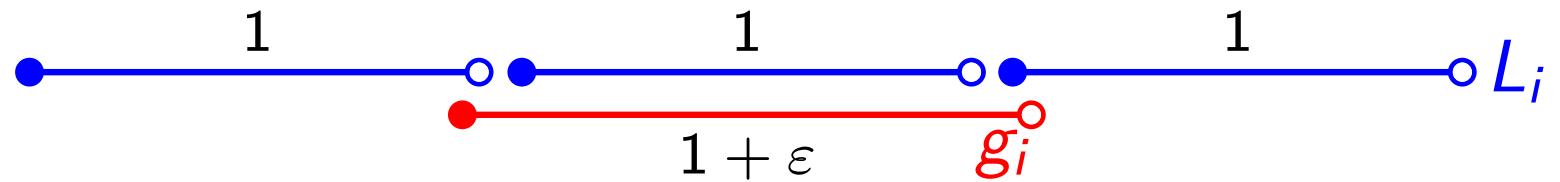
Wie gut/schlecht ist der 2. GA?

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



Wie gut/schlecht ist der 2. GA?

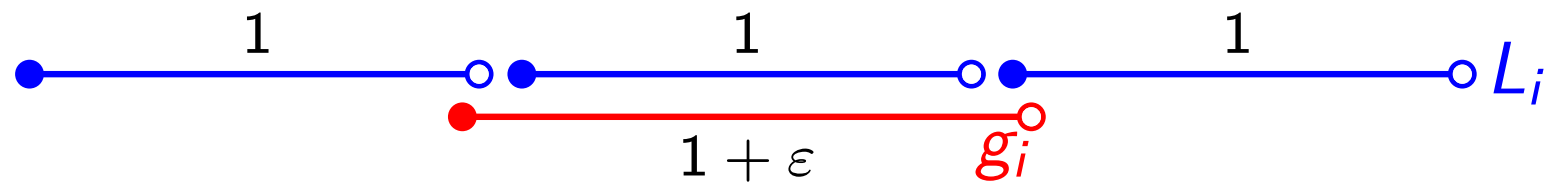
Behauptung: Für $i = 1, \dots, k$ gilt $\ell(L_i) < 3\ell(g_i)$.



Beweis.

Wie gut/schlecht ist der 2. GA?

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

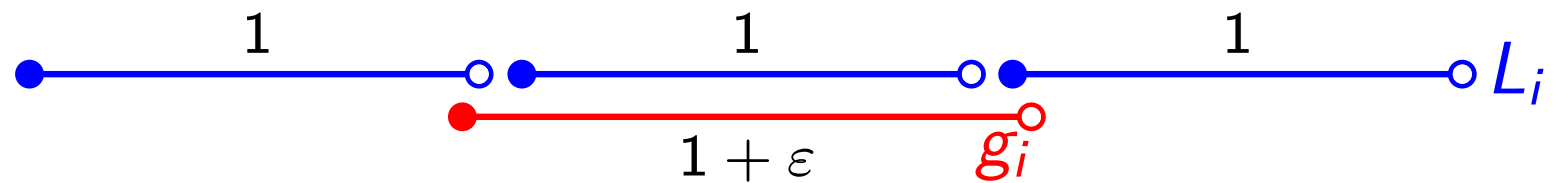


Beweis.

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

Wie gut/schlecht ist der 2. GA?

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

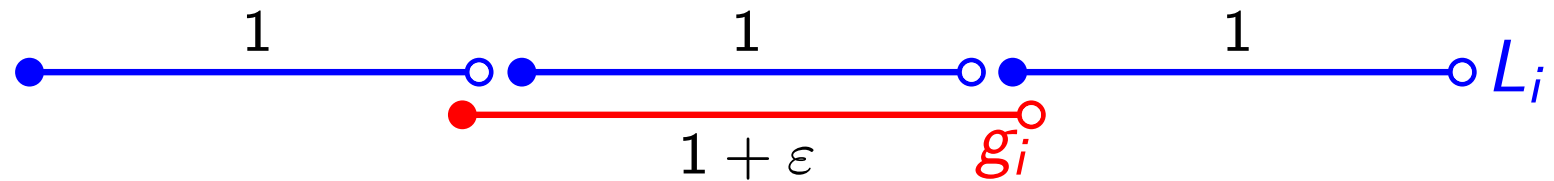


Beweis.

- (a) g_i ist nach Wahl ein längstes Intervall in G_i
- (b) jedes $a \in L_i$ schneidet g_i

Wie gut/schlecht ist der 2. GA?

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

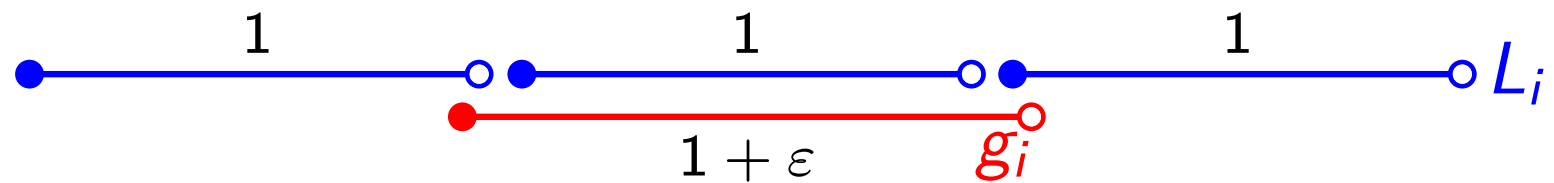


Beweis.

- (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

Wie gut/schlecht ist der 2. GA?

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



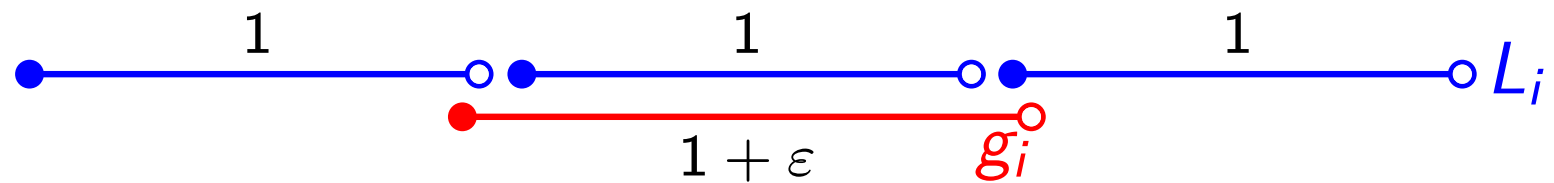
Beweis.

- (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

$\Rightarrow$

Wie gut/schlecht ist der 2. GA?

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



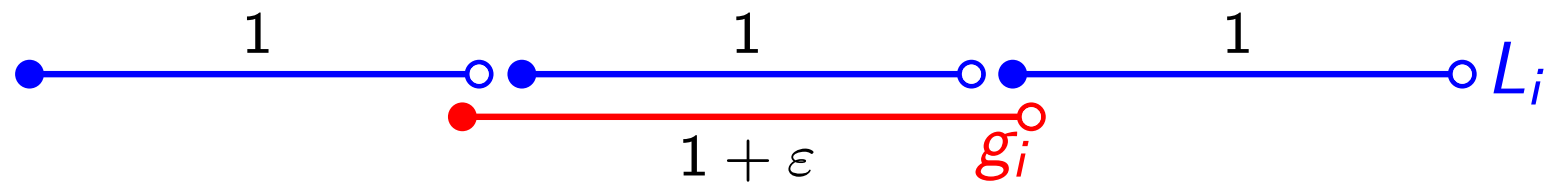
Beweis.

- (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

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

Wie gut/schlecht ist der 2. GA?

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



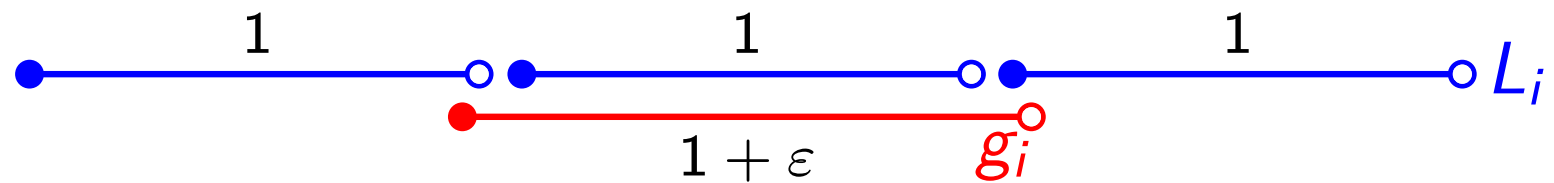
Beweis.

- (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

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

Wie gut/schlecht ist der 2. GA?

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



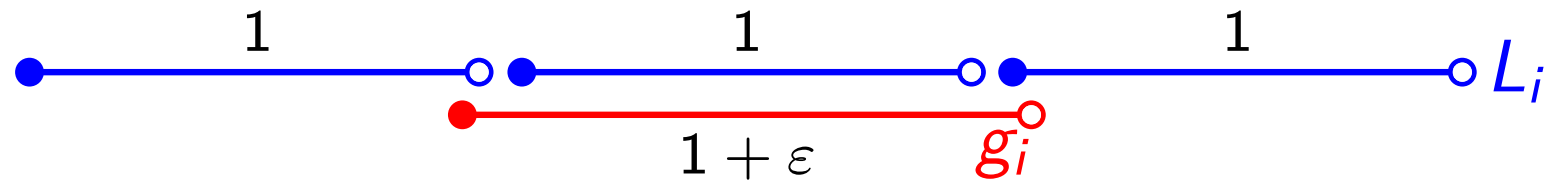
Beweis.

- (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

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

Wie gut/schlecht ist der 2. GA?

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



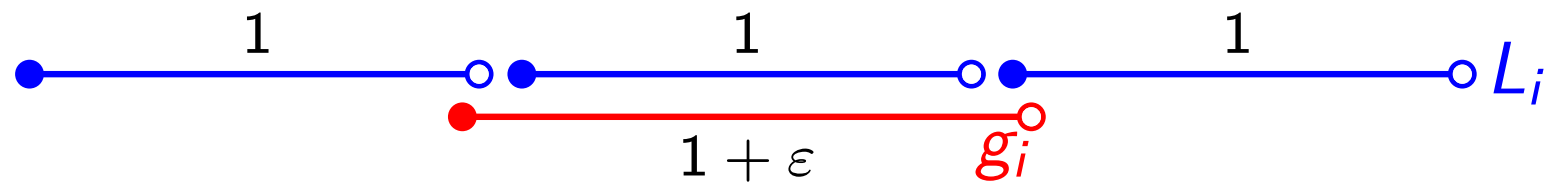
Beweis.

- (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

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

Wie gut/schlecht ist der 2. GA?

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



Beweis.

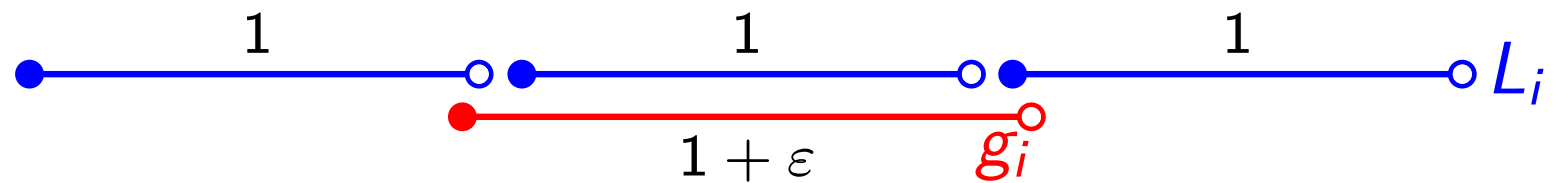
- (a) g_i ist nach Wahl ein längstes Intervall in G_i
- (b) jedes $a \in L_i$ schneidet g_i
- (c) Intervalle in L_i sind paarweise disjunkt

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

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

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

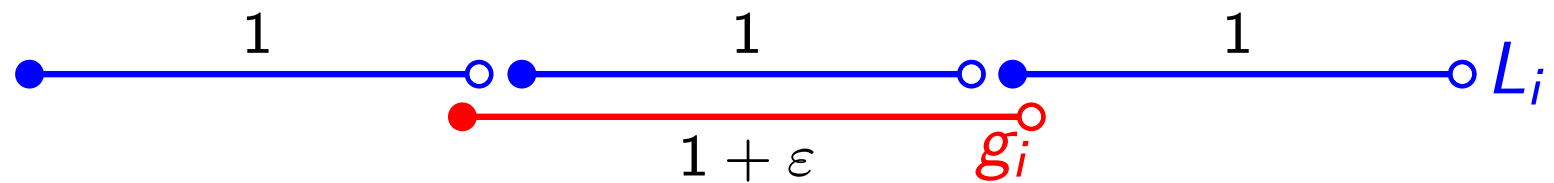
$$\Rightarrow \text{OPT} = \ell(L) = \sum_{i=1}^k \ell(L_i) < 3 \sum_{i=1}^k \ell(g_i) = 3\ell(G)$$

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

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

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. . . *hää?*

Approxim. . . *hää?*

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

Bertrand Russell (1872–1970)

Approxim. . . *h*ä?

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

Sei Π ein *Maximierungsproblem*.

Approxim. . . *hää?*

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

Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

z.B. Ablaufplanung

Approxim. . . 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 $\mapsto \mathbb{Q}_{\geq 0}$.

Approxim. . . 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 $\mapsto \mathbb{Q}_{\geq 0}$.

$\zeta = \ell$

Approxim. . . 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 .

$$\gamma = 1/3$$

Approxim. . . 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

Approxim. . . 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

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

Approxim. . . 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

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

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

Approxim. . . 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

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

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

Approxim. . . 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

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



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

Approxim. . . 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

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



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

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

Approxim. . . 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

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

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

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

Approxim. . . 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

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

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

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

Größe der Instanz I

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

Approxim. . . 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

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

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

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

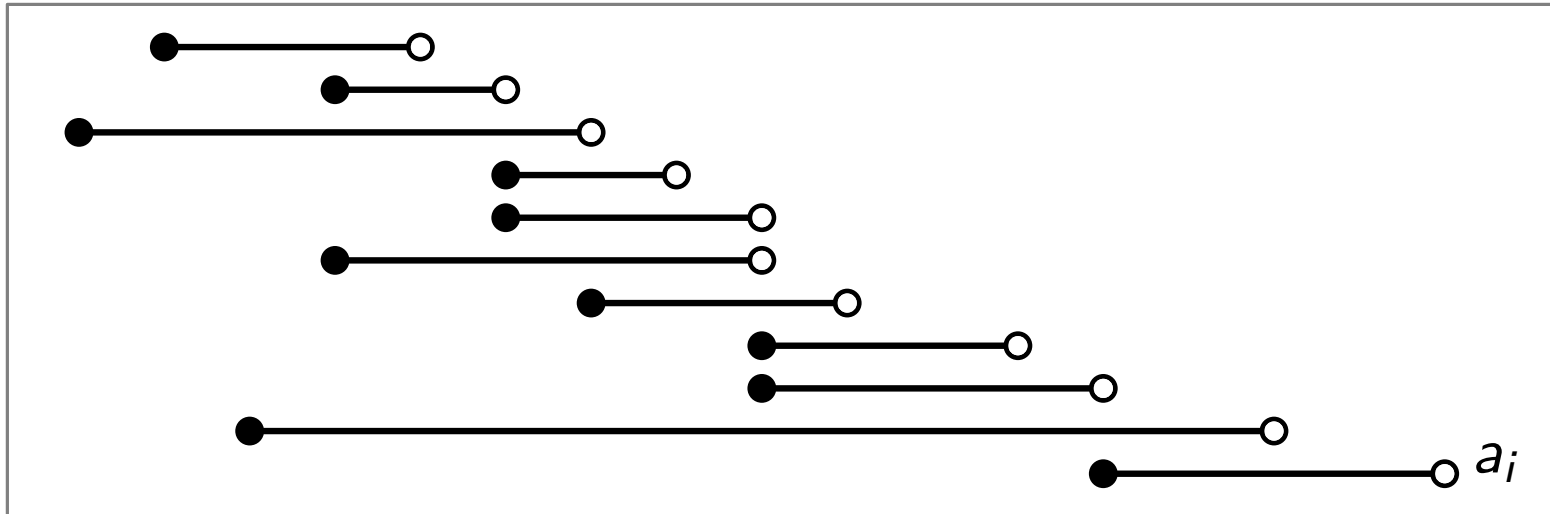
Größe der Instanz I

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

$$O(n \log n)$$

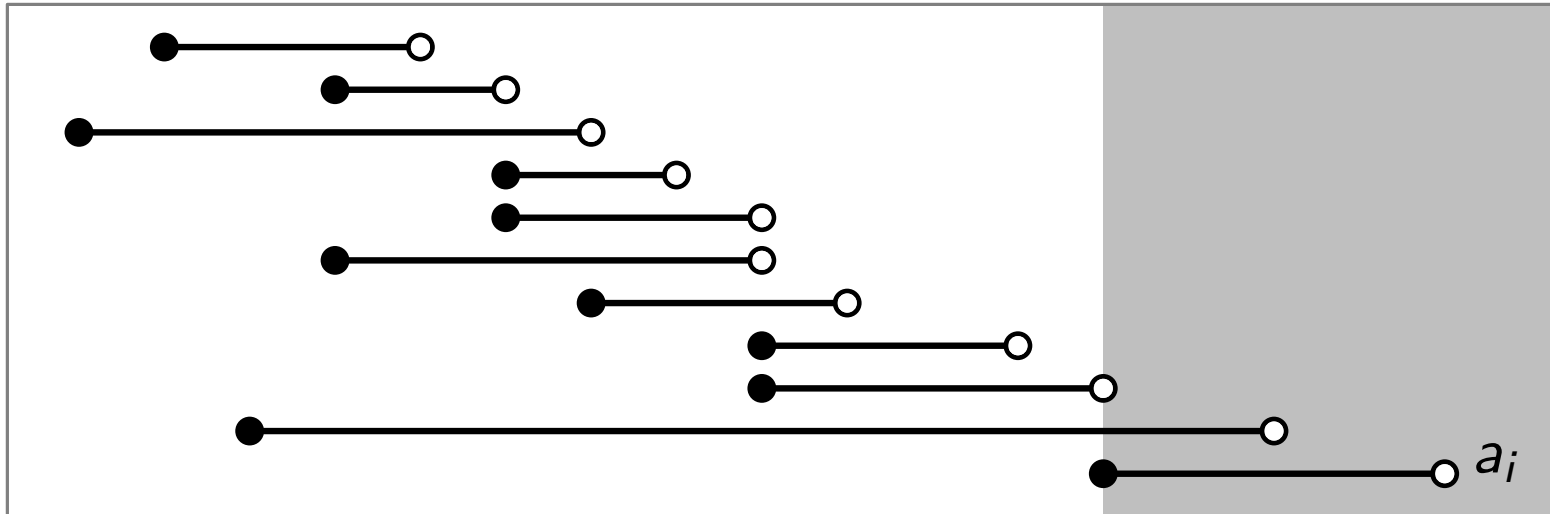
Ein exakter Algorithmus...

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 $A_{n+1} = A$.)



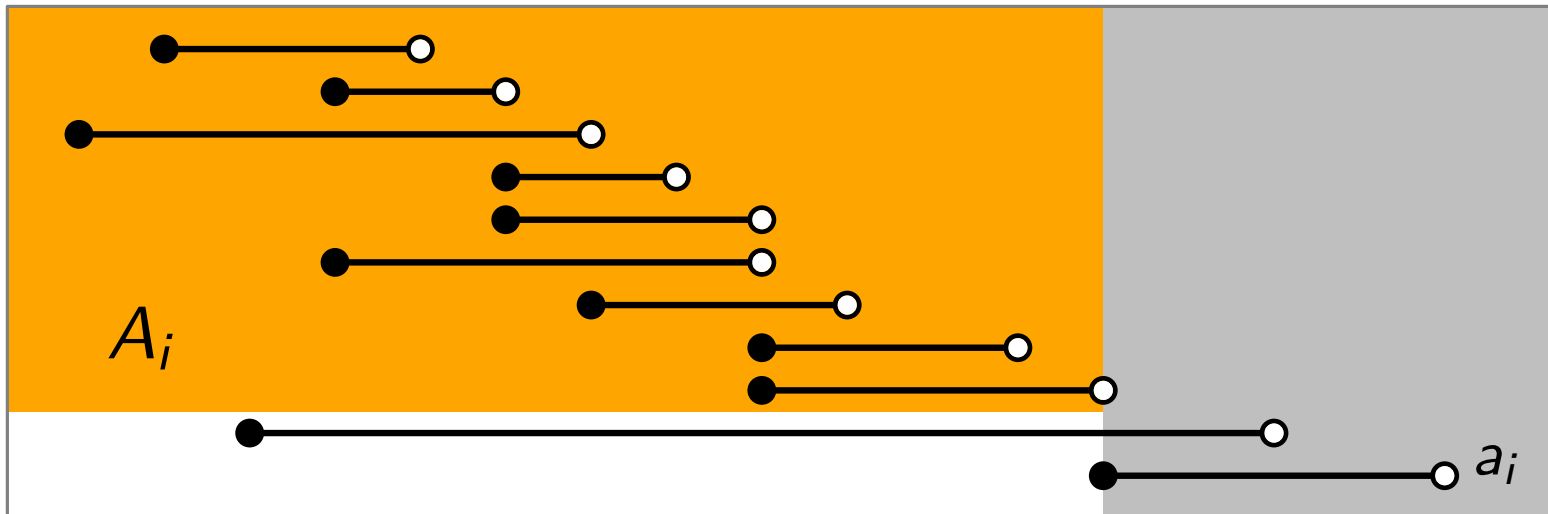
Ein exakter Algorithmus...

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 $A_{n+1} = A$.)



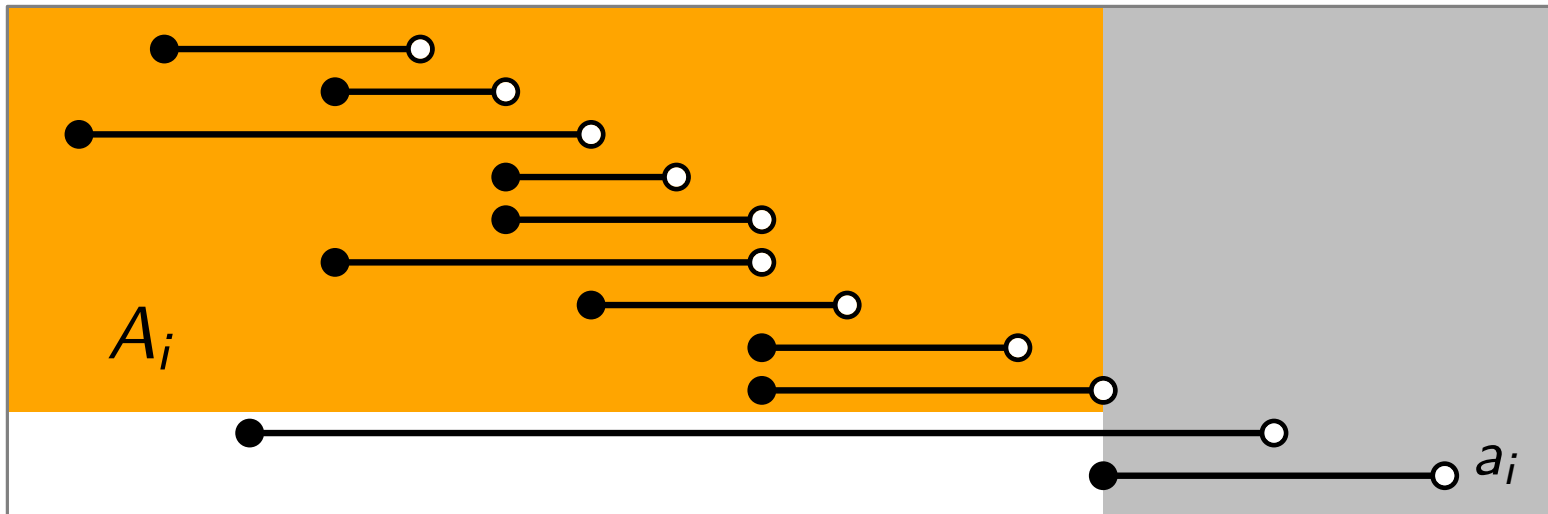
Ein exakter Algorithmus...

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 $A_{n+1} = A$.)



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 $A_{n+1} = A$.)

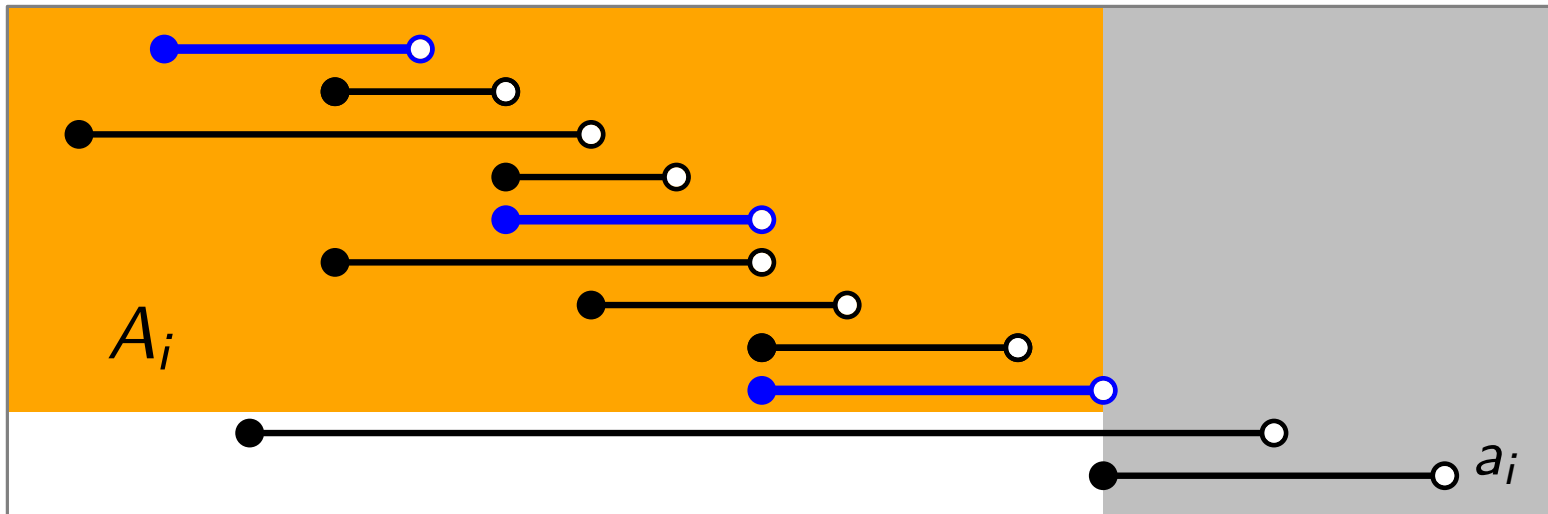


Eine optimale Lösung für A_i besteht aus:

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

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 $A_{n+1} = A$.)

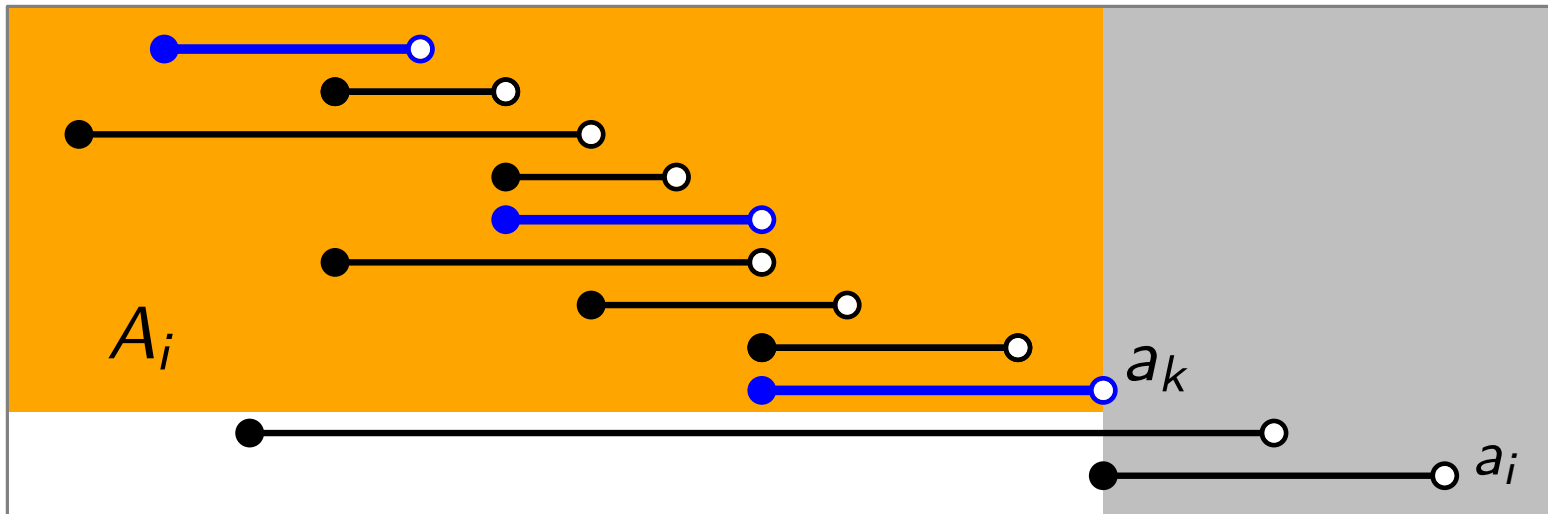


Eine optimale Lösung für A_i besteht aus:

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

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 $A_{n+1} = A$.)

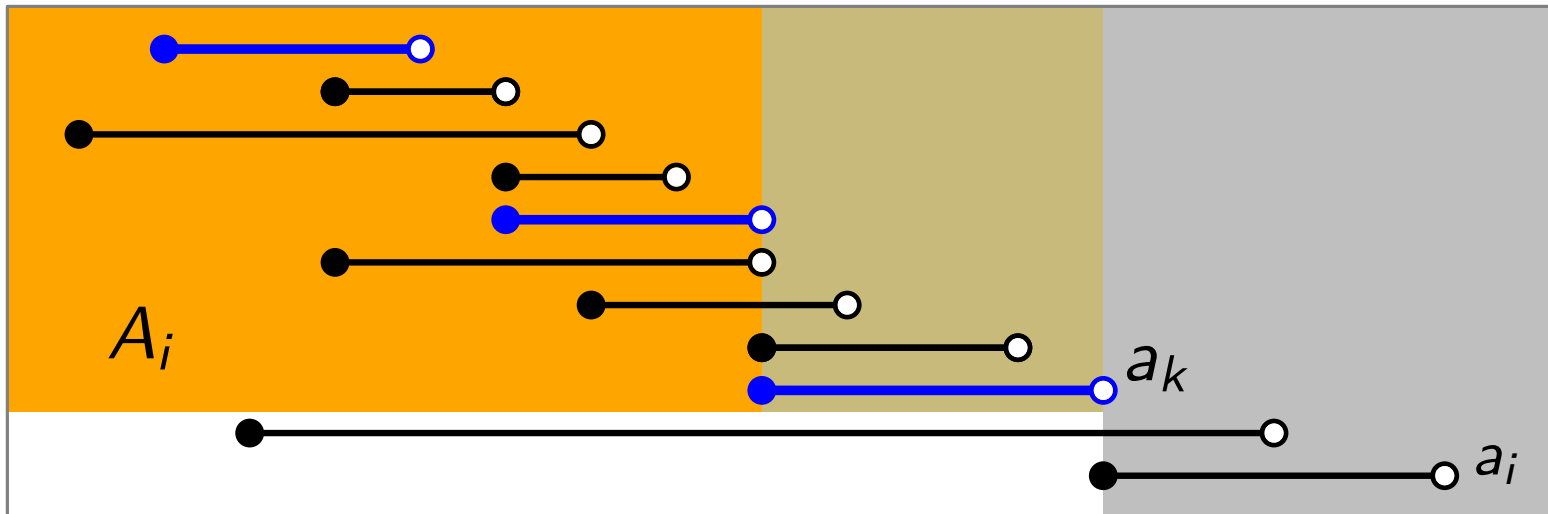


Eine optimale Lösung für A_i besteht aus:

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

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 $A_{n+1} = A$.)

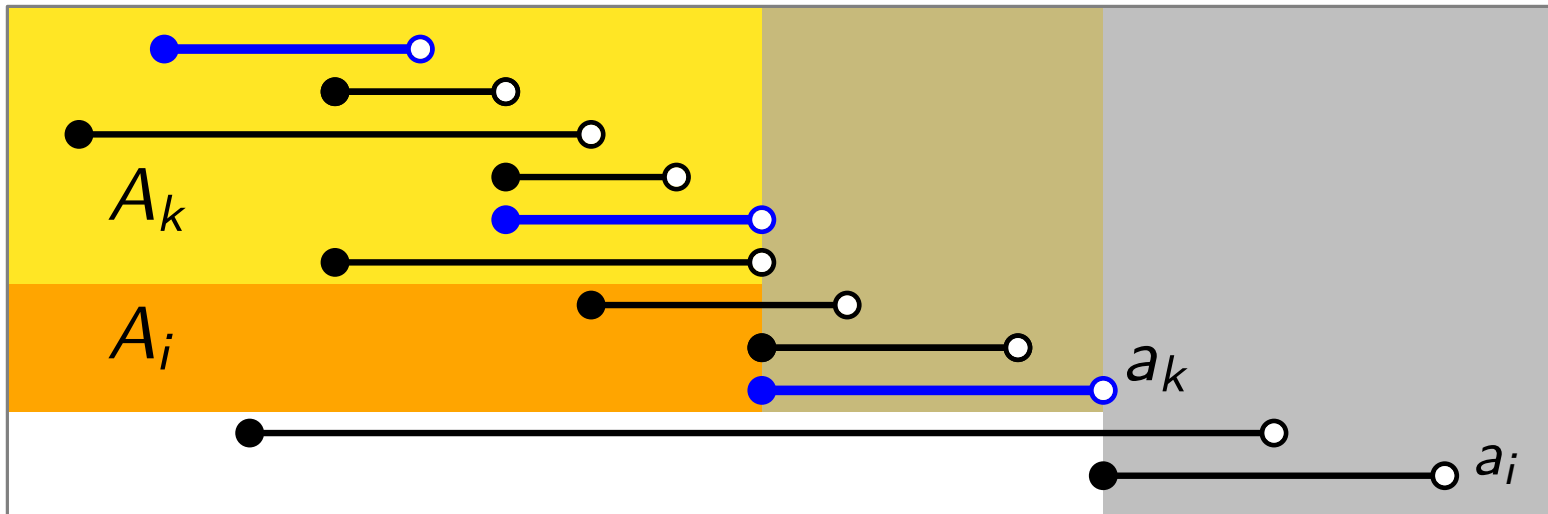


Eine optimale Lösung für A_i besteht aus:

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

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 $A_{n+1} = A$.)

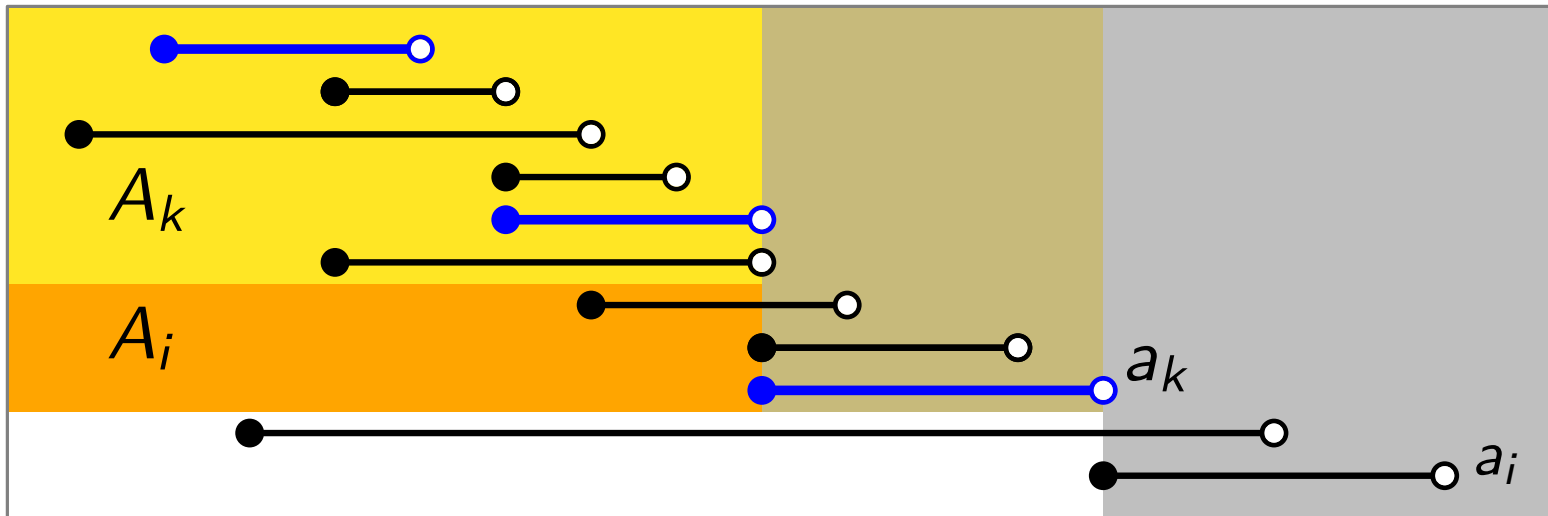


Eine optimale Lösung für A_i besteht aus:

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

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 $A_{n+1} = A$.)



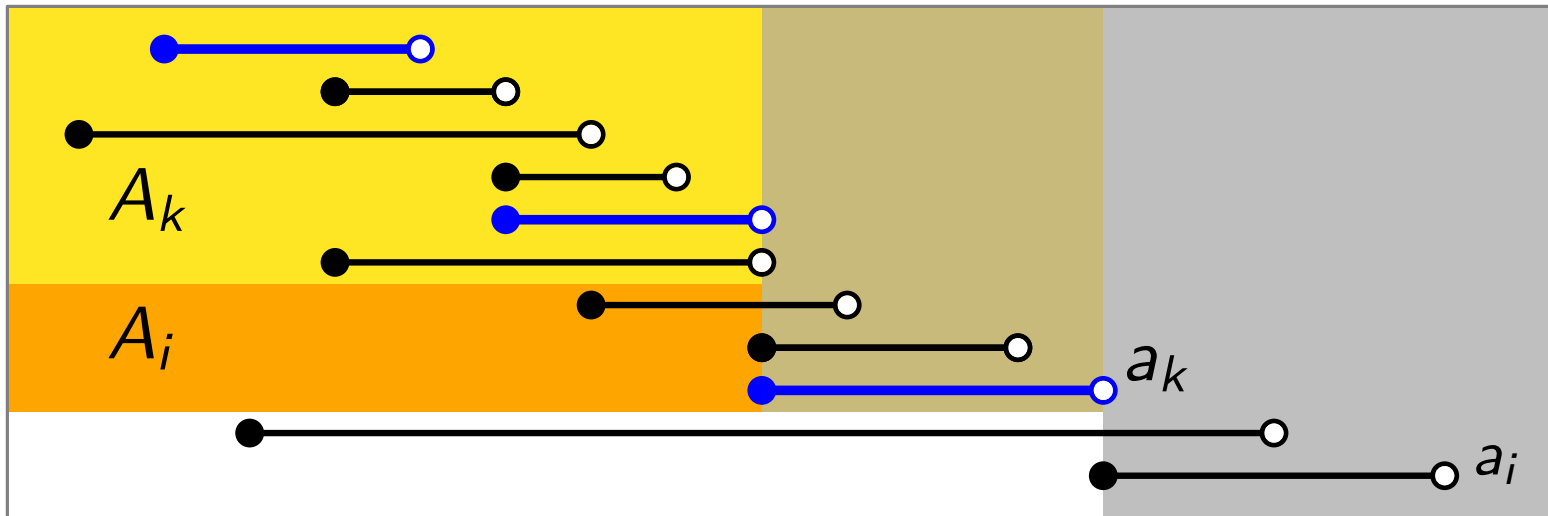
Eine optimale Lösung für A_i besteht aus:

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

} optimale Teilstruktur!

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:

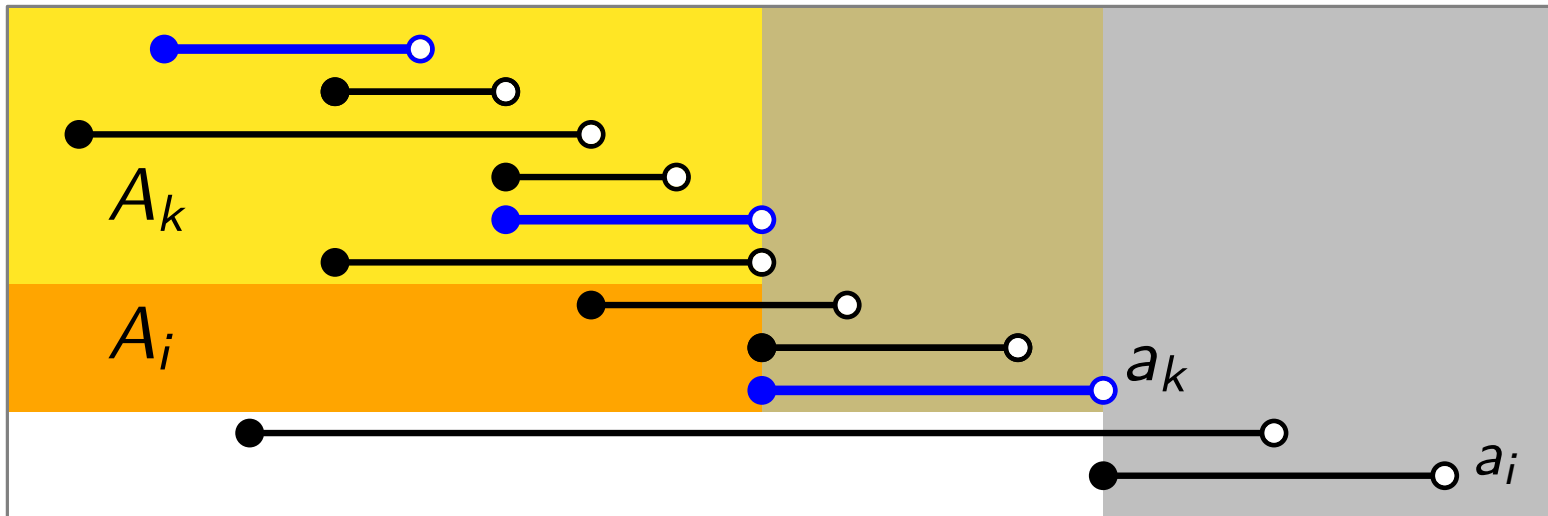
- *einem* letzten Intervall a_k und
- einer optimalen Lösung für A_k .

} optimale Teilstruktur!

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

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 .

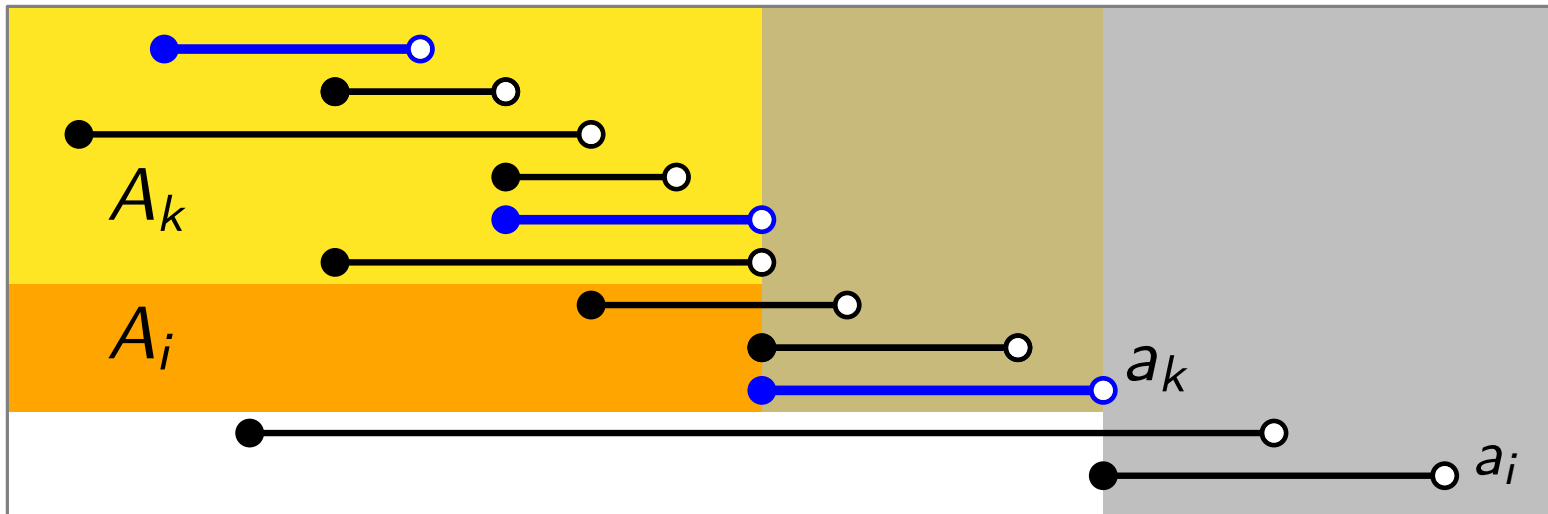
} optimale Teilstruktur!

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

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

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 .

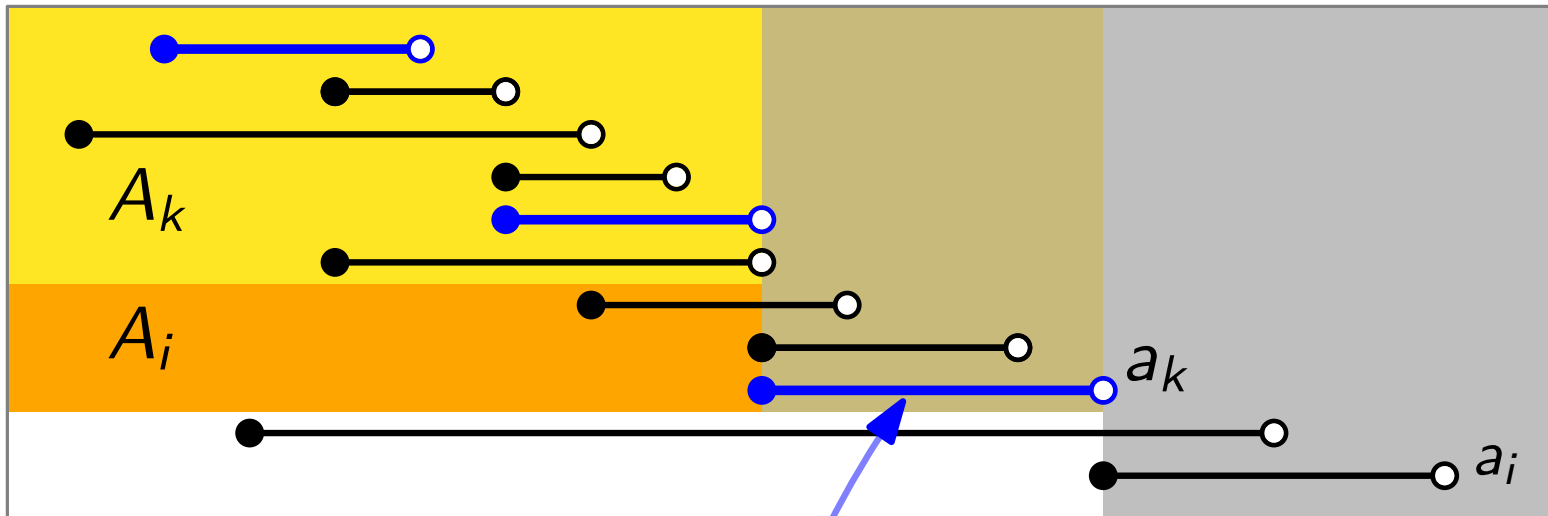
} optimale Teilstruktur!

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

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

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 .

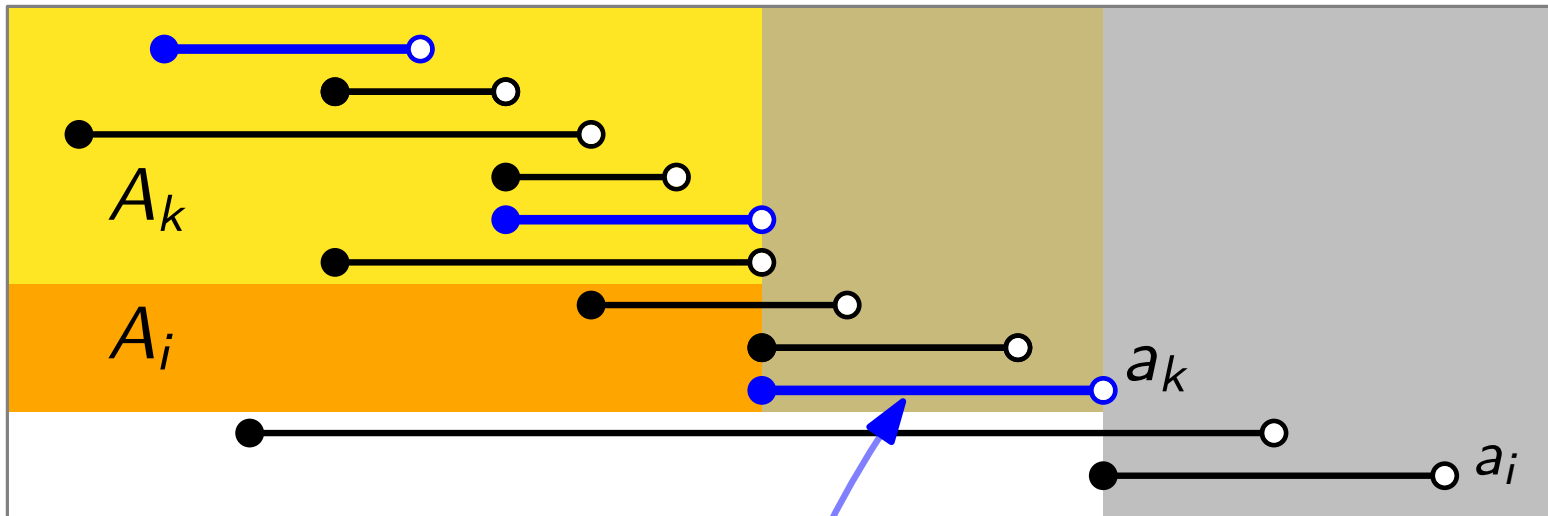
} optimale Teilstruktur!

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

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

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 .

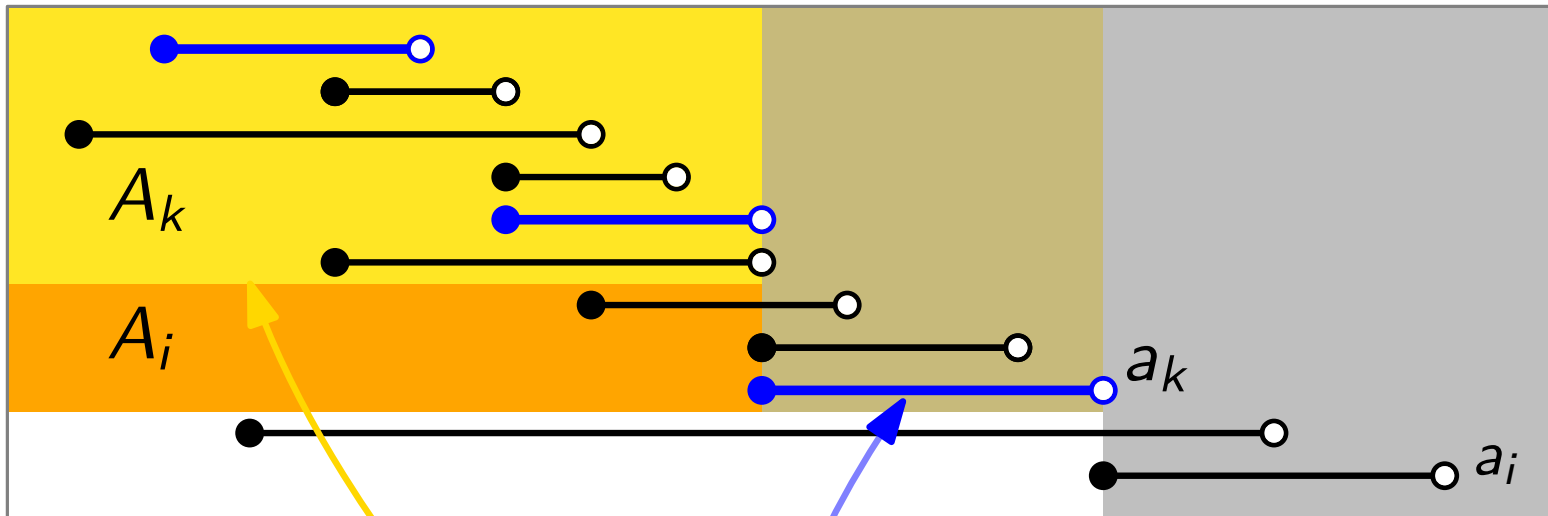
} optimale Teilstruktur!

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

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

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 :

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

...ein Dynamisches Programm!

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

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

...ein Dynamisches Programm!

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

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

Erinnern wir uns...

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

...ein Dynamisches Programm!

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

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

Erinnern wir uns...

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

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

...ein Dynamisches Programm!

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

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

Erinnern wir uns...

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

Also genügt es $c_1, \dots, c_{n+1}$ zu berechnen, wobei $c_1 =$

...ein Dynamisches Programm!

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

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

Erinnern wir uns...

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

Also genügt es $c_1, \dots, c_{n+1}$ zu berechnen, wobei $c_1 = 0$.

...ein Dynamisches Programm!

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

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

Erinnern wir uns...

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

Also genügt es $c_1, \dots, c_{n+1}$ zu berechnen, wobei $c_1 = 0$.

Laufzeit?

...ein Dynamisches Programm!

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

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

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 $c_1 = 0$.

Laufzeit?

...ein Dynamisches Programm!

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

$$c_i = \max_{a_k \in A_i} c_k + \ell(a_k) \quad \text{BERECHNUNG EINES TABELLENEINTRAGS}$$

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 $c_1 = 0$.

Laufzeit?

...ein Dynamisches Programm!

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

$$c_i = \max_{a_k \in A_i} c_k + \ell(a_k) \quad \text{BERECHNUNG EINES TABELLENEINTRAGS}$$

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 $O(\textcolor{red}{n})$

Laufzeit?

...ein Dynamisches Programm!

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

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

Erinnern wir uns...

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

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

Laufzeit?

...ein Dynamisches Programm!

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

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

Erinnern wir uns...

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

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

Laufzeit? $O(\textcolor{red}{n}^2)$

...ein Dynamisches Programm!

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

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

Erinnern wir uns...

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

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

Laufzeit? $O(\textcolor{red}{n}^2)$

Schreiben Sie den Pseudocode!

...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:

...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 $\overset{\text{TABELLE}}{c_1, \dots, c_{n+1}}$ 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.

...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 $\overset{\text{TABELLE}}{c_1, \dots, c_{n+1}}$ 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*.

...ein Dynamisches Programm!

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

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

Erinnern wir uns...

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

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

Laufzeit? $O(\textcolor{red}{n}^2)$

Schreiben Sie den Pseudocode!

Resultate:

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

Trade-Off zwischen Zeit und Qualität!