

Algorithmen und Datenstrukturen

Vorlesung #06 – Dynamische Programmierung

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

22 · Mai · 2019



- ▶ Rückschau: *Divide-and-Conquer*
- ▶ Generelles Prinzip der **Dynamischen Programmierung** (*dynamic programming*)
- ▶ Entwicklung von Ansätzen mit dynamischer Programmierung in drei Beispielen:
 - ▶ Gewichtete Intervallauswahl (*Weighted Interval Scheduling*)
 - ▶ 0/1-Rucksack Problem (*0/1-Knapsack Problem*)
 - ▶ Editierdistanz
- ▶ Zwischendurch: P und NP

- Das **Divide-and-Conquer** Paradigm zur Lösung von Optimierungsproblemen geht wie folgt vor:
 - ▶ Zerlege das Problem in disjunkte Teilprobleme.
 - ▶ Wenn ein Teilproblem klein genug ist, löse es direkt;
 - ▶ Andernfalls benutze Rekursion, um eine Lösung zu erhalten.
 - ▶ Kombiniere Lösungen von Teilproblemen schrittweise zu Lösungen der jeweils übergeordneten Probleme.

- Das **Divide-and-Conquer** Paradigm zur Lösung von Optimierungsproblemen geht wie folgt vor:
 - ▶ Zerlege das Problem in disjunkte Teilprobleme.
 - ▶ Wenn ein Teilproblem klein genug ist, löse es direkt;
 - ▶ Andernfalls benutze Rekursion, um eine Lösung zu erhalten.
 - ▶ Kombiniere Lösungen von Teilproblemen schrittweise zu Lösungen der jeweils übergeordneten Probleme.
- **Beispiele:**
 - ▶ Binäre Suche, *Mergesort*, Strassen-Algorithmus für Matrixmultiplikation

Generelles Prinzip der Dynamischen Programmierung

- ▶ Heute beschäftigen wir uns mit dem Lösungsparadigma **Dynamisches Programmieren**.
- ▶ Die Bezeichnung ›dynamisches Programmieren‹ ist recht unspezifisch und ist auf *Planung über die Zeit* zurückzuführen.
- ▶ Richard Bellman, der Namensgeber, hat den Ausdruck wohl aus strategischen Gründen gewählt.
- ▶ Das Paradigma der **Dynamischen Programmierung** (*dynamical programming*) geht folgendermaßen vor:

Generelles Prinzip der Dynamischen Programmierung

- ▶ Heute beschäftigen wir uns mit dem Lösungsparadigma **Dynamisches Programmieren**.
- ▶ Die Bezeichnung ›dynamisches Programmieren‹ ist recht unspezifisch und ist auf *Planung über die Zeit* zurückzuführen.
- ▶ Richard Bellman, der Namensgeber, hat den Ausdruck wohl aus strategischen Gründen gewählt.
- ▶ Das Paradigma der **Dynamischen Programmierung** (*dynamical programming*) geht folgendermaßen vor:
 - ▶ Für die Bestimmung der Lösung wird eine rekursive Formel aufgestellt, die das Problem auf die Lösung von Teilproblemen zurückführt.
 - ▶ Im Gegensatz zu *Divide-and-Conquer* müssen die Teilprobleme nicht disjunkt sein.
 - ▶ Die Teillösungen werden in einer Tabelle gespeichert und benutzt, um immer größere Problem zu lösen.
 - ▶ Üblicherweise wird die Lösung bei dynamischer Programmierung *bottom-up* bestimmt. In einigen Fällen kann ein *top-down* Ansatz günstiger sein.

Einführendes Beispiel: Fibonacci Zahlen

- ▶ Zur Einführung der Techniken des dynamischen Programmierens betrachten wir die effiziente Bestimmung n -ten **Fibonacci Zahl** F_n .
- ▶ Die Fibonacci Zahlen sind rekursiv definiert:

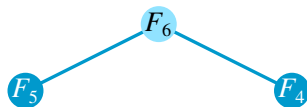
F_6

$$F_n = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ F_{n-1} + F_{n-2} & \text{für } n \geq 2 \end{cases}$$

Einführendes Beispiel: Fibonacci Zahlen

- ▶ Zur Einführung der Techniken des dynamischen Programmierens betrachten wir die effiziente Bestimmung n -ten **Fibonacci Zahl** F_n .
- ▶ Die Fibonacci Zahlen sind rekursiv definiert:

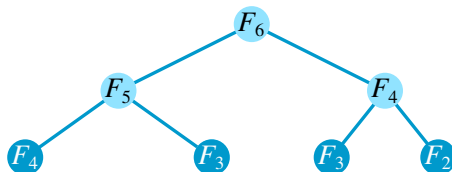
$$F_n = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ F_{n-1} + F_{n-2} & \text{für } n \geq 2 \end{cases}$$



Einführendes Beispiel: Fibonacci Zahlen

- ▶ Zur Einführung der Techniken des dynamischen Programmierens betrachten wir die effiziente Bestimmung n -ten **Fibonacci Zahl** F_n .
- ▶ Die Fibonacci Zahlen sind rekursiv definiert:

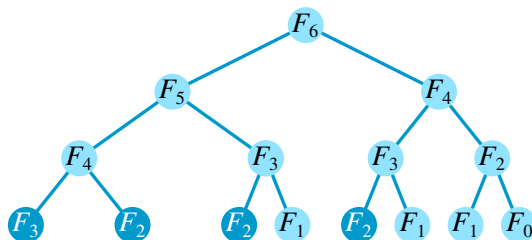
$$F_n = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ F_{n-1} + F_{n-2} & \text{für } n \geq 2 \end{cases}$$



Einführendes Beispiel: Fibonacci Zahlen

- ▶ Zur Einführung der Techniken des dynamischen Programmierens betrachten wir die effiziente Bestimmung n -ten **Fibonacci Zahl** F_n .
- ▶ Die Fibonacci Zahlen sind rekursiv definiert:

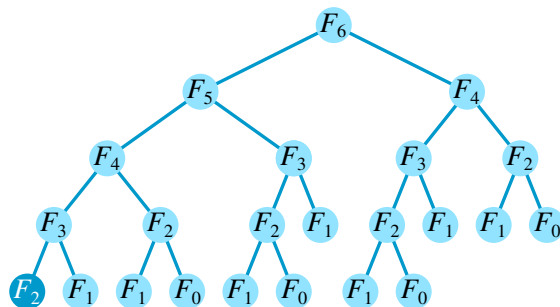
$$F_n = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ F_{n-1} + F_{n-2} & \text{für } n \geq 2 \end{cases}$$



Einführendes Beispiel: Fibonacci Zahlen

- ▶ Zur Einführung der Techniken des dynamischen Programmierens betrachten wir die effiziente Bestimmung n -ten **Fibonacci Zahl** F_n .
- ▶ Die Fibonacci Zahlen sind rekursiv definiert:

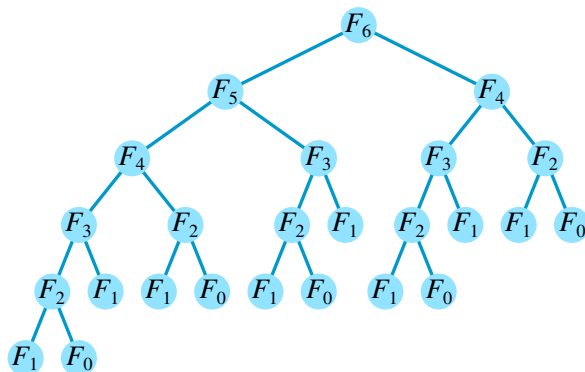
$$F_n = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ F_{n-1} + F_{n-2} & \text{für } n \geq 2 \end{cases}$$



Einführendes Beispiel: Fibonacci Zahlen

- ▶ Zur Einführung der Techniken des dynamischen Programmierens betrachten wir die effiziente Bestimmung n -ten **Fibonacci Zahl** F_n .
- ▶ Die Fibonacci Zahlen sind rekursiv definiert:

$$F_n = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ F_{n-1} + F_{n-2} & \text{für } n \geq 2 \end{cases}$$

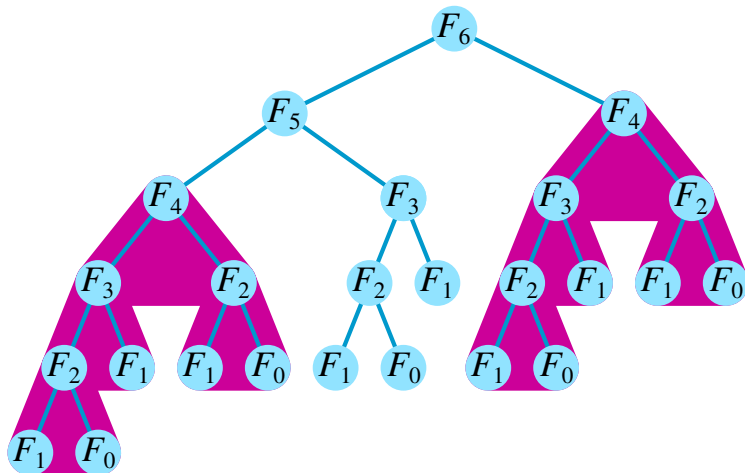


Rekursive Berechnung der Fibonacci-Zahlen

```
public static long fibonacci(int n)
{
    if (n < 2)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

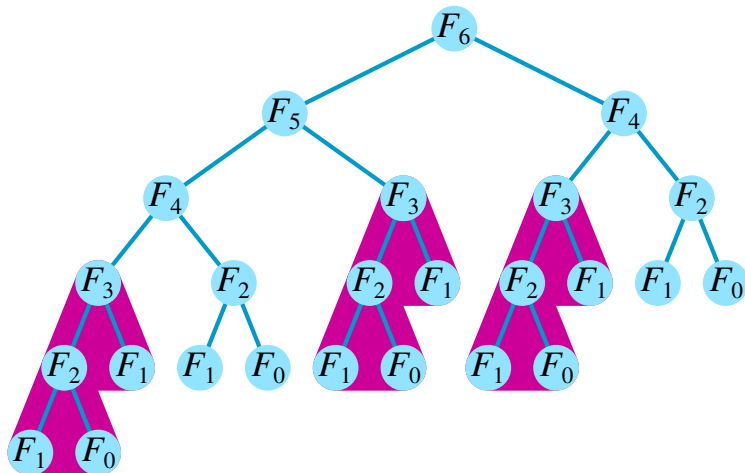
Redundanz in der Berechnung der Fibonacci Zahlen

- Die Berechnung von F_n über diese Rekursionsformel ist nicht effizient, da Teillösungen (F_k für $k < n$) vielfach berechnet werden.



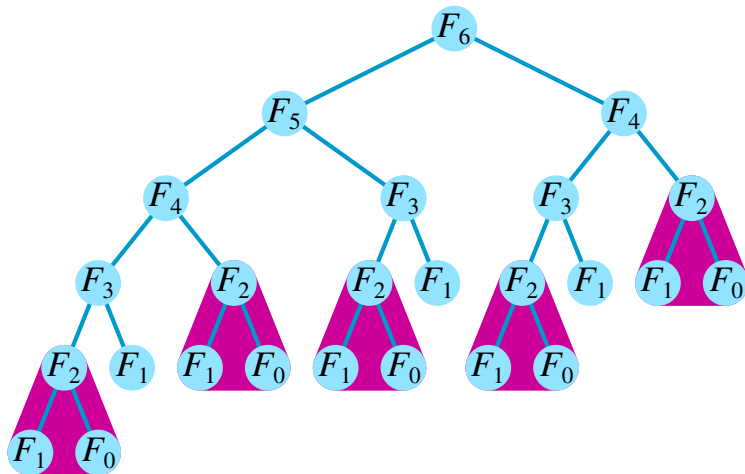
Redundanz in der Berechnung der Fibonacci Zahlen

- Die Berechnung von F_n über diese Rekursionsformel ist nicht effizient, da Teillösungen (F_k für $k < n$) vielfach berechnet werden.



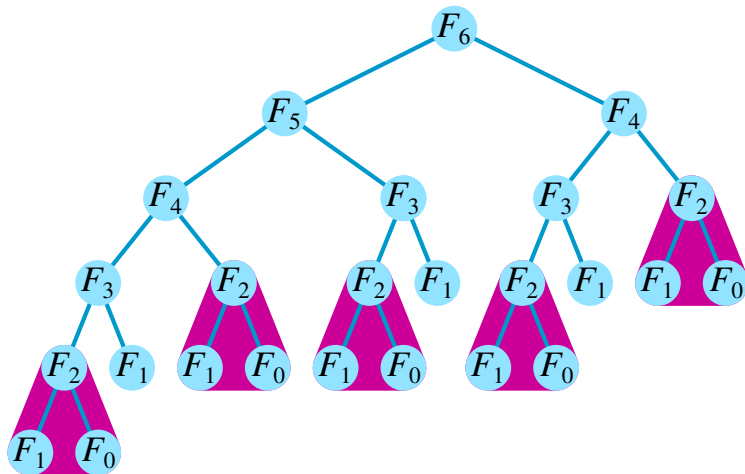
Redundanz in der Berechnung der Fibonacci Zahlen

- Die Berechnung von F_n über diese Rekursionsformel ist nicht effizient, da Teillösungen (F_k für $k < n$) vielfach berechnet werden.



Redundanz in der Berechnung der Fibonacci Zahlen

- Die Berechnung von F_n über diese Rekursionsformel ist nicht effizient, da Teillösungen (F_k für $k < n$) vielfach berechnet werden.
- $F(45) = 1134903170$ calculated with 3672623805 recursive calls.



Laufzeit bei der rekursiven Berechnung der Fibonacci Zahlen

- ▶ Der Berechnungsbaum ist maximal binär verzweigt, also ist die Laufzeit in $O(2^n)$. Eine genauere Abschätzung erhält man mit folgenden Überlegungen:
- ▶ Es gilt $\lim_{n \rightarrow \infty} \frac{F_{n+1}}{F_n} = \phi = \frac{\sqrt{5}+1}{2} \approx 1.6180$.
- ▶ Die Abschätzung $\frac{F_{n+1}}{F_n} \approx 1.6$ gilt schon ab $n = 6$ recht gut.
- ▶ Daher ergibt sich als Wachstumsverhalten $F_n \approx 1.6^n$

Laufzeit bei der rekursiven Berechnung der Fibonacci Zahlen

- ▶ Der Berechnungsbaum ist maximal binär verzweigt, also ist die Laufzeit in $O(2^n)$. Eine genauere Abschätzung erhält man mit folgenden Überlegungen:
- ▶ Es gilt $\lim_{n \rightarrow \infty} \frac{F_{n+1}}{F_n} = \phi = \frac{\sqrt{5}+1}{2} \approx 1.6180$.
- ▶ Die Abschätzung $\frac{F_{n+1}}{F_n} \approx 1.6$ gilt schon ab $n = 6$ recht gut.
- ▶ Daher ergibt sich als Wachstumsverhalten $F_n \approx 1.6^n$
- ▶ Da bei dem Rekursionsbaum an den Blättern die Werte 0 und 1 stehen, muss die Größenordnung der Anzahl der rekursive Aufrufe bei 1.6^n liegen.
- ▶ Die rekursive Berechnung der Fibonacci Zahlen hat also eine **exponentielle** Laufzeit.

Laufzeit bei der rekursiven Berechnung der Fibonacci Zahlen

- ▶ Der Berechnungsbaum ist maximal binär verzweigt, also ist die Laufzeit in $O(2^n)$. Eine genauere Abschätzung erhält man mit folgenden Überlegungen:
- ▶ Es gilt $\lim_{n \rightarrow \infty} \frac{F_{n+1}}{F_n} = \phi = \frac{\sqrt{5}+1}{2} \approx 1.6180$.
- ▶ Die Abschätzung $\frac{F_{n+1}}{F_n} \approx 1.6$ gilt schon ab $n = 6$ recht gut.
- ▶ Daher ergibt sich als Wachstumsverhalten $F_n \approx 1.6^n$
- ▶ Da bei dem Rekursionsbaum an den Blättern die Werte 0 und 1 stehen, muss die Größenordnung der Anzahl der rekursive Aufrufe bei 1.6^n liegen.
- ▶ Die rekursive Berechnung der Fibonacci Zahlen hat also eine **exponentielle** Laufzeit.
- ▶ Das Problem ist, dass dieselben Teillösungen vielfach berechnet werden.
- ▶ Gemäß dem Paradigma der dynamischen Programmierung können die Doppeltberechnungen durch Speicherung der Zwischenlösungen vermieden werden.

Berechnung mit Zwischenspeichern *top-down*

- Wir können ein Feld zur Speicherung der Teillösung verwenden, und Teillösungen nur **nach Bedarf** berechnen, so dass keine Doppelberechnungen stattfinden.

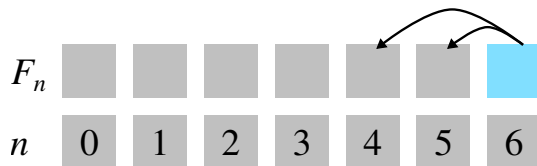
```
public class FibonacciTopDown {  
    private static long[] F;  
  
    public static long fibonacci(int n) {  
        F = new long[n+1];  
        F[0] = 0; F[1] = 1;  
        for (int k = 2; k <= n; k++)  
            F[k] = -1;  
        return fibo(n);  
    }  
  
    public static long fibo(int n) {  
        if (F[n] < 0)  
            F[n] = fibo(n-1) + fibo(n-2);  
        return F[n];  
    }  
}
```

```
public static void main(String[] args)  
{  
    int n = 12;  
    long Fn = fibonacci(n);  
    System.out.println("F(" + n +  
                        ") = " + Fn);  
}
```

- Das Feld F wird benutzt, um Teillösungen zwischenspeichern.
- Der Wert -1 zeigt an, dass der Wert noch nicht berechnet wurde.

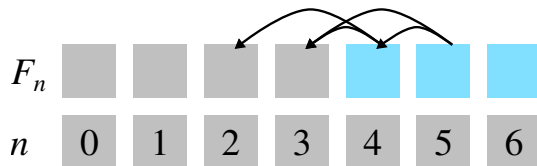
Von *Top-down* zu *Bottom-up*

- Bei dem *top-down* Ansatz gehen die Berechnungsanfragen von oben nach unten.



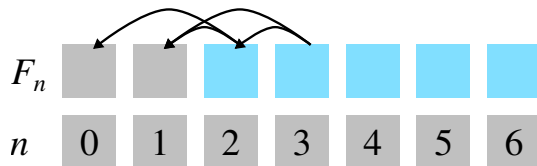
Von *Top-down* zu *Bottom-up*

- ▶ Bei dem *top-down* Ansatz gehen die Berechnungsanfragen von oben nach unten.



Von *Top-down* zu *Bottom-up*

- ▶ Bei dem *top-down* Ansatz gehen die Berechnungsanfragen von oben nach unten.



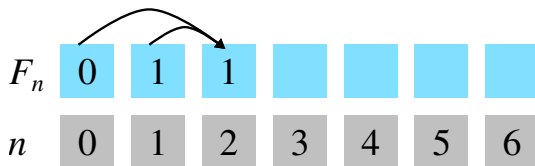
Von *Top-down* zu *Bottom-up*

- ▶ Bei dem *top-down* Ansatz gehen die Berechnungsanfragen von oben nach unten.
- ▶ Die Anfangswerte sind definiert.

F_n	0	1					
n	0	1	2	3	4	5	6

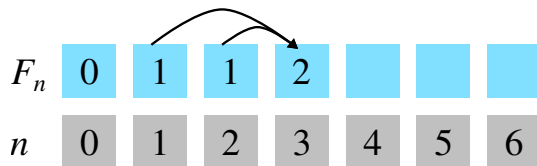
Von *Top-down* zu *Bottom-up*

- ▶ Bei dem *top-down* Ansatz gehen die Berechnungsanfragen von oben nach unten.
- ▶ Die Anfangswerte sind definiert.
- ▶ Von dort werden die Werte *bottom-up* berechnet und gespeichert.



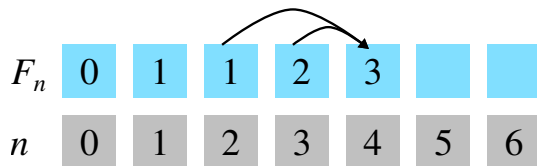
Von *Top-down* zu *Bottom-up*

- ▶ Bei dem *top-down* Ansatz gehen die Berechnungsanfragen von oben nach unten.
- ▶ Die Anfangswerte sind definiert.
- ▶ Von dort werden die Werte *bottom-up* berechnet und gespeichert.



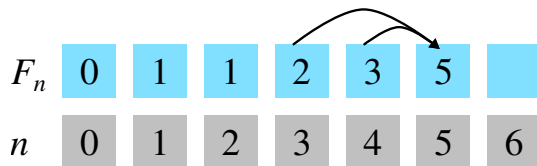
Von *Top-down* zu *Bottom-up*

- ▶ Bei dem *top-down* Ansatz gehen die Berechnungsanfragen von oben nach unten.
- ▶ Die Anfangswerte sind definiert.
- ▶ Von dort werden die Werte *bottom-up* berechnet und gespeichert.



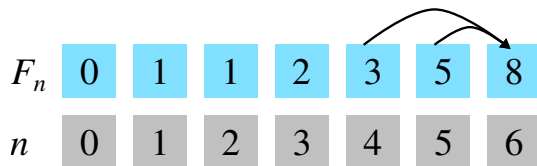
Von *Top-down* zu *Bottom-up*

- ▶ Bei dem *top-down* Ansatz gehen die Berechnungsanfragen von oben nach unten.
- ▶ Die Anfangswerte sind definiert.
- ▶ Von dort werden die Werte *bottom-up* berechnet und gespeichert.



Von *Top-down* zu *Bottom-up*

- ▶ Bei dem *top-down* Ansatz gehen die Berechnungsanfragen von oben nach unten.
- ▶ Die Anfangswerte sind definiert.
- ▶ Von dort werden die Werte *bottom-up* berechnet und gespeichert.



Von *Top-down* zu *Bottom-up*

- ▶ Bei dem *top-down* Ansatz gehen die Berechnungsanfragen von oben nach unten.
- ▶ Die Anfangswerte sind definiert.
- ▶ Von dort werden die Werte *bottom-up* berechnet und gespeichert.
- ▶ Wenn sowieso **alle** Werte des Feldes berechnet werden müssen, kann die Berechnung auch gleich *bottom-up* durchgeführt werden.

F_n	0	1	1	2	3	5	8
n	0	1	2	3	4	5	6

Berechnungen auf Vorrat: *bottom-up*

- ▶ Zur Berechnung von F_n müssen **alle** vorherigen Werte, also alle F_k für $k < n$ berechnet werden.
- ▶ Also kann dies direkt *bottom-up* erledigt werden, ohne auf den 'Bedarf' zu warten.

```
public static long fibonacci(int n)
{
    F = new long[n+1];
    F[0] = 0;
    F[1] = 1;
    for (int k = 2; k <= n; k++)
        F[k] = F[k-1] + F[k-2];
    return F[n];
}
```


Speichereffizienz?

- ▶ Gegenüber der rekursiven Implementation haben wir Laufzeit- gegen Speichereffizienz eingetauscht.
- ▶ Geht es auch effizient in beidem?

Speichereffizienz?

- ▶ Gegenüber der rekursiven Implementation haben wir Laufzeit- gegen Speichereffizienz eingetauscht.
- ▶ Geht es auch effizient in beidem?
- ▶ Man braucht immer nur die letzten beiden Werte!

```
public static long fibonacci(int n)
{
    F = new long[2];
    F[0] = 0;
    F[1] = 1;
    for (int k = 2; k <= n; k++)
        F[k%2] = F[(k-1)%2] + F[(k-2)%2];
    return F[n%2];
}
```

Nebenbemerkung zu Fibonacci

- ▶ Damit haben eine Implementation, die effizient in Laufzeit und Speicherbedarf ist, also ein Optimum.
- ▶ Oder geht es noch besser?

Nebenbemerkung zu Fibonacci

- ▶ Damit haben eine Implementation, die effizient in Laufzeit und Speicherbedarf ist, also ein Optimum.
- ▶ Oder geht es noch besser?
- ▶ Geschlossene Form (für $n > 0$):

$$F_n = \frac{\phi^n - (-\phi^{-n})}{\sqrt{5}} \quad \text{mit} \quad \phi = \frac{\sqrt{5} + 1}{2}$$

```
public static long fibonacci(int n)    // Geschlossene Form!  
{  
    double phi = (Math.sqrt(5)+1) / 2.0;  
    return Math.round( (Math.pow(phi,n) - (-Math.pow(phi,-n)) ) / Math.sqrt(5) );  
}
```

Von Fibonacci zum Dynamischen Programmieren

- ▶ Das Fibonacci Beispiel hat ein paar wichtige Aspekte des dynamischen Programmierens beleuchtet:
- ▶ In einer rekursiven Formel wird eine Lösung auf die Lösungen von Teilproblemen zurückgeführt.
- ▶ Wenn dieselben Teilprobleme vielfach gebraucht werden, kann die Effizienz durch eine Speicherung der Lösungswerte erhöht werden.

Von Fibonacci zum Dynamischen Programmieren

- ▶ Das Fibonacci Beispiel hat ein paar wichtige Aspekte des dynamischen Programmierens beleuchtet:
- ▶ In einer rekursiven Formel wird eine Lösung auf die Lösungen von Teilproblemen zurückgeführt.
- ▶ Wenn dieselben Teilprobleme vielfach gebraucht werden, kann die Effizienz durch eine Speicherung der Lösungswerte erhöht werden.
- ▶ Die Berechnung kann *top-down bei Bedarf* oder
- ▶ *bottom-up auf Vorrat* durchgeführt werden.

Von Fibonacci zum Dynamischen Programmieren

- ▶ Das Fibonacci Beispiel hat ein paar wichtige Aspekte des dynamischen Programmierens beleuchtet:
- ▶ In einer rekursiven Formel wird eine Lösung auf die Lösungen von Teilproblemen zurückgeführt.
- ▶ Wenn dieselben Teilprobleme vielfach gebraucht werden, kann die Effizienz durch eine Speicherung der Lösungswerte erhöht werden.
- ▶ Die Berechnung kann *top-down bei Bedarf* oder
- ▶ *bottom-up auf Vorrat* durchgeführt werden.
- ▶ Der *top-down* Ansatz kann günstig sein, wenn nur manche Teillösungen gebraucht werden und im voraus nicht klar ist, welche das sein werden.
- ▶ Der *bottom-up* Ansatz kann die Speichereffizienz steigern, wenn die Teillösungen nur 'schichtweise' gebraucht werden, also Teillösungen überschrieben werden können.

- ▶ Das Grundprinzip der dynamischen Programmierung lautet: Redundante Berechnung durch Speicherung von Teillösungen vermeiden!

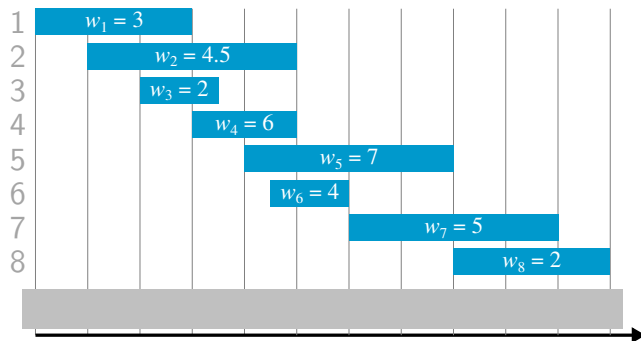
- ▶ Das Grundprinzip der dynamischen Programmierung lautet: Redundante Berechnung durch Speicherung von Teillösungen vermeiden!
- ▶ Um ein Optimierungsproblem per dynamischer Programmierung zu lösen, sucht man zunächst eine **rekursive Formel**.
- ▶ Dafür nimmt man eine optimale Lösung als gegeben an und versucht den Wert der Lösung in Abhängigkeit von Teillösungen darzustellen (Substrukturanalyse). Dabei wird oft eine **Fallunterscheidung** gemacht.

- ▶ Das Grundprinzip der dynamischen Programmierung lautet: Redundante Berechnung durch Speicherung von Teillösungen vermeiden!
- ▶ Um ein Optimierungsproblem per dynamischer Programmierung zu lösen, sucht man zunächst eine **rekursive Formel**.
- ▶ Dafür nimmt man eine optimale Lösung als gegeben an und versucht den Wert der Lösung in Abhängigkeit von Teillösungen darzustellen (Substrukturanalyse). Dabei wird oft eine **Fallunterscheidung** gemacht.
- ▶ Bei komplexeren Problemen werden viele Teillösungen benötigt. Sind darunter sehr viele **gleiche** Teilprobleme, kommt der Vorteil der dynamischen Programmierung zum Tragen.
- ▶ Falls sehr viele **unterschiedliche** Teilprobleme vorkommen, benötigt der Ansatz des dynamischen Programmierens oft zu viel Speicher und der Laufzeitgewinn ist nicht so groß.

- ▶ Mit der Rekursionsformel wird nur der optimale Wert definiert, nicht die zugehörige Lösung selbst.
- ▶ Manchmal kann die Lösung bei der Bestimmung des optimalen Wertes mit gespeichert werden. Oft ist allerdings ein zweiter Durchlauf zur Bestimmung der Lösung notwendig.

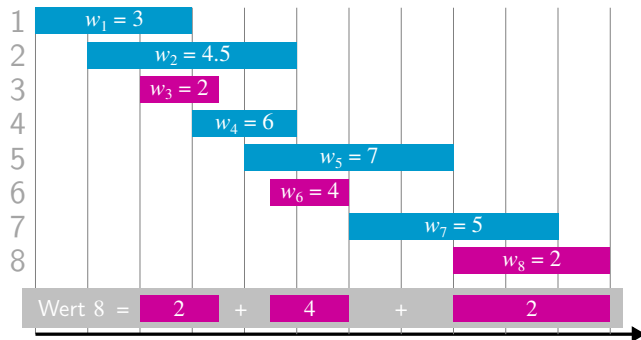
Beispiel 1: Gewichtete Intervallauswahl

- ▶ Als Erweiterung unseres ersten Beispiels der letzten Vorlesung betrachten wir nun die Auswahl **gewichteter** Intervalle.
- ▶ Es gibt Anfragen $1, \dots, n$ zur Nutzung einer Ressource (z.B. Raum, Prozessor, Messgerät) in dem Zeitintervall $I_k = [s_k, f_k)$ und mit dem Gewicht w_i .
- ▶ **Ziel:** Wähle kompatible Intervalle $S \subseteq \{1, \dots, n\}$, so dass das Gewicht der gewählten Intervalle $\sum_{k \in S} w_k$ (= Wert der Lösung) maximal ist.



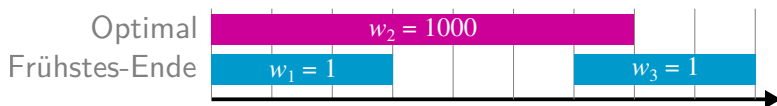
Beispiel 1: Gewichtete Intervallauswahl

- ▶ Als Erweiterung unseres ersten Beispiels der letzten Vorlesung betrachten wir nun die Auswahl **gewichteter** Intervalle.
- ▶ Es gibt Anfragen $1, \dots, n$ zur Nutzung einer Ressource (z.B. Raum, Prozessor, Messgerät) in dem Zeitintervall $I_k = [s_k, f_k)$ und mit dem Gewicht w_i .
- ▶ **Ziel:** Wähle kompatible Intervalle $S \subseteq \{1, \dots, n\}$, so dass das Gewicht der gewählten Intervalle $\sum_{k \in S} w_k$ (= Wert der Lösung) maximal ist.



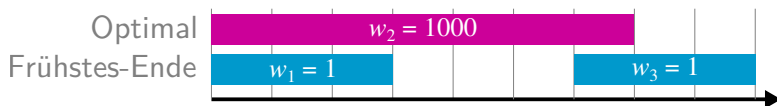
Neues Problem braucht neuen Ansatz

- ▶ Bei der **ungewichteten** Intervallauswahl (entsprechend gleichen Gewichten w_k für alle k) ist der Greedy-Algorithmus Fröhstes-Ende optimal.
- ▶ Bei der **gewichteten** Intervallauswahl kann er grandios scheitern:



Neues Problem braucht neuen Ansatz

- ▶ Bei der **ungewichteten** Intervallauswahl (entsprechend gleichen Gewichten w_k für alle k) ist der Greedy-Algorithmus Fröhstes-Ende optimal.
- ▶ Bei der **gewichteten** Intervallauswahl kann er grandios scheitern:

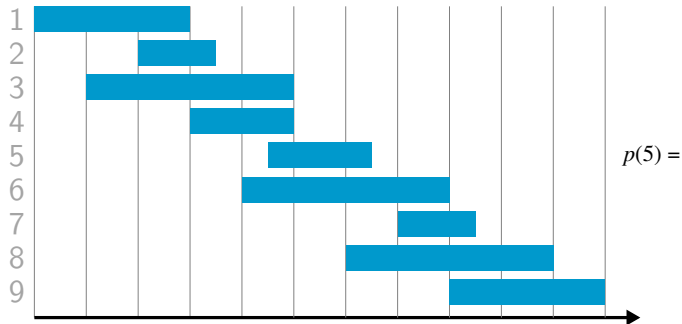


- ▶ Wir wählen einen neuen Ansatz, der die dynamische Programmierung einführt.
- ▶ Dafür überlegt man, wie man die optimale Lösungen aus Lösungen von Teilproblemen ableiten kann.
- ▶ Im Folgenden setzen wir voraus, dass die Intervalle **aufsteigend nach Endzeiten** sortiert sind.

Bester Vorgänger

- Wir definieren zunächst eine Funktion, die zu jedem Intervall den 'besten' Vorgänger (*predecessor*) angibt (oder 0, falls jener nicht existiert).

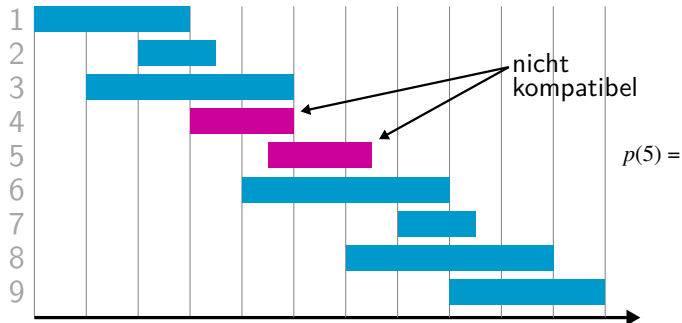
$$p(j) = \begin{cases} \max\{i \in \mathbb{N} \mid i < j \text{ und } i \text{ kompatibel mit } j\} & \text{falls existent} \\ 0 & \text{sonst} \end{cases}$$



Bester Vorgänger

- Wir definieren zunächst eine Funktion, die zu jedem Intervall den 'besten' Vorgänger (*predecessor*) angibt (oder 0, falls jener nicht existiert).

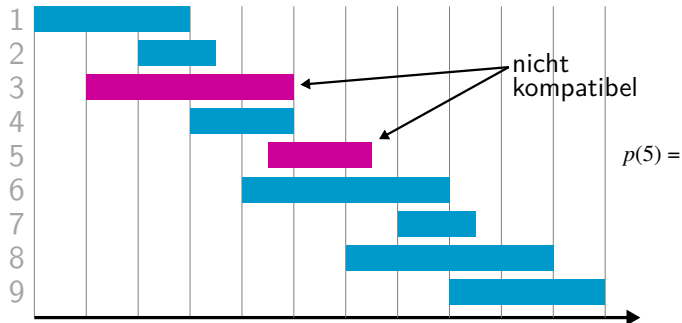
$$p(j) = \begin{cases} \max\{i \in \mathbb{N} \mid i < j \text{ und } i \text{ kompatibel mit } j\} & \text{falls existent} \\ 0 & \text{sonst} \end{cases}$$



Bester Vorgänger

- Wir definieren zunächst eine Funktion, die zu jedem Intervall den 'besten' Vorgänger (*predecessor*) angibt (oder 0, falls jener nicht existiert).

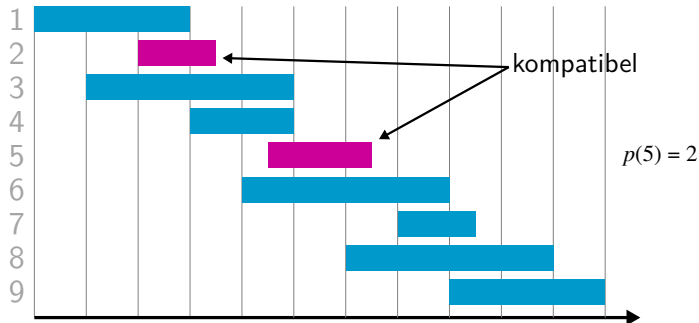
$$p(j) = \begin{cases} \max\{i \in \mathbb{N} \mid i < j \text{ und } i \text{ kompatibel mit } j\} & \text{falls existent} \\ 0 & \text{sonst} \end{cases}$$



Bester Vorgänger

- Wir definieren zunächst eine Funktion, die zu jedem Intervall den 'besten' Vorgänger (*predecessor*) angibt (oder 0, falls jener nicht existiert).

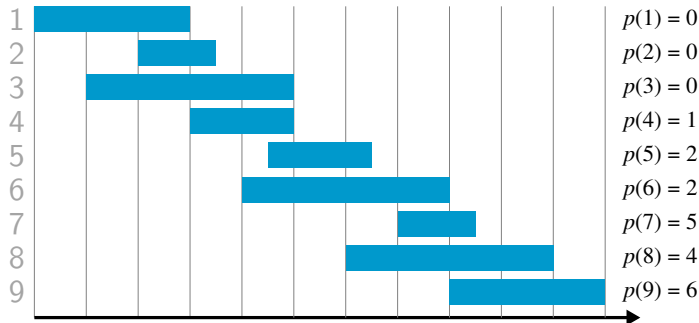
$$p(j) = \begin{cases} \max\{i \in \mathbb{N} \mid i < j \text{ und } i \text{ kompatibel mit } j\} & \text{falls existent} \\ 0 & \text{sonst} \end{cases}$$



Bester Vorgänger

- Wir definieren zunächst eine Funktion, die zu jedem Intervall den 'besten' Vorgänger (*predecessor*) angibt (oder 0, falls jener nicht existiert).

$$p(j) = \begin{cases} \max\{i \in \mathbb{N} \mid i < j \text{ und } i \text{ kompatibel mit } j\} & \text{falls existent} \\ 0 & \text{sonst} \end{cases}$$



- Intervall j ist also mit Intervall $p(j)$ kompatibel und auch mit allen Intervallen vor $p(j)$ durch die Sortierung nach Endzeitpunkten.

- ▶ **Genereller Ansatz:** Wir definieren eine Funktion OPT , die den Wert einer optimalen Lösung zu der gegebenen Eingabe zurückgibt.
- ▶ Bei diesem Problem definieren wir $\text{OPT}(k)$ als den optimalen Wert (maximales Gewicht) für Anfragen $1, \dots, k$.

- ▶ **Genereller Ansatz:** Wir definieren eine Funktion OPT , die den Wert einer optimalen Lösung zu der gegebenen Eingabe zurückgibt.
- ▶ Bei diesem Problem definieren wir $\text{OPT}(k)$ als den optimalen Wert (maximales Gewicht) für Anfragen $1, \dots, k$.
- ▶ **Substrukturanalyse:** Durch Überlegungen zur Optimalität von Teillösungen stellen wir eine Rekursionsgleichung für OPT auf.
- ▶ Diese bildet dann den Ansatzpunkt für die Lösung per **dynamischer Programmierung**.

- ▶ **Genereller Ansatz:** Wir definieren eine Funktion OPT , die den Wert einer optimalen Lösung zu der gegebenen Eingabe zurückgibt.
- ▶ Bei diesem Problem definieren wir $\text{OPT}(k)$ als den optimalen Wert (maximales Gewicht) für Anfragen $1, \dots, k$.
- ▶ **Substrukturanalyse:** Durch Überlegungen zur Optimalität von Teillösungen stellen wir eine Rekursionsgleichung für OPT auf.
- ▶ Diese bildet dann den Ansatzpunkt für die Lösung per **dynamischer Programmierung**.
- ▶ Dabei werden oft Fallunterscheidungen gemacht.
- ▶ Hier unterscheiden wir in der Definition von $\text{OPT}(k)$ danach, ob Anfrage k ausgewählt wird, oder nicht.

Rekursionsgleichung für optimale, gewichtete Intervallauswahl

- **Definition:** Sei $\text{OPT}(k)$ der optimale Wert für Anfragen $1, \dots, k$.
- **1. Fall:** Strategie OPT wählt Anfrage k aus.
 - ▶ Das Gewicht w_k wird dem Wert der Lösung zugefügt.
 - ▶ Anfragen $p(k) + 1, \dots, k - 1$ sind mit der Lösung nicht kompatibel.
 - ▶ Die optimale Lösung muss die optimale Lösung für $1, \dots, p(k)$ umfassen.

Rekursionsgleichung für optimale, gewichtete Intervallauswahl

- **Definition:** Sei $\text{OPT}(k)$ der optimale Wert für Anfragen $1, \dots, k$.
- **1. Fall:** Strategie OPT wählt Anfrage k aus.
 - ▶ Das Gewicht w_k wird dem Wert der Lösung zugefügt.
 - ▶ Anfragen $p(k) + 1, \dots, k - 1$ sind mit der Lösung nicht kompatibel.
 - ▶ Die optimale Lösung muss die optimale Lösung für $1, \dots, p(k)$ umfassen.
- **2. Fall:** Strategie OPT wählt Anfrage k nicht aus.
 - ▶ Die optimale Lösung muss die optimale Lösung für $1, \dots, k - 1$ umfassen.

Rekursionsgleichung für optimale, gewichtete Intervallauswahl

- **Definition:** Sei $\text{OPT}(k)$ der optimale Wert für Anfragen $1, \dots, k$.
- **1. Fall:** Strategie OPT wählt Anfrage k aus.
 - ▶ Das Gewicht w_k wird dem Wert der Lösung zugefügt.
 - ▶ Anfragen $p(k) + 1, \dots, k - 1$ sind mit der Lösung nicht kompatibel.
 - ▶ Die optimale Lösung muss die optimale Lösung für $1, \dots, p(k)$ umfassen.
- **2. Fall:** Strategie OPT wählt Anfrage k nicht aus.
 - ▶ Die optimale Lösung muss die optimale Lösung für $1, \dots, k - 1$ umfassen.
 - ▶ Der optimale Wert wird also durch folgende Rekursionsgleichung beschrieben:

$$\text{OPT}(k) = \begin{cases} 0 & \text{falls } k = 0 \\ \max(\underbrace{w_k + \text{OPT}(p(k))}_{\text{Fall 1}}, \underbrace{\text{OPT}(k-1)}_{\text{Fall 2}}) & \text{sonst} \end{cases}$$

- Die Rekursionsgleichung kann wie folgt umgesetzt werden:

```
1  Sortiere Anfragen nach Endzeit      //  $f_1 \leq \dots \leq f_n$ 
2  Berechne  $p_1, \dots, p_n$ 
3  opt(n)
4
5  procedure opt(k)
6    if  $k = 0$ 
7      return 0
8    else
9      return  $\max(w_k + \text{opt}(p_k), \text{opt}(k - 1))$ 
10 end
```

Kritische Betrachtung des Implementationsansatzes

- ▶ Die Laufzeit der vorgeschlagenen Implementierung ist sehr schlecht.
- ▶ Wie bei der rekursiven Berechnung der Fibonacci Zahlen werden die Lösungen für Teilprobleme **vielfach Neuberechnet**.
- ▶ Die **Rekursionsformel** für OPT liefert die **Basis für den Ansatz der dynamischen Programmierung**. Ein weiteres erforderliches Element ist die **Speicherung der Zwischenlösungen**.
- ▶ Soll *top-down* oder *bottom-up* vorgegangen werden?

Kritische Betrachtung des Implementationsansatzes

- ▶ Die Laufzeit der vorgeschlagenen Implementierung ist sehr schlecht.
- ▶ Wie bei der rekursiven Berechnung der Fibonacci Zahlen werden die Lösungen für Teilprobleme **vielfach Neuberechnet**.
- ▶ Die **Rekursionsformel** für OPT liefert die **Basis für den Ansatz der dynamischen Programmierung**. Ein weiteres erforderliches Element ist die **Speicherung der Zwischenlösungen**.
- ▶ Soll *top-down* oder *bottom-up* vorgegangen werden?
- ▶ Aus der Rekursionsformel für OPT sehen wir, dass **alle** Teillösungen benötigt werden, da $OPT(k)$ von $OPT(k - 1)$ abhängt.
- ▶ Daher ist der *Bottom-up* Ansatz vorzuziehen.

- ▶ Die Werte der Teillösungen $m[k]$ werden *bottom-up* von 0 bis zu dem gesuchten Wert n berechnet werden.

Listing 1: **Algorithmus zur Bestimmung der Intervallauswahl mit maximalem Gewicht über dynamische Programmierung**

```
1  Sortiere Anfragen nach Endzeit      //  $f_1 \leq \dots \leq f_n$   
2  Berechne  $p_1, \dots, p_n$   
3  
4   $m[0] \leftarrow 0$   
5  for  $k = 1$  to  $n$   
6     $m[k] \leftarrow \max( w_k + m[p_k], m[k-1] )$   
7  return  $m[n]$ 
```

Laufzeit für gewichtete Intervallauswahl mit dynamischer Programmierung

Der Algorithmus in Listing 1 bestimmt die Auswahl aus n Intervallen mit maximalem Gewicht in einer Laufzeit in $O(n \log n)$ und mit Speicherbedarf in $O(n)$.

- ▶ Sortieren der Aufträge nach Endzeiten: $O(n \log n)$
- ▶ Berechnung von p_k durch Sortierung nach Anfangszeiten: $O(n \log n)$
- ▶ Initialisierung und Berechnung des Feldes $m[]$ jeweils: $O(n)$

Laufzeit für gewichtete Intervallauswahl mit dynamischer Programmierung

Der Algorithmus in Listing 1 bestimmt die Auswahl aus n Intervallen mit maximalem Gewicht in einer Laufzeit in $O(n \log n)$ und mit Speicherbedarf in $O(n)$.

- ▶ Sortieren der Aufträge nach Endzeiten: $O(n \log n)$
- ▶ Berechnung von p_k durch Sortierung nach Anfangszeiten: $O(n \log n)$
- ▶ Initialisierung und Berechnung des Feldes $m[]$ jeweils: $O(n)$
- ▶ Die Vorbereitung hat also insgesamt eine Laufzeit in $O(n \log n)$ und die eigentliche Intervallauswahl bei gegebenen Sortierungen $O(n)$.
- ▶ Insgesamt ist die Laufzeit also, wie behauptet, in $O(n \log n)$.
- ▶ Der Speicherbedarf für Feld $m[]$ lässt sich nicht (wie im Fibonacci Beispiel S. 11) reduzieren, da die Rekursionsformel mit $p(k)$ auf unterschiedliche Vorlösungen zurückgreift. \square

Die optimale Lösung feststellen

- ▶ Der bisherige Ansatz bestimmt den Wert der optimalen Lösung. Wie bekommen wir die Lösung selbst (ausgewählte Intervalle)?

Die optimale Lösung feststellen

- ▶ Der bisherige Ansatz bestimmt den **Wert** der optimalen Lösung. Wie bekommen wir die **Lösung** selbst (ausgewählte Intervalle)?
- ▶ Dies kann z. B. in einem zweiten Durchlauf gemacht werden.

```
1 procedure findSolution (k)  
2 if  $k = 0$   
3   return  $\emptyset$   
4 else if  $w_k + m[p_k] > m[k-1]$   
5   return  $\{k\} \cup \text{findSolution}(p_k)$   
6 else  
7   return findSolution ( $k-1$ )  
8 end
```

Die optimale Lösung feststellen

- ▶ Der bisherige Ansatz bestimmt den Wert der optimalen Lösung. Wie bekommen wir die Lösung selbst (ausgewählte Intervalle)?
- ▶ Dies kann z. B. in einem zweiten Durchlauf gemacht werden.

```
1 procedure findSolution (k)  
2 if k = 0  
3   return  $\emptyset$   
4 else if  $w_k + m[p_k] > m[k-1]$   
5   return  $\{k\} \cup \text{findSolution}(p_k)$   
6 else  
7   return findSolution (k-1)  
8 end
```

- ▶ Die Lösung muss nicht eindeutig sein.
- ▶ In dem Fall $w_k + m[p_k] = m[k-1]$ (Ambivalenz bei der Maximumsbildung in OPT), kann Intervall *k* ausgewählt werden, muss es aber nicht.

Beispiel 2: Ein Wiedersehen mit dem 0/1-Rucksack Problem

0/1-Rucksackproblem

Es sind K Objekte mit Gewicht w_k und Wert v_k (für $1 \leq k \leq K$) sowie ein Rucksack (*knapsack*) mit einer maximalen Kapazität W gegeben. Wähle Objekte, so dass ihr Gesamtwert maximal ist und ihr Gesamtgewicht die Kapazität nicht überschreitet.

- Formal ist das Ziel $S \subseteq \{1, \dots, n\}$ gemäß folgender Optimierung zu wählen:

$$S \text{ maximiert } \sum_{k \in S} v_k \quad \text{unter der Bedingung} \quad \sum_{k \in S} w_k \leq W$$

Beispiel 2: Ein Wiedersehen mit dem 0/1-Rucksack Problem

0/1-Rucksackproblem

Es sind K Objekte mit Gewicht w_k und Wert v_k (für $1 \leq k \leq K$) sowie ein Rucksack (*knapsack*) mit einer maximalen Kapazität W gegeben. Wähle Objekte, so dass ihr Gesamtwert maximal ist und ihr Gesamtgewicht die Kapazität nicht überschreitet.

- Formal ist das Ziel $S \subseteq \{1, \dots, n\}$ gemäß folgender Optimierung zu wählen:

$$S \text{ maximiert } \sum_{k \in S} v_k \quad \text{unter der Bedingung } \sum_{k \in S} w_k \leq W$$

- Im Gegensatz zu dem teilbaren Rucksackproblem konnte das 0/1-Rucksack Problem **nicht effizient** (per Greedy Ansatz) gelöst werden.

Interlude: NP-Vollständigkeit

- ▶ Die **Komplexitätsklasse P** umfasst diejenigen Probleme, für die es einen Algorithmus mit Laufzeit in $\mathcal{O}(p(n))$ für ein Polynom $p(n)$ in der Eingabegröße n gibt.
- ▶ Grob gesagt, können Probleme in P halbwegs effizient gelöst werden.
- ▶ Zumindest sind Probleme, die nicht in P enthalten sind, für größere Eingaben n meist praktisch nicht lösbar.
- ▶ Beispiel: Ein Algorithmus mit einer Laufzeit von 2^n benötigt bei einer Eingabegröße von $n = 100$ selbst auf einem sehr schnellen Computer mehr als 10^{14} Jahre.

Interlude: NP-Vollständigkeit

- ▶ Die **Komplexitätsklasse P** umfasst diejenigen Probleme, für die es einen Algorithmus mit Laufzeit in $O(p(n))$ für ein Polynom $p(n)$ in der Eingabegröße n gibt.
- ▶ Grob gesagt, können Probleme in P halbwegs effizient gelöst werden.
- ▶ Zumindest sind Probleme, die nicht in P enthalten sind, für größere Eingaben n meist praktisch nicht lösbar.
- ▶ Beispiel: Ein Algorithmus mit einer Laufzeit von 2^n benötigt bei einer Eingabegröße von $n = 100$ selbst auf einem sehr schnellen Computer mehr als 10^{14} Jahre.
- ▶ Die **Komplexitätsklasse NP** umfasst die Probleme, bei denen in polynomieller Laufzeit festgestellt werden kann, ob ein Lösungskandidat tatsächlich eine Lösung darstellt. Die Bestimmung von Lösungen unterliegt keiner Laufzeitbeschränkung.
- ▶ Ein Problem X heißt **NP-schwer** (*NP-hard*), wenn ein beliebiges Problem aus NP in polynomieller Zeit auf eine Lösung von X zurückgeführt werden kann.
- ▶ Ein Problem heißt **NP-vollständig**, wenn es zu NP gehört und NP-schwer ist.

Interlude: NP-Vollständigkeit

- ▶ Offensichtlich gilt $P \subseteq NP$.
- ▶ Es wird von den allermeisten vermutet, dass $P \neq NP$ gilt. Aber dies konnte bisher nicht bewiesen werden (Millenium Problem, 1.000.000 \$ Preisgeld).
- ▶ NP-vollständige Probleme stellen Prüfsteine für die $P=NP$ Hypothese dar.

Interlude: NP-Vollständigkeit

- ▶ Offensichtlich gilt $P \subseteq NP$.
- ▶ Es wird von den allermeisten vermutet, dass $P \neq NP$ gilt. Aber dies konnte bisher nicht bewiesen werden (Millenium Problem, 1.000.000 \$ Preisgeld).
- ▶ NP-vollständige Probleme stellen Prüfsteine für die $P=NP$ Hypothese dar.
- ▶ Wenn für ein einziges NP-vollständiges Problem ein polynomieller Algorithmus gefunden wird, ist $P=NP$ gezeigt und viele relevante Problemstellungen, die z. Z. praktisch nicht lösbar sind, könnten dadurch lösbar werden.
- ▶ Daher sind NP-vollständige Probleme interessante Herausforderungen.
- ▶ Insbesondere werden für solche Probleme oft ›Ersatzansätze‹ gesucht:
 - ▶ Ansätze, die in bestimmten praktischen Fällen eine effiziente Lösungen finden, auch wenn der *worst-case* exponentiell bleibt.
 - ▶ Ansätze, die in effizienter Laufzeit suboptimale, approximative Lösungen bestimmen.

Interlude: NP-Vollständigkeit

- Beispiele für NP-vollständige Probleme:
 - ▶ **Problem des Handlungsreisenden:** Finde in einem vollständigen, gewichteten Graphen einen Zyklus mit minimalem Gewicht, der jeden Knoten genau einmal enthält (*Traveling Salesman Problem*; TSP).

Interlude: NP-Vollständigkeit

- Beispiele für NP-vollständige Probleme:
 - ▶ **Problem des Handlungsreisenden:** Finde in einem vollständigen, gewichteten Graphen einen Zyklus mit minimalem Gewicht, der jeden Knoten genau einmal enthält (*Traveling Salesman Problem*; TSP).
 - ▶ **Hamiltonpfad:** Finde einen Pfad, der jeden Knoten eines gegebenen Graphen genau einmal besucht. (Ebenso 'Hamiltonzyklus')

Interlude: NP-Vollständigkeit

- Beispiele für NP-vollständige Probleme:
 - ▶ **Problem des Handlungsreisenden:** Finde in einem vollständigen, gewichteten Graphen einen Zyklus mit minimalem Gewicht, der jeden Knoten genau einmal enthält (*Traveling Salesman Problem*; TSP).
 - ▶ **Hamiltonpfad:** Finde einen Pfad, der jeden Knoten eines gegebenen Graphen genau einmal besucht. (Ebenso 'Hamiltonzyklus')
 - ▶ **0/1-Rucksack Problem!** (auch bei Beschränkung auf ganzzahlige Gewichte)
 - ▶ Es gibt also nicht viel Hoffnung für einen Ansatz mit dynamischer Programmierung. Wir probieren es trotzdem! (Und beschränken uns dabei auf ganzzahlige Gewichte.)

- **Definition:** Sei $\text{OPT}(k)$ der Wert einer optimalen Lösung für Objekte $1, \dots, k$.
(Beachte: Die Reihenfolge der Objekte in der gegebenen Lösung ist beliebig.)
- **1. Fall** Objekt k ist in der Lösung nicht ausgewählt.
 - ▶ Dann besteht die Lösung in der optimalen Lösung für Teilproblem $1, \dots, k - 1$.

Erster Ansatz für das 0/1-Rucksack Problem

- **Definition:** Sei $\text{OPT}(k)$ der Wert einer optimalen Lösung für Objekte $1, \dots, k$.
(Beachte: Die Reihenfolge der Objekte in der gegebenen Lösung ist beliebig.)
- **1. Fall** Objekt k ist in der Lösung nicht ausgewählt.
 - ▶ Dann besteht die Lösung in der optimalen Lösung für Teilproblem $1, \dots, k - 1$.
- **2. Fall** Objekt k ist ausgewählt.
 - ▶ Ohne weitere Information lässt sich die Lösung nicht auf Teillösungen zurückführen.
 - ▶ Wir wissen nicht, ob Objekt k überhaupt ausgewählt werden konnte, und wir wissen nicht wieviel freie Kapazität vorhanden ist, um weitere Objekte auszuwählen.

Erster Ansatz für das 0/1-Rucksack Problem

- **Definition:** Sei $\text{OPT}(k)$ der Wert einer optimalen Lösung für Objekte $1, \dots, k$.
(Beachte: Die Reihenfolge der Objekte in der gegebenen Lösung ist beliebig.)
- **1. Fall** Objekt k ist in der Lösung nicht ausgewählt.
 - ▶ Dann besteht die Lösung in der optimalen Lösung für Teilproblem $1, \dots, k - 1$.
- **2. Fall** Objekt k ist ausgewählt.
 - ▶ Ohne weitere Information lässt sich die Lösung nicht auf Teillösungen zurückführen.
 - ▶ Wir wissen nicht, ob Objekt k überhaupt ausgewählt werden konnte, und wir wissen nicht wieviel freie Kapazität vorhanden ist, um weitere Objekte auszuwählen.
 - ▶ Wir müssen also die Restkapazität als **weitere Variable** in OPT mitberücksichtigen.

Richtiger Ansatz für das 0/1-Rucksack Problem

- **Definition:** Sei $\text{OPT}(k, W)$ der Wert einer optimalen Lösung \mathbf{O} für Objekte $1, \dots, k$ mit Maximalgewicht W . (Reihenfolge der Objekte in \mathbf{O} ist beliebig.)
 - ▶ Falls $w_k > W$ kann Objekt k nicht ausgewählt werden. Andernfalls:
- **1. Fall:** Objekt k ist in der Lösung \mathbf{O} nicht ausgewählt.
 - ▶ Dann besteht die Lösung \mathbf{O} in der optimalen Lösung für Teilproblem $1, \dots, k - 1$ mit Maximalgewicht W .

Richtiger Ansatz für das 0/1-Rucksack Problem

- **Definition:** Sei $\text{OPT}(k, W)$ der Wert einer optimalen Lösung \mathbf{O} für Objekte $1, \dots, k$ mit Maximalgewicht W . (Reihenfolge der Objekte in \mathbf{O} ist beliebig.)
 - ▶ Falls $w_k > W$ kann Objekt k nicht ausgewählt werden. Andernfalls:
- **1. Fall:** Objekt k ist in der Lösung \mathbf{O} nicht ausgewählt.
 - ▶ Dann besteht die Lösung \mathbf{O} in der optimalen Lösung für Teilproblem $1, \dots, k-1$ mit Maximalgewicht W .
- **2. Fall:** Objekt k ist in der Lösung \mathbf{O} ausgewählt.
 - ▶ Dann besteht die Lösung \mathbf{O} in der optimalen Lösung für Teilproblem $1, \dots, k-1$ mit Maximalgewicht $W - w_k$.

$$\text{OPT}(k, W) = \begin{cases} 0 & \text{falls } k = 0 \\ \text{OPT}(k-1, W) & \text{falls } w_k > W \\ \max(\underbrace{v_k + \text{OPT}(k-1, W - w_k)}_{k \text{ ausgewählt}}, \underbrace{\text{OPT}(k-1, W)}_{k \text{ nicht ausgewählt}}) & \text{sonst} \end{cases}$$

Weitere Entscheidungen zur Implementation

- ▶ OPT greift in der ersten Dimension nur auf Vorgänger zu, d.h. $\text{OPT}(k, \cdot)$ hängt nur von $\text{OPT}(k - 1, \cdot)$ ab.
- ▶ Daher könnte bei dem *bottom-up* Ansatz der Speicherbedarf zur Bestimmung des optimalen Wertes auf zwei Spalten der Matrix M beschränkt werden, analog zu dem Fibonacci Beispiel S. 11.

Weitere Entscheidungen zur Implementation

- ▶ OPT greift in der ersten Dimension nur auf Vorgänger zu, d.h. $\text{OPT}(k, \cdot)$ hängt nur von $\text{OPT}(k-1, \cdot)$ ab.
- ▶ Daher könnte bei dem *bottom-up* Ansatz der Speicherbedarf zur Bestimmung des optimalen Wertes auf zwei Spalten der Matrix M beschränkt werden, analog zu dem Fibonacci Beispiel S. 11.
- ▶ Allerdings ist dann unklar, wie die eigentliche Lösung (welche Objekte ausgewählt werden) ausgelesen werden kann.
- ▶ Dies geschieht ja normalerweise in einem zweiten Durchlauf, wobei das Feld / die Matrix der gespeicherten Teillösungen durchlaufen wird, siehe Seite 25.

Weitere Entscheidungen zur Implementation

- ▶ OPT greift in der ersten Dimension nur auf Vorgänger zu, d.h. $\text{OPT}(k, \cdot)$ hängt nur von $\text{OPT}(k-1, \cdot)$ ab.
- ▶ Daher könnte bei dem *bottom-up* Ansatz der Speicherbedarf zur Bestimmung des optimalen Wertes auf zwei Spalten der Matrix M beschränkt werden, analog zu dem Fibonacci Beispiel S. 11.
- ▶ Allerdings ist dann unklar, wie die eigentliche Lösung (welche Objekte ausgewählt werden) ausgelesen werden kann.
- ▶ Dies geschieht ja normalerweise in einem zweiten Durchlauf, wobei das Feld / die Matrix der gespeicherten Teillösungen durchlaufen wird, siehe Seite 25.
- ▶ Bei dieser Rekursionsformel für OPT werden **nicht unbedingt alle** Teillösungen benötigt.
- ▶ Daher benutzen wir hier den *top-down* Ansatz. Der *bottom-up* Ansatz ist leichter zu implementieren, aber u. U. etwas weniger effizient.

Knapsack Top-Down Implementation

```
1 public class Knapsack {
2
3     public int W, K;
4     public int[] weight;
5     public double[] value;
6     private double[][] M;
7     private Queue<Integer> inventory = new LinkedList<>();
8
9     public Knapsack(int[] weight, double[] value, int W) {
10         this.W = W;
11         this.K = weight.length - 1;
12         this.weight = weight;
13         this.value = value;
14         M = new double[K+1][W+1];
15         for (int w = 0; w <= W; w++) {
16             M[0][w] = 0.0;
17             for (int k = 1; k <= K; k++)
18                 M[k][w] = -1.0;
19         }
20     }
```

Knapsack Top-Down Implementation

```
21 public double opt(int k, int w)
22 {
23     if (M[k][w] < 0)
24         if (weight[k] > w)
25             M[k][w] = opt(k-1, w);
26         else
27             M[k][w] = Math.max( value[k] + opt(k-1, w-weight[k]),
28                                 opt(k-1, w) );
29     return M[k][w];
30 }
31
32 public void findSolution(int k, int w)
33 {
34     if (k == 0) return;
35     else if (weight[k] > w)
36         findSolution(k-1, w);
37     else if (value[k] + M[k-1][w-weight[k]] > M[k-1][w]) {
38         findSolution(k-1, w-weight[k]);
39         inventory.add(k);
40     } else
41         findSolution(k-1, w);
42 }
```

```
public static void main(String[] args)
{
    double[] value = {0, 2, 3, 1, 5, 7, 3, 6};
    int[] weight = {0, 3, 4, 2, 4, 7, 3, 5};
    int maxWeight = 14;

    Knapsack knapsack = new Knapsack(weight, value, maxWeight);
    double optValue = knapsack.opt(knapsack.K, knapsack.W);
    knapsack.findSolution(knapsack.K, knapsack.W);

    System.out.println("Optimal load: ");
    for (int k : knapsack.inventory)
        System.out.println(knapsack.weight[k] + " - " + knapsack.value[k]);
}
```

Laufzeit der Knapsack Implementierung

Die 0/1-Knapsack Implementierung für ganzzahlige Gewichte basierend auf dynamischer Programmierung hat eine Laufzeit in $O(KW)$, wobei K die Anzahl der Objekte und W die Kapazität des Rucksacks ist. Der Speicherbedarf ist ebenfalls in $O(KW)$.

Beweis.

- ▶ Die Methode $opt()$ geht durch die Abfrage in Zeile 23 für jedes Paar (k, w) nur einmal in die Rekursion.
- ▶ Somit ist die Laufzeit von $opt()$ und von der Initialisierung in $O(KW)$. \square

Laufzeit der Knapsack Implementierung

Die 0/1-Knapsack Implementierung für ganzzahlige Gewichte basierend auf dynamischer Programmierung hat eine Laufzeit in $O(KW)$, wobei K die Anzahl der Objekte und W die Kapazität des Rucksacks ist. Der Speicherbedarf ist ebenfalls in $O(KW)$.

Beweis.

- ▶ Die Methode $opt()$ geht durch die Abfrage in Zeile 23 für jedes Paar (k, w) nur einmal in die Rekursion.
- ▶ Somit ist die Laufzeit von $opt()$ und von der Initialisierung in $O(KW)$. \square
- ▶ 0/1-Knapsack ist NP-vollständig und die Laufzeit ist in $O(KW)??$

Laufzeit der Knapsack Implementierung

Die 0/1-Knapsack Implementierung für ganzzahlige Gewichte basierend auf dynamischer Programmierung hat eine Laufzeit in $O(KW)$, wobei K die Anzahl der Objekte und W die Kapazität des Rucksacks ist. Der Speicherbedarf ist ebenfalls in $O(KW)$.

Beweis.

- ▶ Die Methode $opt()$ geht durch die Abfrage in Zeile 23 für jedes Paar (k, w) nur einmal in die Rekursion.
- ▶ Somit ist die Laufzeit von $opt()$ und von der Initialisierung in $O(KW)$. \square
- ▶ 0/1-Knapsack ist NP-vollständig und die Laufzeit ist in $O(KW)??$
- ▶ Auflösung: Diese Abschätzung 'zählt nicht', da sie nicht nur von der **Anzahl** der Eingabeobjekte, sondern auch von einem **Eingabewert** abhängt.

Pseudo-polynomieller Algorithmus

Ein Algorithmus zur Lösung eines Problems, das als Eingaben ganze Zahlen hat, heißt **pseudopolynomiell**, wenn seine Laufzeit durch ein Polynom in der Eingabegröße und dem größten Absolutwert der Eingabezahlen beschränkt ist.

- ▶ Die Knapsack Implementierung ist also pseudopolynomiell.
- ▶ Nach der formalen Definition darf eine Laufzeit-Komplexität nur von der Bitanzahl abhängen, die benötigt wird, um die Eingabedaten zu kodieren.

Pseudo-polynomieller Algorithmus

Ein Algorithmus zur Lösung eines Problems, das als Eingaben ganze Zahlen hat, heißt **pseudopolynomiell**, wenn seine Laufzeit durch ein Polynom in der Eingabegröße und dem größten Absolutwert der Eingabezahlen beschränkt ist.

- ▶ Die Knapsack Implementierung ist also pseudopolynomiell.
- ▶ Nach der formalen Definition darf eine Laufzeit-Komplexität nur von der Bitanzahl abhängen, die benötigt wird, um die Eingabedaten zu kodieren.
- ▶ Wenn die Kapazitätsgrenze W mit n Bits kodiert wird, kann die Grenze bis zu $W = 2^n - 1$ betragen. Die Laufzeit in Abhängigkeit von der Eingabelänge in Bits ist daher $\mathcal{O}(K2^n)$, also exponentiell!
- ▶ Genau genommen müssten noch die Bits berücksichtigt werden, die benötigt werden, um die K Objekte zu kodieren. Dies macht allerdings nur einen konstanten Faktor aus, wenn nur W variiert wird.

Beispiel 3: Editierdistanz

- ▶ Die **Editierdistanz** (*edit distance*, auch Levenshtein-Distanz) ist ein Maß für die Ähnlichkeit zwischen zwei Zeichenketten.
- ▶ Sie dient als Grundlage für den *diff* Befehl und approximative *String-Matching* Algorithmen.
- ▶ Das Verfahren findet auch Anwendung in der Bioinformatik zur Analyse von DNA- und RNA-Sequenzen. Dort wird das Distanzmaß noch etwas angepasst, um die biologischen Gegebenheiten besser zu modellieren.

Beispiel 3: Editierdistanz

- ▶ Die Editierdistanz zwischen zwei Strings a und b ist die minimale Anzahl von elementaren Buchstabenoperationen, die notwendig sind, um a in b umzuwandeln.
- ▶ Die elementaren Buchstabenoperationen sind:
 - ▶ **D**: Einen Buchstaben löschen (*delete*)
 - ▶ **I**: Einen Buchstaben einfügen (*insert*)
 - ▶ **S**: Einen Buchstaben ersetzen (*substitute*)

Beispiel 3: Editierdistanz

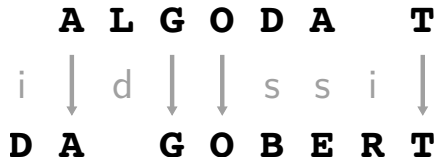
- ▶ Die Editierdistanz zwischen zwei Strings a und b ist die minimale Anzahl von elementaren Buchstabenoperationen, die notwendig sind, um a in b umzuwandeln.
- ▶ Die elementaren Buchstabenoperationen sind:
 - ▶ **D**: Einen Buchstaben löschen (*delete*)
 - ▶ **I**: Einen Buchstaben einfügen (*insert*)
 - ▶ **S**: Einen Buchstaben ersetzen (*substitute*)
- ▶ Das Editieren kann buchstabenweise von vorne entlang des Strings a durchgeführt werden. Dann wird noch folgende Aktion verwendet:
 - ▶ -: Einen Buchstaben unverändert übernehmen
- ▶ Das Übernehmen eines Buchstaben zählt **nicht** als Operation im Sinne der Editierdistanz.

- ▶ Wie kann man ALGODAT in DAGOBERT verwandeln?

A L G O D A T

D A G O B E R T

- ▶ Wie kann man ALGODAT in DAGOBERT verwandeln?



- ▶ Diese Umwandlung ergibt eine Distanz von 5 (zweimal Einfügen, einmal Löschen, zweimal Ersetzen).
- ▶ Die Editierdistanz ist die **kleinste** Anzahl von Buchstabenoperationen. Geht es noch kürzer als in dem Beispiel?

- ▶ Gemäß dem Ansatz der dynamischen Programmierung definieren wir eine Funktion OPT .
- ▶ Wir führen zunächst einige Schreibweisen ein.
- ▶ Für einen String a bezeichnet $a[1 : i]$ den String, der aus den ersten i Zeichen von a besteht.
- ▶ $a[1 : 0]$ ist der leere String.
- ▶ Desweiteren bezeichnet $a[i]$ (für $i > 0$) den i -ten Buchstaben von a und

$$a[i] \neq b[j] \quad = \quad \begin{cases} 0 & \text{falls die Zeichen } a[i] \text{ und } b[j] \text{ gleich sind} \\ 1 & \text{sonst} \end{cases}$$

- ▶ Gemäß dem Ansatz der dynamischen Programmierung definieren wir eine Funktion OPT .
- ▶ Wir führen zunächst einige Schreibweisen ein.
- ▶ Für einen String a bezeichnet $a[1 : i]$ den String, der aus den ersten i Zeichen von a besteht.
- ▶ $a[1 : 0]$ ist der leere String.
- ▶ Desweiteren bezeichnet $a[i]$ (für $i > 0$) den i -ten Buchstaben von a und

$$a[i] \neq b[j] \quad = \quad \begin{cases} 0 & \text{falls die Zeichen } a[i] \text{ und } b[j] \text{ gleich sind} \\ 1 & \text{sonst} \end{cases}$$

- ▶ Wir betrachten zunächst die Randfälle.

Editierdistanz durch Dynamisches Programmieren

- ▶ $\text{OPT}(i, j)$ sei die Editierdistanz zwischen den Substrings $a[1 : i]$ und $b[1 : j]$.
- ▶ Für $j = 0$ soll String $a[1 : i]$ in einen leeren String umgewandelt werden. Dazu müssen alle Zeichen von $a[1 : i]$ gelöscht werden (also i -mal die Operation **D** ausführen): $\text{OPT}(i, 0) = i$.

Editierdistanz durch Dynamisches Programmieren

- ▶ $\text{OPT}(i, j)$ sei die Editierdistanz zwischen den Substrings $a[1 : i]$ und $b[1 : j]$.
- ▶ Für $j = 0$ soll String $a[1 : i]$ in einen **leeren** String umgewandelt werden. Dazu müssen alle Zeichen von $a[1 : i]$ gelöscht werden (also i -mal die Operation **D** ausführen): $\text{OPT}(i, 0) = i$.
- ▶ Für $i = 0$ soll ein **leerer** String in den String $b[1 : j]$ umgewandelt werden. Dazu müssen alle Zeichen des Strings $b[1 : j]$ eingefügt werden (also j -mal die Operation **I** ausführen): $\text{OPT}(0, j) = j$.

Editierdistanz durch Dynamisches Programmieren

- ▶ $\text{OPT}(i, j)$ sei die Editierdistanz zwischen den Substrings $a[1 : i]$ und $b[1 : j]$.
- ▶ Für $j = 0$ soll String $a[1 : i]$ in einen **leeren** String umgewandelt werden. Dazu müssen alle Zeichen von $a[1 : i]$ gelöscht werden (also i -mal die Operation **D** ausführen): $\text{OPT}(i, 0) = i$.
- ▶ Für $i = 0$ soll ein **leerer** String in den String $b[1 : j]$ umgewandelt werden. Dazu müssen alle Zeichen des Strings $b[1 : j]$ eingefügt werden (also j -mal die Operation **I** ausführen): $\text{OPT}(0, j) = j$.
- ▶ Für $i, j > 0$ betrachten wir alle möglichen Operationen, addieren die Editierkosten (=1 für **D**, **I** und **S**) zu der jeweiligen Teillösung und wählen das Minimum:

$$\begin{aligned} \text{OPT}(i, j) = \min(& \text{OPT}(i, j - 1) + 1, && \text{einfügen von } b[j] \\ & \text{OPT}(i - 1, j) + 1, && \text{löschen von } a[i] \\ & \text{OPT}(i - 1, j - 1) + (a[i] \neq b[j]) && \text{ersetzen oder übernehmen} \\ &) \end{aligned}$$

Matrix der gespeicherten Teillösungen

- Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.

		D	A	G	O	B	E	R	T
A									
L									
G									
O									
D									
A									
T									

Matrix der gespeicherten Teillösungen

- Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.

		D	A	G	O	B	E	R	T
A									
L									
G									
O									
D									
A									
T									

Editierdistanz
von "**ALG**"
zu "**DAGO**"

Matrix der gespeicherten Teillösungen

- Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.

		D	A	G	O	B	E	R	T
A									
L									
G									
O									
D									
A									
T									

Editierdistanz
von "**ALGOD**"
zu "**DA**"

Matrix der gespeicherten Teillösungen

- Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.

		D	A	G	O	B	E	R	T
A									
L									
G									
O									
D									
A									
T									

Editierdistanz
von "**ALGODAT**"
zu "**DAGOBERT**"

Matrix der gespeicherten Teillösungen

- Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.

		D	A	G	O	B	E	R	T
A									
L									
G									
O									
D									
A									
T									

Diagram illustrating the edit distance matrix for the strings $a = \text{ALGODAT}$ and $b = \text{DAGOBERT}$. The matrix is a 9x9 grid. The first row and column are labeled with the characters of the strings. The cell at row 'A' and column 'D' is highlighted in blue, indicating the edit distance between 'A' and 'D'. An arrow points to this cell with the text: Editierdistanz von "" zu "D".

Matrix der gespeicherten Teillösungen

- Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.

		D	A	G	O	B	E	R	T
A		1							
L									
G									
O									
D									
A									
T									

← Editierdistanz
von ""
zu "D"
= 1 (1x Einfügen)

Matrix der gespeicherten Teillösungen

- Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.

		D	A	G	O	B	E	R	T
A		1							8
L									
G									
O									
D									
A									
T									

Editierdistanz
von ""
zu "**DAGOBERT**"
= 8 (8x Einfügen)

Matrix der gespeicherten Teillösungen

- Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.

		D	A	G	O	B	E	R	T
A		1							8
L									
G									
O									
D	5								
A									
T									

← Editierdistanz von "ALGOD" zu ""
= 5 (5x Löschen)

Matrix der gespeicherten Teillösungen

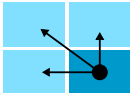
- ▶ Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.
- ▶ Die Randfälle sind einfach.

		D	A	G	O	B	E	R	T
	0	1	2	3	4	5	6	7	8
A	1								
L	2								
G	3								
O	4								
D	5								
A	6								
T	7								

Matrix der gespeicherten Teillösungen

- ▶ Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.
- ▶ Die Randfälle sind einfach.
- ▶ Die Editierdistanz der Strings wird in Eintrag (7,8) der Matrix stehen.
- ▶ Um den Wert zu bestimmen, brauchen wir Teillösungen.

		D	A	G	O	B	E	R	T
	0	1	2	3	4	5	6	7	8
A	1								
L	2								
G	3								
O	4								
D	5								
A	6								
T	7								



Matrix der gespeicherten Teillösungen

- ▶ Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.
- ▶ Die Randfälle sind einfach.
- ▶ Die Editierdistanz der Strings wird in Eintrag (7,8) der Matrix stehen.
- ▶ Um den Wert zu bestimmen, brauchen wir Teillösungen. Und zwar **alle** Teillösungen.

		D	A	G	O	B	E	R	T
	0	1	2	3	4	5	6	7	8
A	1								
L	2								
G	3								
O	4								
D	5								
A	6								
T	7								

Matrix der gespeicherten Teillösungen

- ▶ Wir betrachten die Matrix der Teillösungen für $a = \text{ALGODAT}$ und $b = \text{DAGOBERT}$.
- ▶ Die Randfälle sind einfach.
- ▶ Die Editierdistanz der Strings wird in Eintrag (7,8) der Matrix stehen.
- ▶ Um den Wert zu bestimmen, brauchen wir Teillösungen. Und zwar **alle** Teillösungen.

		D	A	G	O	B	E	R	T
	0	1	2	3	4	5	6	7	8
A	1								
L	2								
G	3								
O	4								
D	5								
A	6								
T	7								

Überlegungen zur Implementation

- ▶ Die Betrachtung hat gezeigt, dass zur Bestimmung der Editierdistanz der gegebenen Strings **alle** Teillösungen, die in der Matrix repräsentiert sind, benötigt werden.
- ▶ Daher bringt die “Berechnung bei Bedarf” des *top-down* Ansatzes keinen Vorteil.
- ▶ Es ist also **günstiger**, direkt alle Werte der Matrix *bottom-up* zu bestimmen.

Überlegungen zur Implementation

- ▶ Die Betrachtung hat gezeigt, dass zur Bestimmung der Editierdistanz der gegebenen Strings **alle** Teillösungen, die in der Matrix repräsentiert sind, benötigt werden.
- ▶ Daher bringt die “Berechnung bei Bedarf” des *top-down* Ansatzes keinen Vorteil.
- ▶ Es ist also **günstiger**, direkt alle Werte der Matrix **bottom-up** zu bestimmen.
- ▶ Wie bei unseren vorigen Beispiel, wird zunächst nur der **Wert** der Lösung bestimmt, also die Editierdistanz.
- ▶ Um die Editiersequenz auszugeben, müsste wieder ein zweiter Durchlauf erfolgen, der anhand der gespeicherten $D[][]$ Werte die optimale Sequenz rekonstruiert. (Dies funktioniert allerdings nicht für die Speicher-effiziente Variante.)

Implementation der Editierdistanz

```
public class EditDistance
{
    private String a, b;
    private int an, bn;
    private int D[][];

    public EditDistance(String a, String b)
    {
        this.a = a;
        this.b = b;
        an = a.length();
        bn = b.length();
        D = new int[an + 1][bn + 1];
        for (int i = 0; i <= an; i++) {
            D[i][0] = i;
        }
        for (int j = 0; j <= bn; j++)
            D[0][j] = j;
    }
}
```

```
public int distance()
{
    for (int i = 1; i <= an; i++) {
        for (int j = 1; j <= bn; j++) {
            int d1 = D[i][j - 1] + 1;
            int d2 = D[i - 1][j] + 1;
            int d3 = D[i - 1][j - 1] + (a.charAt(i-1) == b.charAt(j-1) ? 0 : 1);
            D[i][j] = Math.min(Math.min(d1, d2), d3);
        }
    }
    return D[an][bn];
}
```

- ▶ In Java gibt `a.charAt(i-1)` das i -te Zeichen des Strings `a` zurück. Dadurch ergibt sich eine Diskrepanz zu der OPT-Formel.

Matrix der gespeicherten Teillösungen

		D	A	G	O	B	E	R	T
	0	1	2	3	4	5	6	7	8
A	1	1	1	2	3	4	5	6	7
L	2	2	2	2	3	4	5	6	7
G	3	3	3	2	3	4	5	6	7
O	4	4	4	3	2	3	4	5	6
D	5	4	5	4	3	3	4	5	6
A	6	5	4	5	4	4	4	5	6
T	7	6	5	5	5	5	5	5	5

Generell:

- ▶ Schöning U. *Algorithmik (Spektrum Lehrbuch)*. Spektrum Akademischer Verlag; 2001. ISBN: 978-3827410924
- ▶ Kleinberg J, Tardos E. *Algorithm Design*. Pearson Education Limited; Auflage: Pearson New International Edition (30. Juli 2013). ISBN: 978-1292023946

Anderes Vorlesungsmaterial:

- ▶ Wayne K. Vorlesung *Theory of Algorithms* (COS 423), Princeton University 2013.
<https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures.php>
- ▶ Röglin H. *Skript zur Vorlesung Randomisierte und Approximative Algorithmen*, Universität Bonn, <http://www.roeglin.org/teaching/WS2011/RandomisierteAlgorithmen/RandomisierteAlgorithmen.pdf>
- ▶ Skiena S. Vorlesung *Algorithms Lecture #16* (CSE 373/548), State University of New York, Stony Brook, 2012.
<https://www3.cs.stonybrook.edu/~algorithm/video-lectures>

Danksagung I

Bei der Darstellung vom *weighted interval scheduling* und dem Rucksack Problem habe ich einige Ideen von den großartigen Folien von Kevin Wayne zu seiner Vorlesung *Theory of Algorithms* (COS 423, Princeton University 2013) aufgenommen. (Seine Vorlesung orientiert sich seinerseits an dem Buch von Kleinberg & Tardos.)

Index

Divide-and-Conquer, 2

Dynamical Programming, 3

Dynamische Programmierung, 3

Editierdistanz, 38

Fibonacci Zahl, 4

Komplexitätsklasse NP, 27

Komplexitätsklasse P, 27

NP-schwer, 27

NP-Vollständigkeit, 27

Traveling Salesman Problem, 29