19. Gierige Algorithmen

- Gierige Algorithmen sind eine Algorithmenmethode, um so genannte Optimierungsprobleme zu lösen.

- Bei einem Optimierungsproblem gibt es zu jeder *Probleminstanz* viele mögliche oder zulässige Lösungen. Lösungen haben Werte gegeben durch eine Zielfunktion. Gesucht ist dann eine möglichst gute zulässige Lösung. Also eine Lösung mit möglichst kleinem oder möglichst großem Wert (Minimierungs- bzw. Maximierungsproblem).

Gierige (Greedy) Algorithmen

Gierige Strategie für Optimierungsprobleme:

- Aufbau einer Lösung in „kleinen“ Schritten
- In jedem dieser Schritte wird entsprechend eines definierten Optimierungskriteriums eine irreversible Entscheidung getroffen

Frage 1:

Wann kann eine solche Strategie zu einer optimalen Lösung führen?

Frage 2:

Wie beweist man, dass ein gieriger Algo. eine optimale Lösung liefert?

Gierige Algorithmen – Minimale Spannbäume

- Beispiel: Minimale Spannbäume:

  1. Probleminstanz – gewichteter, ungerichteteter, zusammenhängender Graph

  2. Zulässige Lösungen – Spannbäume

  3. Zielfunktion – Gewicht eines Spannbaums

  4. Gesucht – Spannbaum minimalen Gewichts

Gierige Algorithmen – Idee und Prim

1. Gierige Algorithmen bestimmen Lösung durch sukzessives Er-weitern von bereits gefundenen Teillösungen.

2. Erweitern geschieht durch lokal optimale Wahlen.

3. Analyse muss dann zeigen, dass lokal optimale Wahlen zu global optimalen Lösungen führt.

1. Algorithmus von Prim bestimmt minimalen Spannbaum durch sukzessives Hinzufügen von Kanten.

2. Prims Algorithmus wählt möglichst leichte Kante, die isolierten Knoten mit Teilbaum verbindet.

3. Analyse mit Hilfe von Schnitten zeigte, dass Teilbaum immer in einem minimalen Spannbaum enthalten ist.

Gierige Algorithmen – Idee und Kruskal

1. Gierige Algorithmen bestimmen Lösung durch sukzessives Erweitern von bereits gefundenen Teillösungen.

2. Erweitern geschieht durch lokal optimale Wahlen.

3. Analyse muss dann zeigen, dass lokal optimale Wahlen zu global optimalen Lösungen führt.

1. Algorithmus von Kruskal bestimmt minimalen Spannbaum durch sukzessives Hinzufügen von Kanten.

2. Kruskals Algorithmus wählt möglichst leichte Kante, die Zusammenhangskomponenten verbindet.

3. Analyse mit Hilfe von Schnitten zeigte, dass Teilbaum immer in einem minimalen Spannbaum enthalten ist.

Gieriges 1-Prozessor-Scheduling (1)

- Gegeben sind n Jobs $j_1,\dots,j_n$ mit Dauer $t_1,\dots,t_n$.
- Jeder der Jobs muss auf einem einzigen Prozessor abgearbeitet werden. Der Prozessor kann zu jedem Zeitpunkt nur einen Job bearbeiten. Ein einmal begonnener Job darf nicht abgebrochen werden.
- Gesucht ist eine Reihenfolge, in der die Jobs abgearbeitet werden, so dass die durchschnittliche Bearbeitungszeit der Jobs möglichst gering ist.

Gieriges 1-Prozessor-Scheduling (2)

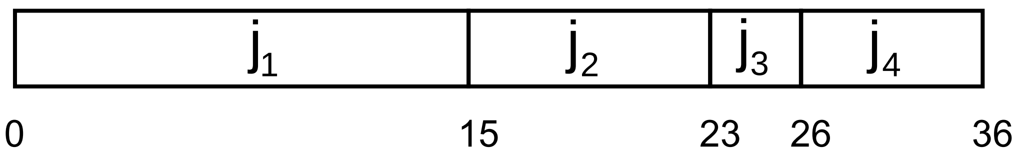
- Bearbeitungszeit eines Jobs ist der Zeitpunkt, an dem der Job vollständig bearbeitet wurde.

- Werden die Jobs in Reihenfolge $j_{\pi(1)},\ldots,j_{\pi(n)}$ ausgeführt, wobei π eine Permutation ist, so ist die Bearbeitungszeit von Job $j_{\pi(1)}$ genau $t_{\pi(1)}$, die Bearbeitungszeit von Job $j_{\pi(2)}$ ist $t_{\pi(1)}+t_{\pi(2)}$, usw.

Gieriges 1-Prozessor-Scheduling - Beispiel

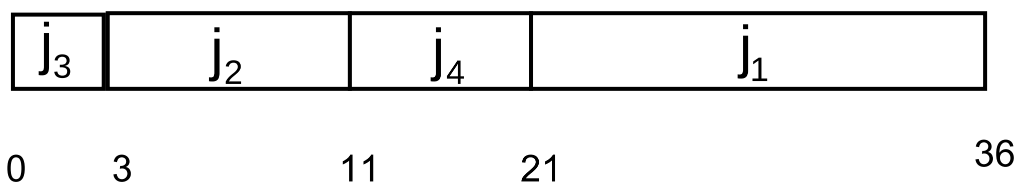
Job	Laufzeit
j_1	15
j_2	8
j_3	3
j_4	10

Schedule Nr. 1:



Durchschnittliche Beendigung: 25

Schedule Nr. 2:



Durchschnittliche Beendigung: 17,75

Gieriges 1-Prozessor-Scheduling (3)

Lemma 19.1: Bei Permutation π ist die durchschnittliche Bearbeitungszeit gegeben durch

$$\frac{1}{n}\sum_{i=1}^{n}(n-i+1)\,t_{\pi(i)}$$

Lemma 19.2: Eine Permutation π führt genau dann zu einer minimalen durchschnittlichen Bearbeitungszeit, wenn die Folge $(t_{\pi(1)},\ldots,t_{\pi(n)})$ aufsteigend sortiert ist.

Gieriges Mehr-Prozessor-Scheduling

- Gegeben sind n Jobs $j_1,\ldots,j_n$ mit Dauer $t_1,\ldots,t_n$ und m identische Prozessoren.

- Jeder der Jobs muss auf einem einzigen Prozessor abgearbeitet werden. Jeder Prozessor kann zu jedem Zeitpunkt nur einen Job bearbeiten. Ein einmal begonnener Job darf nicht abgebrochen werden.

- Gesucht ist eine Aufteilung der Jobs auf die Prozessoren und für jeden Prozessor eine Reihenfolge der ihm zugewiesenen Jobs, so dass durchschnittliche Bearbeitungszeit der Jobs möglichst gering ist.

- Bearbeitungszeit eines Jobs wie vorher definiert.

Gieriges Mehr-Prozessor-Scheduling (2)

Lemma 19.3: Die Permutation π sei so gewählt, dass die Folge $(t_{\pi(1)},\ldots,t_{\pi(n)})$ aufsteigend sortiert ist. Weiter werde dann Job $j_{\pi(i)}$ auf dem Prozessor mit Nummer i mod m ausgeführt (wobei wir für alle i mit i mod $m = 0$ Prozessor m verwenden). Das so konstruierte Scheduling minimiert dann die durchschnittliche Bearbeitungszeit.

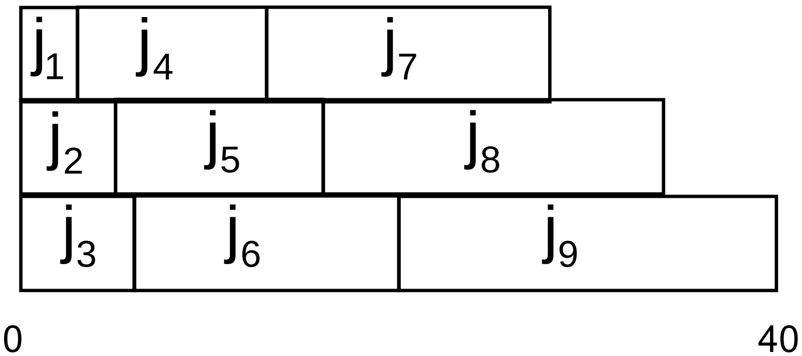
Beweisidee:

1. Hat Prozessor i mehr als einen Job mehr als Prozessor j, verbessert sich die durchschnittliche Bearbeitungszeit, wenn ein Job von i nach j verschoben wird.
2. Der Tausch zweier Jobs an derselben Ausführungsposition in zwei Prozessoren mit derselben Anzahl an Jobs verändert nicht die durchschnittliche Bearbeitungszeit.
3. Die durchschnittliche Bearbeitungszeit innerhalb eines Prozessors ist minimal, wenn die Jobs nach aufsteigender Bearbeitungszeit ausgeführt werden.

Gieriges Mehr-Prozessor-Scheduling - Beispiel

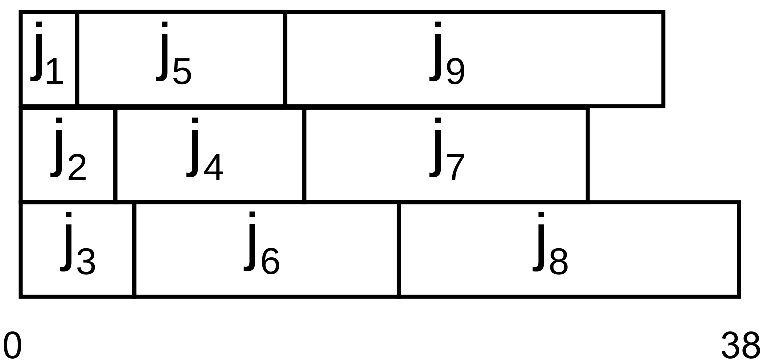
Job	Laufzeit
j_1	3
j_2	5
j_3	6
j_4	10
j_5	11
j_6	14
j_7	15
j_8	18
j_9	20

Schedule Nr. 1 (Methode aus Lemma 19.3):



Durchschnittliche Beendigung: 18,33

Schedule Nr. 2:



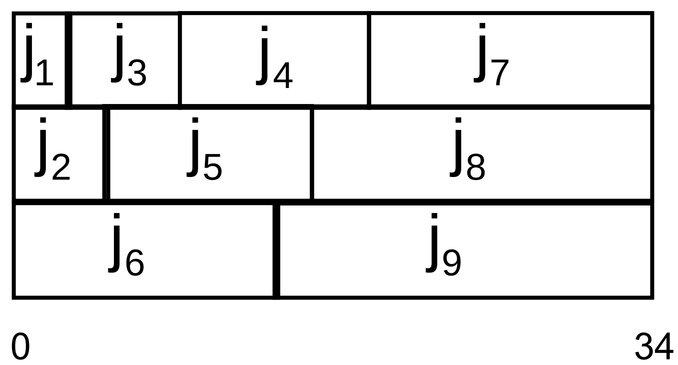
Durchschnittliche Beendigung: 18,33

Gierig ist nicht immer optimal

Job	Laufzeit
j_1	3
j_2	5
j_3	6
j_4	10
j_5	11
j_6	14
j_7	15
j_8	18
j_9	20

- Betrachten dasselbe Szenario wie vorher mit Jobs und Prozessoren.
- Wollen aber jetzt den Zeitpunkt minimieren, an dem alle Jobs beendet sind.

Optimaler Schedule:



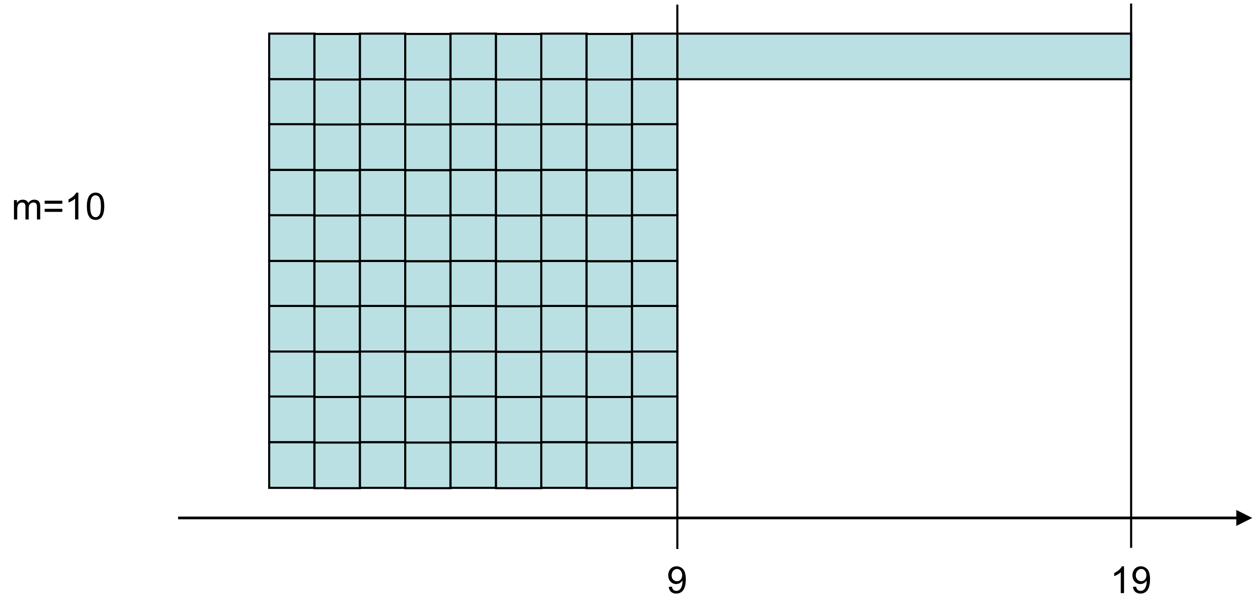
Greedy Scheduling (vorige Folie): 40

Gierig ist nicht immer optimal

Geht es schlimmer? Ja!

Beispiel: m Prozessoren, m(m-1) Jobs der Länge 1,
ein Job der Länge m

Greedy Schedule für m=10:



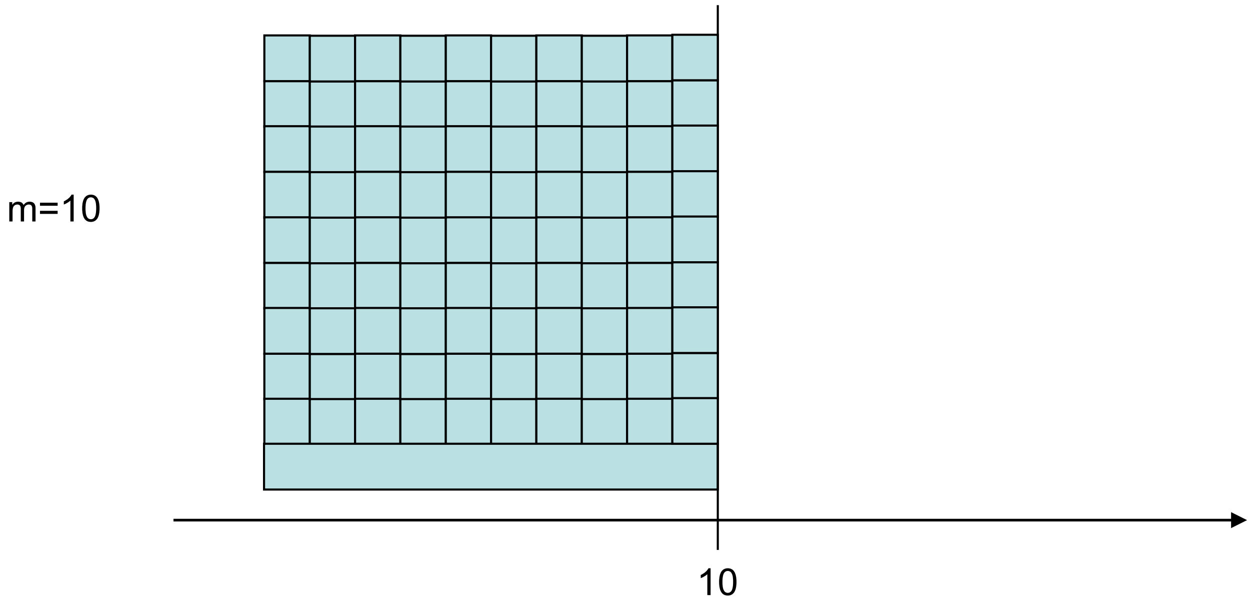
Gierig ist nicht immer optimal

Greedy Schedule: 2m-1 Zeiteinheiten

Optimaler Schedule: m Zeiteinheiten

Greedy kann also Faktor 2-1/m schlechter als OPT sein.

Das ist aber für alle Instanzen auch der worst case.



Gierig ist nicht immer optimal

Greedy Schedule über absteigend sortierte Jobzeiten:

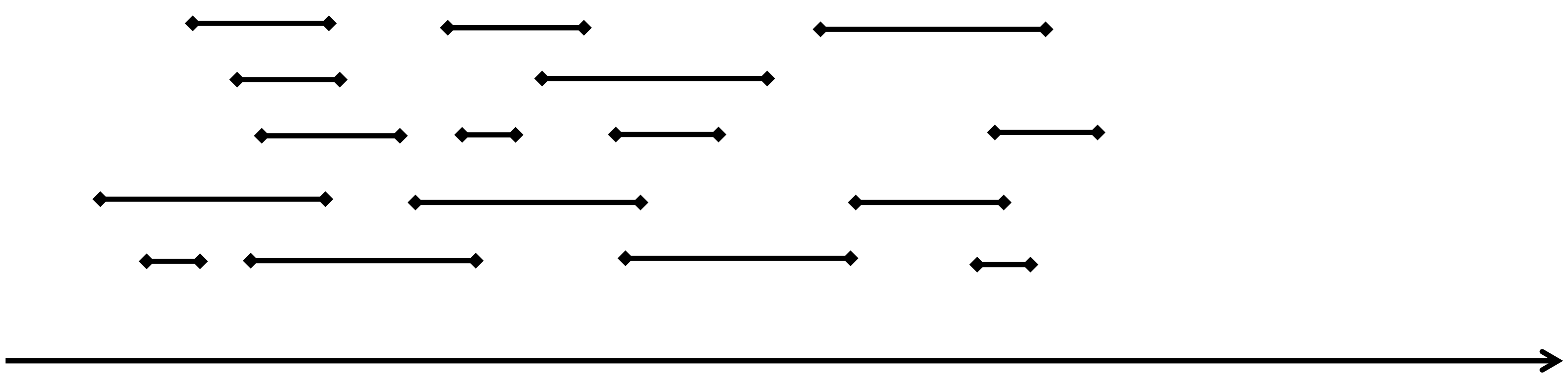
Kann sogar nie schlechter als $4/3 \cdot OPT$ sein.
Beweis ist sehr aufwändig.

4/3-Resultat ist bestmöglich.
Beispiel: m Maschinen, $n=2m+1$ Jobs: jeweils 2 Jobs der Länge $m+1, m+2, \ldots, 2m$ und ein Job der Länge m
Vergleich zu OPT: Übung.

Anwendungsbeispiel: Intervall Scheduling

Intervall Scheduling:

- eine Ressource (Hörsaal, Parallelrechner, ...)
- Menge von Anfragen der Art:
  Kann ich die Ressource für den Zeitraum [t_1 ,t_2] nutzen?

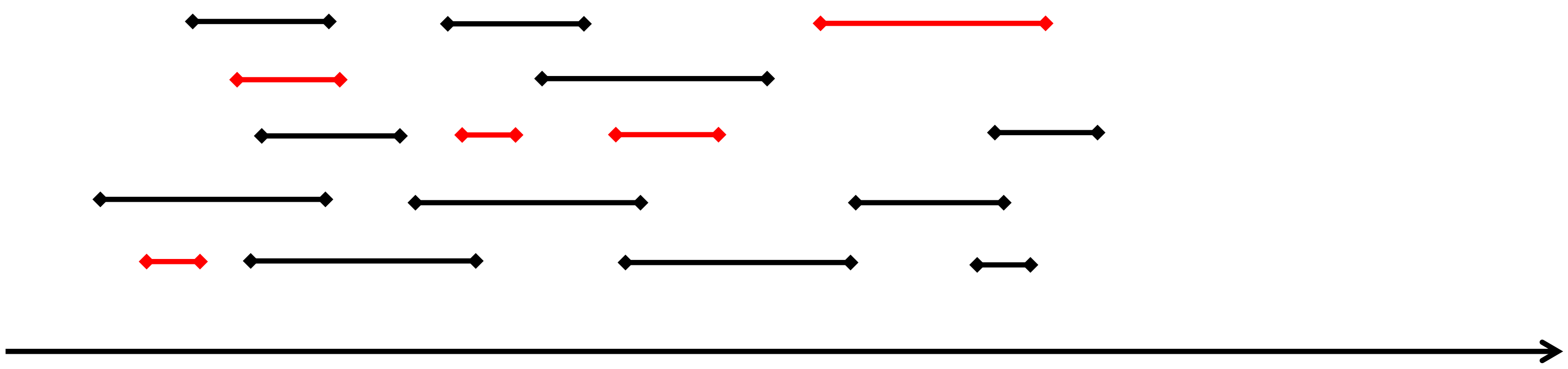


- *Optimierungs-Ziel:* Möglichst viele Anfragen erfüllen

Gierige Algorithmen – Intervall Scheduling

Intervall Scheduling:

- eine Ressource (Hörsaal, Parallelrechner, ...)
- Menge von Anfragen der Art:
  Kann ich die Ressource für den Zeitraum [t_1 ,t_2] nutzen?

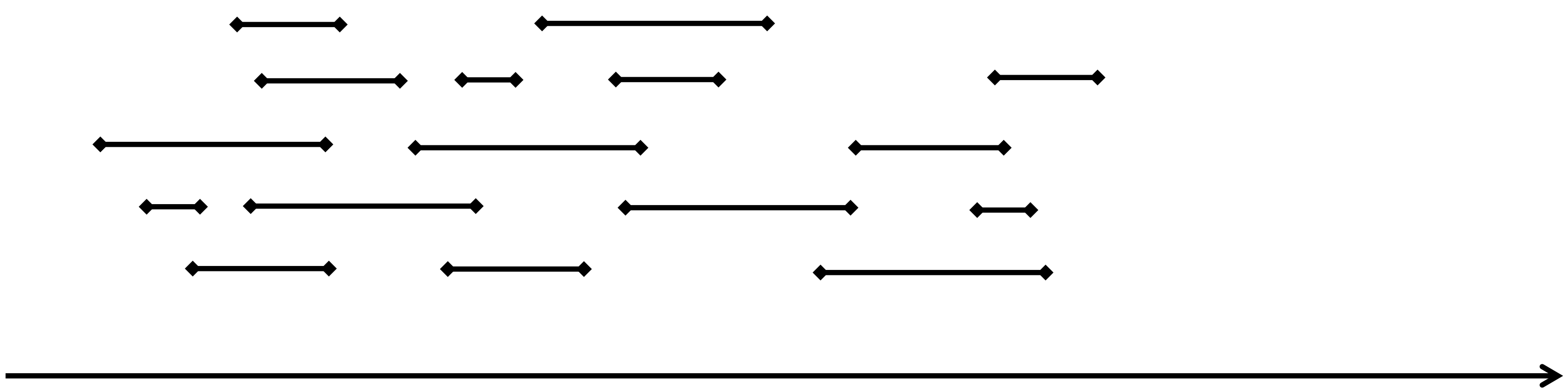


- *Optimierungs-Ziel:* Möglichst viele Anfragen erfüllen

Gierige Algorithmen – Intervall Scheduling

Definition:

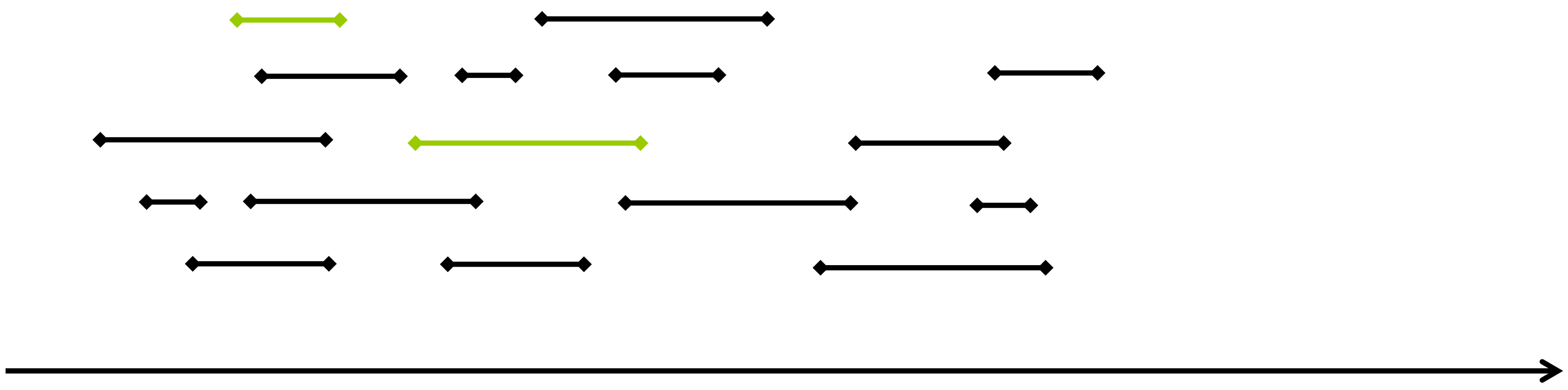
- Zwei Anfragen heißen kompatibel, wenn sich die Intervalle nicht überschneiden.



Gierige Algorithmen – Intervall Scheduling

Definition:

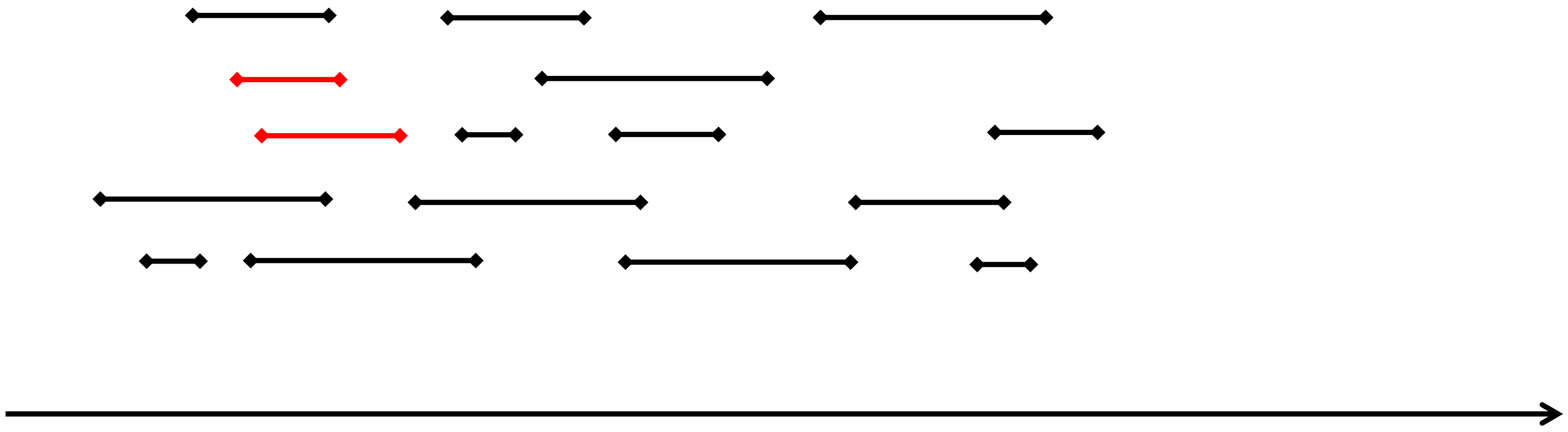
- Zwei Anfragen heißen kompatibel, wenn sich die Intervalle nicht überschneiden.



Gierige Algorithmen – Intervall Scheduling

Definition:

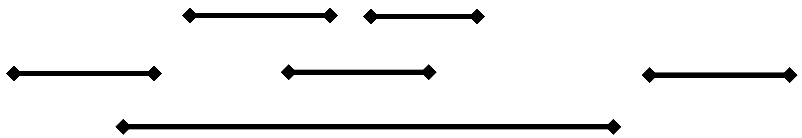
- Zwei Anfragen heißen kompatibel, wenn sich die Intervalle nicht überschneiden.



Gierige Algorithmen – Intervall Scheduling

Generelle Überlegung:

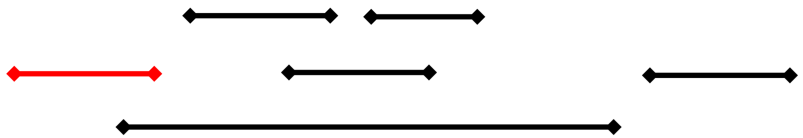
- Wähle erste Anfrage i_1 „geschickt“
- Ist i_1 akzeptiert, weise alle Anfragen zurück, die nicht kompatibel sind
- Wähle nächste Anfrage i_2 und weise alle Anfragen zurück, die nicht mit i_2 kompatibel sind
- Mache weiter, bis keine Anfragen mehr übrig sind



Gierige Algorithmen – Intervall Scheduling

Generelle Überlegung:

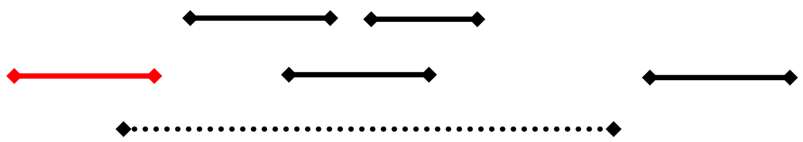
- Wähle erste Anfrage i_1 „geschickt“
- Ist i_1 akzeptiert, weise alle Anfragen zurück, die nicht kompatibel sind
- Wähle nächste Anfrage i_2 und weise alle Anfragen zurück, die nicht mit i_2 kompatibel sind
- Mache weiter, bis keine Anfragen mehr übrig sind



Gierige Algorithmen – Intervall Scheduling

Generelle Überlegung:

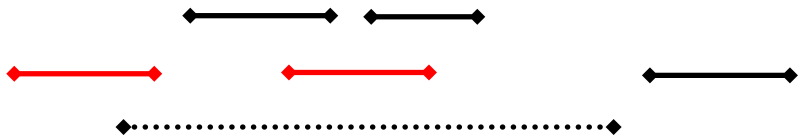
- Wähle erste Anfrage i_1 „geschickt“
- Ist i_1 akzeptiert, weise alle Anfragen zurück, die nicht kompatibel sind
- Wähle nächste Anfrage i_2 und weise alle Anfragen zurück, die nicht mit i_2 kompatibel sind
- Mache weiter, bis keine Anfragen mehr übrig sind



Gierige Algorithmen – Intervall Scheduling

Generelle Überlegung:

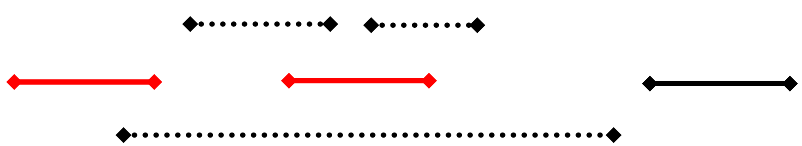
- Wähle erste Anfrage i_1 „geschickt"
- Ist i_1 akzeptiert, weise alle Anfragen zurück, die nicht kompatibel sind
- Wähle nächste Anfrage i_2 und weise alle Anfragen zurück, die nicht mit i_2 kompatibel sind
- Mache weiter, bis keine Anfragen mehr übrig sind



Gierige Algorithmen – Intervall Scheduling

Generelle Überlegung:

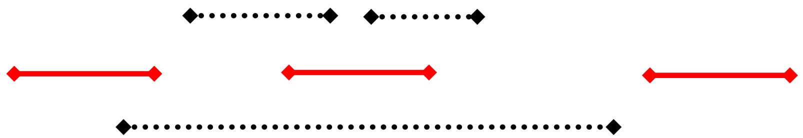
- Wähle erste Anfrage i_1 „geschickt“
- Ist i_1 akzeptiert, weise alle Anfragen zurück, die nicht kompatibel sind
- Wähle nächste Anfrage i_2 und weise alle Anfragen zurück, die nicht mit i_2 kompatibel sind
- Mache weiter, bis keine Anfragen mehr übrig sind



Gierige Algorithmen – Intervall Scheduling

Generelle Überlegung:

- Wähle erste Anfrage i_1 „geschickt“
- Ist i_1 akzeptiert, weise alle Anfragen zurück, die nicht kompatibel sind
- Wähle nächste Anfrage i_2 und weise alle Anfragen zurück, die nicht mit i_2 kompatibel sind
- Mache weiter, bis keine Anfragen mehr übrig sind



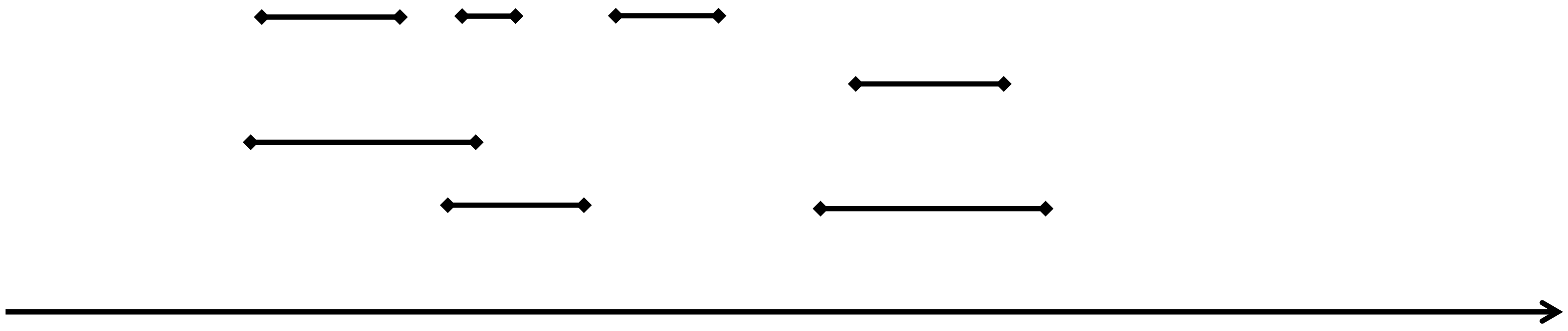
Gierige Algorithmen – Intervall Scheduling

Was ist nun aber eine „geschickte“ Auswahlstrategie ?

Gierige Algorithmen – Intervall Scheduling

Strategie 1:

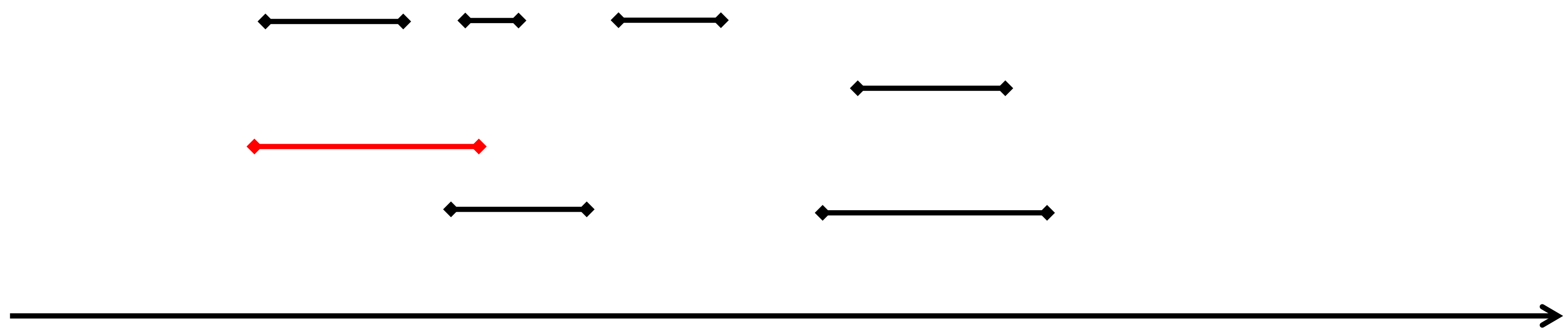
- Wähle immer die Anfrage, die am frühesten beginnt



Gierige Algorithmen – Intervall Scheduling

Strategie 1:

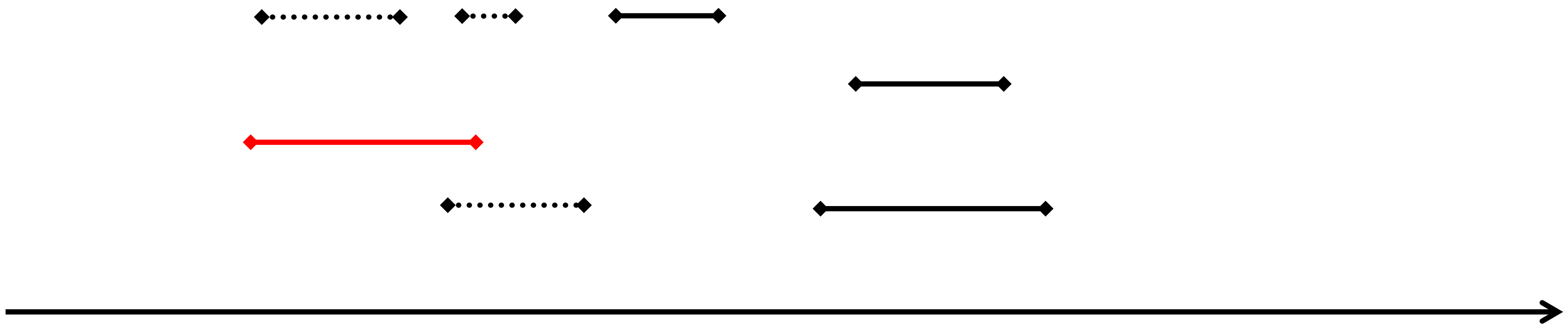
- Wähle immer die Anfrage, die am frühesten beginnt



Gierige Algorithmen – Intervall Scheduling

Strategie 1:

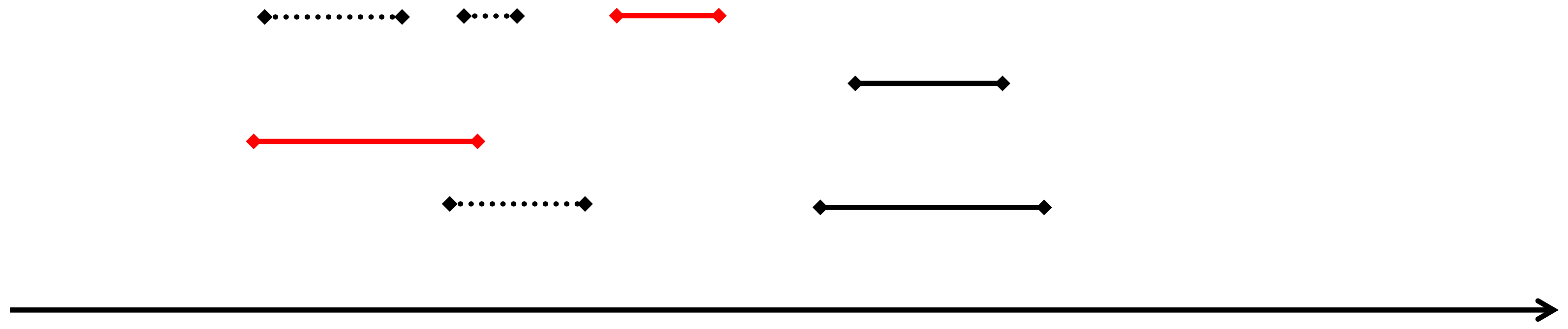
- Wähle immer die Anfrage, die am frühesten beginnt



Gierige Algorithmen – Intervall Scheduling

Strategie 1:

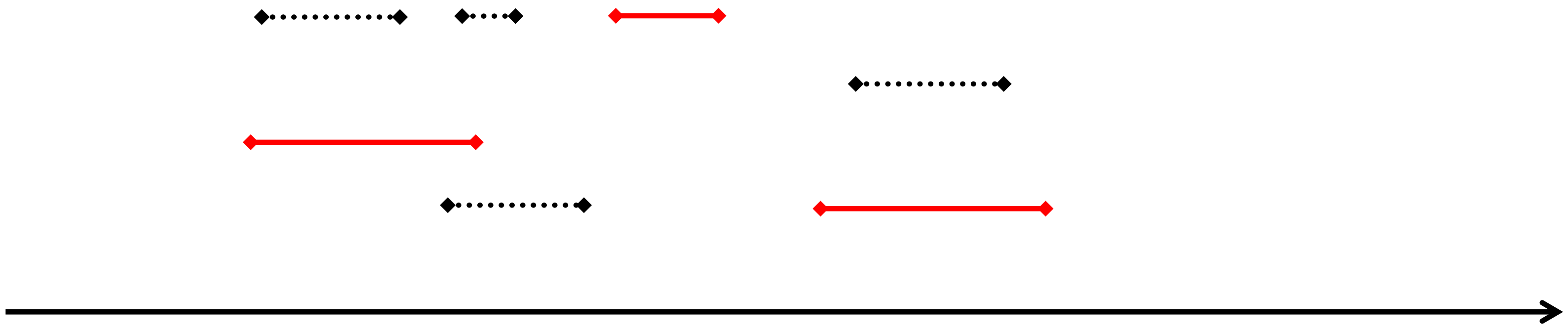
- Wähle immer die Anfrage, die am frühesten beginnt



Gierige Algorithmen – Intervall Scheduling

Strategie 1:

- Wähle immer die Anfrage, die am frühesten beginnt

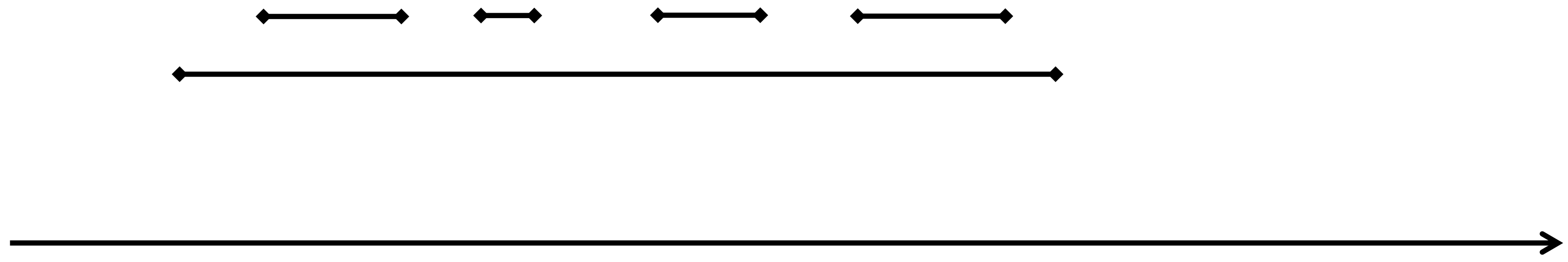


Gierige Algorithmen – Intervall Scheduling

Strategie 1:

- Wähle immer die Anfrage, die am frühesten beginnt

Optimalität ?

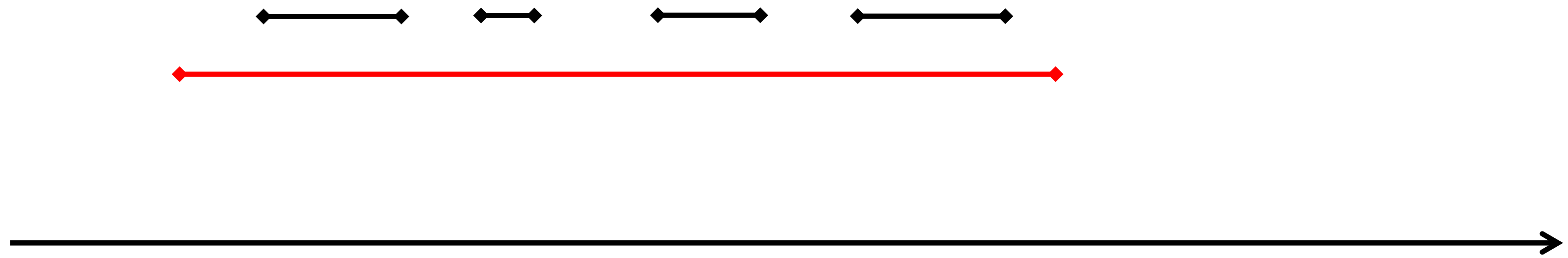


Gierige Algorithmen – Intervall Scheduling

Strategie 1:

- Wähle immer die Anfrage, die am frühesten beginnt

Optimalität ?

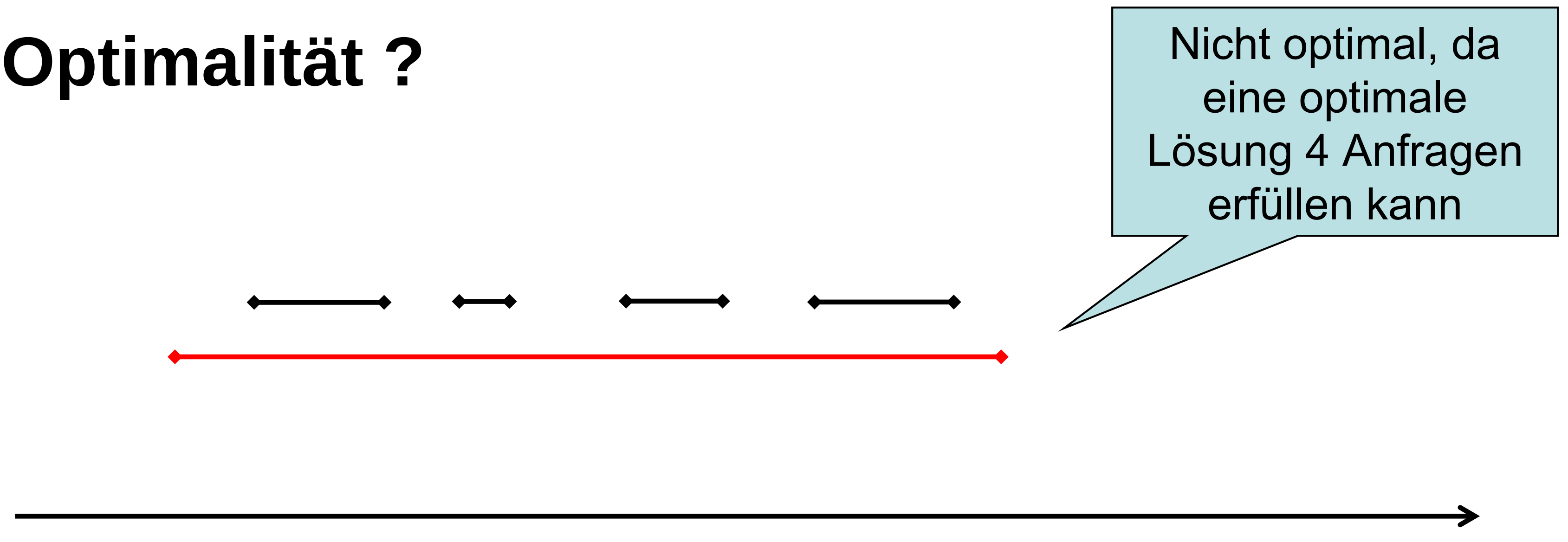


Gierige Algorithmen – Intervall Scheduling

Strategie 1:

- Wähle immer die Anfrage, die am frühesten beginnt

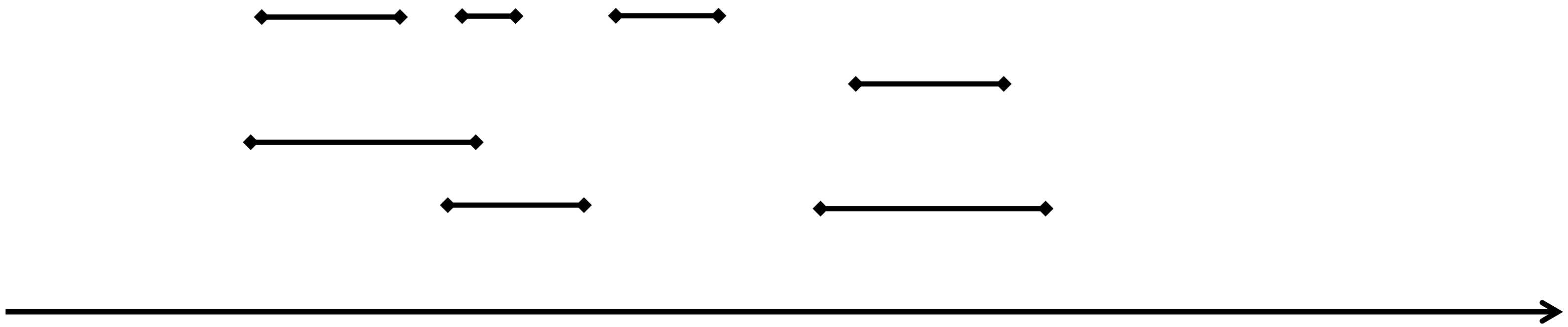
Optimalität ?



Gierige Algorithmen – Intervall Scheduling

Strategie 2:

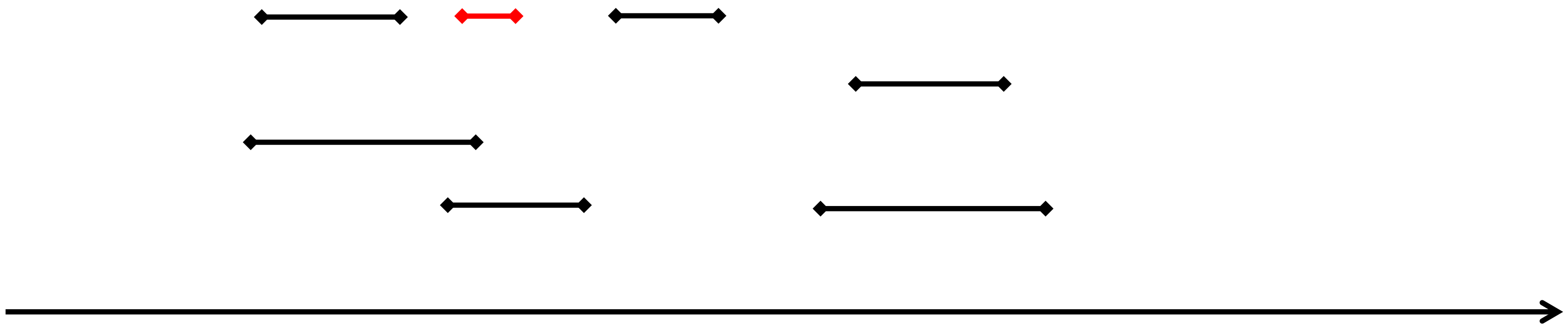
- Wähle immer das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 2:

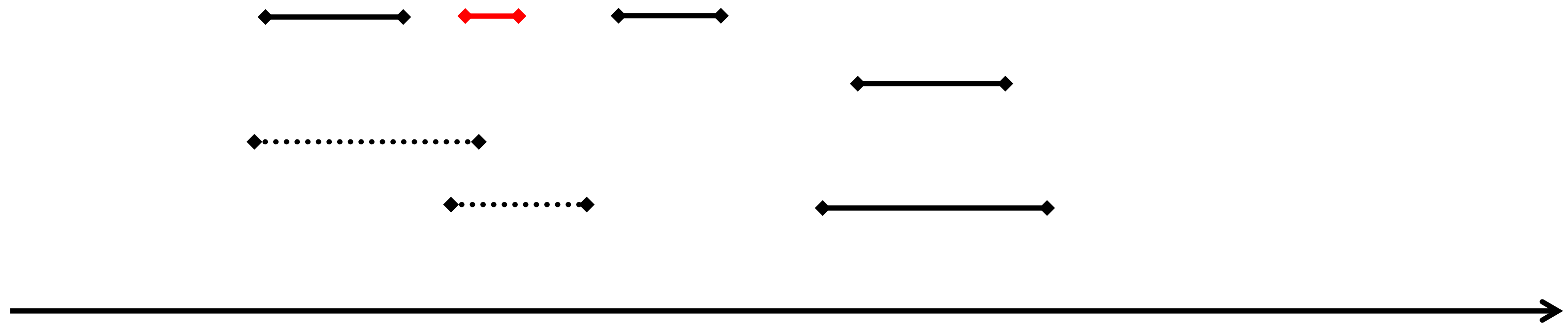
- Wähle immer das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 2:

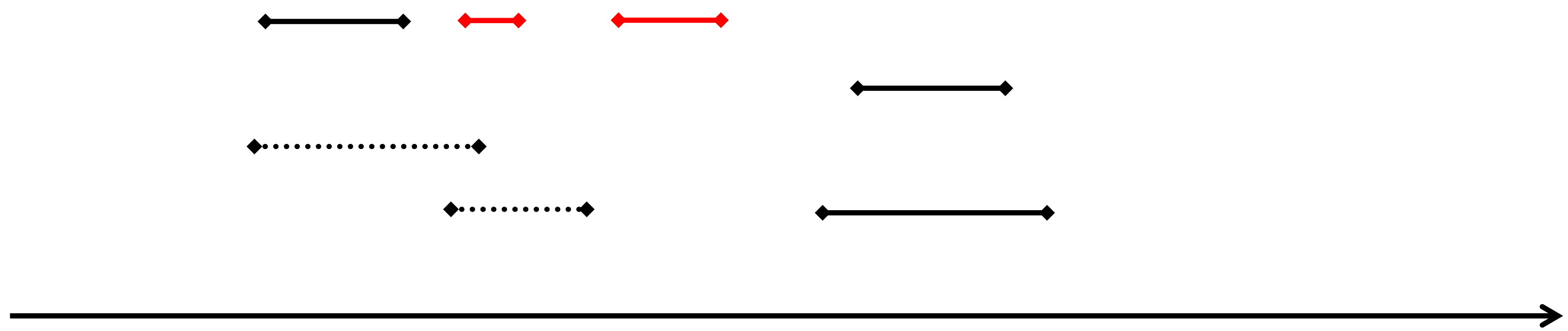
- Wähle immer das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 2:

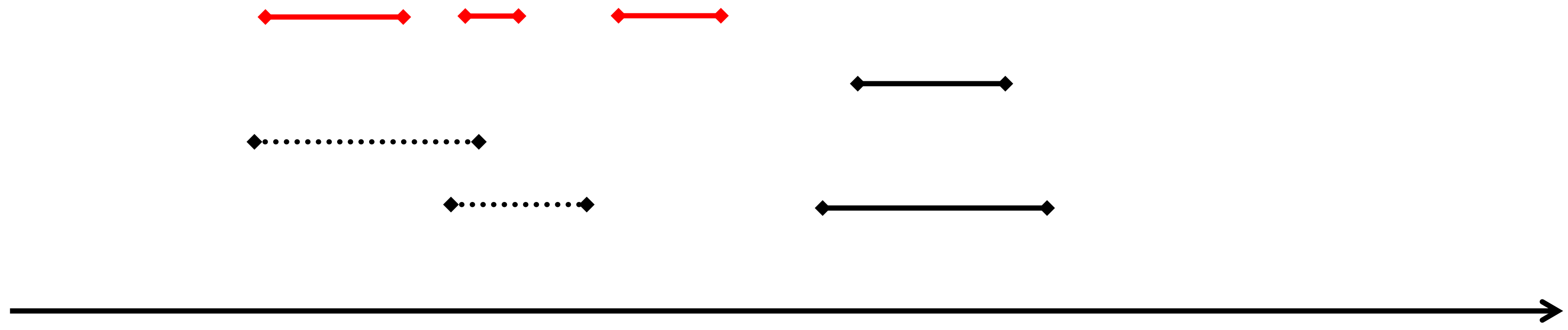
- Wähle immer das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

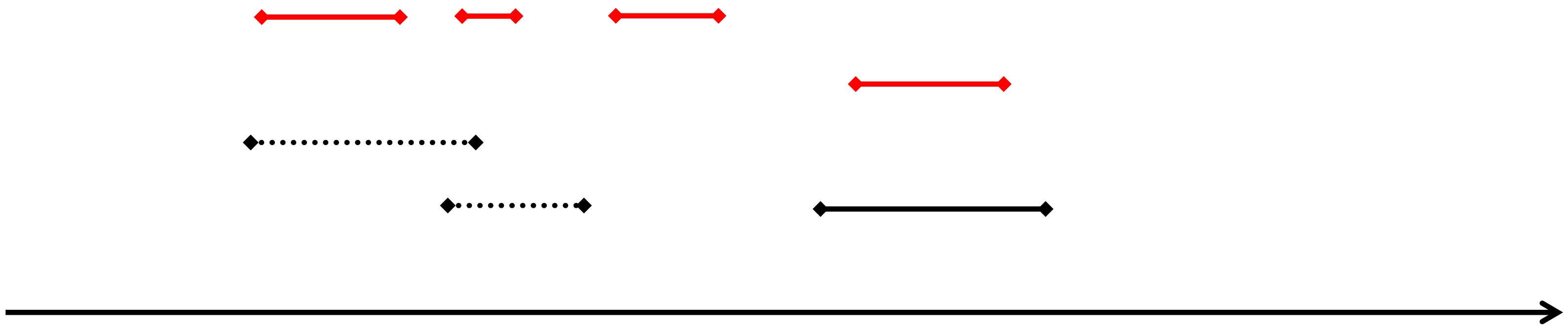
Strategie 2:

- Wähle immer das kürzeste Intervall



Strategie 2:

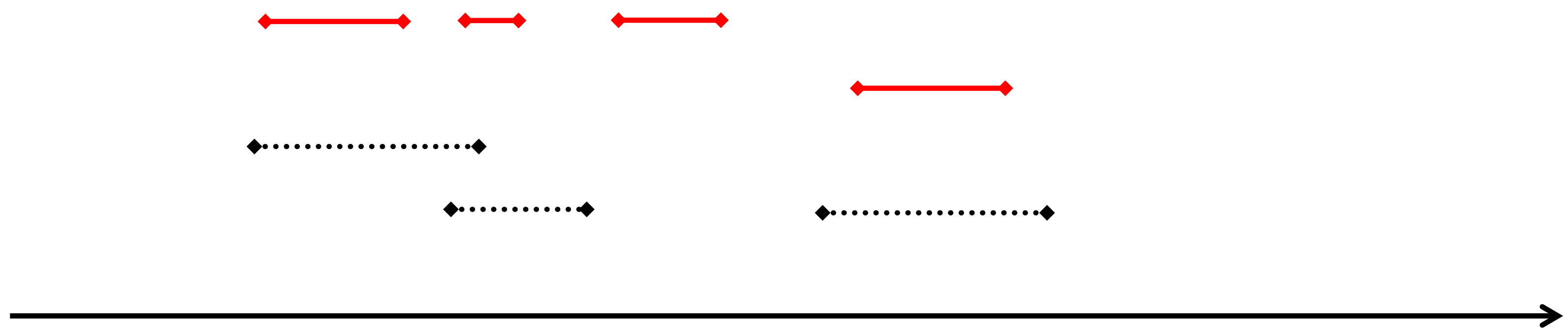
- Wähle immer das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 2:

- Wähle immer das kürzeste Intervall

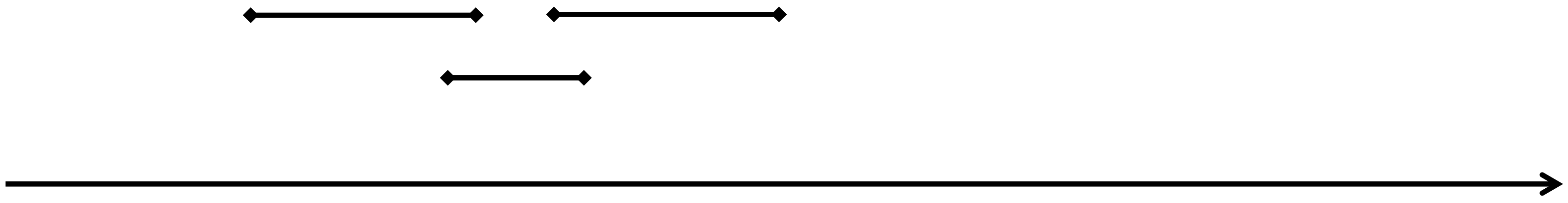


Gierige Algorithmen – Intervall Scheduling

Strategie 2:

- Wähle immer das kürzeste Intervall

Optimalität?

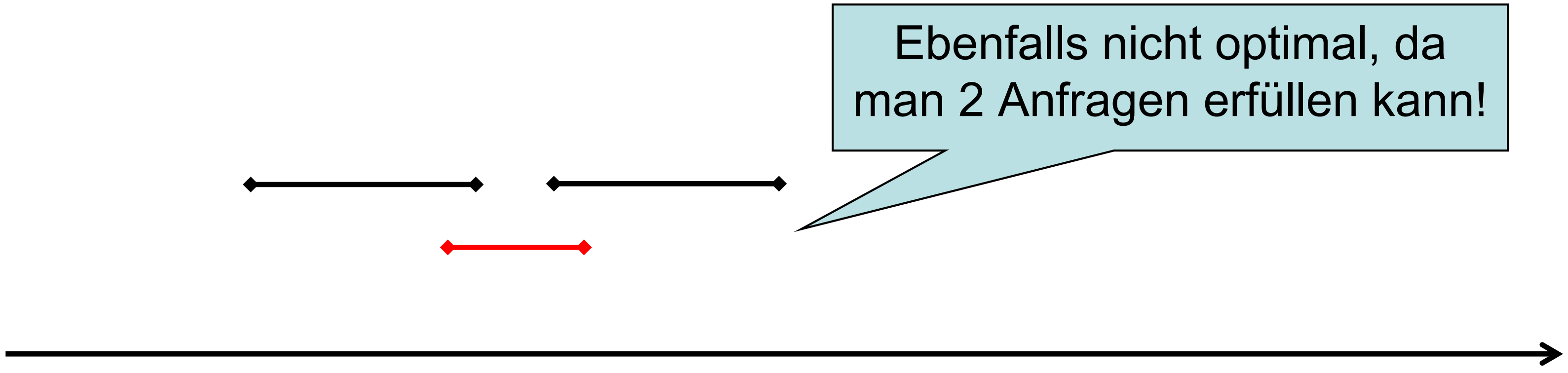


Gierige Algorithmen – Intervall Scheduling

Strategie 2:

- Wähle immer das kürzeste Intervall

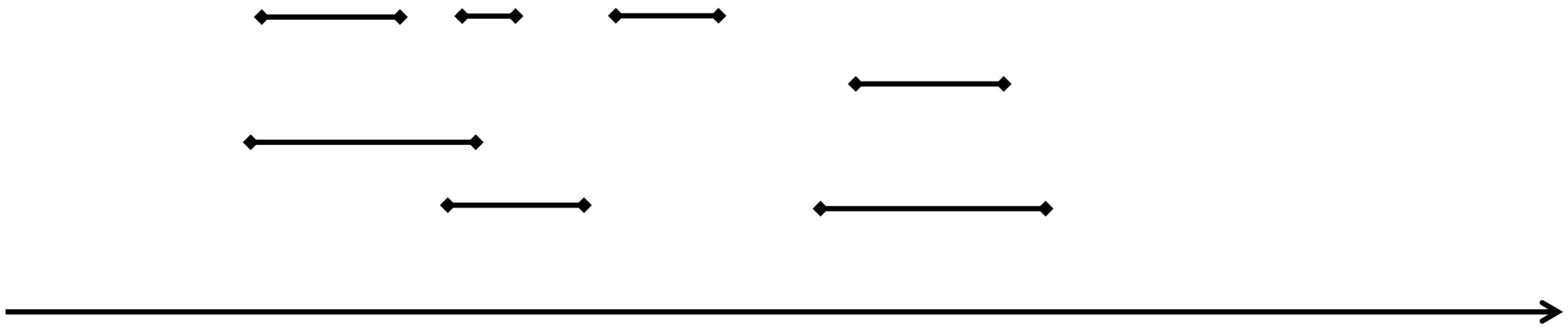
Optimalität?



Gierige Algorithmen – Intervall Scheduling

Strategie 3:

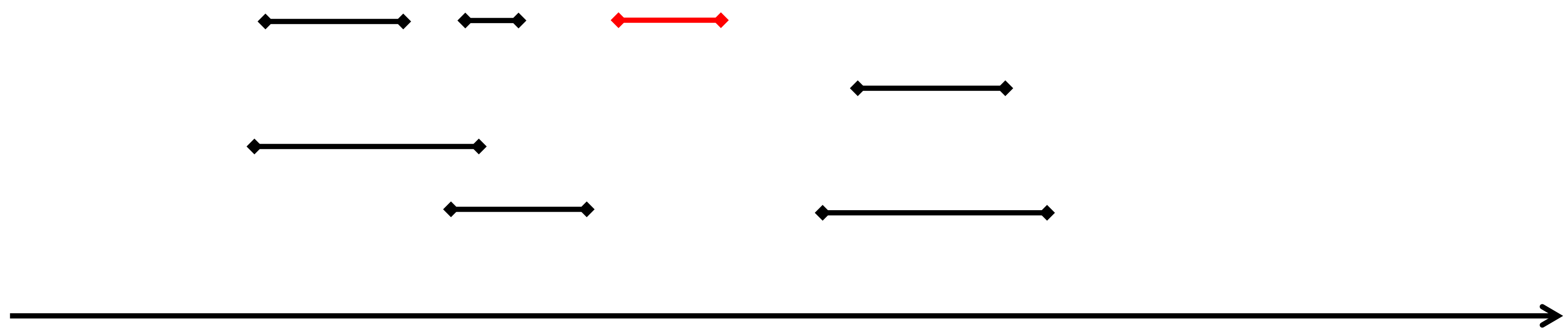
- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 3:

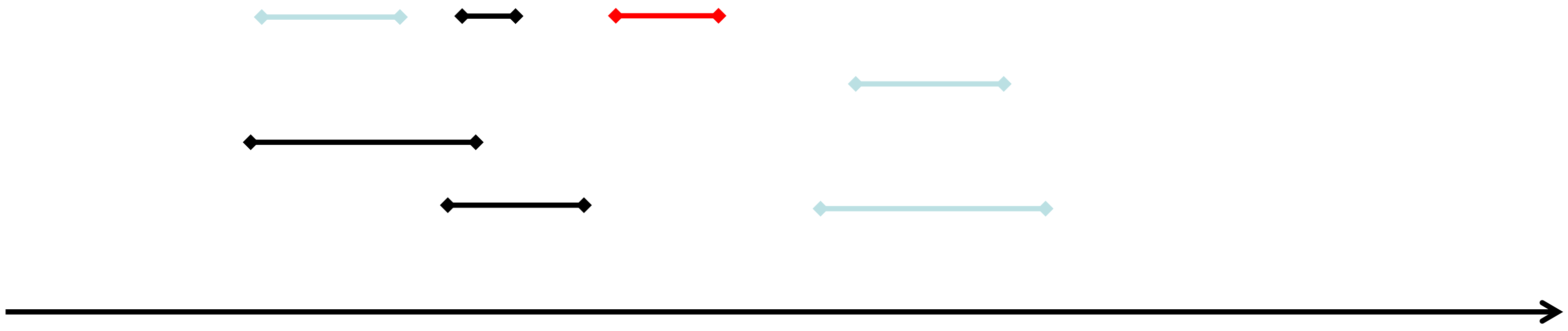
- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 3:

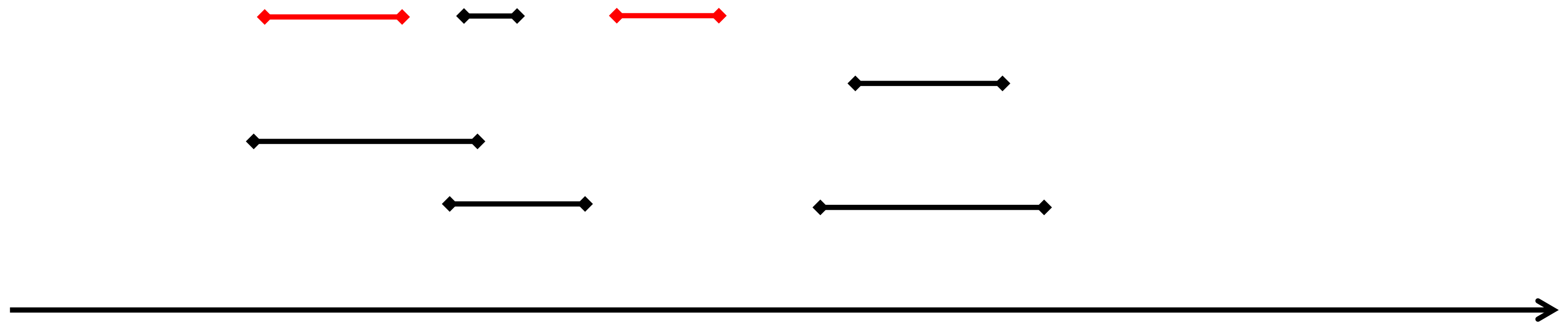
- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 3:

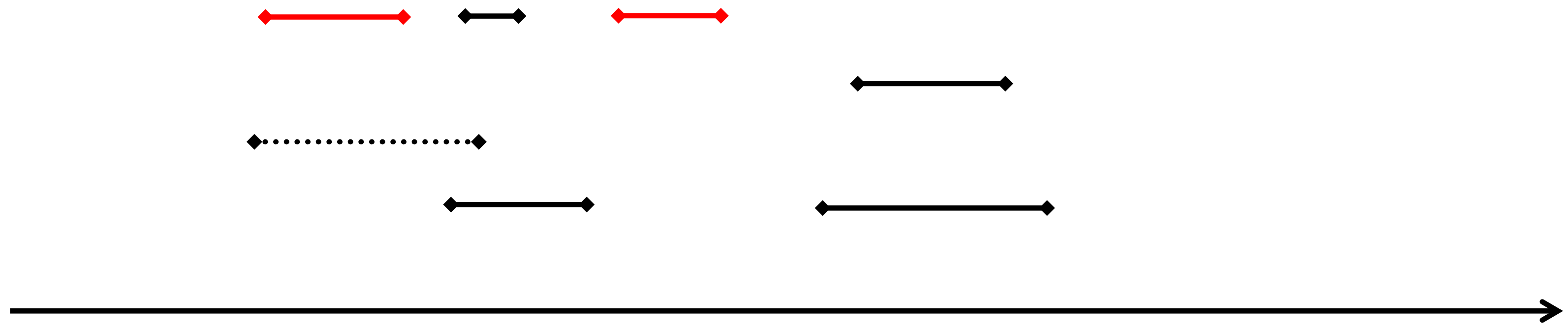
- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 3:

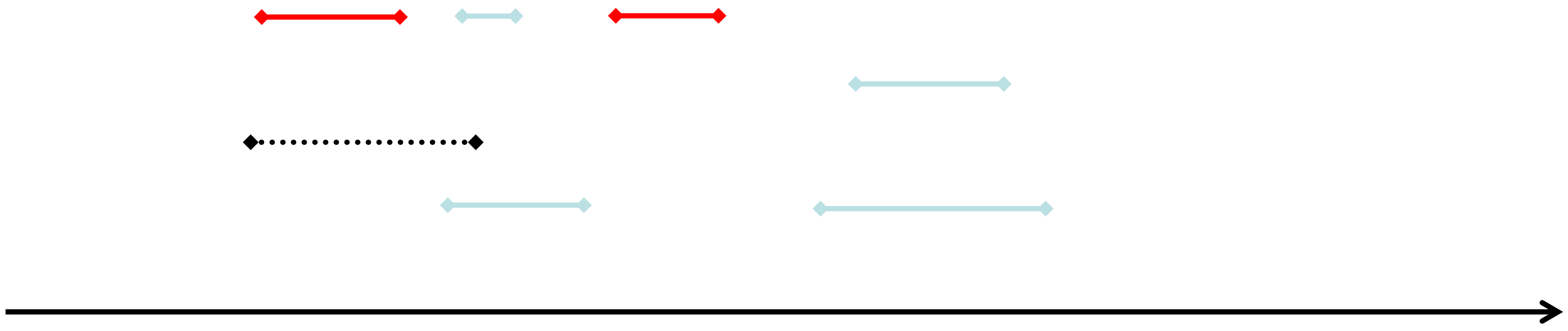
- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 3:

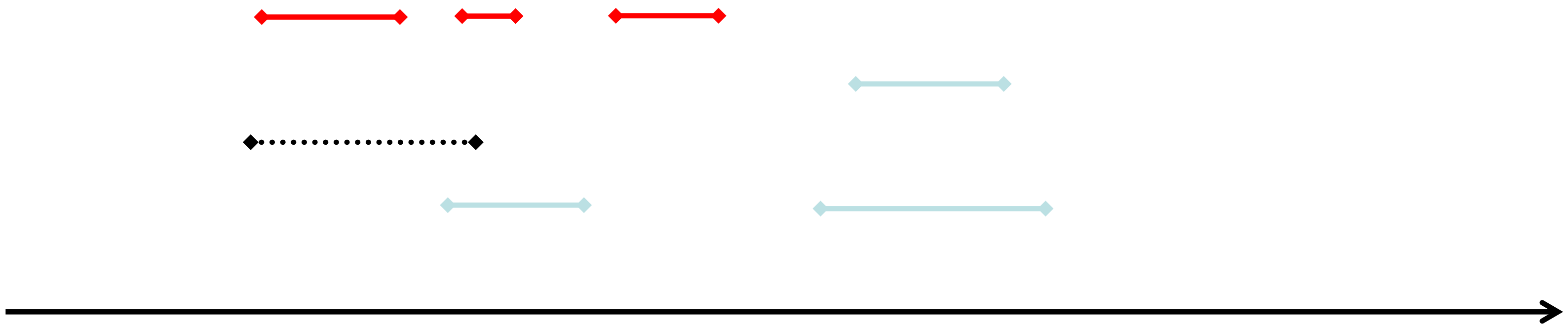
- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 3:

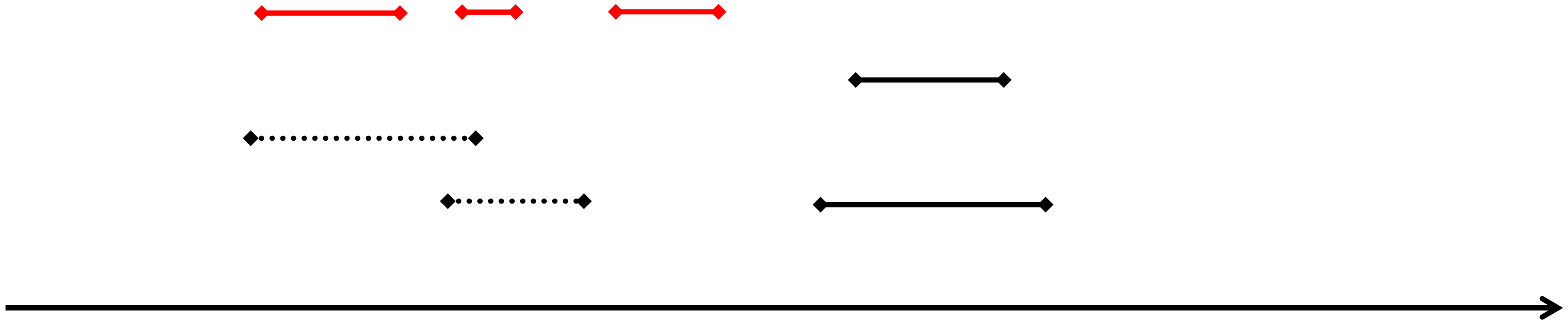
- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

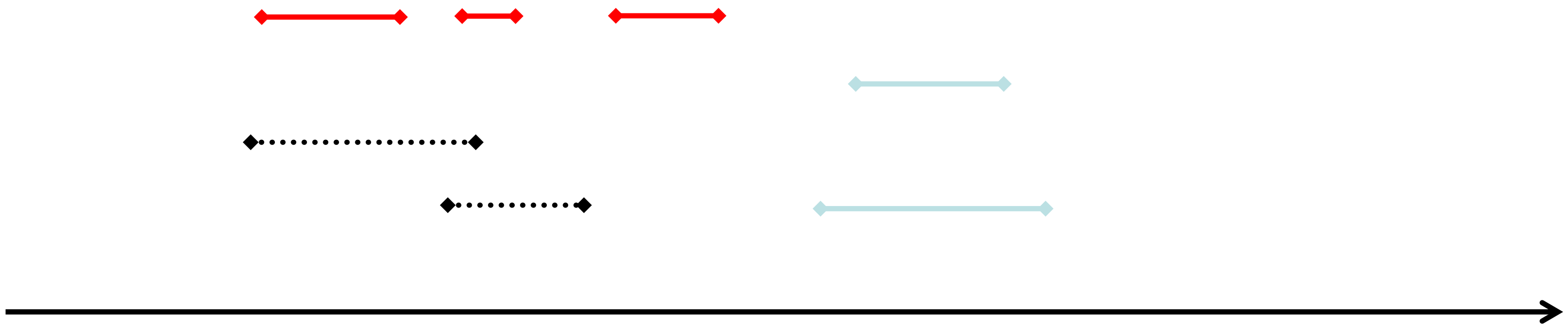
Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall



Strategie 3:

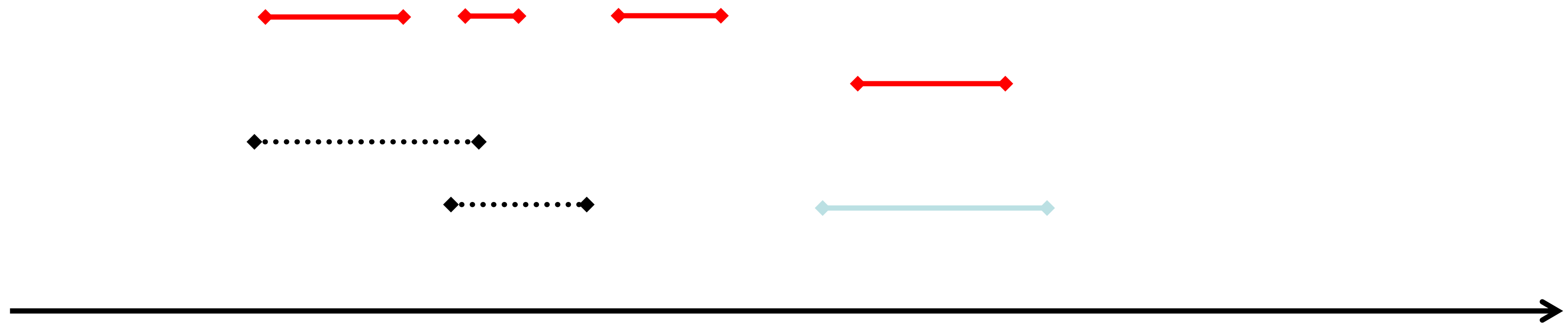
- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 3:

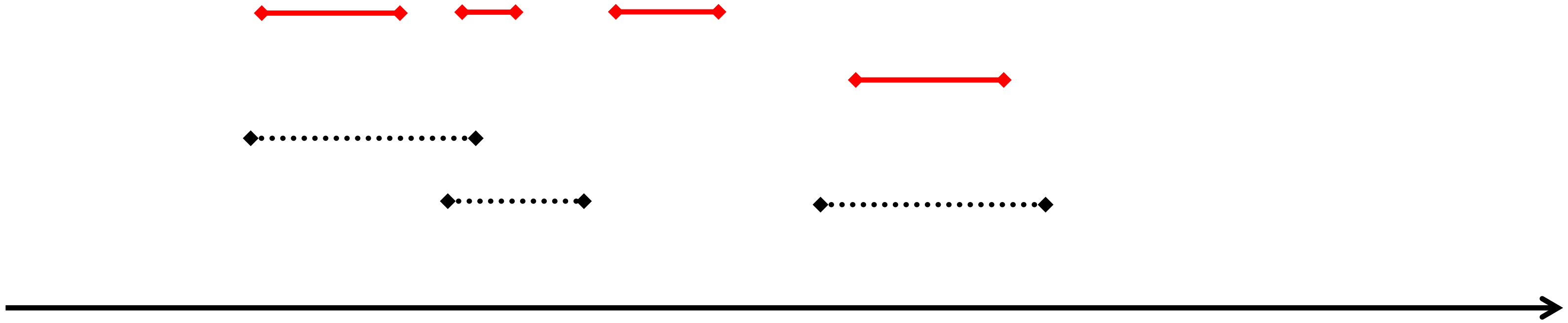
- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall



Gierige Algorithmen – Intervall Scheduling

Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall

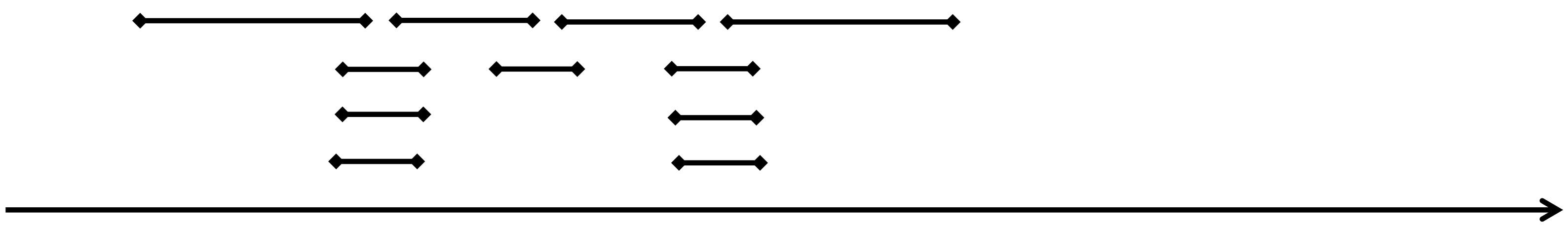


Gierige Algorithmen – Intervall Scheduling

Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall

Optimalität?

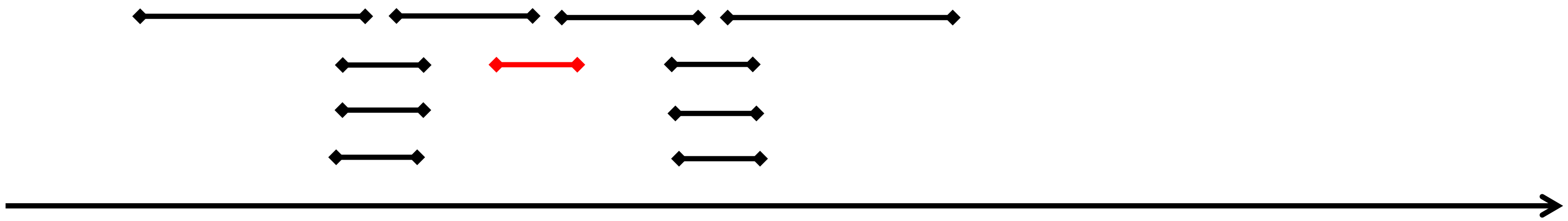


Gierige Algorithmen – Intervall Scheduling

Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall

Optimalität?

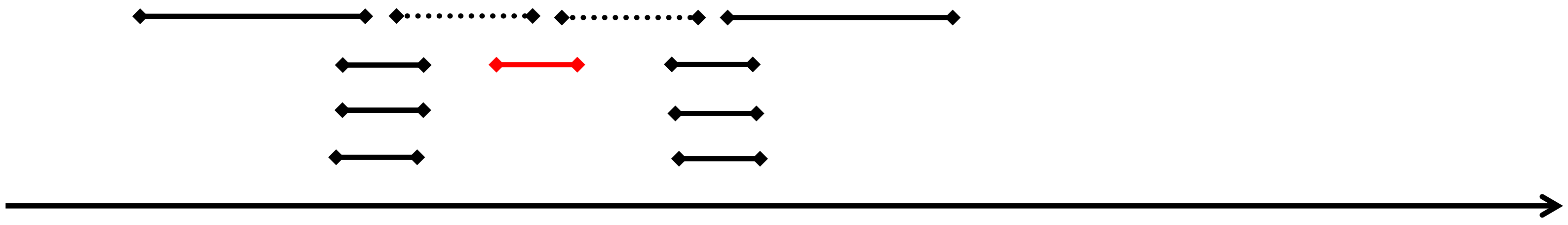


Gierige Algorithmen – Intervall Scheduling

Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall

Optimalität?

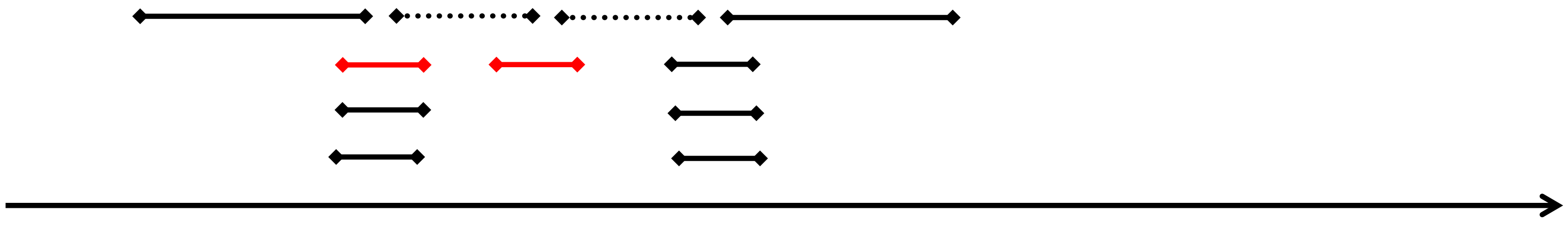


Gierige Algorithmen – Intervall Scheduling

Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall

Optimalität?

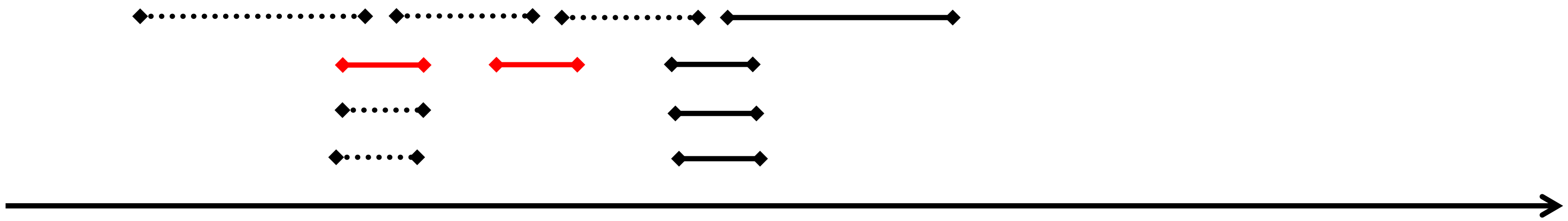


Gierige Algorithmen – Intervall Scheduling

Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall

Optimalität?

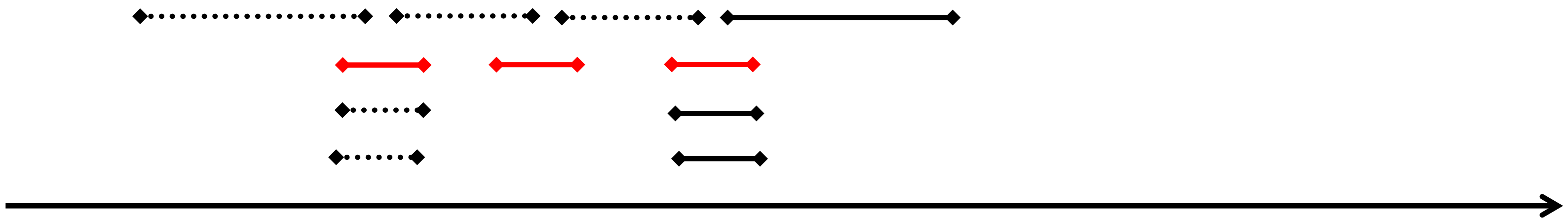


Gierige Algorithmen – Intervall Scheduling

Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall

Optimalität?

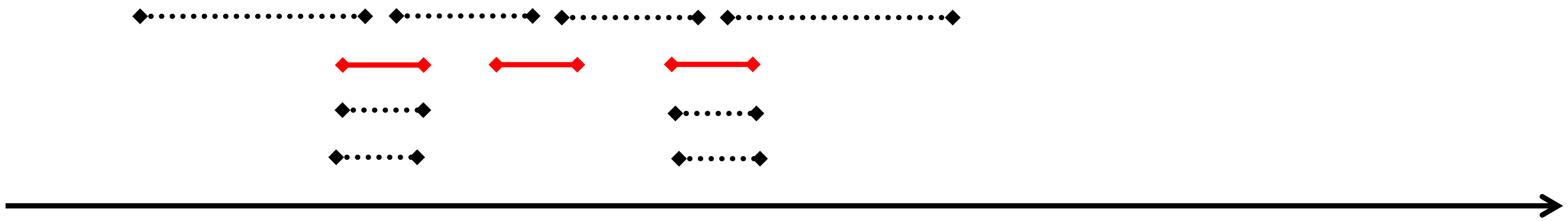


Gierige Algorithmen – Intervall Scheduling

Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall

Optimalität?

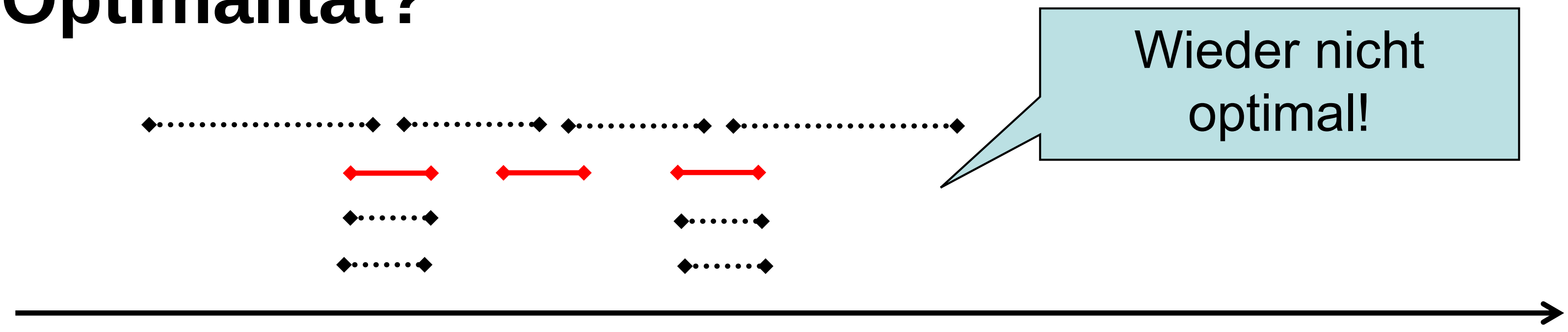


Gierige Algorithmen – Intervall Scheduling

Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall

Optimalität?



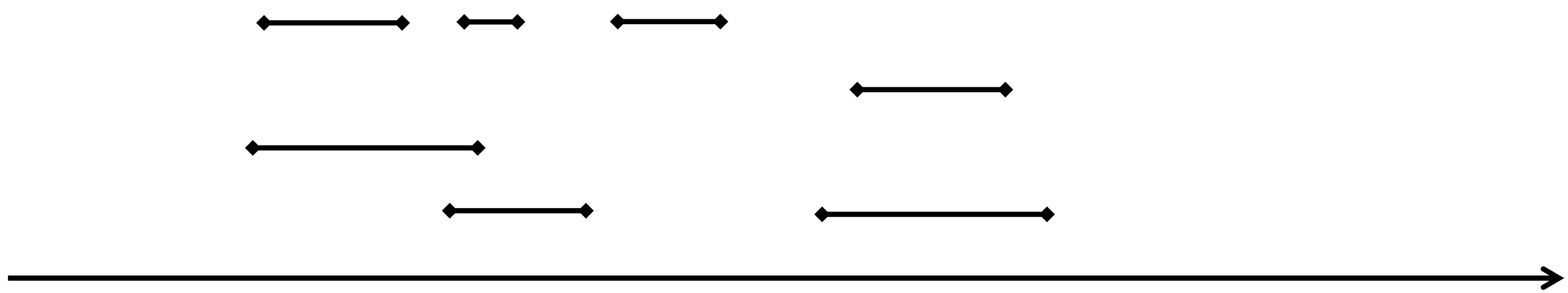
Gierige Algorithmen – Intervall Scheduling

Worauf muss man achten?

- Ressource muss möglichst früh wieder frei werden!

Neue Strategie:

- Nimm die Anfrage, die am frühesten fertig ist.



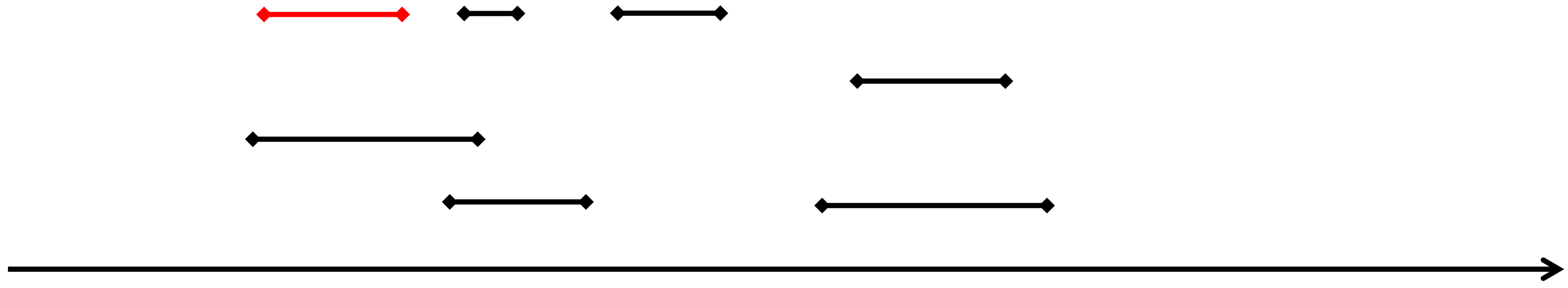
Gierige Algorithmen – Intervall Scheduling

Worauf muss man achten?

- Ressource muss möglichst früh wieder frei werden!

Neue Strategie:

- Nimm die Anfrage, die am frühesten fertig ist.



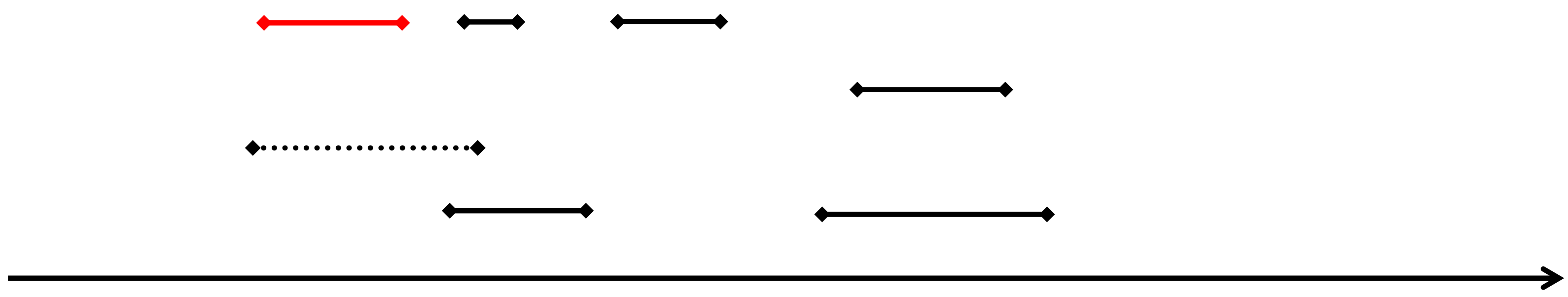
Gierige Algorithmen – Intervall Scheduling

Worauf muss man achten?

- Ressource muss möglichst früh wieder frei werden!

Neue Strategie:

- Nimm die Anfrage, die am frühesten fertig ist.



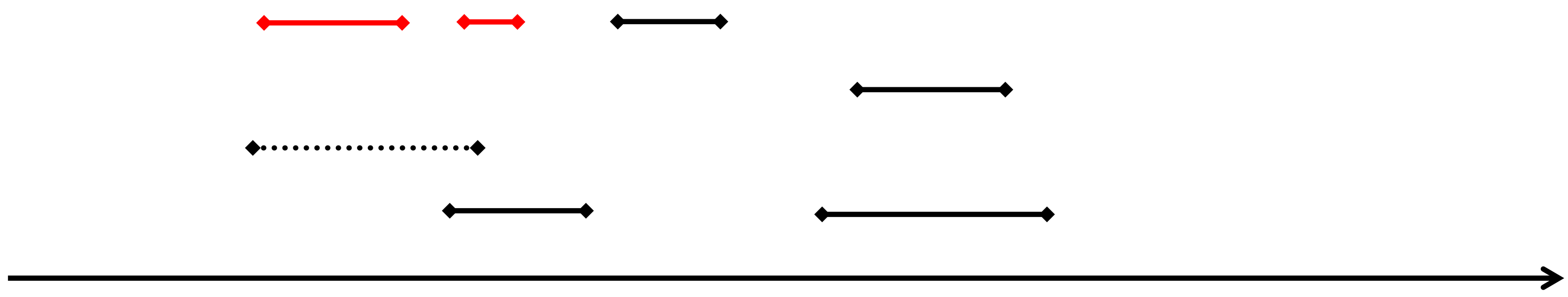
Gierige Algorithmen – Intervall Scheduling

Worauf muss man achten?

- Ressource muss möglichst früh wieder frei werden!

Neue Strategie:

- Nimm die Anfrage, die am frühesten fertig ist.



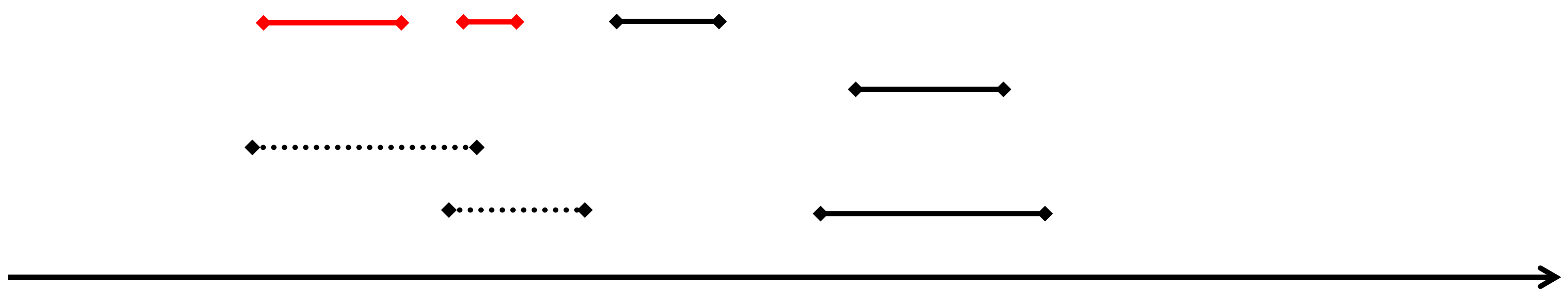
Gierige Algorithmen – Intervall Scheduling

Worauf muss man achten?

- Ressource muss möglichst früh wieder frei werden!

Neue Strategie:

- Nimm die Anfrage, die am frühesten fertig ist.



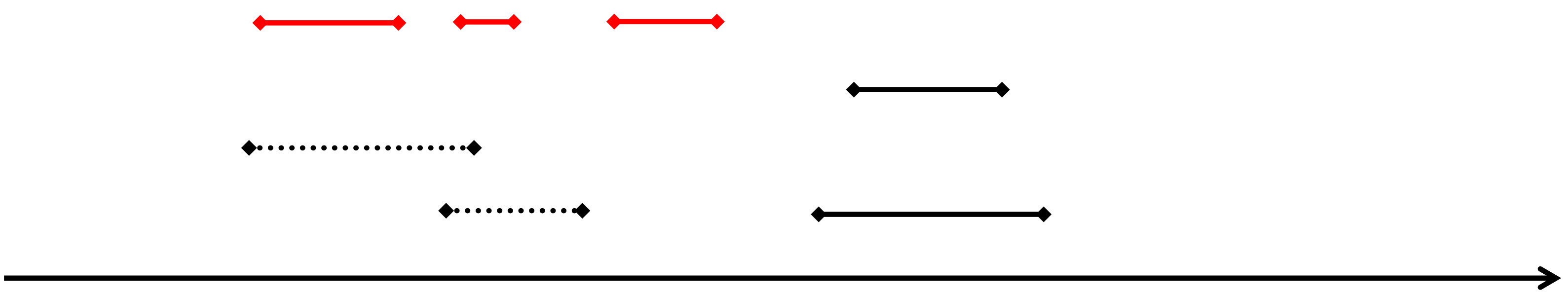
Gierige Algorithmen – Intervall Scheduling

Worauf muss man achten?

- Ressource muss möglichst früh wieder frei werden!

Neue Strategie:

- Nimm die Anfrage, die am frühesten fertig ist.

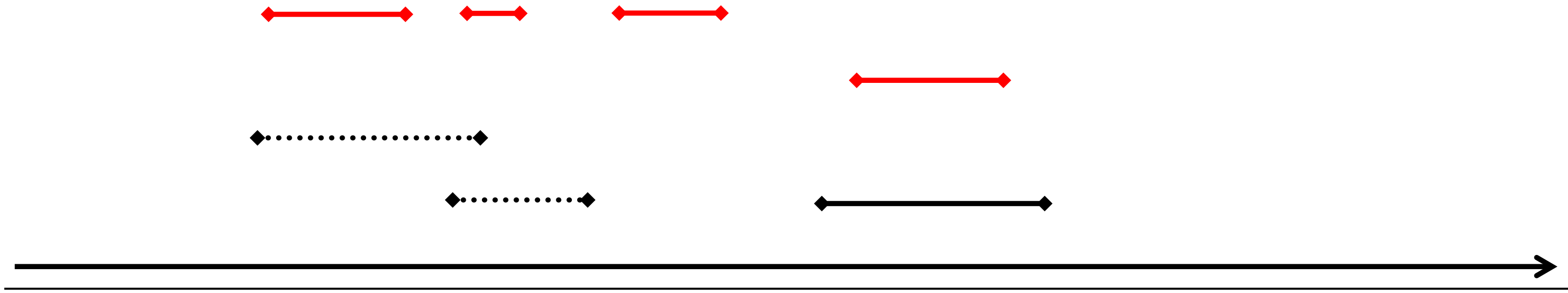


Worauf muss man achten?

- Ressource muss möglichst früh wieder frei werden!

Neue Strategie:

- Nimm die Anfrage, die am frühesten fertig ist.



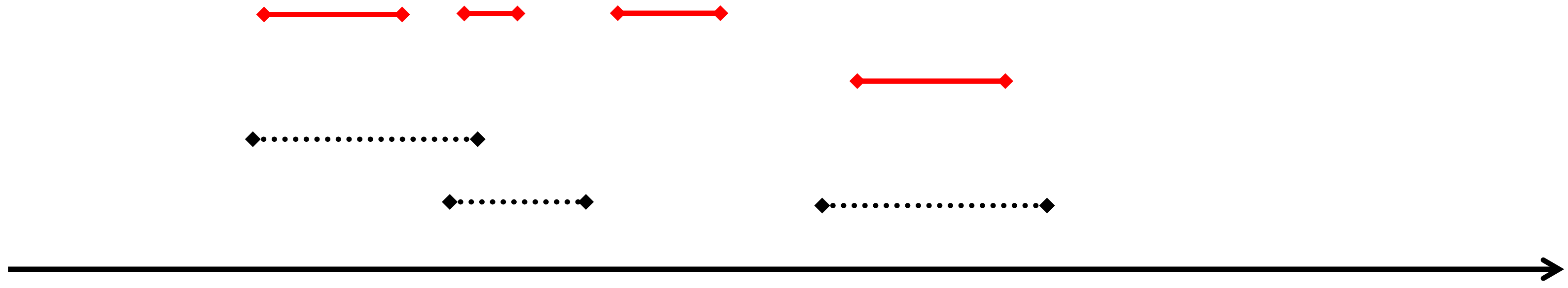
Gierige Algorithmen – Intervall Scheduling

Worauf muss man achten?

- Ressource muss möglichst früh wieder frei werden!

Neue Strategie:

- Nimm die Anfrage, die am frühesten fertig ist.



Gierige Algorithmen – Intervall Scheduling

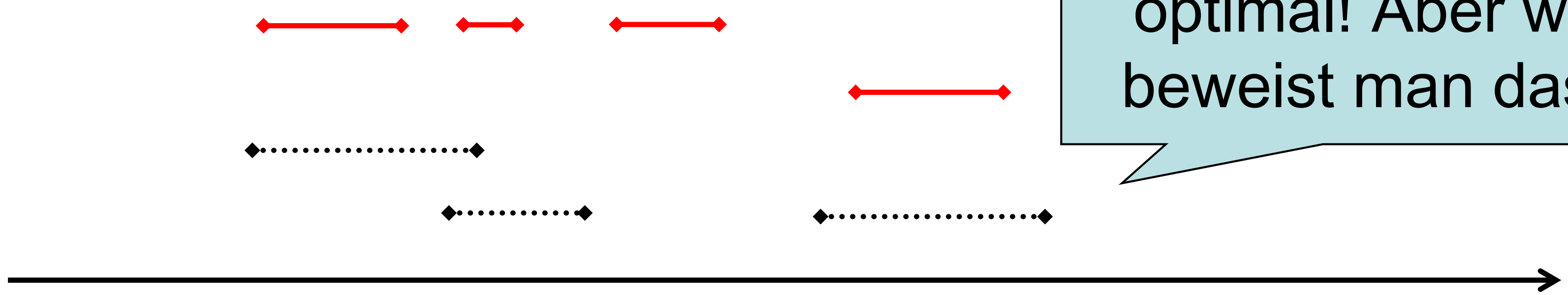
Worauf muss man achten?

- Ressource muss möglichst früh wieder frei werden!

Neue Strategie:

- Nimm die Anfrage, die am frühesten fertig ist.

Diese Strategie ist optimal! Aber wie beweist man das?



Gierige Algorithmen – Intervall Scheduling

Formale Problemformulierung:

- Problem: Intervall Scheduling
- Eingabe: Felder s und f, die die Intervalle [s[i], f[i]] beschreiben
- Ausgabe: Indizes der ausgewählten Intervalle

o.B.d.A. nehmen wir an:

- Eingabe ist nach Intervallendpunkten sortiert,

  d.h.: $f[1] \leq f[2] \leq \ldots \leq f[n]$

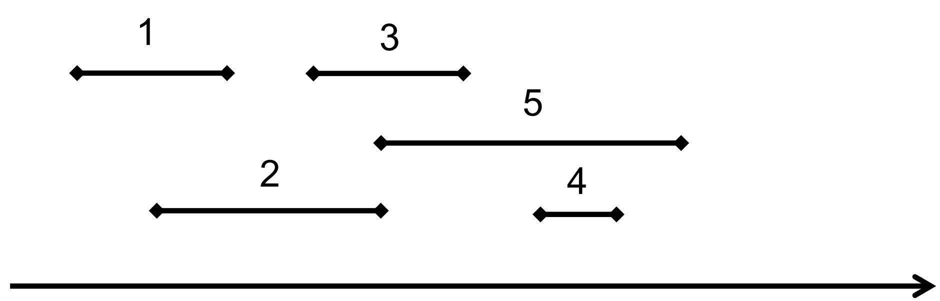
Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)

1. n $\leftarrow$ length[s]
2. A $\leftarrow$ {1}
3. j $\leftarrow$ 1
4. **for** i $\leftarrow$ 2 **to** n **do**
5.   **if** s[i] $\geq$ f[j] **then**
6.   A $\leftarrow$ A $\cup$ {i}
7.   j $\leftarrow$ i

7. **return** A

s	1	2	4	7	5
f	3	5	6	8	9

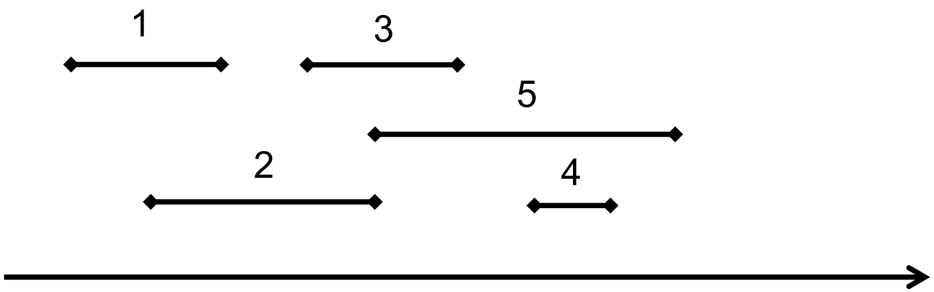


Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)

1. $n \leftarrow$ length[s]
2. $A \leftarrow \{1\}$
3. $j \leftarrow 1$
4. **for** $i \leftarrow 2$ **to** n **do**
5.   **if** $s[i] \geq f[j]$ **then**
6. $A \leftarrow A \cup \{i\}$
7. $j \leftarrow i$
8. **return** A

s	1	2	4	7	5
f	3	5	6	8	9

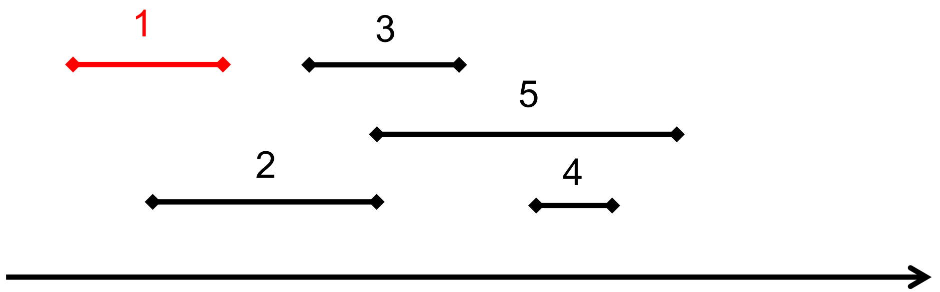


Gierige Algorithmen – Intervall Scheduling

s	1	2	4	7	5
f	3	5	6	8	9

IntervalScheduling(s,f)

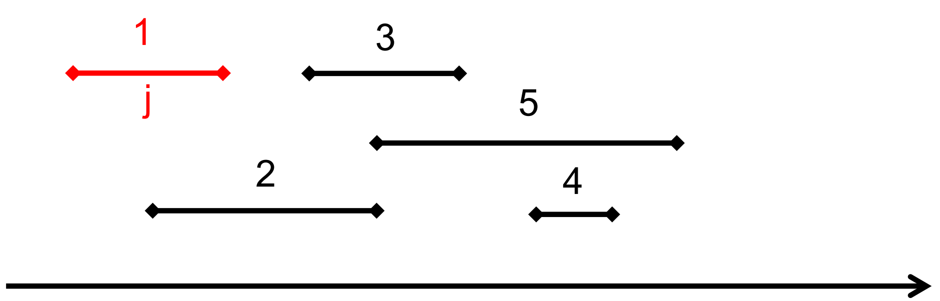
1. n ← length[s]
2. A ← {1}
3. j ← 1
4. **for** i ← 2 **to** n **do**
5.   **if** s[i] ≥ f[j] **then**
6.   A ← A ∪ {i}
7.   j ← i
8. **return** A



Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)
1. n ← length[s]
2. A ← {1}
3. j ← 1
4. **for** i ← 2 **to** n **do**
5.   **if** s[i] ≥ f[j] **then**
6.   A ← A ∪ {i}
7.   j ← i
8. **return** A

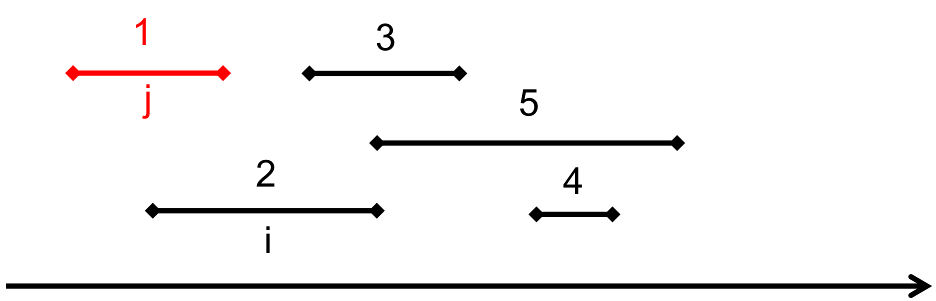
s	1	2	4	7	5
f	3	5	6	8	9



Gierige Algorithmen – Intervall Scheduling

```
IntervalScheduling(s,f)
n ← length[s]
A ← {1}
j ← 1
for i ← 2 to n do
    if s[i] ≥ f[j] then
        A ← A ∪ {i}
        j ← i
return A
```

s	1	2	4	7	5
f	3	5	6	8	9

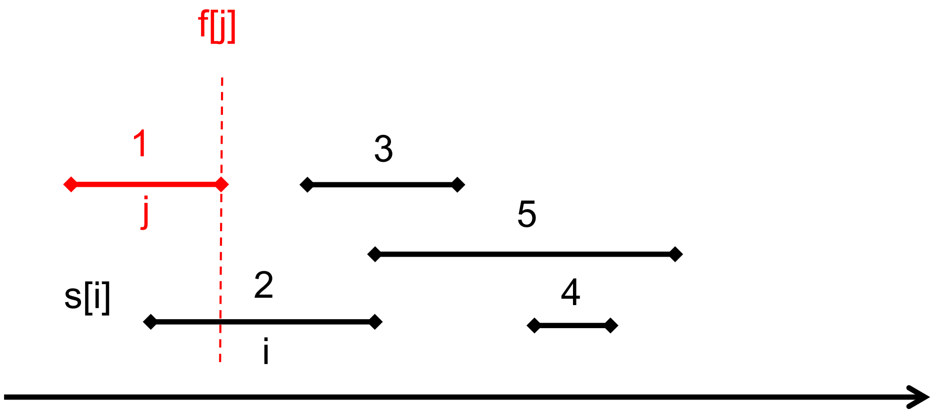


Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)

1. n ← length[s]
2. A ← {1}
3. j ← 1
4. **for** i ← 2 **to** n **do**
5. **if** s[i] ≥ f[j] **then**
6. A ← A ∪ {i}
7. j ← i
8. **return** A

s	1	2	4	7	5
f	3	5	6	8	9

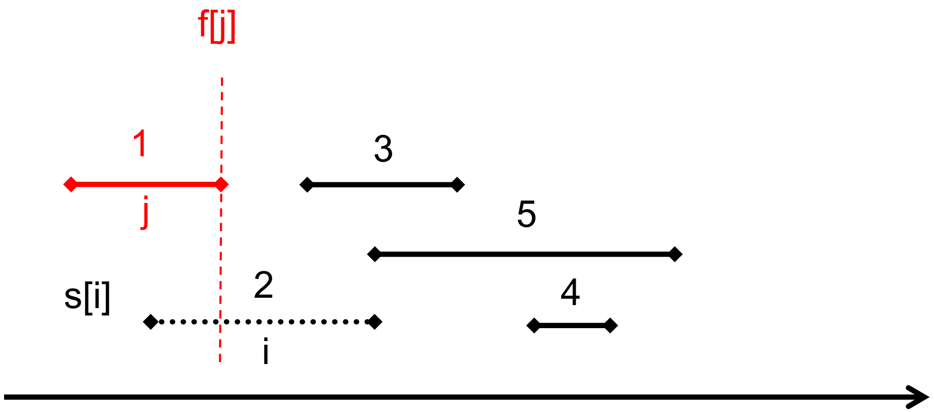


Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)

1. n ← length[s]
2. A ← {1}
3. j ← 1
4. **for** i ← 2 **to** n **do**
5.   **if** s[i] ≥ f[j] **then**
6.   A ← A ∪ {i}
7.   j ← i
8. **return** A

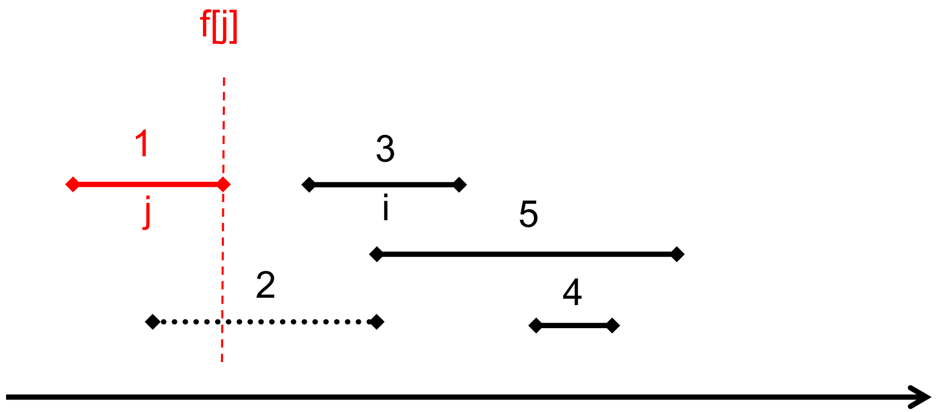
s	1	2	4	7	5
f	3	5	6	8	9



Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)
1. n ← length[s]
2. A ← {1}
3. j ← 1
4. **for** i ← 2 **to** n **do**
5.   **if** s[i] ≥ f[j] **then**
6.   A ← A ∪ {i}
7.   j ← i
8. **return** A

s	1	2	4	7	5
f	3	5	6	8	9

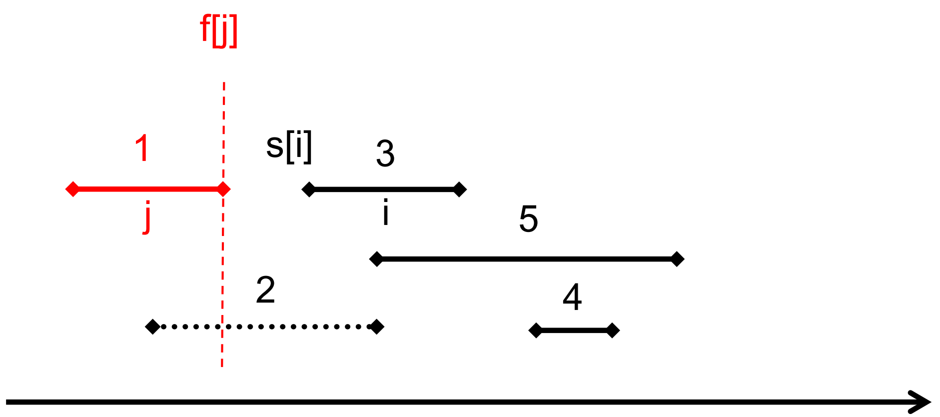


Gierige Algorithmen – Intervall Scheduling

s	1	2	4	7	5
f	3	5	6	8	9

IntervalScheduling(s,f)

1. n $\leftarrow$ length[s]
2. A $\leftarrow$ {1}
3. j $\leftarrow$ 1
4. **for** i $\leftarrow$ 2 **to** n **do**
5. **if** s[i] $\geq$ f[j] **then**
6. A $\leftarrow$ A $\cup$ {i}
7. j $\leftarrow$ i
8. **return** A

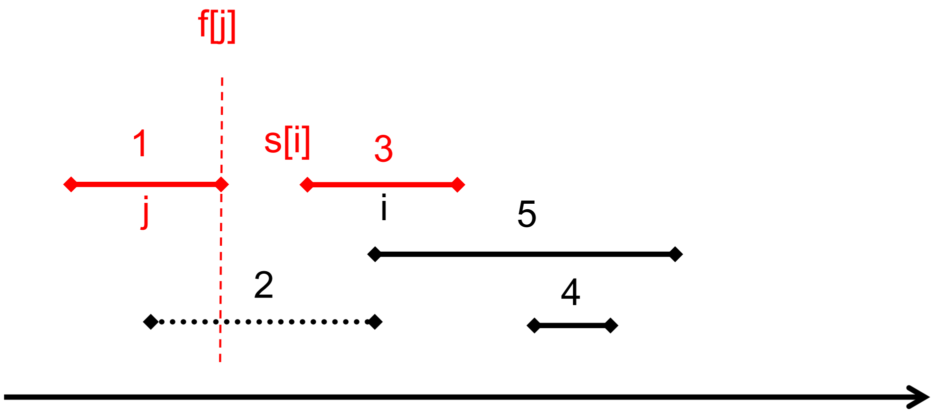


Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)

1. n $\leftarrow$ length[s]
2. A $\leftarrow$ {1}
3. j $\leftarrow$ 1
4. **for** i $\leftarrow$ 2 **to** n **do**
5.   **if** s[i] $\geq$ f[j] **then**
6.   A $\leftarrow$ A $\cup$ {i}
7.   j $\leftarrow$ i
8. **return** A

s	1	2	4	7	5
f	3	5	6	8	9

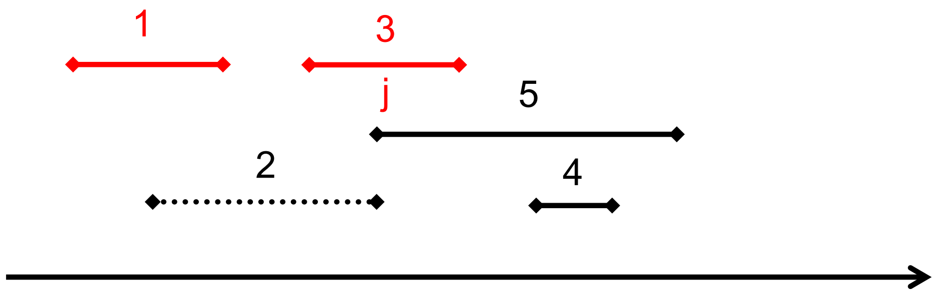


Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)

1. n $\leftarrow$ length[s]
2. A $\leftarrow$ {1}
3. j $\leftarrow$ 1
4. **for** i $\leftarrow$ 2 **to** n **do**
5.   **if** s[i] $\geq$ f[j] **then**
6.   A $\leftarrow$ A $\cup$ {i}
7.   j $\leftarrow$ i
8. **return** A

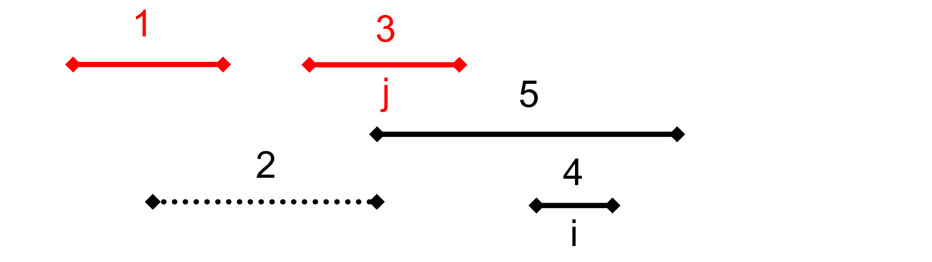
s	1	2	4	7	5
f	3	5	6	8	9



Gierige Algorithmen – Intervall Scheduling

s	1	2	4	7	5
f	3	5	6	8	9

IntervalScheduling(s,f)
1. n ← length[s]
2. A ← {1}
3. j ← 1
4. **for** i ← 2 **to** n **do**
5.   **if** s[i] ≥ f[j] **then**
6.   A ← A ∪ {i}
7.   j ← i
8. **return** A

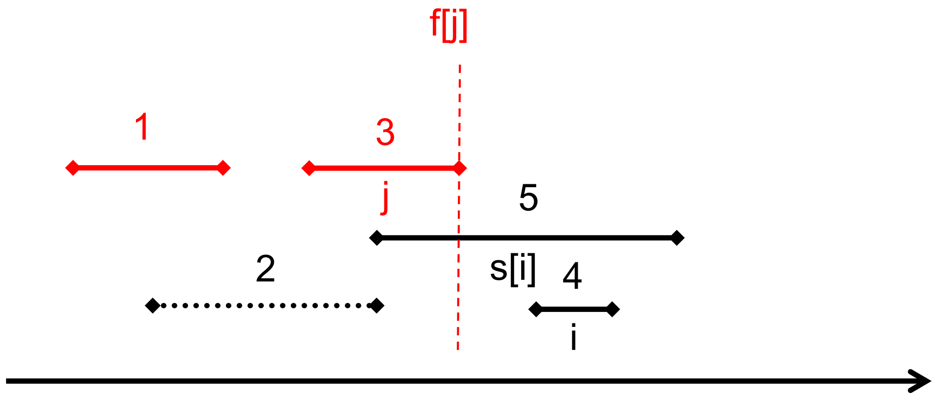


Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)

1. n ← length[s]
2. A ← {1}
3. j ← 1
4. **for** i ← 2 **to** n **do**
5. **if** s[i] ≥ f[j] **then**
6. A ← A ∪ {i}
7. j ← i
8. **return** A

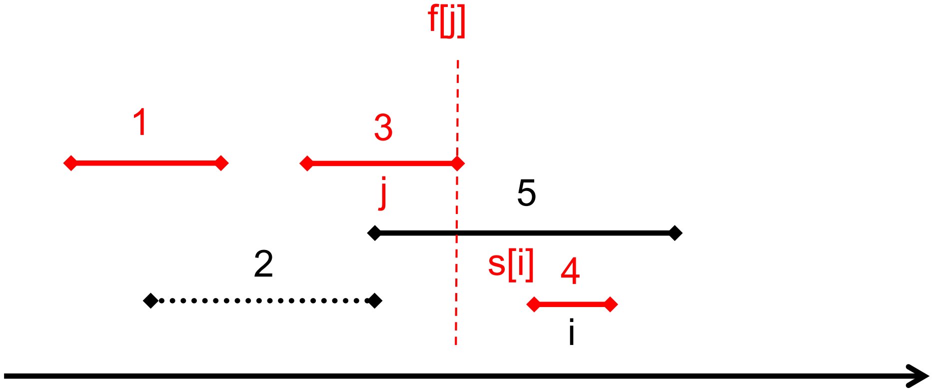
s	1	2	4	7	5
f	3	5	6	8	9



Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)
1. n ← length[s]
2. A ← {1}
3. j ← 1
4. **for** i ← 2 **to** n **do**
5.   **if** s[i] ≥ f[j] **then**
6.   A ← A ∪ {i}
7.   j ← i
8. **return** A

s	1	2	4	7	5
f	3	5	6	8	9

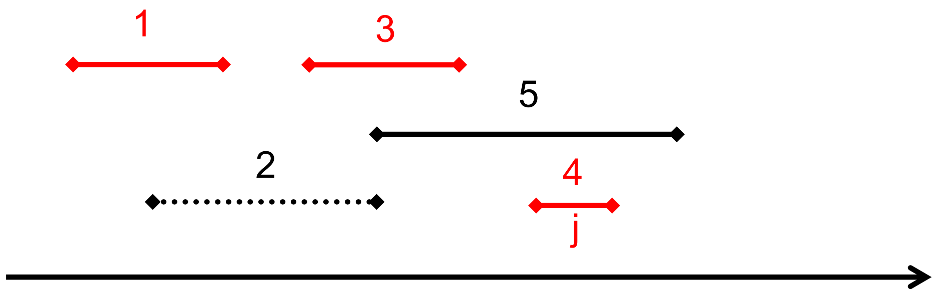


Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)

1. n $\leftarrow$ length[s]
2. A $\leftarrow$ {1}
3. j $\leftarrow$ 1
4. **for** i $\leftarrow$ 2 **to** n **do**
5.   **if** s[i] $\geq$ f[j] **then**
6.   A $\leftarrow$ A $\cup$ {i}
7.   j $\leftarrow$ i
8. **return** A

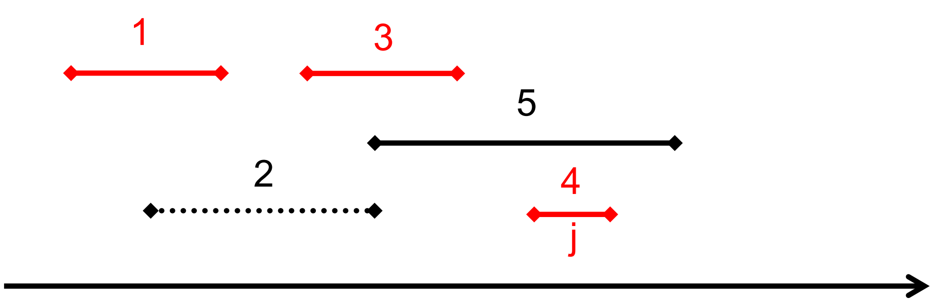
s	1	2	4	7	5
f	3	5	6	8	9



Gierige Algorithmen – Intervall Scheduling

s	1	2	4	7	5
f	3	5	6	8	9

```
IntervalScheduling(s,f)
n ← length[s]
A ← {1}
j ← 1
for i ← 2 to n do
    if s[i] ≥ f[j] then
        A ← A ∪ {i}
        j ← i
return A
```

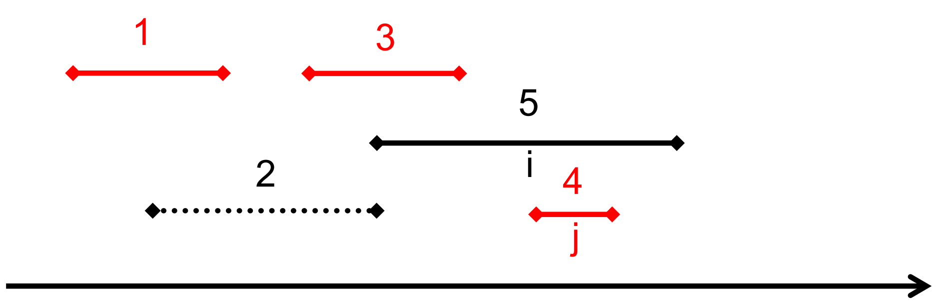


Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)

1. n ← length[s]
2. A ← {1}
3. j ← 1
4. **for** i ← 2 **to** n **do**
5.   **if** s[i] ≥ f[j] **then**
6.   A ← A ∪ {i}
7.   j ← i
8. **return** A

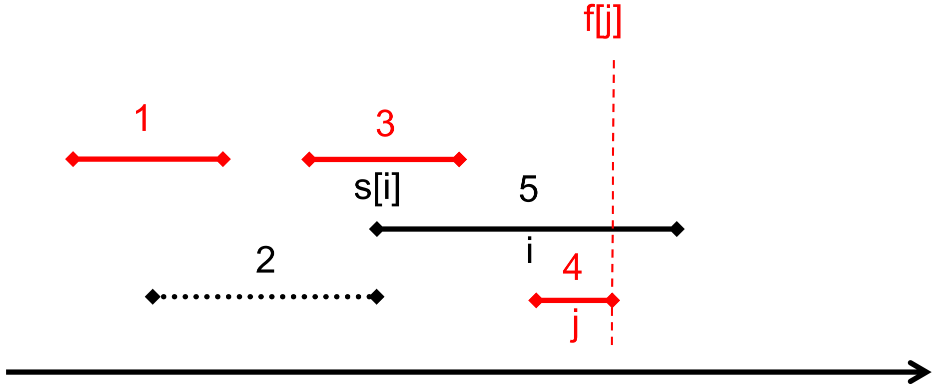
s	1	2	4	7	5
f	3	5	6	8	9



Gierige Algorithmen – Intervall Scheduling

s	1	2	4	7	5
f	3	5	6	8	9

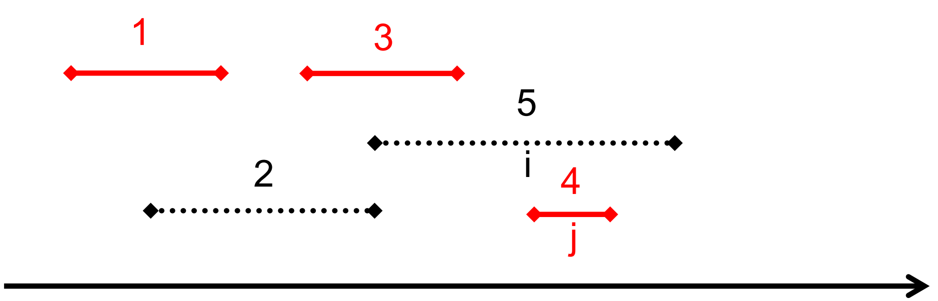
IntervalScheduling(s,f)
1. n ← length[s]
2. A ← {1}
3. j ← 1
4. **for** i ← 2 **to** n **do**
5.   **if** s[i] ≥ f[j] **then**
6.   A ← A ∪ {i}
7.   j ← i
8. **return** A



Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)
1. n ← length[s]
2. A ← {1}
3. j ← 1
4. **for** i ← 2 **to** n **do**
5. **if** s[i] ≥ f[j] **then**
6. A ← A ∪ {i}
7. j ← i
8. **return** A

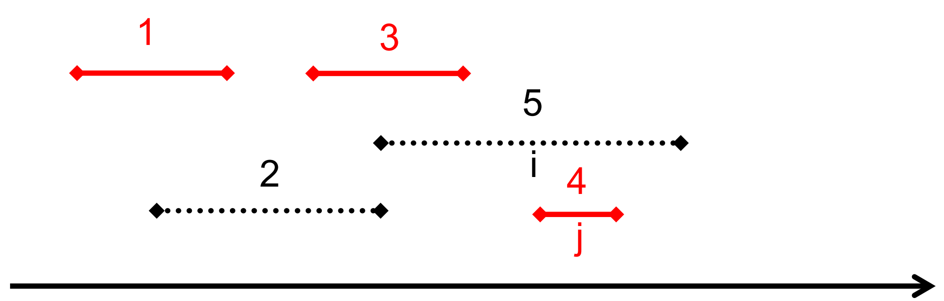
s	1	2	4	7	5
f	3	5	6	8	9



Gierige Algorithmen – Intervall Scheduling

s	1	2	4	7	5
f	3	5	6	8	9

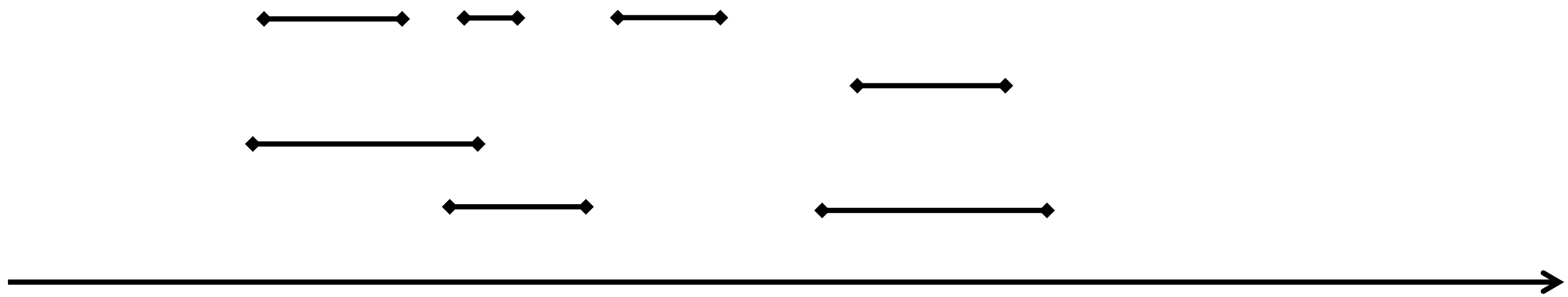
```
IntervalScheduling(s,f)
n ← length[s]
A ← {1}
j ← 1
for i ← 2 to n do
    if s[i] ≥ f[j] then
        A ← A ∪ {i}
        j ← i
return A
```



Gierige Algorithmen – Intervall Scheduling

Wie können wir Optimalität zeigen?

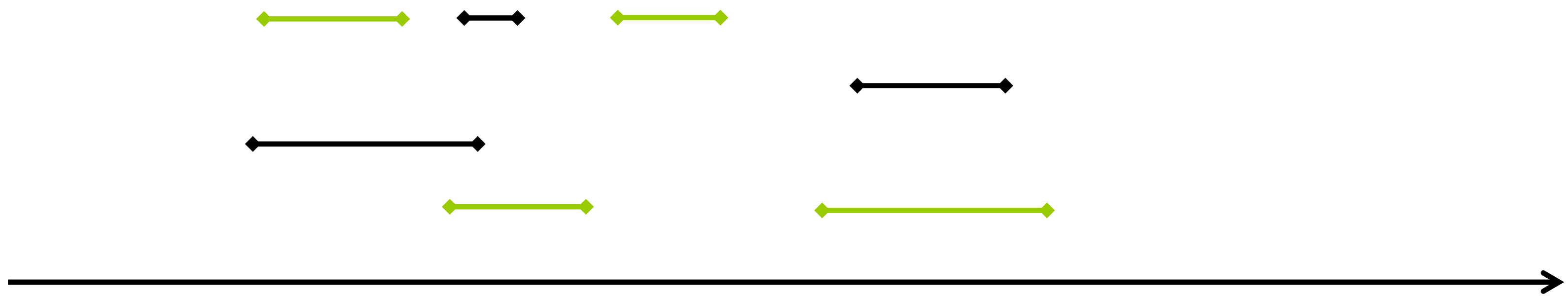
- Sei O optimale Menge von Intervallen
- Können wir A = O zeigen ?



Gierige Algorithmen – Intervall Scheduling

Wie können wir Optimalität zeigen?

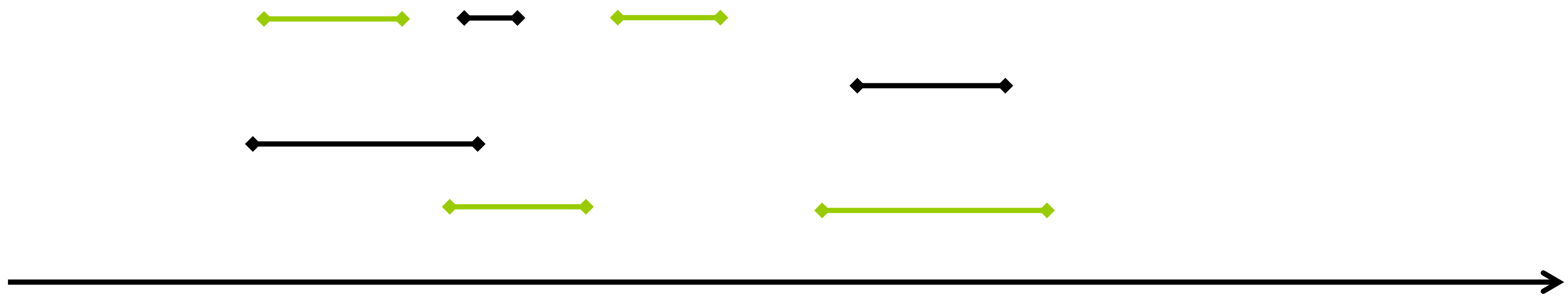
- Sei O optimale Menge von Intervallen
- Können wir A = O zeigen ?



Gierige Algorithmen – Intervall Scheduling

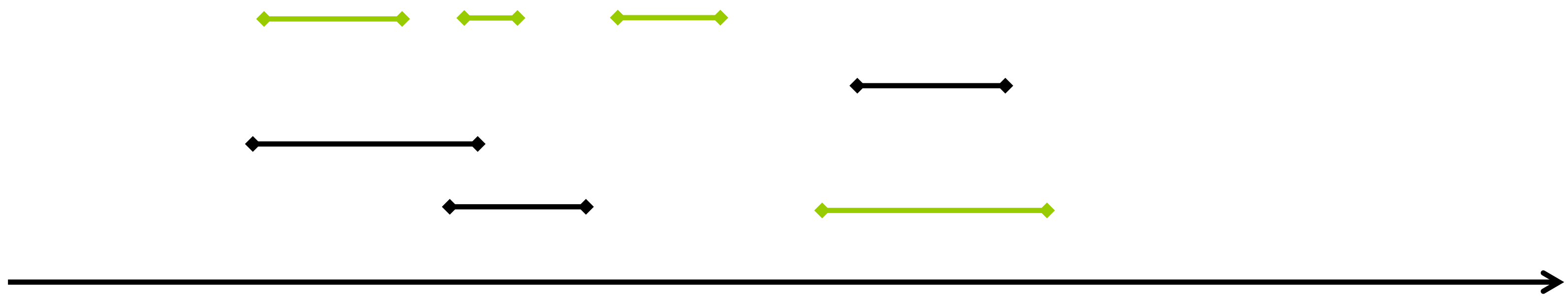
Wie können wir Optimalität zeigen?

- Sei O optimale Menge von Intervallen
- u. U. viele optimale Lösungen



Wie können wir Optimalität zeigen?

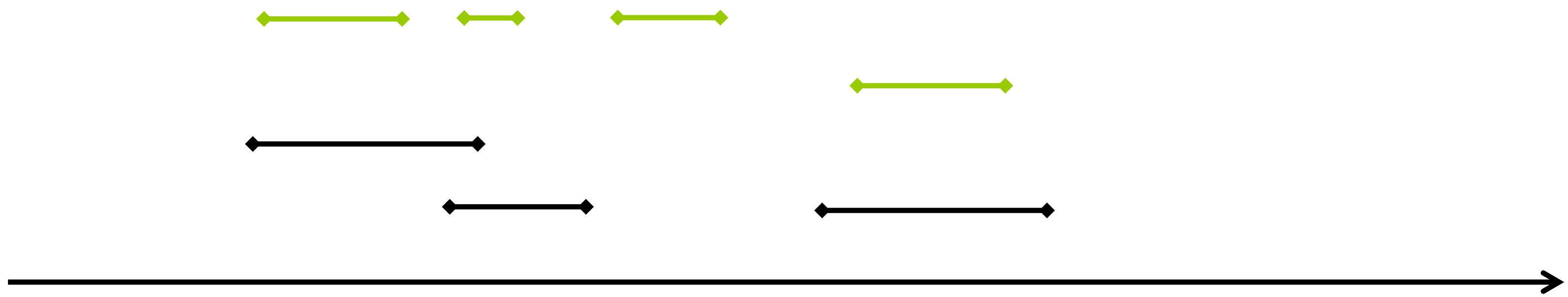
- Sei O optimale Menge von Intervallen
- u. U. viele optimale Lösungen



Gierige Algorithmen – Intervall Scheduling

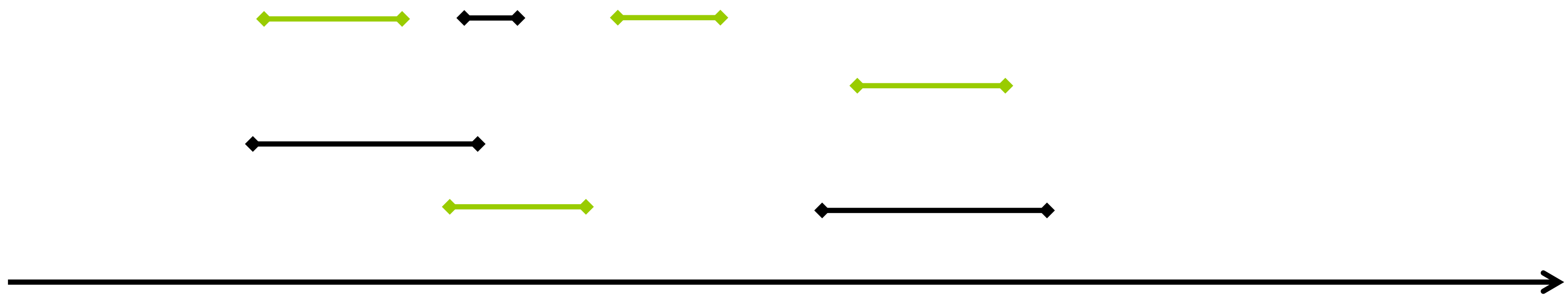
Wie können wir Optimalität zeigen?

- Sei O optimale Menge von Intervallen
- u. U. viele optimale Lösungen



Wie können wir Optimalität zeigen?

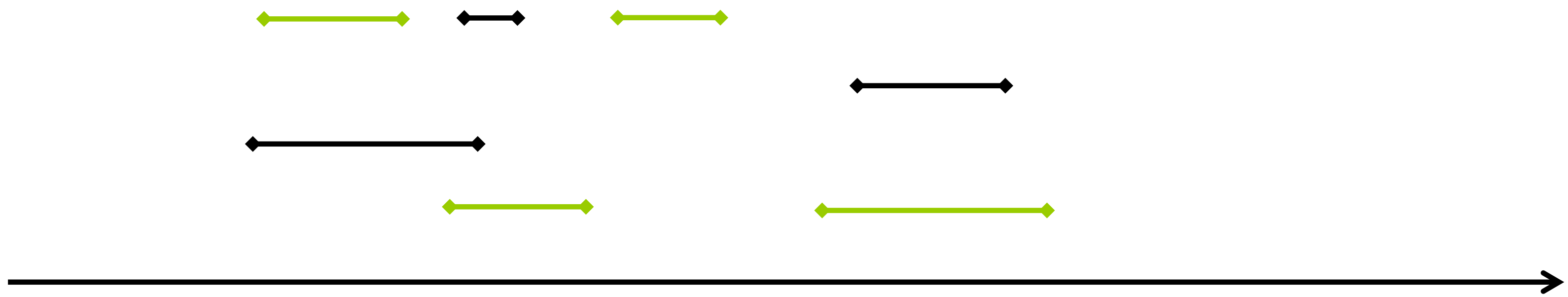
- Sei O optimale Menge von Intervallen
- u. U. viele optimale Lösungen



Gierige Algorithmen – Intervall Scheduling

Wie können wir Optimalität zeigen?

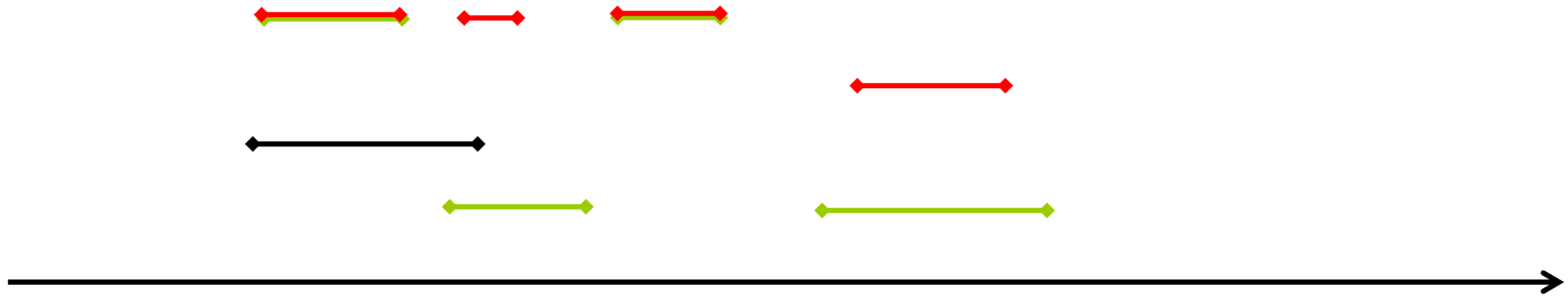
- Sei O optimale Menge von Intervallen
- u. U. viele optimale Lösungen



Wie können wir Optimalität zeigen?

- Sei O optimale Menge von Intervallen
- u. U. viele optimale Lösungen

  $\Rightarrow$ Wir zeigen: |A| = |O|



Beobachtung:

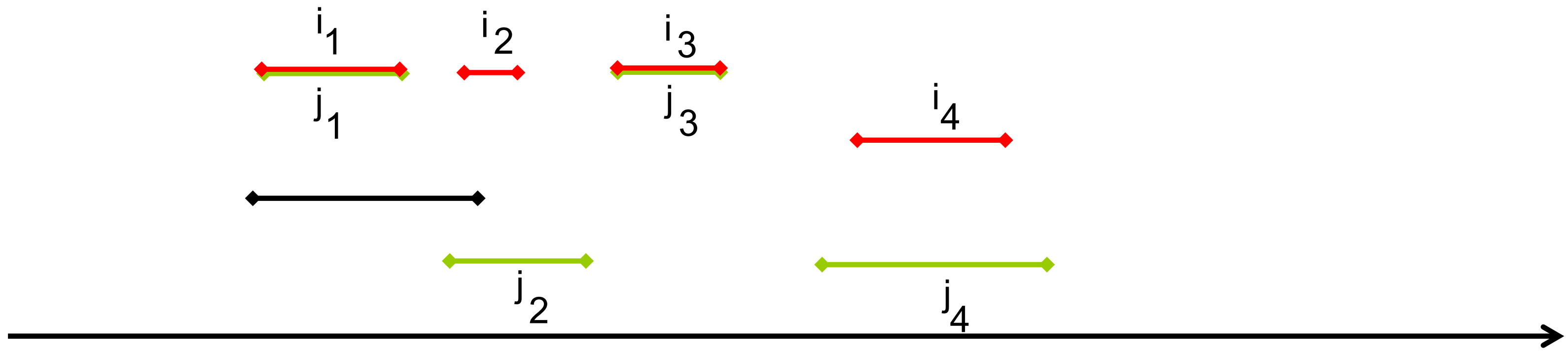
A ist eine Menge von kompatiblen Anfragen,

d.h. die Menge A bildet eine zulässige bzw. gültige Lösung.

Gierige Algorithmen – Intervall Scheduling

Notation:

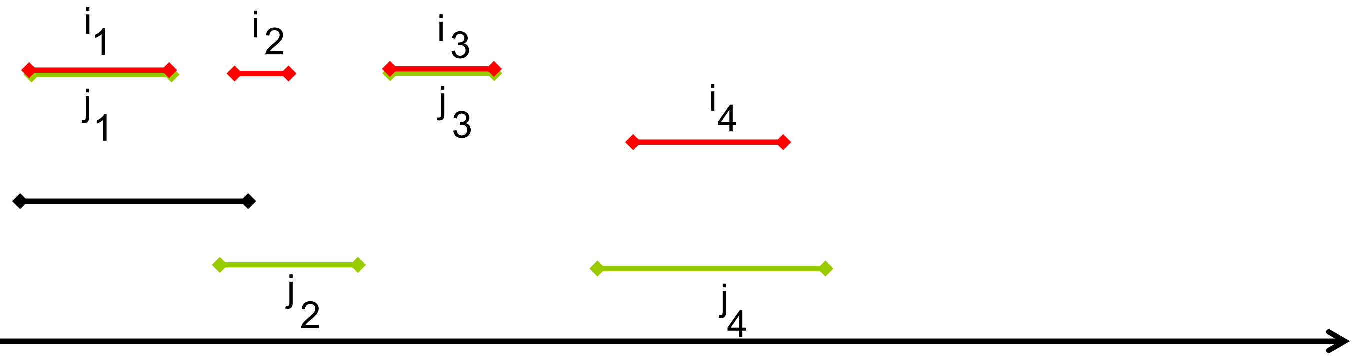
- $i_1, \ldots, i_k$ Intervalle von A in Ordnung des Hinzufügens
- $j_1,\ldots, j_m$ Intervalle von O sortiert nach Endpunkt



Gierige Algorithmen – Intervall Scheduling

Wir zeigen: Der gierige Algorithmus „liegt vorn“:

- D.h. jedes Intervall in A gibt die Ressource mindestens so früh wieder frei wie das korrespondierende Intervall in O
- Dies ist wahr für das erste Intervallpaar, da $i_1 \leq j_1$ und damit $f[i_1] \leq f[j_1]$.
- Zu zeigen: Dies gilt für alle anderen Intervallpaare !



Gierige Algorithmen – Intervall Scheduling

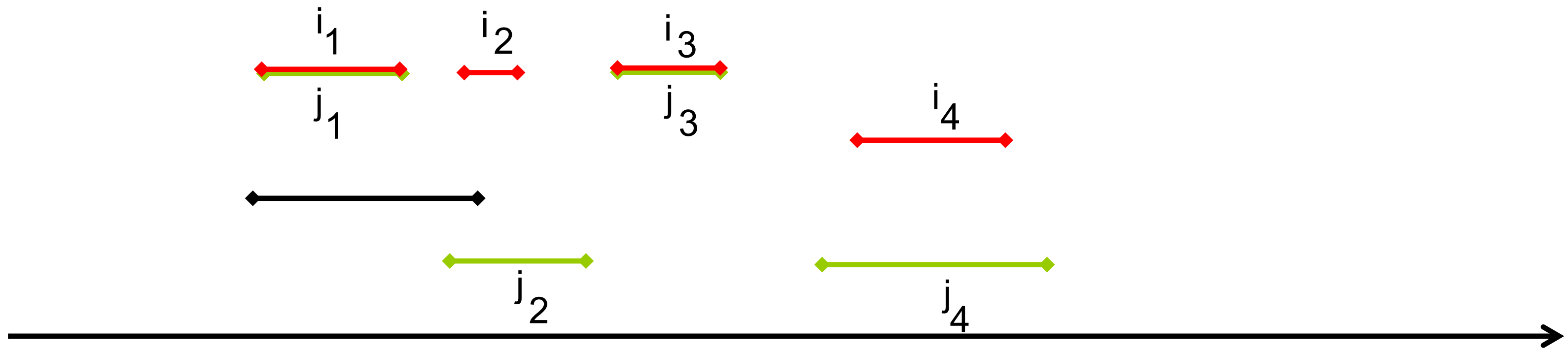
Lemma 19.4:

Für alle $r \leq k$ gilt $i_r \leq j_r$ (und damit $f[i_r] \leq f[j_r]$).

Beweis: vollständige Induktion

- Angenommen, $i_r \leq j_r$ (erfüllt für r=1).

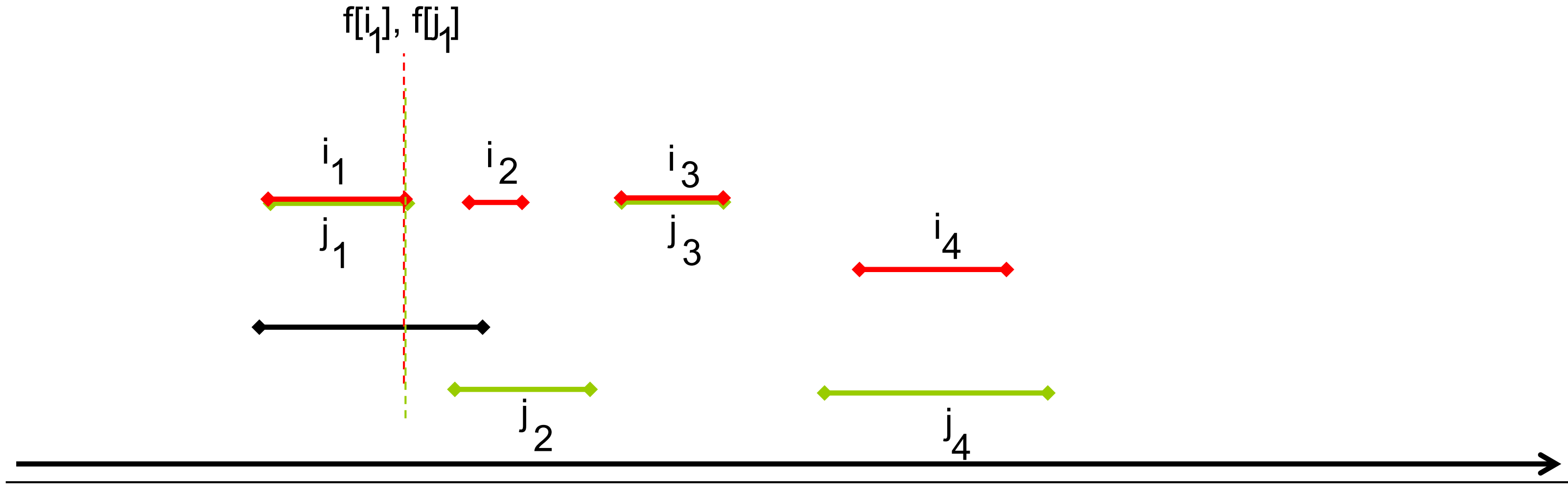
  $s[j_{r+1}] \geq f[j_r] \Rightarrow s[j_{r+1}] \geq f[i_r] \Rightarrow i_{r+1} \leq j_{r+1}$



Gierige Algorithmen – Intervall Scheduling

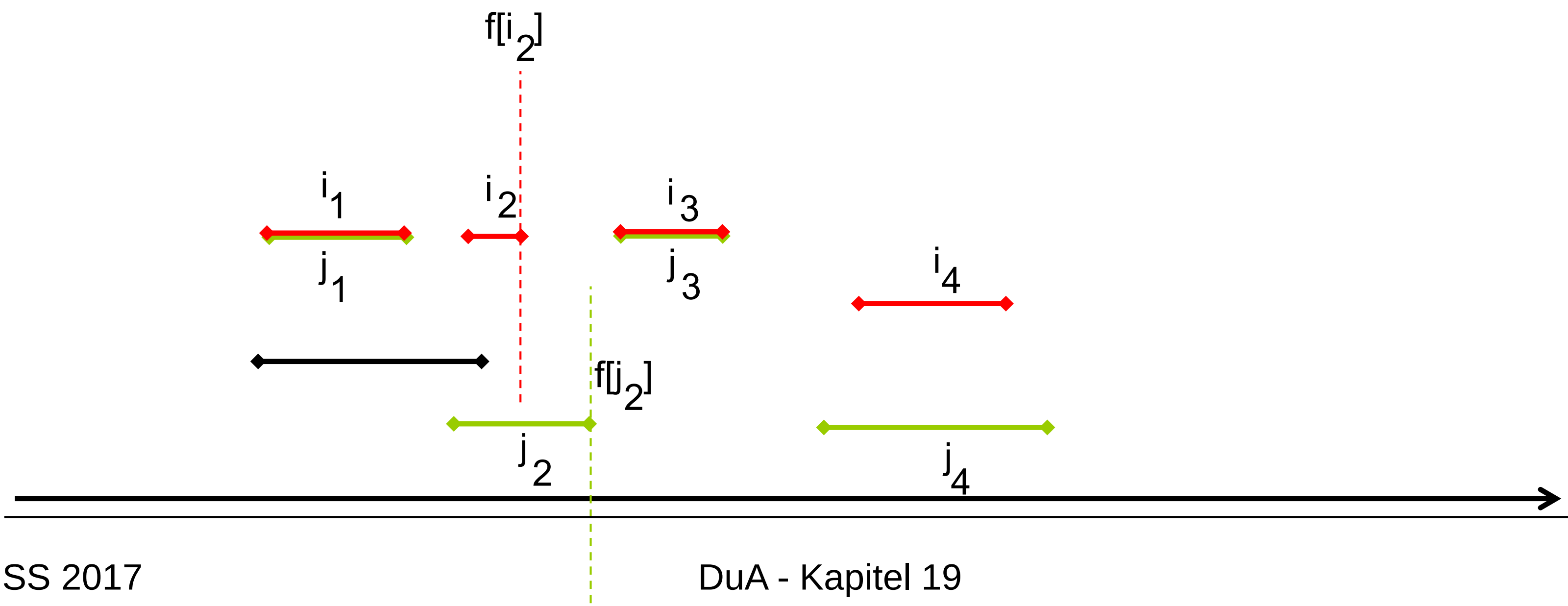
Lemma 19.4:

Für alle $r \leq k$ gilt $i_r \leq j_r$ (und damit $f[i_r] \leq f[j_r]$).



Lemma 19.4:

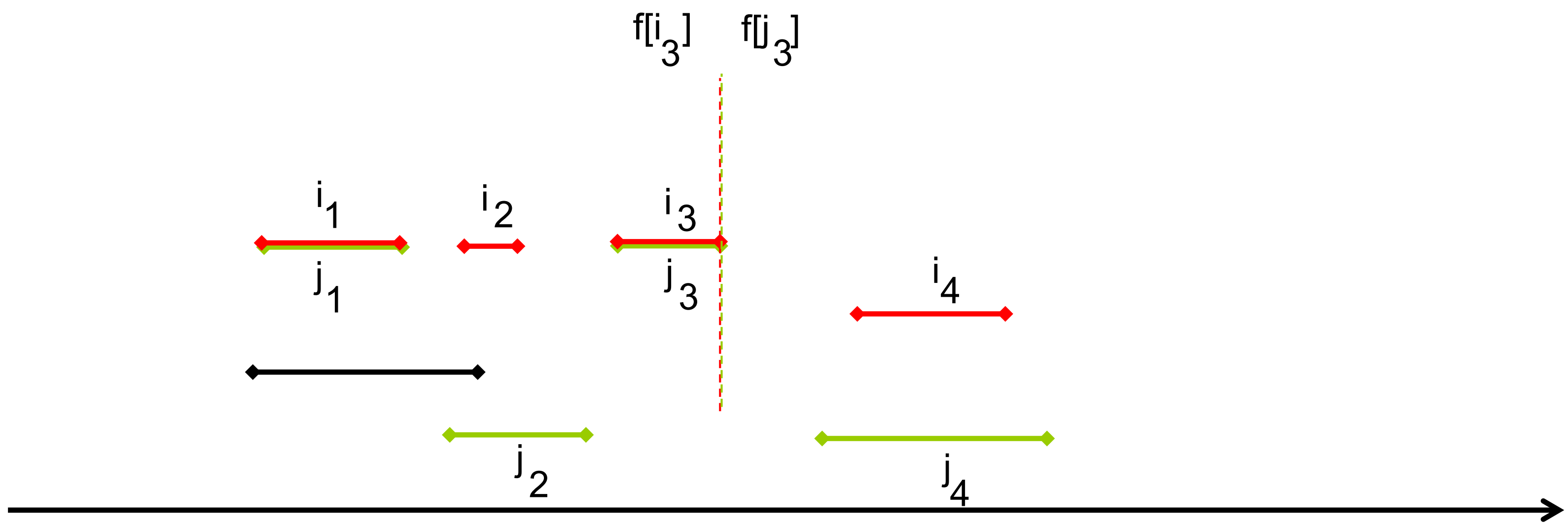
Für alle $r \leq k$ gilt $i_r \leq j_r$ (und damit $f[i_r] \leq f[j_r]$).



Gierige Algorithmen – Intervall Scheduling

Lemma 19.4:

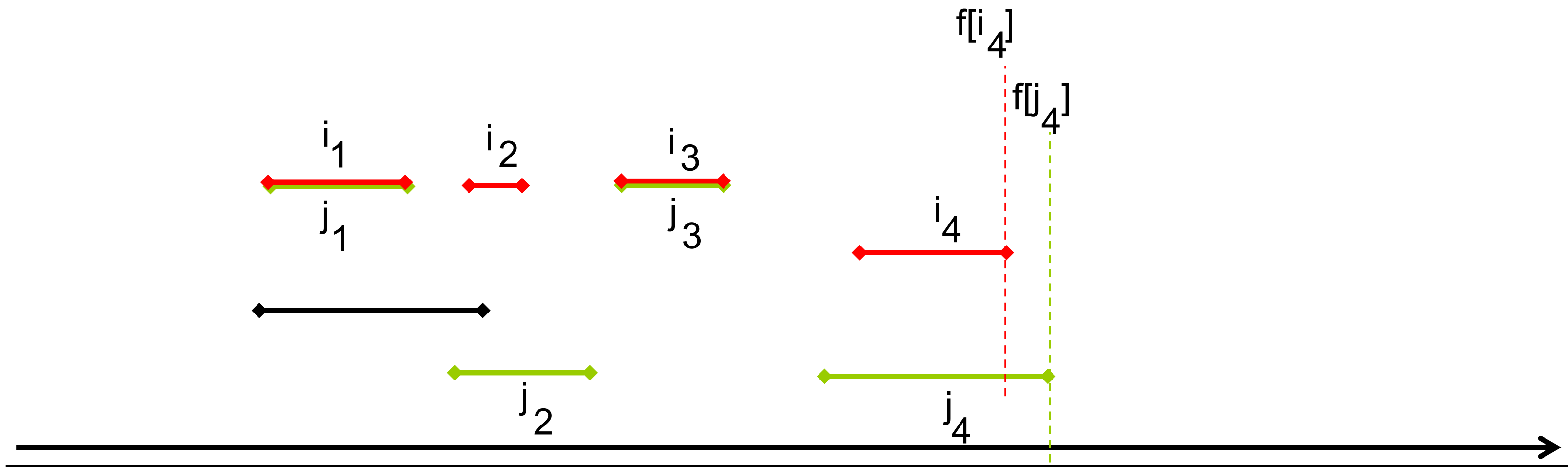
Für alle $r \leq k$ gilt $i_r \leq j_r$ (und damit $f[i_r] \leq f[j_r]$).



Gierige Algorithmen – Intervall Scheduling

Lemma 19.4:

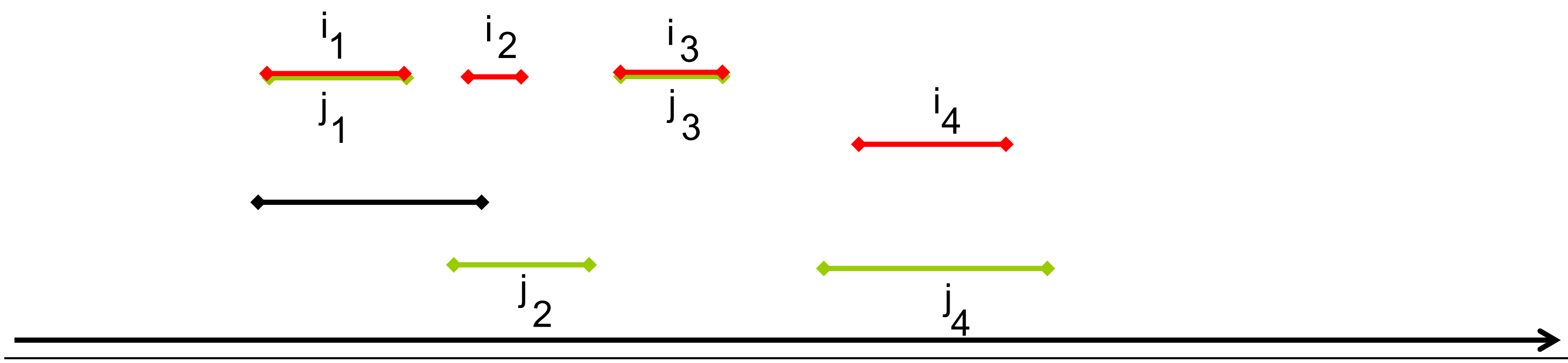
Für alle $r \leq k$ gilt $i_r \leq j_r$ (und damit $f[i_r] \leq f[j_r]$).



Gierige Algorithmen – Intervall Scheduling

Satz 19.5:

Die von Algorithmus IntervalScheduling berechnete Lösung A ist optimal.



Gierige Algorithmen – Intervall Scheduling

IntervalScheduling(s,f)

1. n $\leftarrow$ length[s]
2. A $\leftarrow$ {1}
3. j $\leftarrow$ 1
4. **for** i $\leftarrow$ 2 **to** n **do**
5.   **if** s[i] $\geq$ f[j] **then**
6.   A $\leftarrow$ A $\cup$ {i}
7.   j $\leftarrow$ i
8. **return** A

Gierige Algorithmen – Intervall Scheduling

```
IntervalScheduling(s,f)
n ← length[s]
A ← {1}                                   Θ(1)
j ← 1
for i ← 2 to n do
    if s[i] ≥ f[j] then
        A ← A ∪ {i}
        j ← i
return A
```

Gierige Algorithmen – Intervall Scheduling

$$\begin{array}{l} \text{IntervalScheduling}(s,f) \\ \left.\begin{array}{l} n \leftarrow \text{length}[s] \\ A \leftarrow \{1\} \\ j \leftarrow 1 \end{array}\right\} \Theta(1) \\ \left.\begin{array}{l} \textbf{for } i \leftarrow 2 \textbf{ to } n \textbf{ do} \\ \quad \textbf{if } s[i] \ge f[j] \textbf{ then} \\ \quad\quad A \leftarrow A \cup \{i\} \\ \quad\quad j \leftarrow i \end{array}\right\} \Theta(n) \\ \textbf{return } A \end{array}$$

Gierige Algorithmen – Intervall Scheduling

```
IntervalScheduling(s,f)
n ← length[s]
A ← {1}                          Θ(1)
j ← 1
for i ← 2 to n do
    if s[i] ≥ f[j] then
        A ← A ∪ {i}              Θ(n)
        j ← i
return A                         Θ(1)
```

Gierige Algorithmen – Intervall Scheduling

```
IntervalScheduling(s,f)
n ← length[s]
A ← {1}
j ← 1
for i ← 2 to n do
    if s[i] ≥ f[j] then
        A ← A ∪ {i}
        j ← i
return A
```

$\Theta(1)$

$\Theta(n)$

$\Theta(1)$

$\Theta(n)$

Gierige Algorithmen – Intervall Scheduling

Satz 19.6:

Algorithmus IntervalScheduling berechnet in $\Theta(n)$ Zeit eine optimale Lösung, wenn die Eingabe nach Endzeit der Intervalle (rechter Endpunkt) sortiert ist. Die Sortierung kann in $\Theta(n \log n)$ Zeit berechnet werden.

Anwendungsbeispiel: Datenkompression

Datenkompression

- Reduziert Größen von Files
- Viele Verfahren für unterschiedliche Anwendungen: MP3, MPEG, JPEG, …
- Wie funktioniert Datenkompression?

Zwei Typen von Kompression:

- Verlustbehaftete Kompression (Bilder, Musik, Filme,…)
- Verlustfreie Kompression (Programme, Texte, …)

Datenkompression

- Reduziert Größen von Files
- Viele Verfahren für unterschiedliche Anwendungen: MP3, MPEG, JPEG, …
- Wie funktioniert Datenkompression?

Zwei Typen von Kompression:

- Verlustbehaftete Kompression (Bilder, Musik, Filme,…)
- Verlustfreie Kompression (Programme, Texte, …)

Gierige Algorithmen – Datenkompression

Kodierung:

- Computer arbeiten auf Bits (Symbole 0 und 1), nutzen also das Alphabet {0,1}
- Menschen nutzen umfangreichere Alphabete (z.B. Alphabete von Sprachen)
- Darstellung auf Rechner erfordert Umwandlung in Bitfolgen

Gierige Algorithmen – Datenkompression

Beispiel:

- Alphabet Σ={a,b,c,d,…,x,y,z, ,.,:,!,?,&} (32 Zeichen)
- 5 Bits pro Symbol: 2^5 = 32 Möglichkeiten

a	b	…	z		.	:	!	?	&
00000	00001		11001	11010	11011	11100	11101	11110	11111

Gierige Algorithmen – Datenkompression

Beispiel:

- Alphabet Σ={a,b,c,d,…,x,y,z, ,.,:,!,?,&} (32 Zeichen)
- 5 Bits pro Symbol: $2^5 = 32$ Möglichkeiten

a	b	…	z		.	:	!	?	&
00000	00001		11001	11010	11011	11100	11101	11110	11111

Fragen?

- Sind 4 Bits pro Symbol nicht genug ?
- Müssen wir im Durchschnitt 5 Bits für jedes Vorkommen eines Symbols in langen Texten verwenden?

Gierige Algorithmen – Datenkompression

Beobachtung:

- Nicht jeder Buchstabe kommt gleich häufig vor
- Z.B. kommen x,y und z in der deutschen Sprache viel seltener vor als e, n oder r

Gierige Algorithmen – Datenkompression

Beobachtung:

- Nicht jeder Buchstabe kommt gleich häufig vor
- Z.B. kommen x,y und z in der deutschen Sprache viel seltener vor als e, n oder r

Idee:

- Benutze kurze Bitstrings für Symbole die häufig vorkommen

Gierige Algorithmen – Datenkompression

Beobachtung:

- Nicht jeder Buchstabe kommt gleich häufig vor
- Z.B. kommen x,y und z in der deutschen Sprache viel seltener vor als e, n oder r

Idee:

- Benutze kurze Bitstrings für Symbole die häufig vorkommen

Effekt:

- Gesamtlänge der Kodierung einer Symbolfolge (eines Textes) wird reduziert

Gierige Algorithmen – Datenkompression

Grundlegendes Problem:

- Eingabe: Text über dem Alphabet Σ
- Gesucht: Eine binäre Kodierung von Σ, so dass die Länge des Textes in dieser Kodierung minimiert wird

Beispiel:

- Σ={0,1,2,…,9}
- Text = 00125590004356789 (17 Zeichen)

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Gierige Algorithmen – Datenkompression

Grundlegendes Problem:

- Eingabe: Text über dem Alphabet Σ
- Gesucht: Eine binäre Kodierung von Σ, so dass die Länge des Textes in dieser Kodierung minimiert wird

Beispiel:

- $\Sigma=\{0,1,2,\ldots,9\}$
- Text = 00125590004356789 (17 Zeichen)

Länge der Kodierung:
$4 \cdot 17 = 68$ Bits

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Gierige Algorithmen – Datenkompression

Grundlegendes Problem:

- Eingabe: Text über dem Alphabet Σ
- Gesucht: Eine binäre Kodierung von Σ, so dass die Länge des Textes in dieser Kodierung minimiert wird

Beispiel:

- $\Sigma=\{0,1,2,\ldots,9\}$
- Text = 00125590004356789 (17 Zeichen)

0	1	2	3	4	5	6	7	8	9
0	11000	11001	11010	11011	10	11100	11101	11110	11111

Grundlegendes Problem:

- Eingabe: Text über dem Alphabet Σ
- Gesucht: Eine binäre Kodierung von Σ, so dass die Länge des Textes in dieser Kodierung minimiert wird.

Beispiel:

- $\Sigma=\{0,1,2,\ldots,9\}$
- Text = 00125590004356789 (17 Zeiche

Länge der Kodierung:
$5 \cdot 1 + 3 \cdot 2 + 9 \cdot 5 = 56$ Bits

0	1	2	3	4	5	6	7	8	9
0	11000	11001	11010	11011	10	11100	11101	11110	11111

Gierige Algorithmen – Datenkompression

Grundlegendes Problem:

- Eingabe: Text über dem Alphabet Σ
- Gesucht: Eine binäre Kodierung von Σ, so dass die Länge des Textes in dieser Kodierung minimiert wird

Beispiel:

- $\Sigma=\{0,1,2,\ldots,9\}$
- Text = 00125590004356789 (17 Zeiche

Länge der Kodierung:
$5 \cdot 1 + 3 \cdot 2 + 9 \cdot 5 = 56$ Bits

0	1	2	3	4	5	6	7	8	9
0	11000	11001	11010	11011	10	11100	11101	11110	11111

Gierige Algorithmen – Datenkompression

Grundlegendes Problem:

- Eingabe: Text über dem Alphabet Σ
- Gesucht: Eine binäre Kodierung von Σ, so dass die Länge des Textes in dieser Kodierung minimiert wird.

Beispiel:

- $\Sigma=\{0,1,2,\ldots,9\}$
- Text = 00125590004356789 (17 Zeichen)

Länge der Kodierung:
$5 \cdot 1 + 3 \cdot 2 + 9 \cdot 5 = 56$ Bits

0	1	2	3	4	5	6	7	8	9
0	11000	11001	11010	11011	10	11100	11101	11110	11111

Gierige Algorithmen – Datenkompression

Grundlegendes Problem:

- Eingabe: Text über dem Alphabet Σ
- Gesucht: Eine binäre Kodierung von Σ, so dass die Länge des Textes in dieser Kodierung minimiert wird.

Beispiel:

- $\Sigma=\{0,1,2,\ldots,9\}$
- Text = 00125590004356789 (17 Zeichen)

Länge der Kodierung:
$5 \cdot 1 + 3 \cdot 2 + 9 \cdot 5 = 56$ Bits

0	1	2	3	4	5	6	7	8	9
0	11000	11001	11010	11011	10	11100	11101	11110	11111

Gierige Algorithmen – Datenkompression

MorseCode:

- Elektrische Pulse über Kabel
- Punkte (kurze Pulse)
- Striche(Lange Pulse)

Beispiele aus dem MorseCode:

- e ist 0 (ein einzelner Punkt)
- t ist 1 (ein einzelner Strich)
- a ist 01 (Punkt – Strich)

Problem:

- Ist 0101 eta, aa, etet, oder aet ?

Gierige Algorithmen – Datenkompression

Problem Mehrdeutigkeit:

- Ist die Kodierung eines Buchstabens ein Präfix der Kodierung eines anderen Buchstabens, dann kann es passieren, dass die Kodierung nicht eindeutig ist.

Beispiel:

- e = 0, a = 01
- 0 ist Präfix von 01

Präfix-Kodierung:

Eine Präfix-Kodierung für ein Alphabet Σ ist eine Funktion γ, die jeden Buchstaben $x \in \Sigma$ auf eine endliche Sequenz von 0 und 1 abbildet, so dass für $x,y \in \Sigma$, $x \neq y$, die Sequenz $\gamma(x)$ *kein* Präfix der Sequenz $\gamma(y)$ ist.

Gierige Algorithmen – Datenkompression

Präfix-Kodierung:

Eine Präfix-Kodierung für ein Alphabet Σ ist eine Funktion γ, die jeden Buchstaben $x \in \Sigma$ auf eine endliche Sequenz von 0 und 1 abbildet, so dass für $x,y \in \Sigma$, $x \neq y$, die Sequenz $\gamma(x)$ *kein* Präfix der Sequenz $\gamma(y)$ ist.

Beispiel (Präfix-Kodierung):

$x \in \Sigma$	0	1	2	3	4	5	6	7	8	9
$\gamma(x)$	00	0100	0110	0111	1001	1010	1011	1101	1110	1111

Definition (Frequenz)

- Die Frequenz $f[x]$ eines Buchstaben $x \in \Sigma$ bezeichnet den Bruchteil der Buchstaben im Text, die x sind.

Beispiel:

- $\Sigma = \{0,1,2\}$
- Text =„0010022001“ (10 Zeichen)
- $f[0] = 3/5$
- $f[1] = 1/5$
- $f[2] = 1/5$

Definition (Kodierungslänge)

Die Kodierungslänge eines Textes mit n Zeichen bzgl. einer Kodierung γ ist definert als

$$\text{Kodierungslänge} = \sum_{x \in \Sigma} n \cdot f[x] \cdot |\gamma(x)|$$

Beispiel:

- $\Sigma = \{a,b,c,d\}$
- $\gamma(a) = 0$; $\gamma(b) = 101$; $\gamma(c) = 110$; $\gamma(d) = 111$
- Text = „aacdaabb“
- Kodierungslänge = 16

Definition (Kodierungslänge)

Die Kodierungslänge eines Textes mit

Anzahl der Vorkommen von x im Text

einer Kodierung γ ist definiert als

$$\text{Kodierungslänge} = \sum_{x\in\Sigma} n\cdot f[x]\cdot|\gamma(x)|$$

Beispiel:

- $\Sigma = \{a,b,c,d\}$
- $\gamma(a) = 0$; $\gamma(b) = 101$; $\gamma(c) = 110$; $\gamma(d) = 111$
- Text = „aacdaabb“
- Kodierungslänge = 16

Gierige Algorithmen – Datenkompression

Definition (Kodierungslänge)

Die Kodierungslänge eines Textes mit n Zeichen bezüglich einer Kodierung γ ist definiert als

Länge der Codierung von x

$$\text{Kodierungslänge} = \sum_{x \in \Sigma} n \cdot f[x] \cdot |\gamma(x)|$$

Beispiel:

- $\Sigma = \{a,b,c,d\}$
- $\gamma(a) = 0$; $\gamma(b) = 101$; $\gamma(c) = 110$; $\gamma(d) = 111$
- Text = „aacdaabb“
- Kodierungslänge = 16

Definition (durchschn. Kodierungslänge)

Die durchschnittliche Kodierungslänge eines Buchstabens in einem Text mit n Zeichen und bzgl. einer Kodierung γ ist definiert als

$$ABL(\gamma) = \sum_{x \in \Sigma} f[x] \cdot |\gamma(x)|$$

Beispiel:

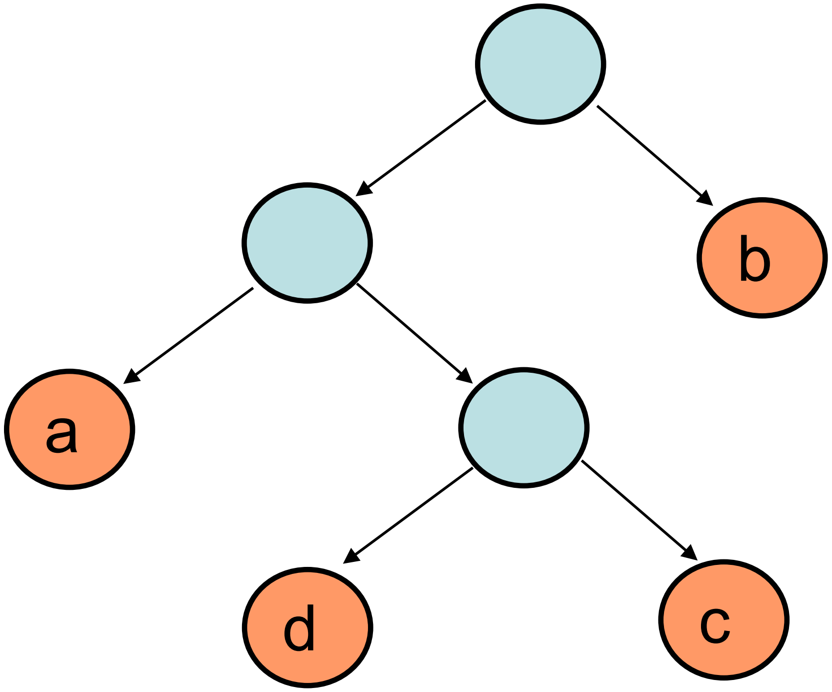
- $\Sigma = \{a,b,c,d\}$
- $\gamma(a) = 0$; $\gamma(b) = 101$; $\gamma(c) = 110$; $\gamma(d) = 111$
- Text = „aacdaabb“
- Durchschnittliche Kodierungslänge = 16/8 = 2

Problem einer optimalen Präfix-Kodierung:

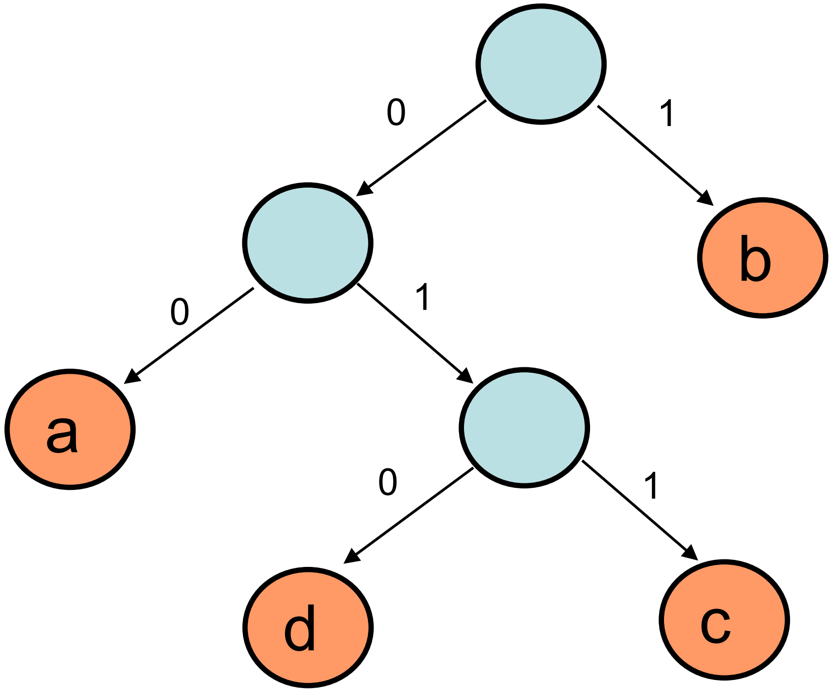
- Eingabe:
  Alphabet Σ und für jedes $x \in \Sigma$ seine Frequenz f[x]

- Ausgabe:
  Eine Kodierung γ, die ABL(γ) minimiert

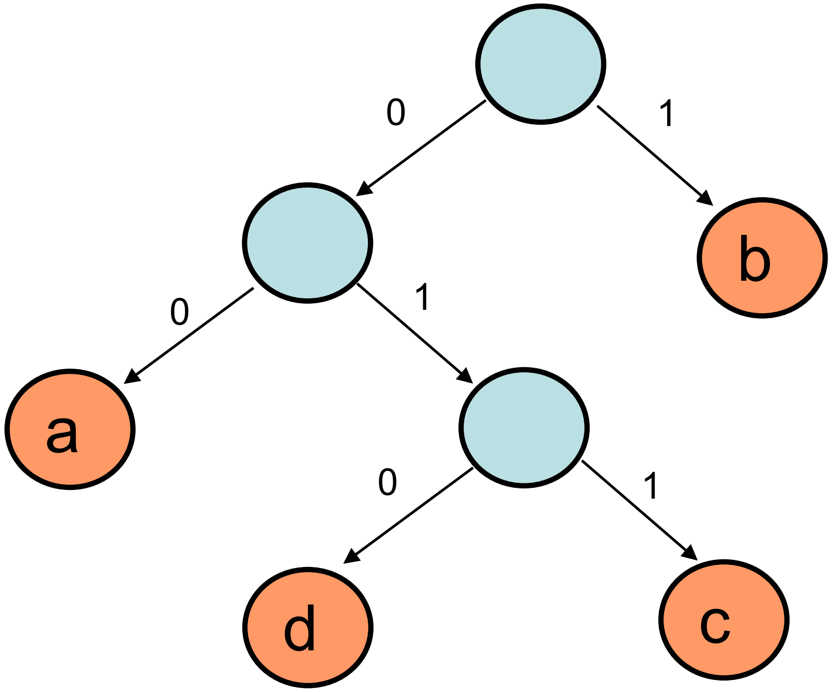
Binärbäume und Präfix-Kodierungen:



Binärbäume und Präfix-Kodierungen:



Binärbäume und Präfix-Kodierungen:



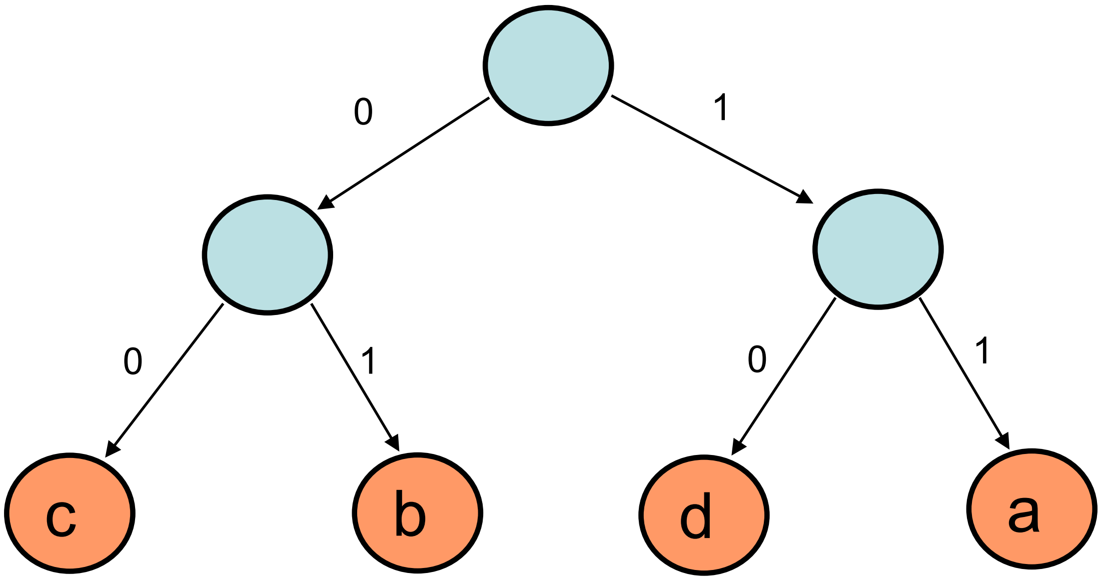
$x \in \Sigma$	$\gamma(x)$
a	00
b	1
c	011
d	010

Präfix-Kodierungen und Binärbäume:

$x \in \Sigma$	$\gamma(x)$
a	11
b	01
c	00
d	10

Präfix-Kodierungen und Binärbäume:

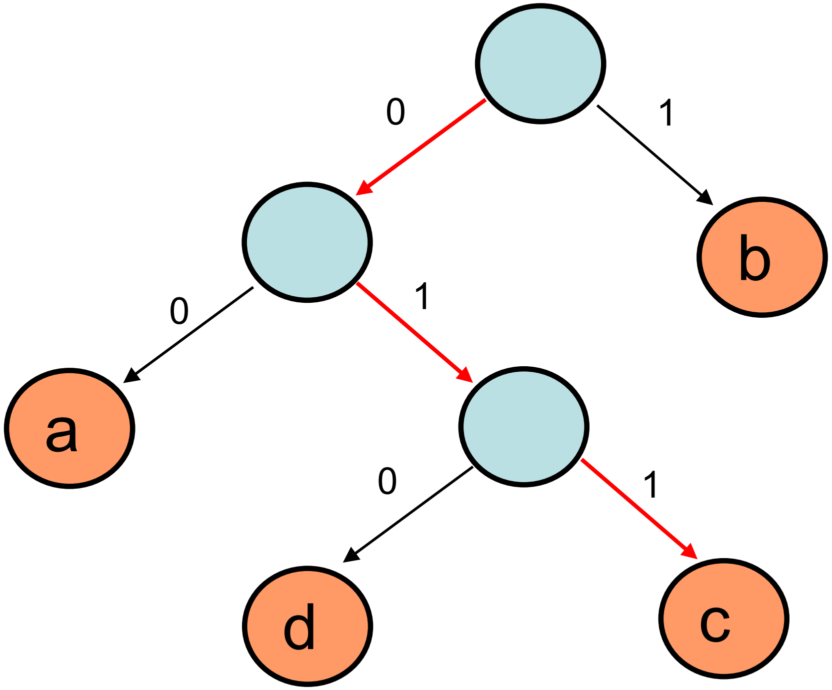
$x \in \Sigma$	$\gamma(x)$
a	11
b	01
c	00
d	10



Gierige Algorithmen – Datenkompression

Definition:

Die Tiefe eines Baumknotens ist die Länge seines Pfades zur Wurzel.



Tiefe(c) = 3

Gierige Algorithmen – Datenkompression

Neue Problemformulierung:

- Suche Binärbaum T, dessen Blätter die Symbole aus Σ sind und der

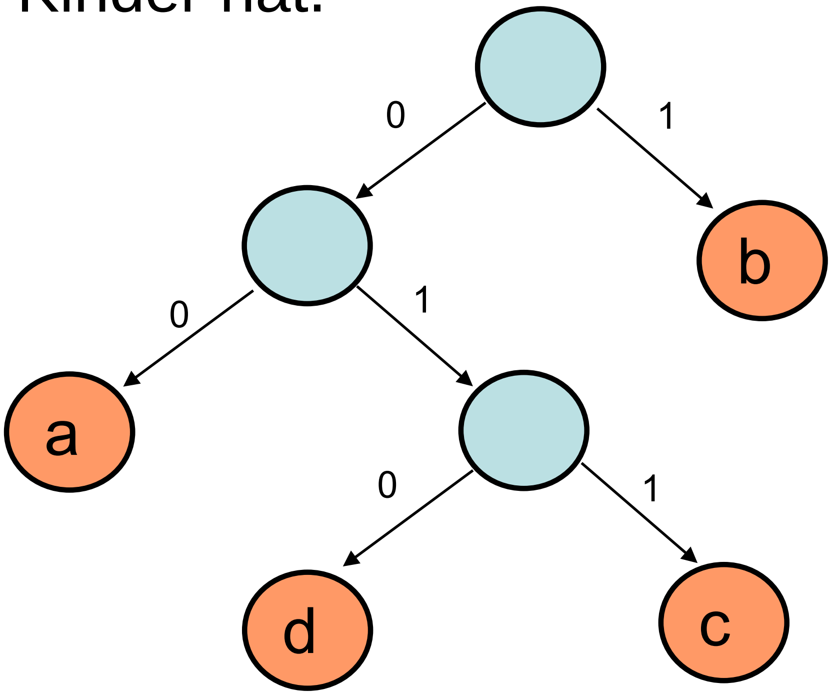
$$ABL(T) = \sum_{x \in \Sigma} f[x] \cdot \mathrm{Tiefe}_T(x)$$

minimiert.

Gierige Algorithmen – Datenkompression

Definition:

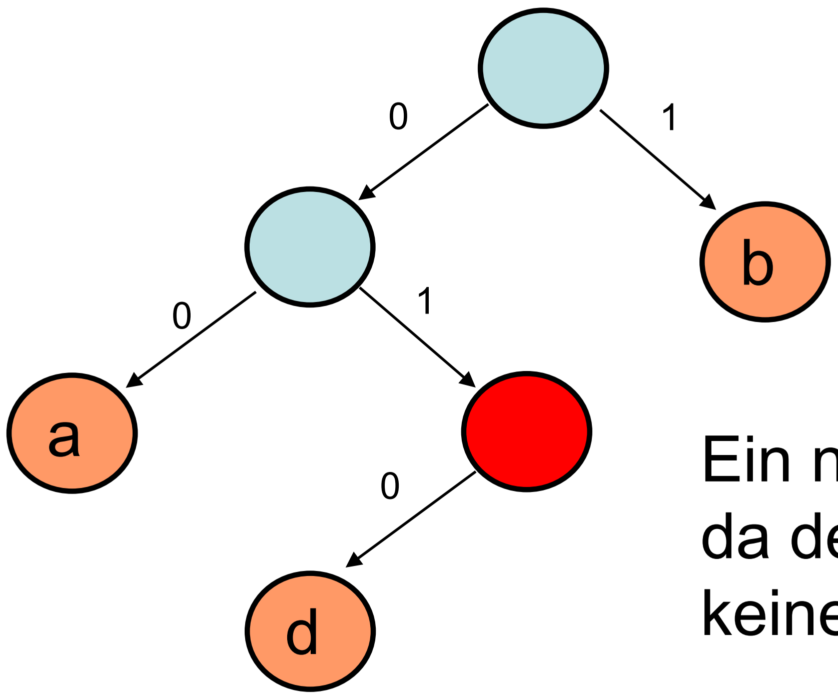
Ein Binärbaum heißt voll, wenn jeder innere Knoten genau zwei Kinder hat.



Ein voller Binärbaum

Definition:

Ein Binärbaum heißt voll, wenn jeder innere Knoten genau zwei Kinder hat.

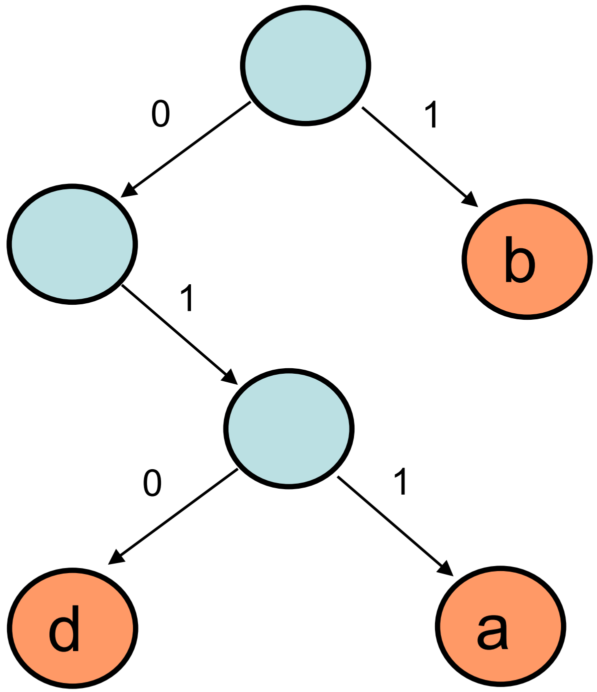


Ein nicht voller Binärbaum, da der rote innere Knoten keine zwei Kinder hat

Gierige Algorithmen – Datenkompression

Lemma 19.7:

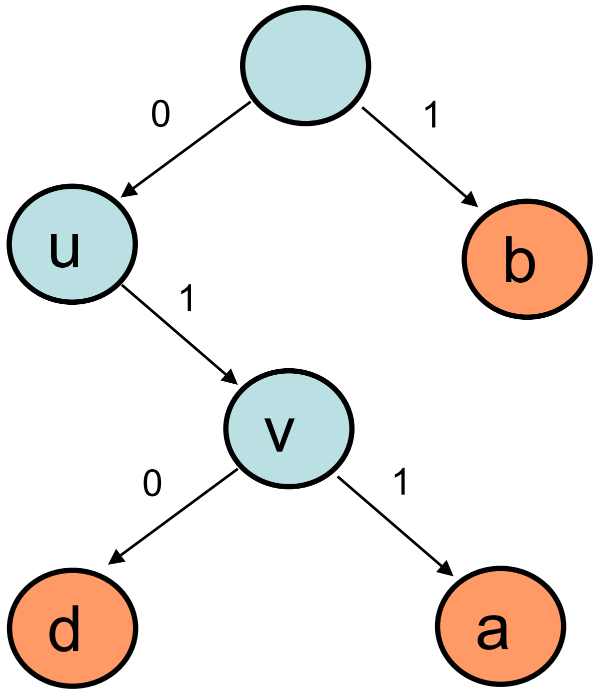
Der Binärbaum, der einer optimalen Präfix-Kodierung entspricht, ist voll.



Gierige Algorithmen – Datenkompression

Lemma 19.7:

Der Binärbaum, der einer optimalen Präfix-Kodierung entspricht, ist voll.



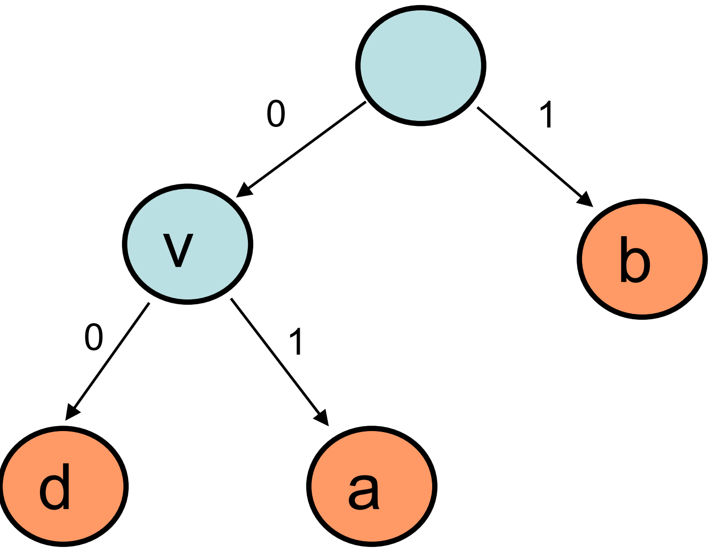
Beweis:

- Annahme: T ist optimal und hat inneren Knoten u mit einem Kind v
- Ersetze u durch v
- Dies verkürzt die Tiefe einiger Knoten, erhöht aber keine Tiefe
- Damit verbessert man die Kodierung

Gierige Algorithmen – Datenkompression

Lemma 19.7:

Der Binärbaum, der einer optimalen Präfix-Kodierung entspricht, ist voll.



<u>Beweis:</u>

- Annahme: T ist optimal und hat inneren Knoten u mit einem Kind v
- Ersetze u durch v
- Dies verkürzt die Tiefe einiger Knoten, erhöht aber keine Tiefe
- Damit verbessert man die Kodierung

Gierige Algorithmen – Datenkompression

Ein Gedankenexperiment:

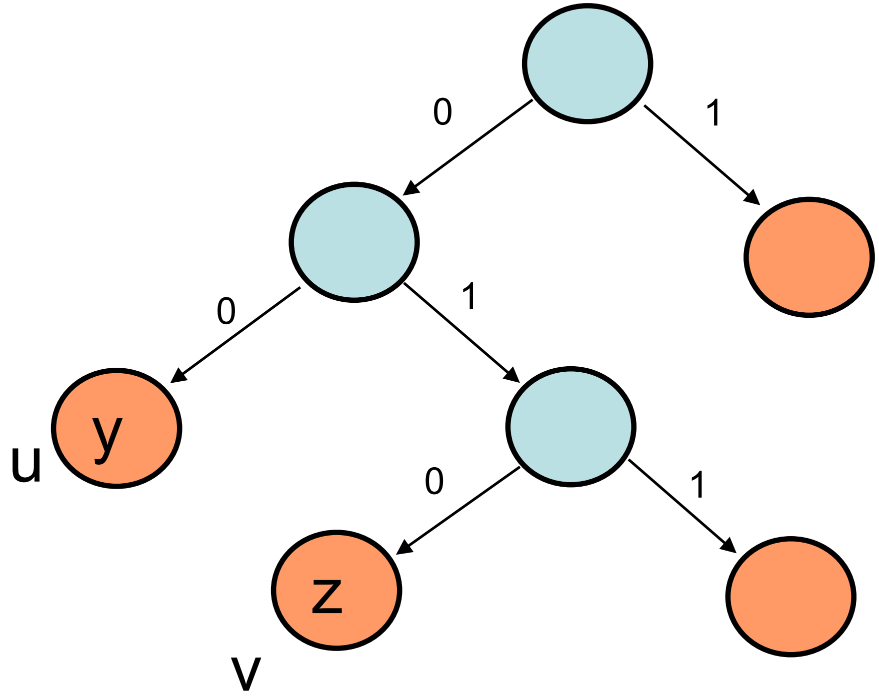
- Angenommen, jemand gibt uns den optimalen Baum T*, aber nicht die Bezeichnung der Blätter
- Wie schwierig ist es, die Bezeichnungen zu finden?

Gierige Algorithmen – Datenkompression

Lemma 19.8:

Seien u und v Blätter von T* mit Tiefe(u) <Tiefe(v).

Seien u bzw. v in einer optimalen Kodierung mit $y\in\Sigma$ bzw. $z\in\Sigma$ bezeichnet. Dann gilt $f[y]\geq f[z]$.

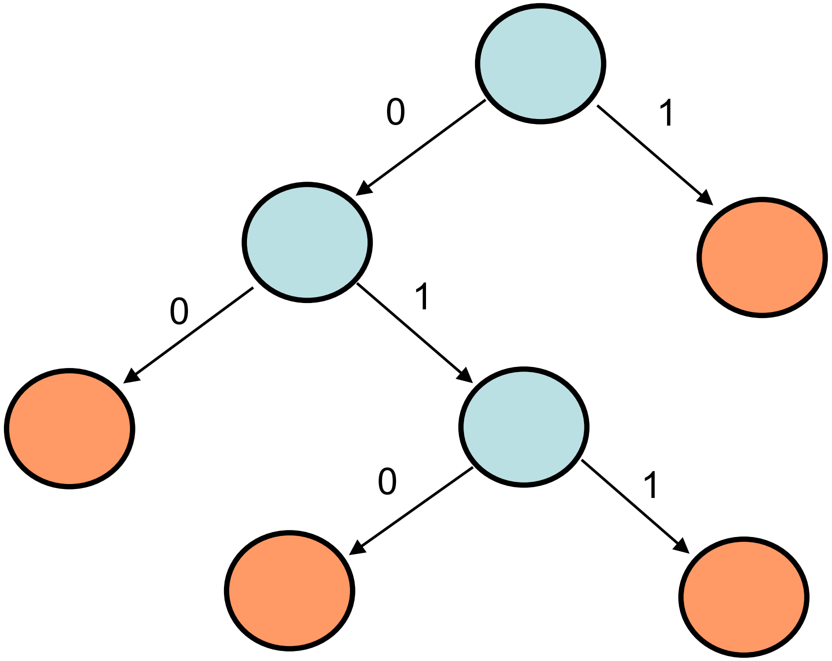


Gierige Algorithmen – Datenkompression

Lemma 19.8:

Seien u und v Blätter von T* mit Tiefe(u) <Tiefe(v).

Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet. Dann gilt $f[y] \geq f[z]$.



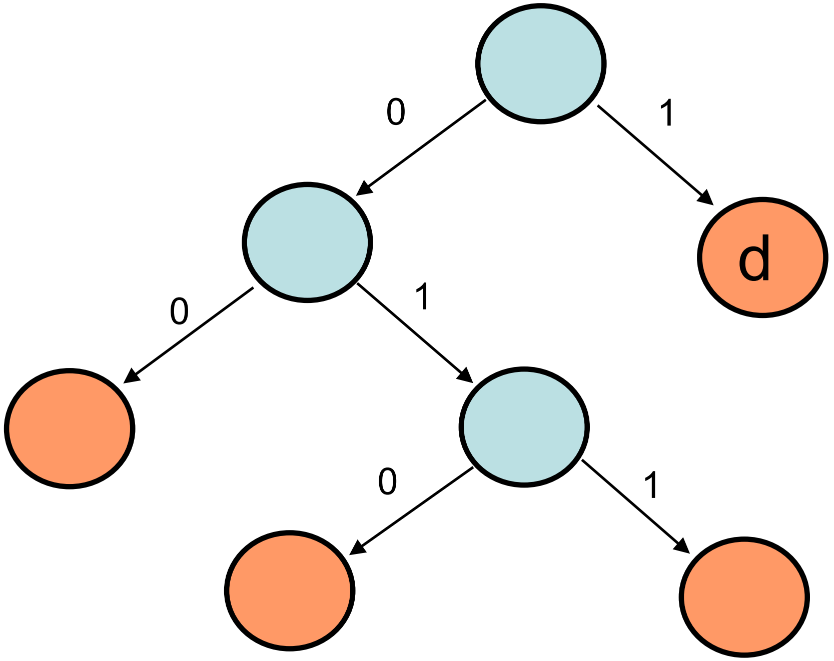
$x \in \Sigma$	f[x]
a	10%
b	12%
c	18%
d	60%

Gierige Algorithmen – Datenkompression

Lemma 19.8:

Seien u und v Blätter von T* mit Tiefe(u) <Tiefe(v).

Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet. Dann gilt $f[y] \geq f[z]$.



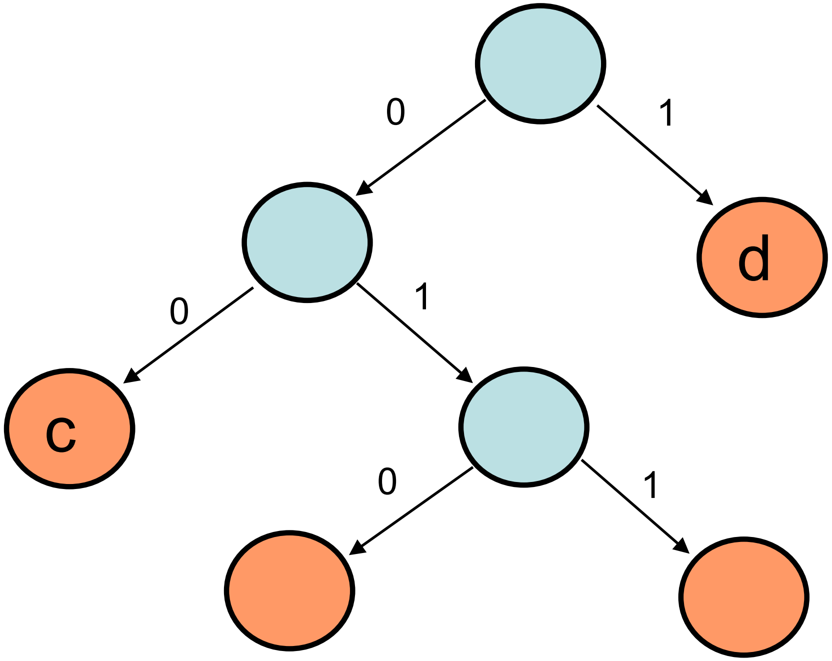
$x \in \Sigma$	f[x]
a	10%
b	12%
c	18%
d	60%

Gierige Algorithmen – Datenkompression

Lemma 19.8:

Seien u und v Blätter von T* mit Tiefe(u) <Tiefe(v).

Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet. Dann gilt $f[y] \geq f[z]$.



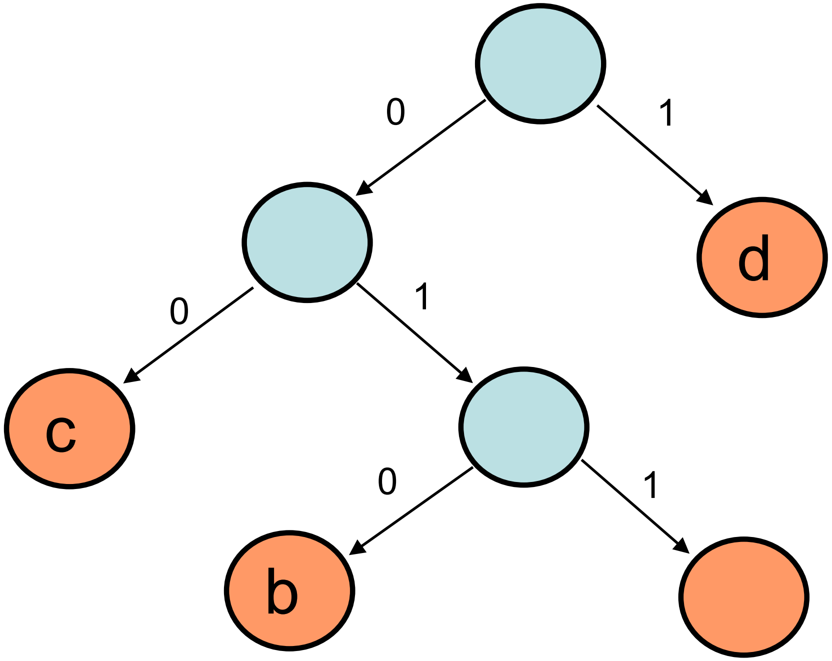
$x \in \Sigma$	f[x]
a	10%
b	12%
c	18%
d	60%

Gierige Algorithmen – Datenkompression

Lemma 19.8:

Seien u und v Blätter von T* mit Tiefe(u) <Tiefe(v).

Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet. Dann gilt $f[y] \geq f[z]$.



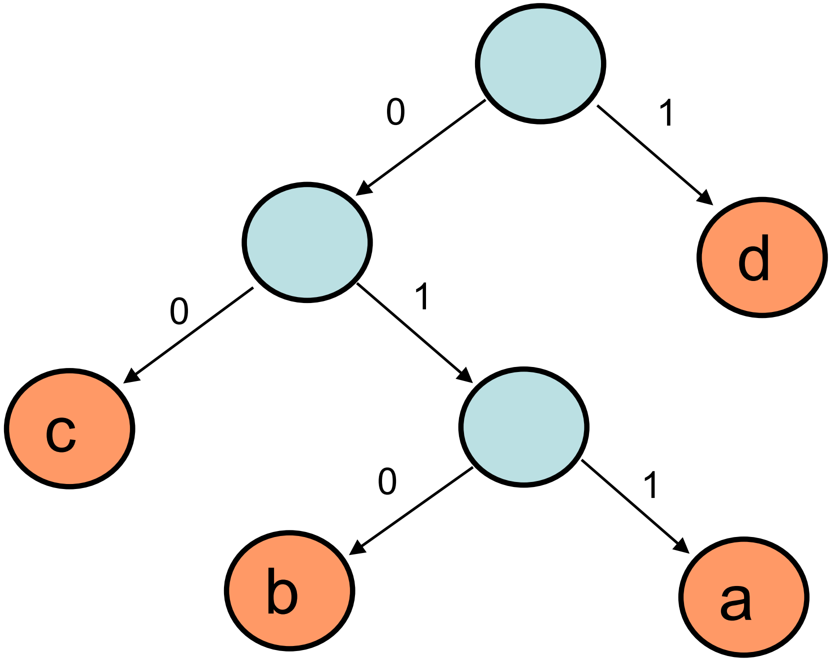
$x \in \Sigma$	f[x]
a	10%
b	12%
c	18%
d	60%

Gierige Algorithmen – Datenkompression

Lemma 19.8:

Seien u und v Blätter von T* mit Tiefe(u) <Tiefe(v).

Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet. Dann gilt $f[y] \geq f[z]$.



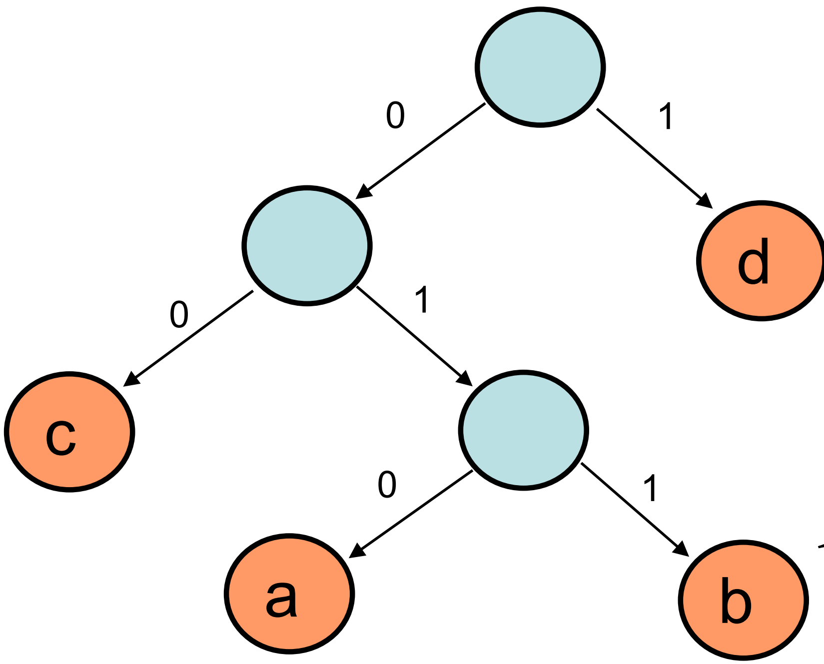
$x \in \Sigma$	f[x]
a	10%
b	12%
c	18%
d	60%

Gierige Algorithmen – Datenkompression

Lemma 19.8:

Seien u und v Blätter von T* mit Tiefe(u) <Tiefe(v).

Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet. Dann gilt $f[y] \geq f[z]$.

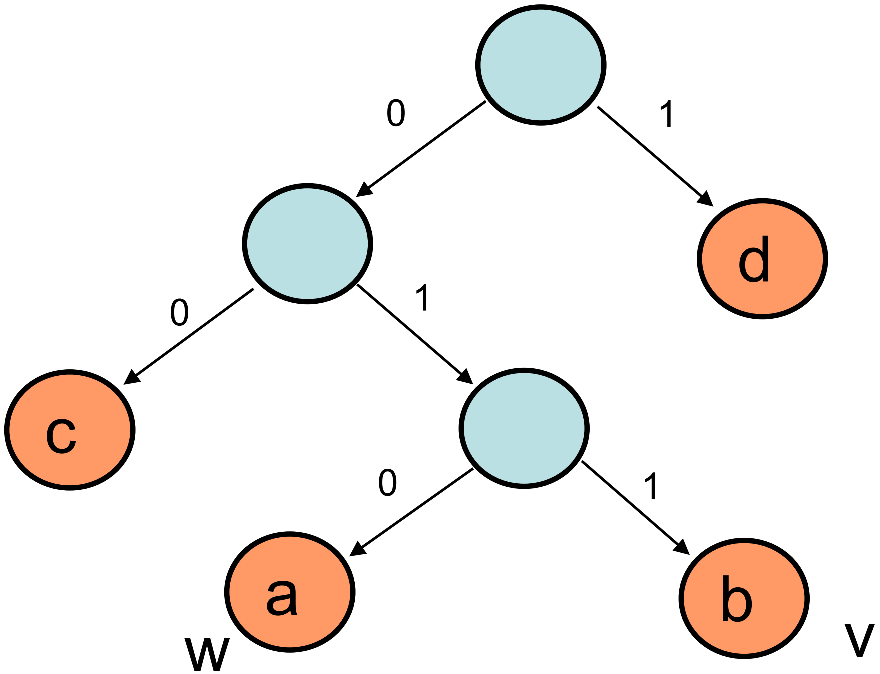


$x \in \Sigma$	f[x]
a	10%
b	12%
[illegible]	[illegible]%
[illegible]	[illegible]%

Anordnung innerhalb einer Tiefenstufe ist egal!

Beobachtung

Sei v der tiefste Blattknoten in T*. Dann hat v einen Bruder und dieser ist ebenfalls ein Blattknoten.

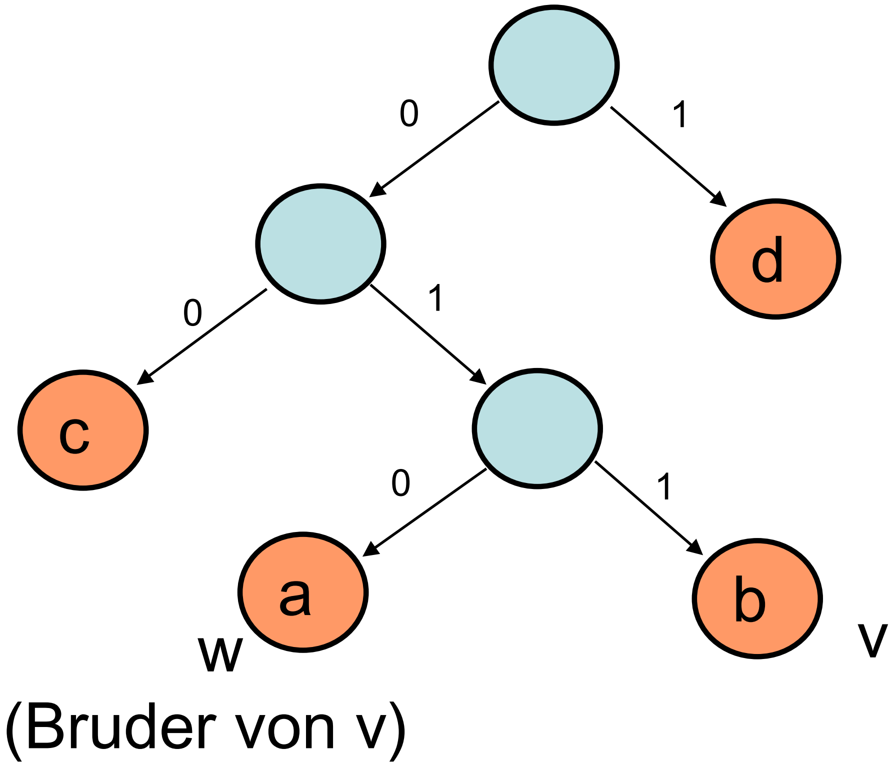


(Bruder von v)

Gierige Algorithmen – Datenkompression

Beobachtung

Sei v der tiefste Blattknoten in T*. Dann hat v einen Bruder und dieser ist ebenfalls ein Blattknoten.



Beweis:

- Da ein optimaler Baum voll ist, hat v einen Bruder w
- Wäre w kein Blatt, dann hätte ein Nachfolger von w größere Tiefe als v

Lemma 19.9:

Es gibt eine optimale Präfix-Kodierung mit zugehörigem Baum T*, so dass die beiden Blattknoten, denen die Symbole mit den kleinsten Frequenzen zugewiesen wurden, Bruderknoten in T* sind.

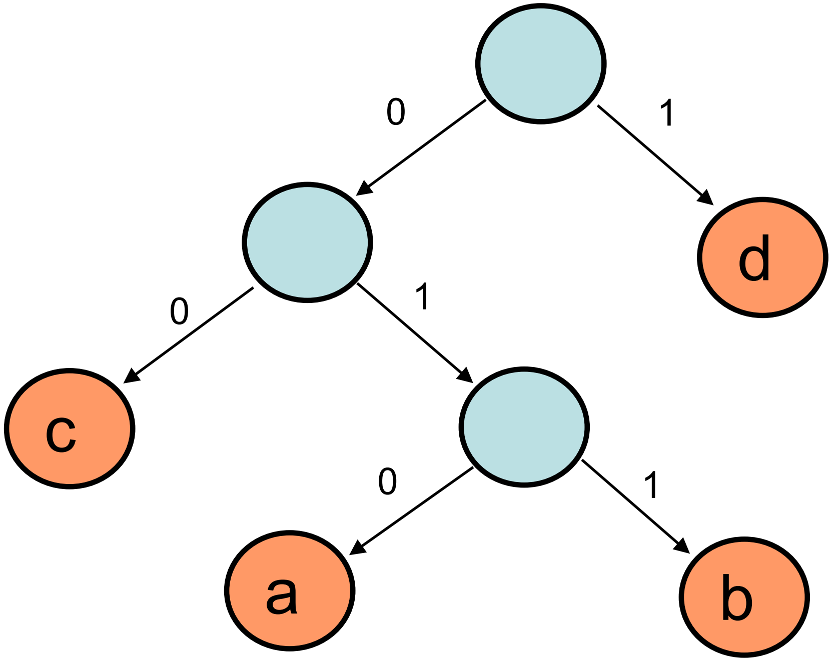
Beweis:

- Nach Lemma 19.8 ist das Symbol $y \in \Sigma$ mit minimalem f[y] in einem tiefsten Knoten v zu finden.
- Da v ein Blatt ist, hat v einen Bruder w, welcher auch ein Blatt ist.
- Sollte w nicht einem Symbol z mit zweitkleinster Frequenz zugewiesen sein, dann kann z dorthin getauscht werden ohne ABL(T*) zu erhöhen.

Gierige Algorithmen – Datenkompression

Lemma 19.9

Es gibt eine optimale Präfix-Kodierung mit zugehörigem Baum T*, so dass die beiden Blattknoten, denen die Symbole mit den kleinsten Frequenzen zugewiesen wurden, Bruder-knoten in T* sind.

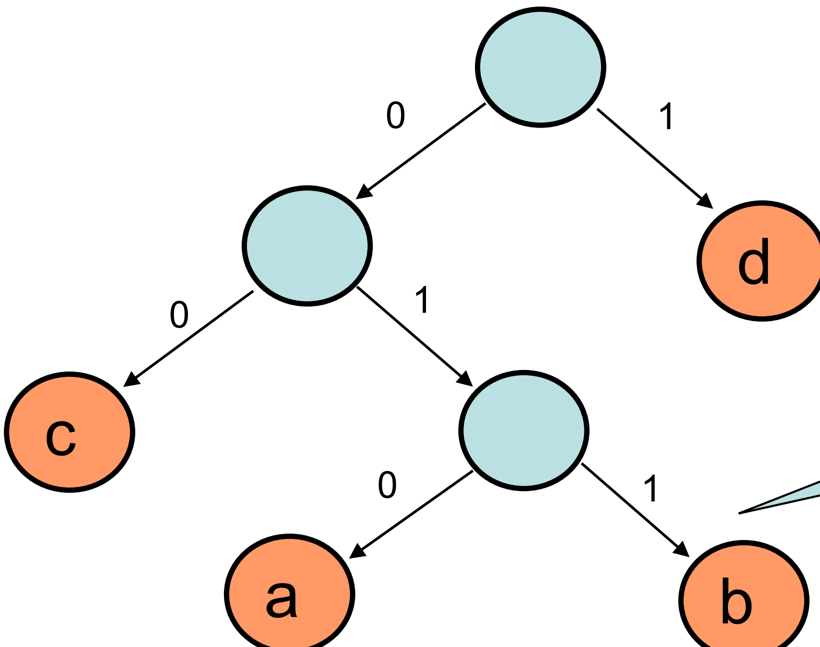


$x \in \Sigma$	f[x]
a	10%
b	12%
c	18%
d	60%

Gierige Algorithmen – Datenkompression

Lemma 19.9

Es gibt eine optimale Präfix-Kodierung mit zugehörigem Baum T*, so dass die beiden Blattknoten, denen die Symbole mit den kleinsten Frequenzen zugewiesen wurden, Bruder-knoten in T* sind.

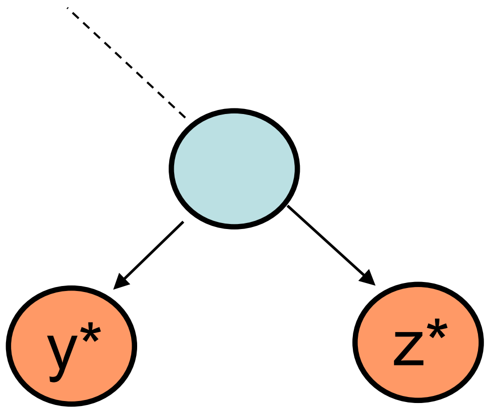


a und b erfüllen die Bedingungen der Behauptung!

$x \in \Sigma$	f[x]
a	10%
[illegible]	[illegible]%
[illegible]	[illegible]%
d	60%

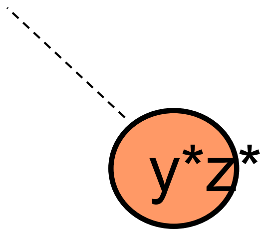
Idee des Algorithmus:

- Die beiden Symbole y* und z* mit den niedrigsten Frequenzen sind Bruderknoten
- Fasse y* und z* zu einem neuen Symbol zusammen
- Löse das Problem für die übrigen n-1 Symbole (z.B. rekursiv)



Idee des Algorithmus:

- Die beiden Symbole y* und z* mit den niedrigsten Frequenzen sind Bruderknoten
- Fasse y* und z* zu einem neuen Symbol zusammen
- Löse das Problem für die übrigen n-1 Symbole (z.B. rekursiv)



Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)

1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

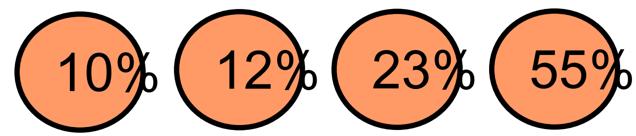
Huffman(Σ)
1. $n \leftarrow |\Sigma|$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** $n-1$ **do**
4. $x \leftarrow$ deleteMin(Q)
5. $y \leftarrow$ deleteMin(Q)
6. $z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. $n \leftarrow |\Sigma|$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $x \leftarrow$ deleteMin(Q)
5. $y \leftarrow$ deleteMin(Q)
6. $z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

Q:



Gierige Algorithmen – Datenkompression

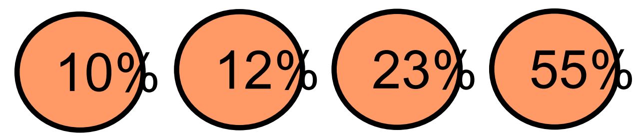
$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)

1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

i=1

Q:

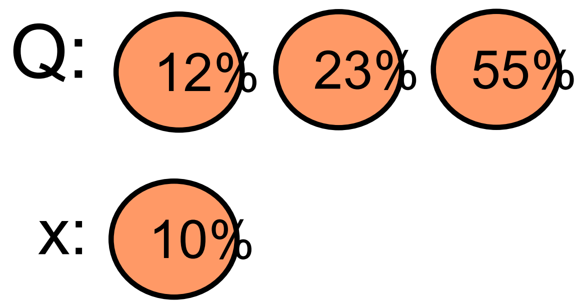


Gierige Algorithmen – Datenkompression

```
Huffman(Σ)
1. n ← | Σ |
2. Q ← Σ       /* Priority Queue bzgl. f[x] */
3. for i ← 1 to n-1 do
4.     x ← deleteMin(Q)
5.     y ← deleteMin(Q)
6.     z←new BinTree(x, f[x]+f[y],y)
7.     f[z]← f[x] + f[y]
8.     Q ← Q ∪{z}
9. return deleteMin(Q)
```

i=1

$x\in\Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

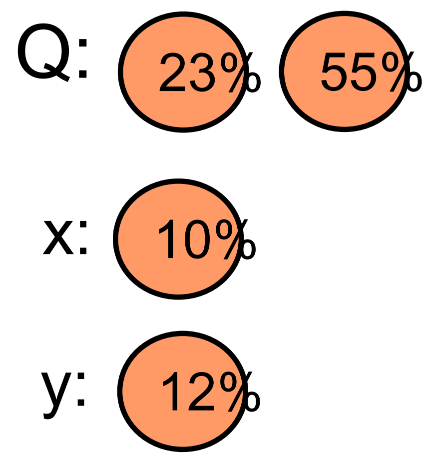


Gierige Algorithmen – Datenkompression

x∈Σ	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n ← | Σ |
2. Q ← Σ   /* Priority Queue bzgl. f[x] */
3. **for** i ← 1 **to** n-1 **do**
4.   x ← deleteMin(Q)
5.   y ← deleteMin(Q)
6.   z←**new** BinTree(x, f[x]+f[y],y)
7.   f[z]← f[x] + f[y]
8.   Q ← Q ∪{z}
9. **return** deleteMin(Q)

i=1

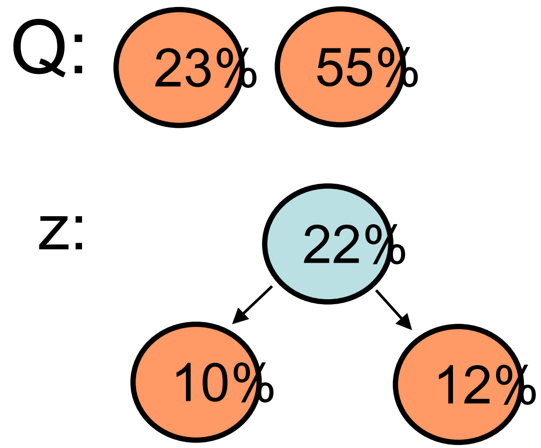


Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

i=1

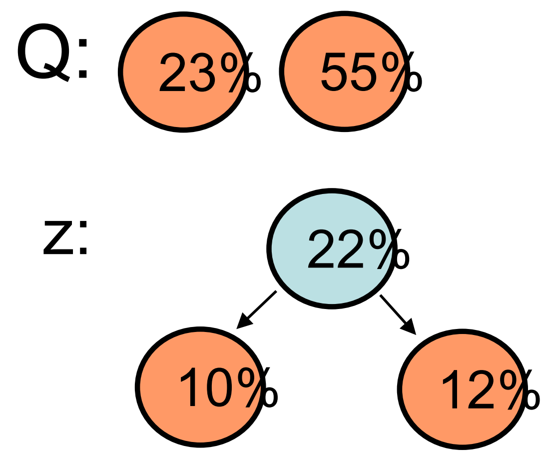


Gierige Algorithmen – Datenkompression

x∈Σ	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n ← | Σ |
2. Q ← Σ /* Priority Queue bzgl. f[x] */
3. **for** i ← 1 **to** n-1 **do**
4. x ← deleteMin(Q)
5. y ← deleteMin(Q)
6. z←**new** BinTree(x, f[x]+f[y],y)
7. f[z]← f[x] + f[y]
8. Q ← Q ∪{z}
9. **return** deleteMin(Q)

i=1



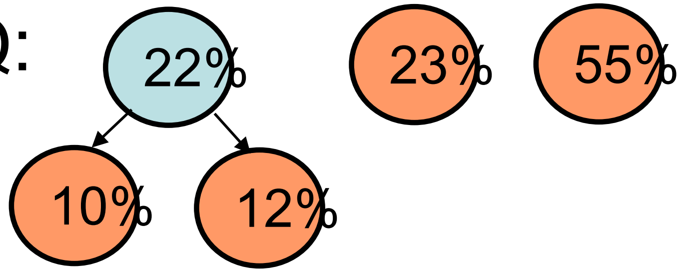
Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

i=1

Q:

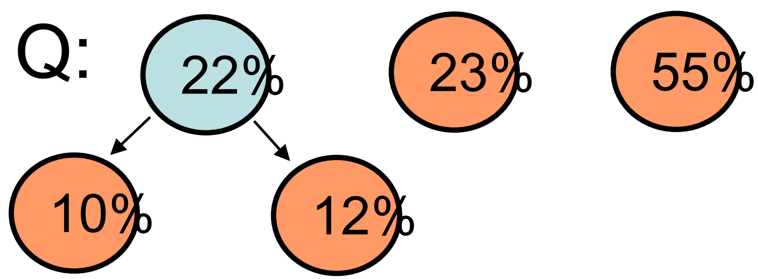


Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup${z}
9. **return** deleteMin(Q)

i=2

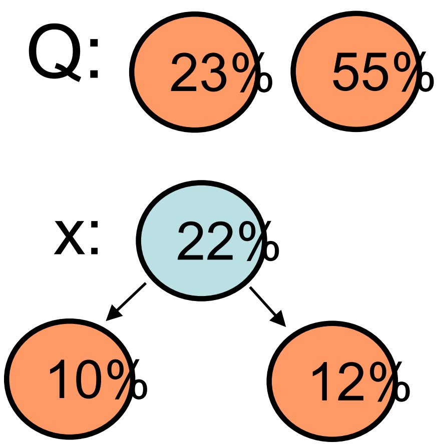


Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

i=2

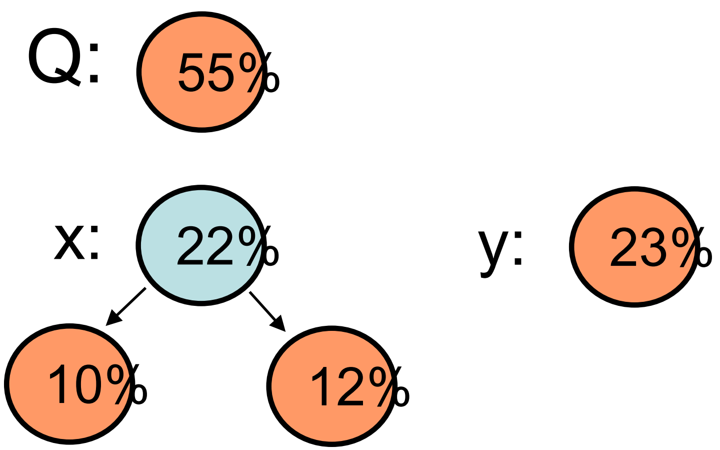


Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup${z}
9. **return** deleteMin(Q)

i=2

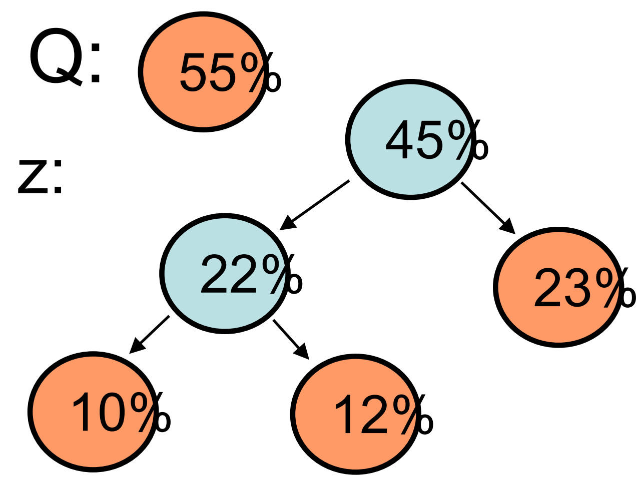


Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

i=2

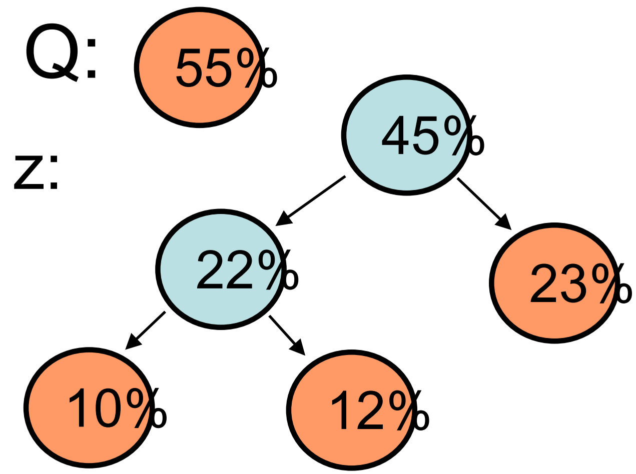


Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

i=2



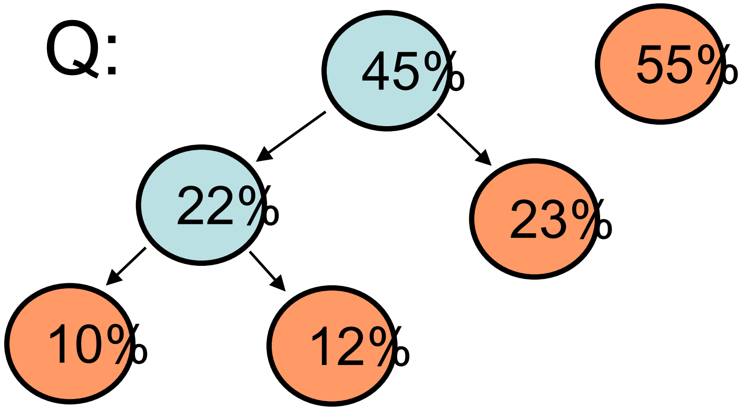
Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup${z}
9. **return** deleteMin(Q)

i=2

Q:



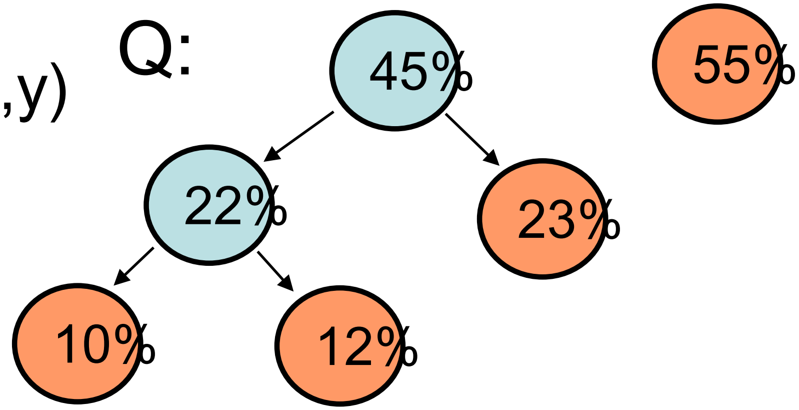
Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup${z}
9. **return** deleteMin(Q)

i=3

Q:

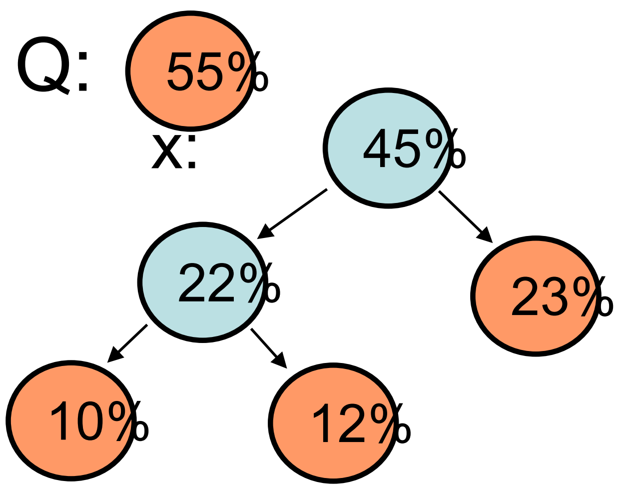


Gierige Algorithmen – Datenkompression

x∈Σ	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n ← | Σ |
2. Q ← Σ   /* Priority Queue bzgl. f[x] */
3. **for** i ← 1 **to** n-1 **do**
4.   x ← deleteMin(Q)
5.   y ← deleteMin(Q)
6.   z←**new** BinTree(x, f[x]+f[y],y)
7.   f[z]← f[x] + f[y]
8.   Q ← Q ∪{z}
9. **return** deleteMin(Q)

i=3

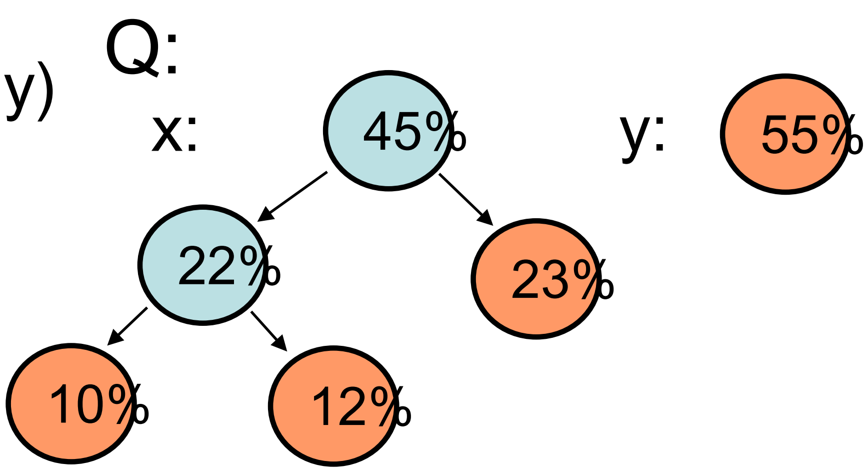


Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

i=3



Gierige Algorithmen – Datenkompression

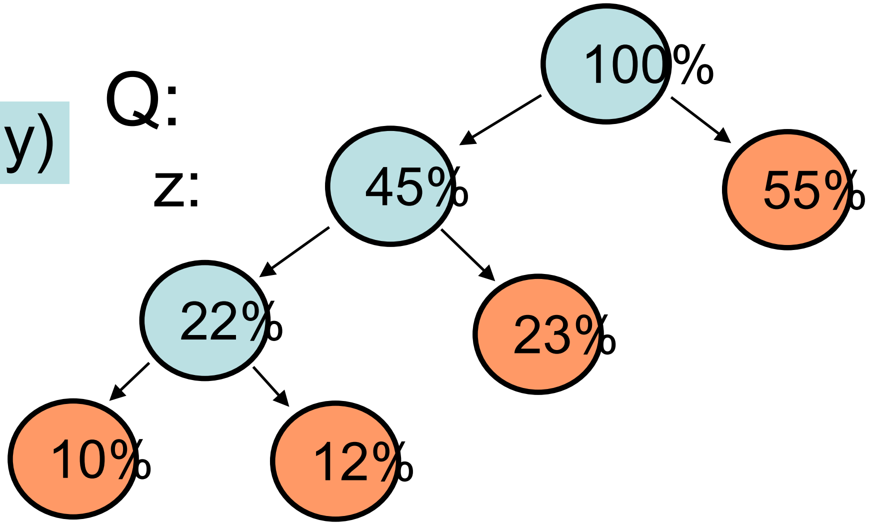
$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

i=3

Q:

z:

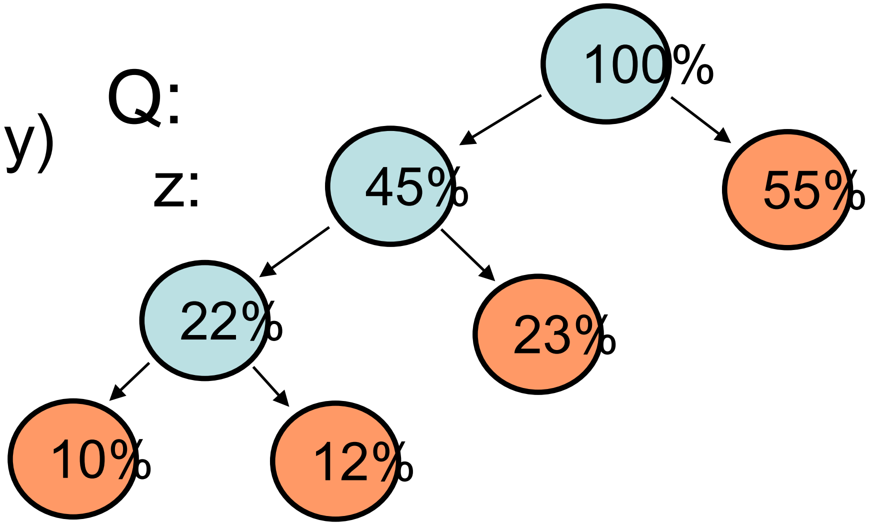


Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

i=3



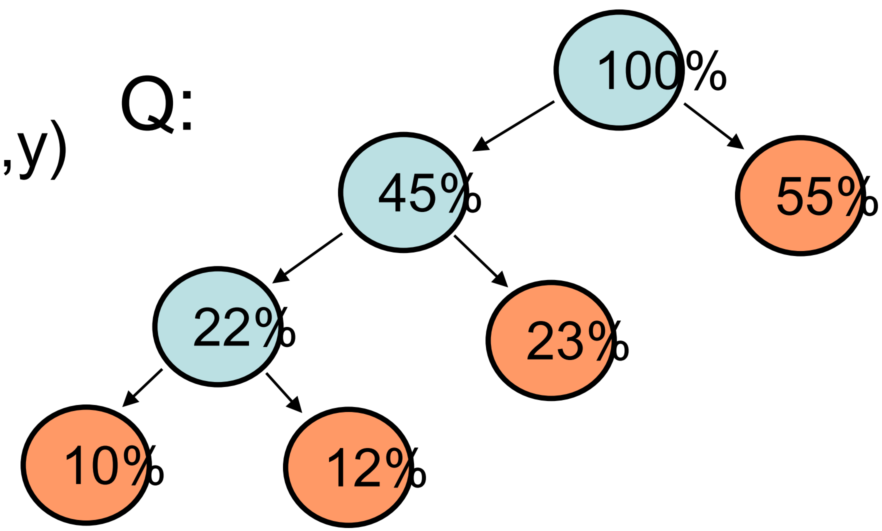
Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

i=3

Q:



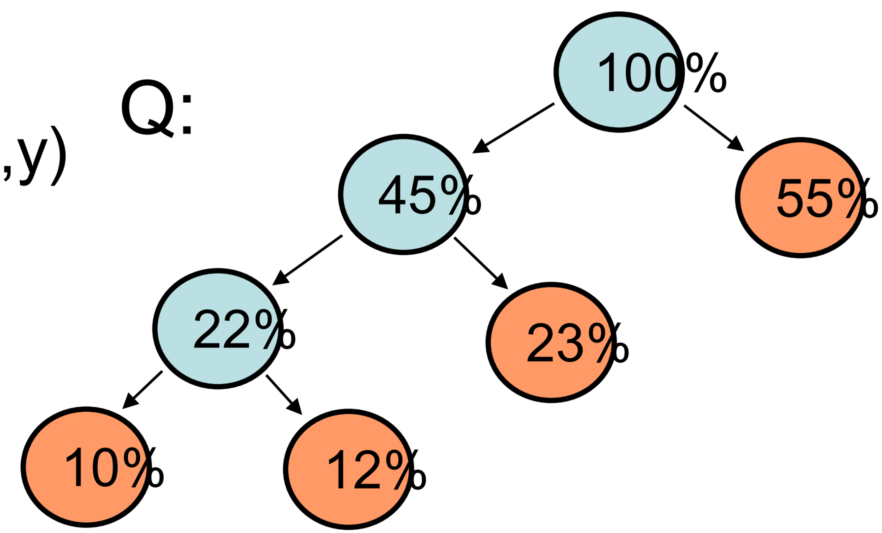
Gierige Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow$ Σ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup$\{z\}
9. **return** deleteMin(Q)

i=3

Q:



Gierige Algorithmen – Datenkompression

Satz 19.10

Algorithmus Huffman(Σ) berechnet eine optimale Präfix-Kodierung.

Beweis:

- Durch Induktion über Anzahl Symbole in Σ.
- $|\Sigma|=2$: Algo Huffman() offensichtlich optimal
- $n \rightarrow n+1$: seien x und y die Symbole mit kleinsten Frequenzen in Σ. Verschmelze x und y zu einem Symbol z mit $f[z]=f[x]+f[y]$ und sei $\Sigma' = (\Sigma \setminus \{x,y\}) \cup \{z\}$.
- Betrachte den durch Huffman(Σ') konstruierten Baum T' für Σ' und ersetze den Knoten z mit einem Knoten mit Kindern x und y. Der resultierende Baum T hat die Kosten
$$ABL(T) = \Sigma_{x \in \Sigma} f[x] \cdot Tiefe_T(x) = ABL(T') + f[x]+f[y].$$
- Angenommen, T sei nicht optimal für Σ, aber ein Baum T'' ist es. Nach Lemma 19.9 sind (o.B.d.A.) x und y Bruderknoten in T''. Deren Verschmelzung zu einem Knoten mit Symbol z ergibt dann einen Baum T''' für Σ' mit
$$ABL(T''') = ABL(T'')-f[x]-f[y] < ABL(T)-f[x]-f[y] = ABL(T')$$ Widerspruch!
- Wir wissen: Algo Huffman(Σ') konstruiert optimalen Baum für Σ'. Dann konstruiert Huffman(Σ) auch optimalen Baum für Σ.