

# Optimierung – Dynamische Programmierung

Algorithmen und Datenstrukturen  
VU 186.866, 5.5h, 8 ECTS, 2023S  
Letzte Änderung: 25. Mai 2023

Quiz

Vorlesungsfolien

ac  ALGORITHMS AND  
COMPLEXITY GROUP



Informatics

# Optimierung: Roadmap

## Branch-and-Bound

**Dynamische Programmierung:** Dynamische Programmierung kann dann eingesetzt werden, wenn das Problem aus vielen gleichartigen Teilproblemen besteht und eine optimale Lösung sich aus optimalen Lösungen der Teilprobleme zusammensetzt.

## Approximation(algorithmen)

## Heuristische Verfahren

# Grundlagen

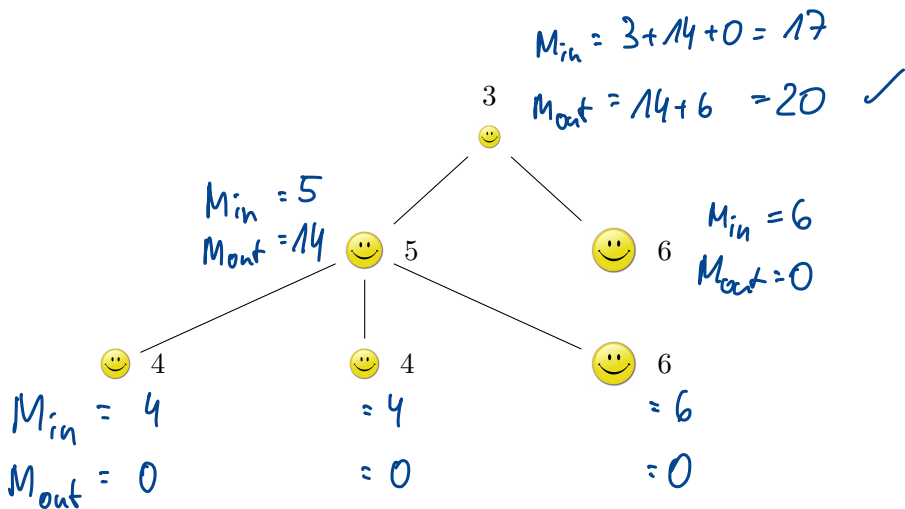
**Dynamische Programmierung:** Teile das Problem in eine Folge von überlappenden Teilproblemen auf und erstelle und speichere Lösungen für immer größere Teilprobleme unter Verwendung der abgespeicherten Lösungen.

**Optimalitätsprinzip von Bellman:** Dynamische Programmierung führt zu einem optimalen Ergebnis genau dann, wenn es sich aus den optimalen Ergebnissen der Subprobleme zusammensetzt.

**Effizienz:** Hängt von der Vorgehensweise bei der Aufteilung und Ermittlung der Lösungen für die einzelnen Teilprobleme ab.

**Wesentlicher Aspekt:** Speicherung (*memoization*) von Ergebnissen für Subprobleme zur Wiederverwendung.

# Beispiel: Weighted Independent Set auf Bäumen



# Geschichte der dynamischen Programmierung

**Bellman:** [1950er] Leistete Pionierarbeit bei der systematischen Untersuchung der dynamischen Programmierung.

## Etymologie:

- Dynamische Programmierung = Zeitablauf planen.
- Verteidigungsminister war ablehnend gegenüber mathematischer Forschung.
- Bellman suchte einen eindrucksvollen Namen, um eine Konfrontation zu vermeiden.

„it's impossible to use dynamic in a pejorative sense“  
„something not even a Congressman could object to“

Referenz: Bellman, R. E. Eye of the Hurricane, An Autobiography.

# Anwendung der dynamischen Programmierung

## Bereiche:

- Bioinformatik
- Informationstheorie
- Operations Research
- Informatik: Theorie, Grafik, Künstliche Intelligenz, Compilerbau ...

## Einige bekannte Algorithmen:

- Bellman-Ford-Algorithmus für das Finden kürzester Pfade in Graphen
- Effiziente Methode für das Rucksack-Problem
- Needleman-Wunsch und Smith-Waterman Algorithmen für Genomsequenz-Alignment

# Überblick

Einführendes Beispiel: Fibonacci

Gewichtetes Interval Scheduling

Segmented Least Squares

Rucksackproblem

Kürzeste Pfade

## Einführendes Beispiel



# Fibonacci-Zahlen

1, 1, 2, 3, 5, 8, 13, ...

$$F_{n-1} + F_{n-2}$$

Folge von Fibonacci-Zahlen:  $F_1 = F_2 = 1$   $F_n = F_{n-1} + F_{n-2} \quad \forall n \geq 3$

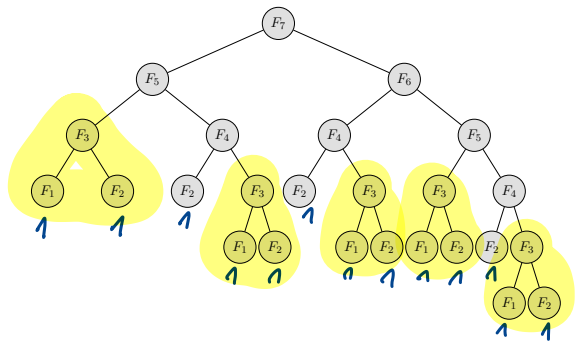
Einfacher rekursiver Algorithmus:

```
Fibonacci(n):  
if n = 1 oder n = 2  
    return 1  
else  
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```

# Fibonacci-Zahlen: Laufzeit

$T_i = \# \text{ Blätter im Aufrufbaum}$   
 $\# \text{ rek. Aufrufe} = \# \text{ innere Knoten} = T_i - 1$

- Gesamtanzahl der Aufrufe für die  $i$ -te Fibonacci-Zahl entspricht der  $i$ -ten Fibonacci-Zahl (Beispiele:  $F_{10} = 55$ ,  $F_{20} = 6765$ ,  $F_{30} = 832040$ ,  $F_{40} = 102334155$ ).
- Exponentielle Zeitkomplexität, da  $F_n = F_{n-1} + F_{n-2} \geq 2F_{n-2}$  und folglich  $F_n \geq 2^{\frac{n}{2}-1}$  (mit  $F_2 = 2^0$ ) und daher  $F_n \geq (\sqrt{2})^{n-2}$



$$T_n \geq 2T_{n-2} \geq 2 \cdot 2 \cdot T_{n-4} \dots \geq 2^{\frac{n-2}{2}} = \left(2^{\frac{1}{2}}\right)^{n-2} = \sqrt{2}^{n-2} \leadsto \text{untere Schranke } \Omega(\sqrt{2}^n)$$

# Dynamische Programmierung (Rekursiv)

**Speicherung:** Die berechneten Fibonacci-Zahlen zwischenspeichern (z.B. in einem Array  $F$ ) und in der Berechnung wiederverwenden.

```
for  $i \leftarrow 1$  bis  $n$   
   $F[i] \leftarrow$  leer
```

„top down“

```
Fibonacci( $n$ ):
```

↪ memoization

```
if  $F[n]$  ist leer
```

↪ nur, wenn  $F_n$  noch nicht benutzt wurde

```
  if  $n = 1$  oder  $n = 2$ 
```

```
     $F[n] \leftarrow 1$ 
```

```
  else
```

```
     $F[n] \leftarrow$  Fibonacci( $n - 1$ ) + Fibonacci( $n - 2$ )
```

```
  return  $F[n]$ 
```

**Laufzeit:**  $O(n)$  (maximal zwei rekursive Aufrufe pro Arrayeintrag)

# Dynamische Programmierung (Iterativ)

„bottom-up“

**Speicherung:** Die berechneten Fibonacci-Zahlen zwischenspeichern (z.B. in einem Array  $F$ ) und in der Berechnung wiederverwenden.

```
Linear-Fibonacci( $n$ ):  
F[1]  $\leftarrow$  1  
F[2]  $\leftarrow$  1  
for  $i \leftarrow 3$  bis  $n$   
    F[ $i$ ]  $\leftarrow$  F[ $i - 1$ ] + F[ $i - 2$ ]  
return F[ $n$ ]
```

$\rightarrow$  keine Rekursion

**Laufzeit:**  $O(n)$  (konstanter Aufwand für jeden Schleifendurchlauf)

## Quiz

Frage 1: In welchem Bereich bewegt sich der Beschleunigungsfaktor des iterativen Algorithmus Linear-Fibonacci gegenüber dem einfachen rekursiven Algorithmus Fibonacci bei der Berechnung von  $F_{40}$ ?

$$F_{40} = 102\,334\,155$$

- zwischen 10 und 1.000
- zwischen 1.000 und 100.000
- zwischen 100.000 und 10.000.000 ✓
- zwischen 10.000.000 und 1.000.000.000

gemessen ca. 250.000

## Quiz Auflösung

Frage 1: In welchem Bereich bewegt sich der Beschleunigungsfaktor des iterativen Algorithmus Linear-Fibonacci gegenüber dem einfachen rekursiven Algorithmus Fibonacci bei der Berechnung von  $F_{40}$ ?

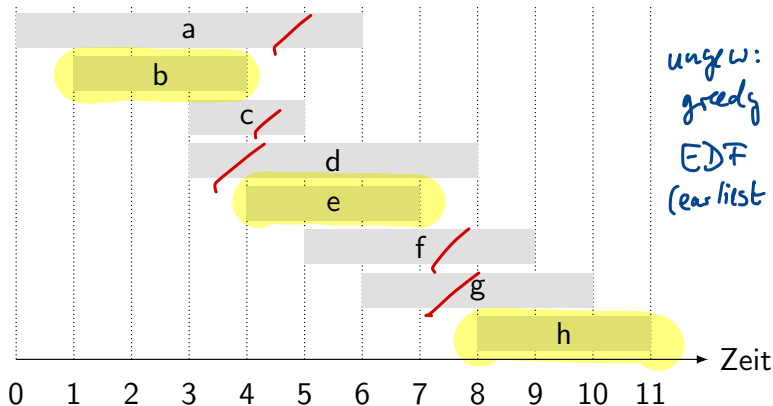
- ✗ zwischen 10 und 1.000
- ✗ zwischen 1.000 und 100.000
- ✓ zwischen 100.000 und 10.000.000
- ✗ zwischen 10.000.000 und 1.000.000.000

## Gewichtetes Interval Scheduling

# Gewichtetes Interval Scheduling

## Gewichtetes Interval Scheduling:

- Job  $j$  startet zum Zeitpunkt  $s_j$ , endet zum Zeitpunkt  $f_j$  und hat ein **Gewicht**  $w_j > 0$ . *→ z.B. Profit, wenn ausgeführt*
- Zwei Jobs sind **kompatibel**, wenn sie sich nicht überlappen.
- Ziel: Finde eine Teilmenge **maximalen Gewichts** von paarweise kompatiblen Jobs.



ungew:  
greedy-Algo.  
EDF  
(earliest deadline first)

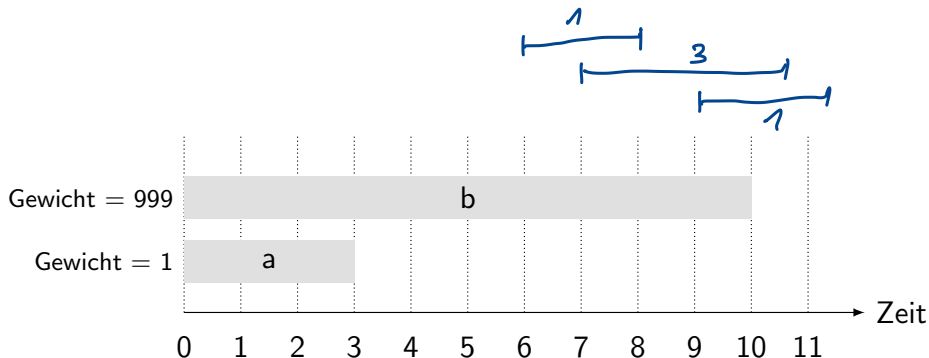


# Interval Scheduling: Rückblick

**Wiederholung:** Greedy-Algorithmus funktioniert, wenn alle Gewichte gleich 1 sind.

- Berücksichtige Jobs in aufsteigender Reihenfolge der Beendigungszeit.
- Füge Job zur Teilmenge hinzu, wenn er kompatibel mit dem zuvor ausgewählten Job ist.

**Beobachtung:** Greedy-Algorithmus scheitert, wenn beliebige Gewichte erlaubt sind.



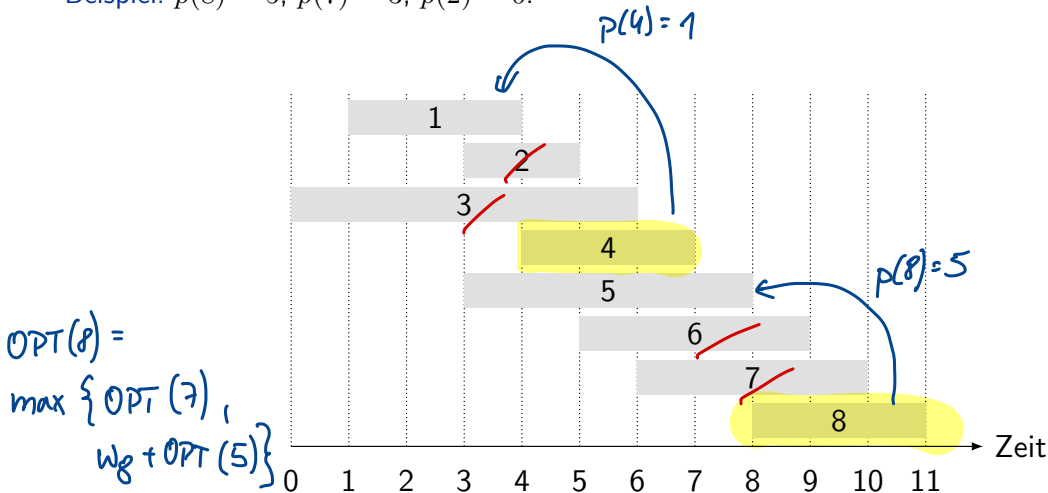
# Gewichtetes Interval Scheduling

Jobs  $J_1, J_2, \dots, J_n$

Notation: Ordne Jobs aufsteigend sortiert nach Beendigungszeit:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

Definition:  $p(j) =$  größter Index  $i < j$ , sodass Job  $i$  kompatibel zu Job  $j$  ist.

Beispiel:  $p(8) = 5$ ,  $p(7) = 3$ ,  $p(2) = 0$ .



# Dynamische Programmierung: Binäre Auswahl

Notation:  $OPT(j)$  = Wert der optimalen Lösung für das Problem, bestehend aus den Jobs  $1, 2, \dots, j$ .  $J_1, \dots, J_j$

Wir unterscheiden zwei Fälle:

- Fall 1:  $OPT(j)$  wird erreicht mit einer Lösung, die den Job  $j$  enthält.
- Fall 2:  $OPT(j)$  wird erreicht mit einer Lösung, die den Job  $j$  nicht enthält.

Konsequenz:

$$\downarrow \\ OPT(j) = OPT(j-1)$$

- Fall 1: Die Lösung kann nicht die inkompatiblen Jobs  $\{p(j) + 1, p(j) + 2, \dots, j - 1\}$  enthalten. Daher gilt dann  $OPT(j) = w_j + OPT(p(j))$ .
- Fall 2: Es gilt  $OPT(j) = OPT(j - 1)$ , da wir wissen, dass die Lösung den Job  $j$  nicht enthält. Also gilt:

$$OPT(j) = \begin{cases} 0 & \text{wenn } j = 0 \\ \max \{w_j + OPT(p(j)), OPT(j - 1)\} & \text{sonst} \end{cases}$$

# Gewichtetes Interval Scheduling: Brute-Force-Ansatz

## Brute-Force-Algorithmus:

- Eingabe:  $n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$
- Sortiere Jobs nach Beendigungszeit, sodass  $f_1 \leq f_2 \leq \dots \leq f_n$ .
- Berechne  $p(1), p(2), \dots, p(n)$

Compute-Opt( $j$ ):

**if**  $j = 0$

**return** 0

**else**

**return**  $\max(w_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1))$

Fall 1

Fall 2



## Gewichtetes Interval Scheduling: Speicherung *Memorization*

**Speicherung:** Speichere Ergebnisse jedes Teilproblems in einem Cache. Berechnung nur, wenn noch nicht gespeichert.

Allgemein:

- Eingabe:  $n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$
- Sortiere Jobs nach Beendigungszeit, sodass  $f_1 \leq f_2 \leq \dots \leq f_n$ .
- Berechne  $p(1), p(2), \dots, p(n)$

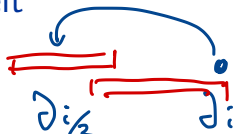
```
for  $j \leftarrow 1$  bis  $n$   
   $M[j] \leftarrow$  leer  
 $M[0] \leftarrow 0$ 
```

```
M-Compute-Opt( $j$ ):  
  if  $M[j]$  ist leer  
     $M[j] \leftarrow \max(w_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j - 1))$   
  return  $M[j]$ 
```

*↪ sehr ähnl. zum rekursiven Alg.  
aber sehr viel effizient!*

■ *globales Array*

# Gewichtetes Interval Scheduling: Laufzeit



in jedem  
Vert  $p(i)$   
eine bin. Suche

**Behauptung:** Version mit Speicherung benötigt  $O(n \log n)$  Zeit.

- Sortiere nach Beendigungszeit:  $O(n \log n)$ .
- Berechne  $p(\cdot)$ :  $O(n \log n)$  mittels binärer Suche (für jedes Intervall) auf der nach Beendigungszeit sortierten Folge. *benötigen  $p(1), \dots, p(n) \rightarrow n$  Werte*
- M-Compute-Opt( $j$ ): Jeder Aufruf benötigt  $O(1)$  Zeit (ohne die Rekursion) und
  - (i) liefert entweder einen existierenden Wert  $M[j]$
  - (ii) oder berechnet einen neuen Eintrag  $M[j]$  und macht zwei rekursive Aufrufe.
- Maß für den Fortschritt  $\varphi$  = die Anzahl der nicht leeren Einträge in  $M[\cdot]$ .
  - Am Anfang gilt  $\varphi = 0$ , danach  $\varphi \leq n$ .
  - (ii) Erhöht  $\varphi$  um 1.  *$\rightarrow$  Fall (ii) tritt nur  $n$ -mal auf  $\Rightarrow$   $\leftarrow 2n$  rek. Aufrufe*
- Die gesamte Laufzeit von M-Compute-Opt( $n$ ) ist  $O(n)$ .

# Gewichtetes Interval Scheduling: Bottom-up

Bottom-up dynamische Programmierung: Iterative Lösung.

Allgemein:

- Eingabe:  $n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$ .
- Sortiere Jobs nach Beendigungszeit, sodass  $f_1 \leq f_2 \leq \dots \leq f_n$ .
- Berechne  $p(1), p(2), \dots, p(n)$

Idee: alle Vorgänger-  
einträge sind  
schon berechnet  
→ muss aber nur auf  
kleinere Teilprobleme  
zurückgreifen

Iterative-Compute-Opt():

$M[0] \leftarrow 0$

for  $j \leftarrow 1$  bis  $n$

$M[j] \leftarrow \max(w_j + M[p(j)], M[j-1])$

es gilt  $p(j) < j$ ,  $j-1 < j$

↓ ↓

Laufzeit: Die Laufzeit von Iterative-Compute-Opt liegt in  $O(n)$  (Schleife von 1 bis  $n$ ).

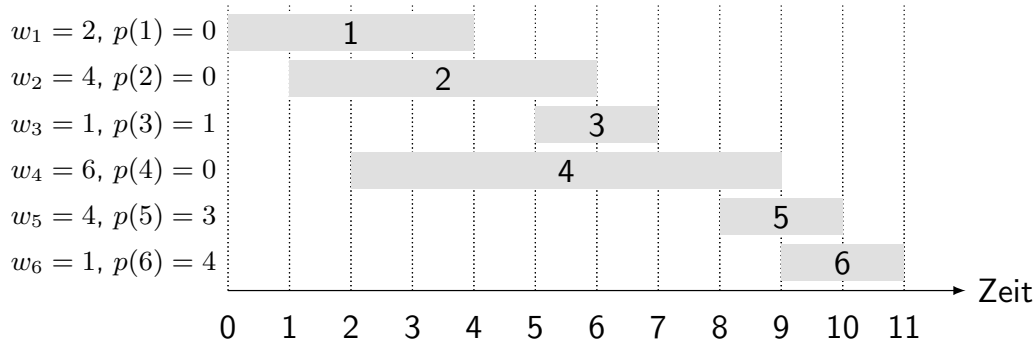
inkl. Vorberechnung von  $p(\cdot)$  und Sortieren dann  $O(n \log n)$



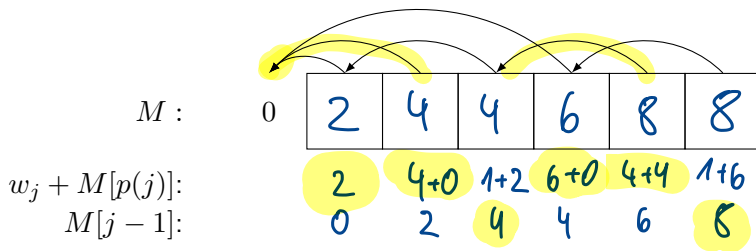
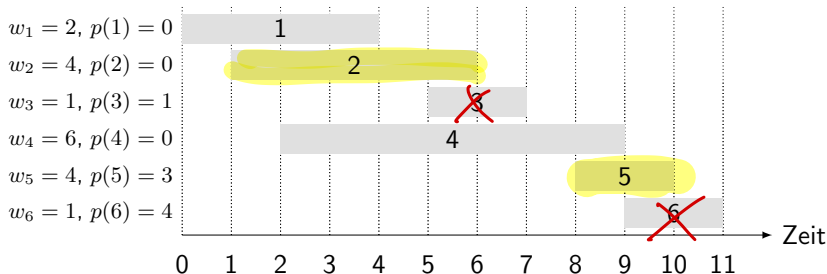
# Gewichtetes Interval Scheduling: Beispiel

Gegeben:

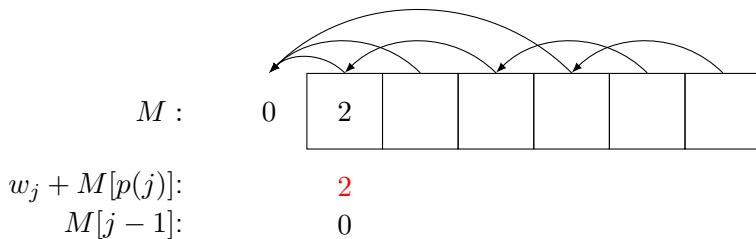
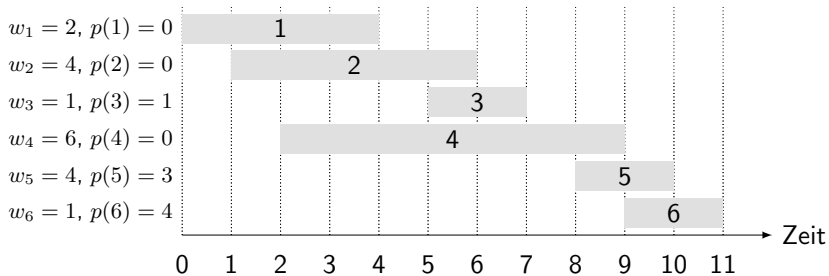
- $n = 6$  Jobs mit Gewichten  $w_i, i = 1 \dots n$ .
- Jobs sind schon sortiert nach Beendigungszeit.



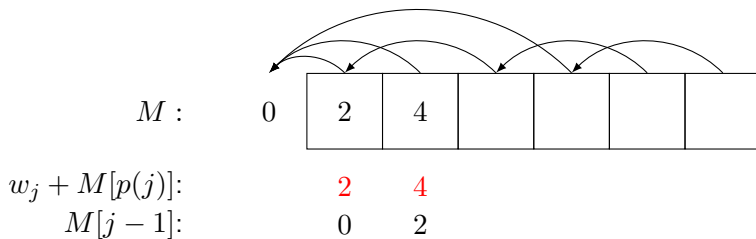
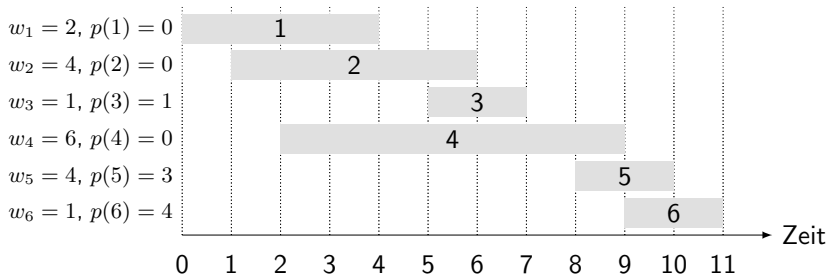
# Gewichtetes Interval Scheduling: Beispiel



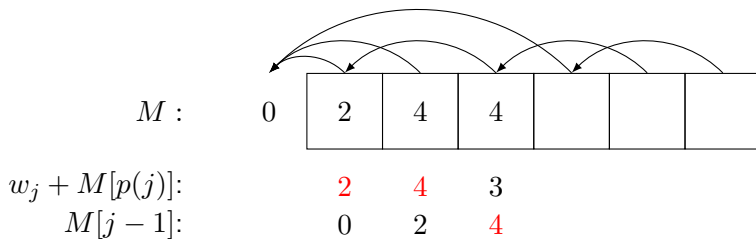
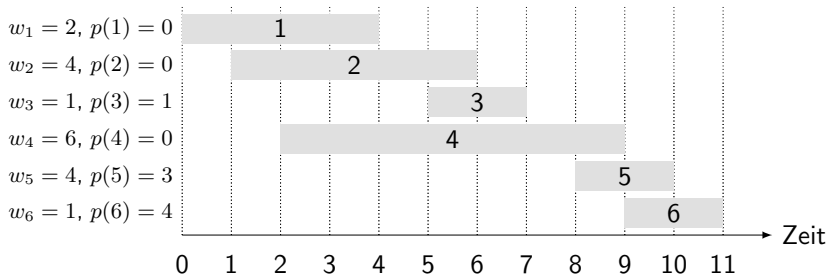
# Gewichtetes Interval Scheduling: Beispiel



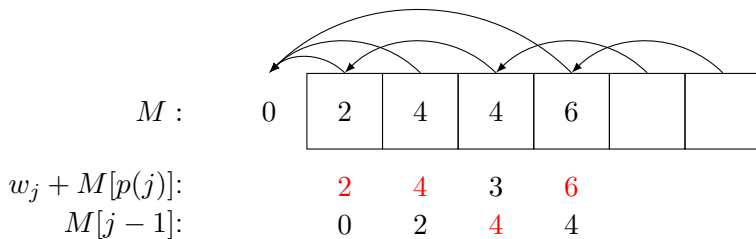
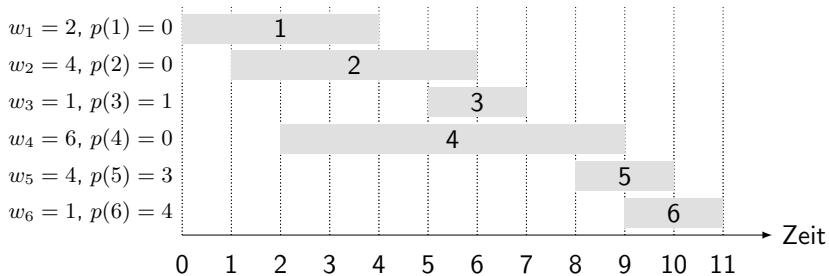
# Gewichtetes Interval Scheduling: Beispiel



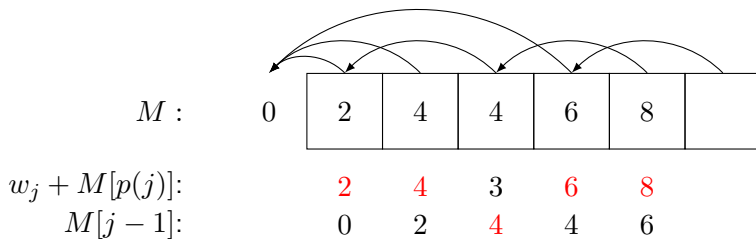
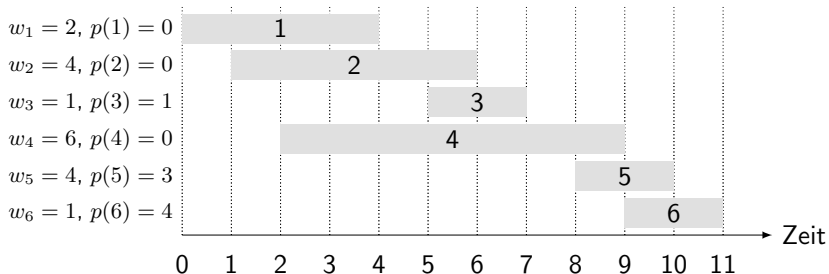
# Gewichtetes Interval Scheduling: Beispiel



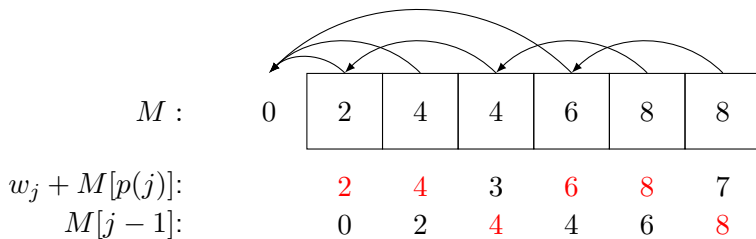
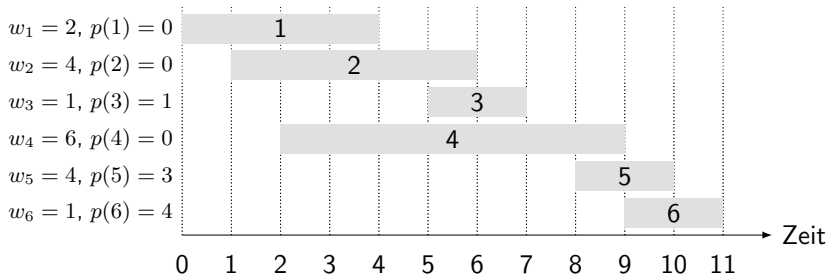
# Gewichtetes Interval Scheduling: Beispiel



# Gewichtetes Interval Scheduling: Beispiel

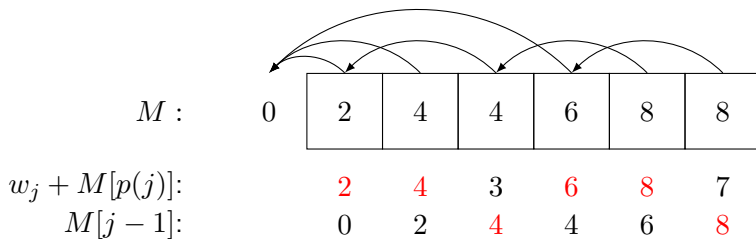


# Gewichtetes Interval Scheduling: Beispiel






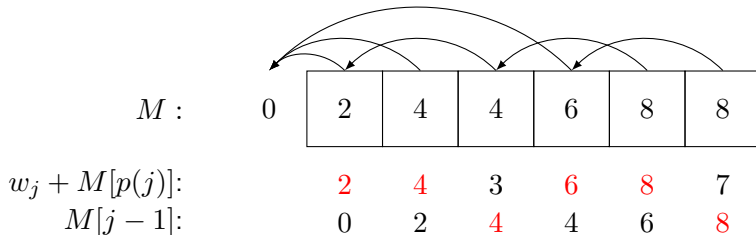
# Quiz



Frage 2: Welche Jobs bilden die Lösungsmenge des Beispiels?

- {2, 5} 
- {1, 2, 4, 5}
- {4}
- {3, 6}

## Quiz Auflösung



Frage 2: Welche Jobs bilden die Lösungsmenge des Beispiels?

- ✓ {2, 5}
- ✗ {1, 2, 4, 5}
- ✗ {4}
- ✗ {3, 6}

# Gewichtetes Interval Scheduling: Finden einer Lösung

**Frage:** Algorithmus berechnet den optimalen Wert. Wie bekommen wir aber die Lösung?

**Antwort:** Durch eine Nachbearbeitung (Backtracking).

**Ablauf:**

- M-Compute-Opt(n) oder Iterative-Compute-Opt(n) ausführen
- Find-Solution(n) ausführen

```
Find-Solution(j):
```

```
if  $j = 0$ 
```

```
    Keine Ausgabe
```

```
elseif  $w_j + M[p(j)] > M[j - 1]$ 
```

```
    Gib  $j$  aus
```

```
    Find-Solution( $p(j)$ )
```

```
else
```

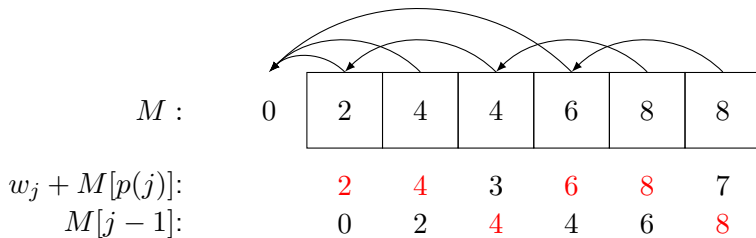
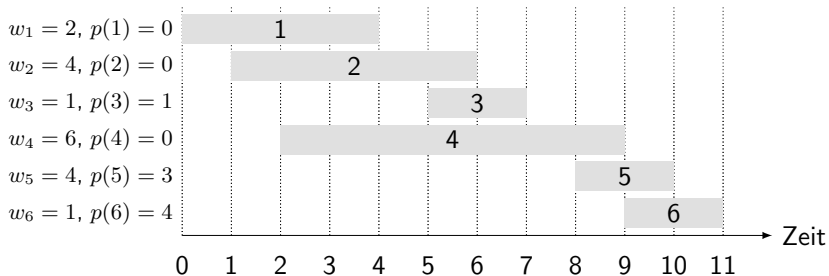
```
    Find-Solution( $j - 1$ )
```

} haben  $j_j$  ausgewählt

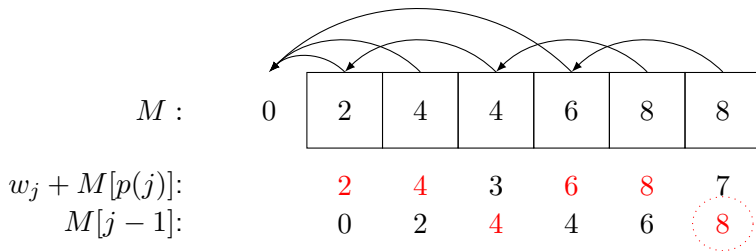
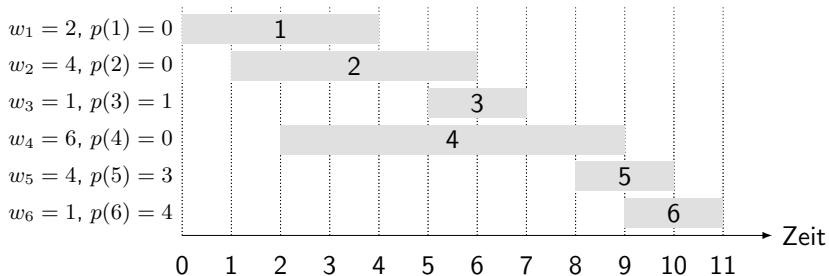
}  $j_j$  nicht ausgewählt

- Anzahl der rekursiven Aufrufe  $\leq n \Rightarrow$  Laufzeit  $O(n)$ .

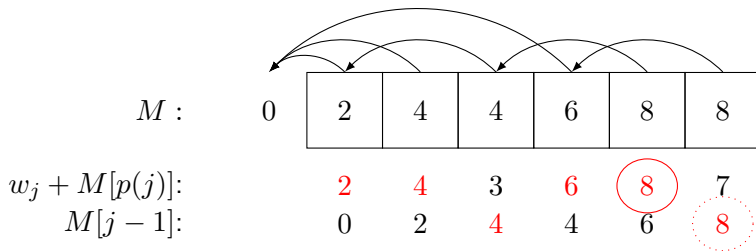
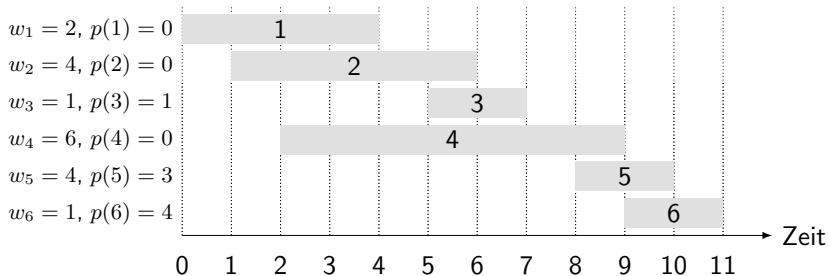
# Gewichtetes Interval Scheduling: Beispiel



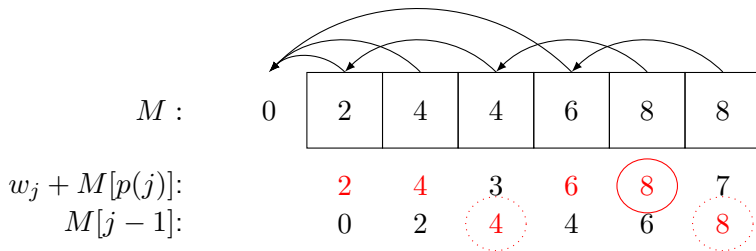
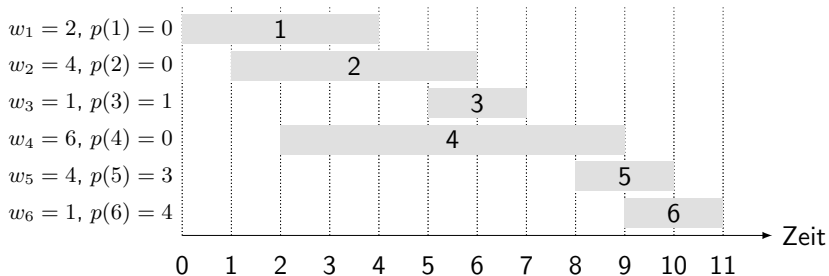
# Gewichtetes Interval Scheduling: Beispiel



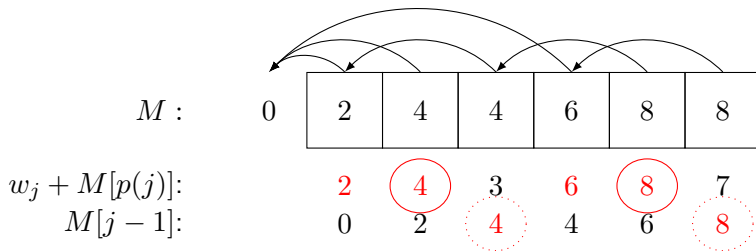
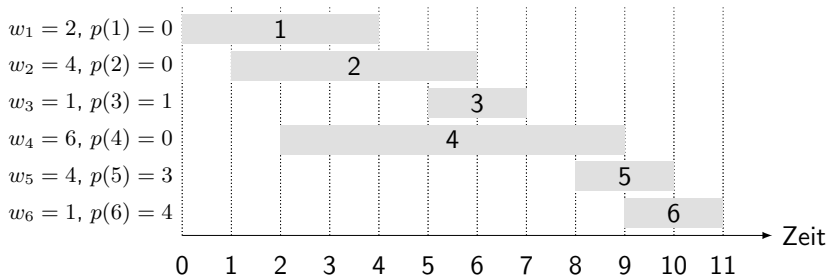
# Gewichtetes Interval Scheduling: Beispiel



# Gewichtetes Interval Scheduling: Beispiel

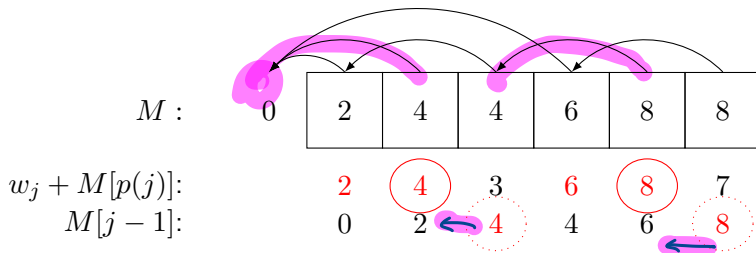
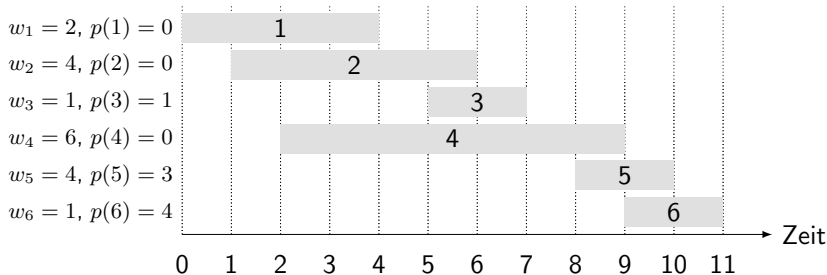


# Gewichtetes Interval Scheduling: Beispiel





# Gewichtetes Interval Scheduling: Beispiel



Ergebnis: 2 und 5.

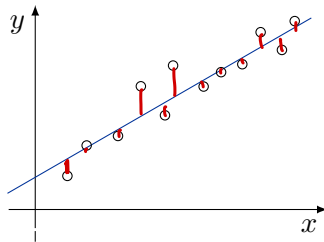
## Segmented Least Squares

# Least Squares

↪ Regressionsgerade

- Fundamentales Problem in der Statistik und der Numerischen Analyse.
- Gegeben:  $n$  Punkte in der Ebene:  $(x_1, y_1), \dots, (x_n, y_n)$ .
- Finde eine Gerade  $y = ax + b$ , welche die Summe des quadrierten Fehlers minimiert:

$$\text{Err} = \sum_{i=1}^n (y_i - (ax_i + b))^2$$



Analytische Lösung: der minimale Fehler ist erreicht, wenn

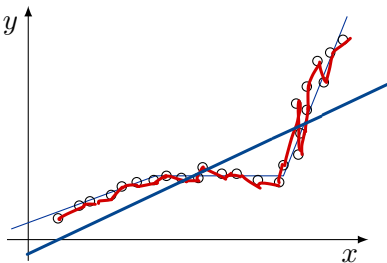
$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i) (\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

# Segmented Least Squares

- Punkte durch eine Folge von Geradensegmenten annähern.
- Gegeben:  $n$  Punkte in der Ebene  $(x_1, y_1), \dots, (x_n, y_n)$  so dass  $x_1 < x_2 < \dots < x_n$ ,
- finde eine Folge von Geraden welche eine bestimmte Funktion  $f(x)$  minimiert.

Frage: Was ist eine angemessene Wahl für  $f(x)$ ? Die Funktion  $f(x)$  sollte sowohl Genauigkeit als auch Sparsamkeit gewährleisten.

■ Höhe des Fehlers ■ Anzahl der Geraden



sehr genau,  
nicht sparsam

nur sparsam  
nicht genau

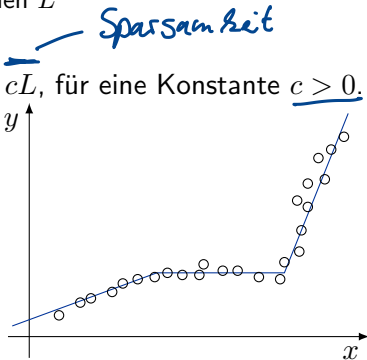
# Segmented Least Squares

- Punkte durch eine Folge von Geradensegmenten annähern.
- Gegeben:  $n$  Punkte in der Ebene  $(x_1, y_1), \dots, (x_n, y_n)$  so dass  $x_1 < x_2 < \dots < x_n$ ,
- finde eine Folge von Geraden welche:
  - die Summe der quadrierten Fehler  $E$  in jedem Segment
  - die Anzahl der Geraden  $L$

minimiert.

- Tradeoff Funktion:  $E + cL$ , für eine Konstante  $c > 0$ .

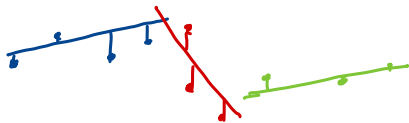
Genauigkeit



bestimmt den trade-off

hier:  $L=3$

# Dynamischer Ansatz: Segmented Least Squares



## Notation

- $OPT(j)$  = minimale Kosten für die Punkte  $p_1, p_{i+1}, \dots, p_j$
- $e(i, j)$  = minimale Summe des quadrierten Fehlers für  $p_i, p_{i+1}, \dots, p_j$

↳ falls  $p_i, \dots, p_j$  durch ein Segm. approximiert  
→ s. Folie 39

## Berechnen von $OPT(j)$ :

- letztes Segment nutzt die Punkte  $p_i, p_{i+1}, \dots, p_j$  für ein bestimmtes  $i$   
(wir suchen das beste  $i$ )
- Kosten =  $OPT(i-1) + e(i, j) + c$

$$OPT(j) = \begin{cases} 0 & \text{falls } j = 0 \\ \min_{1 \leq i \leq j} \{OPT(i-1) + e(i, j) + c\} & \text{sonst} \end{cases}$$

↑  
alle Mögl.  
für Start des letzten  
Segments

neue  
Fehler  
von  $i$  bis  $j$

Kosten für ein neues Segm

# Segmented Least Squares: Algorithmus

bottom-up

```
Segmented-Least-Squares(  $P = \{p_1, p_2, \dots, p_n\}$  )
   $M[0] = 0$ 
   $O(n^2)$  Iterationen { for  $j \leftarrow 1$  bis  $n$ 
                       for  $i \leftarrow 1$  bis  $j$ 
                       berechne Fehler  $e(i, j)$  für Punkte  $p_i, \dots, p_j$  }  $O(n)$  Laufzeit
   $O(n)$  Iterationen [ for  $j \leftarrow 1$  to  $n$ 
                        $M[j] = \min_{1 \leq i \leq j} (M[i-1] + e(i, j) + c)$ 
                       return  $M[n]$   $O(n)$  Mögl. ]  $O(n^2)$ 
```

Laufzeit.  $O(n^3)$ .

- Flaschenhals ist das Berechnen von  $e(i, j)$  für  $O(n^2)$  Paare,  $O(n)$  pro Paar mit der Formel für Least Squares.
- Finden einer Lösung analog zu Interval Scheduling durch Rückverfolgen der Minimierung
- Kann mithilfe geschickterer Vorberechnung und Wiederverwendung von Zwischenergebnissen zu  $O(n^2)$  verbessert werden.

# Zwischenfazit

- wir betrachten polynomielle Anzahl Teilprobleme (Arraygröße)
- optimale Lösung lässt sich aus opt. Lösungen geeigneter Teilprobleme zusammensetzen
- Teilprobleme lassen sich von klein nach groß aufzählen und rekursiv lösen, d. h.

for  $j = 1, \dots, n$

$M[j] = f(M[1], \dots, M[j-1])$  // z.B. binäre Auswahl oder lineare Auswahl

return  $M[n]$

- Rekonstruktion der besten Lösung (nicht nur des Wertes) durch Backtracking