Algorithmen und Datenstrukturen

Vorlesung #06 – Dynamische Programmierung

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

23 · Mai · 2023

Themen der heutigen Vorlesung

- 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

Divide-and-Conquer

- 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

Dynamische Programmierung

- Lösungsparadigma **Dynamisches Programmieren**.
- Wird häufig für *Optimierungs*probleme verwendet.
- 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.
- Einfache Grundidee, zum Teil sehr hohe Effizienzsteigerung

Generelles Prinzip der Dynamischen Programmierung

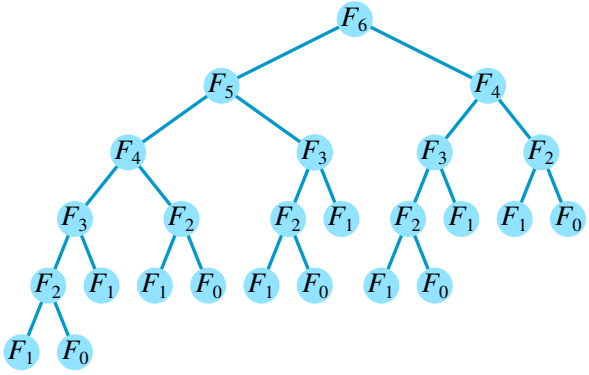
- Ansatz der **Dynamischen Programmierung** (*dynamic programming*):
- Rekursive Formel, die das Problem auf die Lösung von Teilproblemen zurückführt.
- Die Teilprobleme brauchen nicht disjunkt zu sein (im Ggs. zu *Divide-and-Conquer*).
- Deren Lösungen werden gespeichert und für 'größere' Probleme wiederverwendet.
- Üblicherweise wird die Lösung *bottom-up* (*tabulation*) bestimmt.
- In einigen Fällen kann ein *top-down* (*memoization*) 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:

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

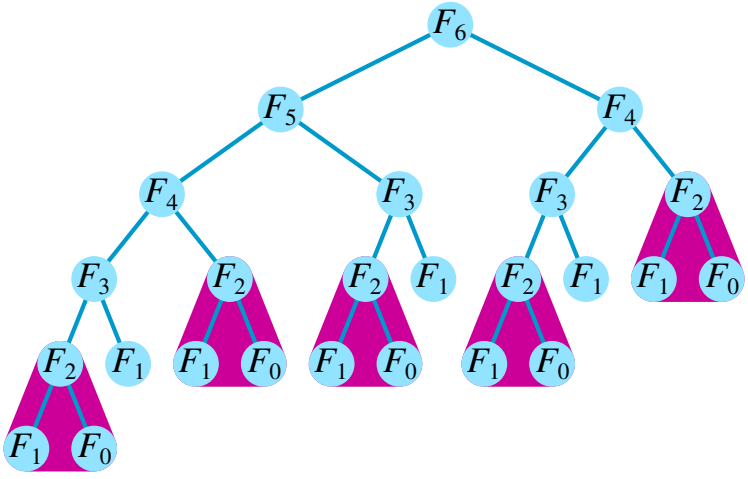


Rekursive Berechnung der Fibonacci-Zahlen

```
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 $(F_k \text{ für } k < n)$ vielfach berechnet werden.
- `F(45) = 1134903170 calculated with 3672623805 recursive calls.`



Berechnung mit Zwischenspeichern *top-down*

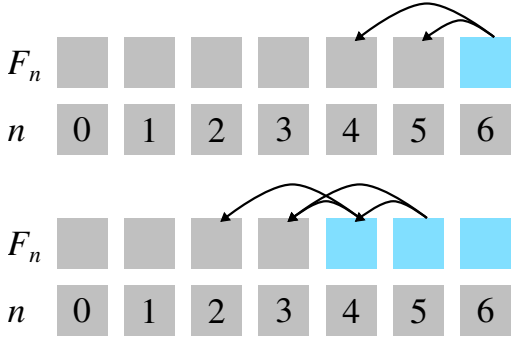
- Speichere berechnete Werte (Teillösungen) in Feld F.
- Noch nicht gespeicherte Werte werden bei Bedarf rekursiv berechnet und gespeichert.

```
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);
  }
}
```

- Der Wert -1 zeigt an, dass der Wert noch nicht berechnet wurde.
- Dieser Ansatz vermeidet Doppeltberechnungen.

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.



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?

- Die Laufzeit ist nun effizient.
- Kann der lineare Speicherbedarf verringert werden?
- 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];
}
```

Von Fibonacci zum Dynamischen Programmieren

- Das Fibonacci Beispiel erfüllt die Voraussetzung des dynamischen Programmierens:

1. Rückführung einer optimalen Lösung auf Teilprobleme *(optimal substructure)*
2. Dabei treten *dieselben* Teilprobleme vielfach auf *(overlapping subproblems)*

- Dann kann die Effizienz durch eine Speicherung der Lösungswerte erhöht werden.
- Die Berechnung kann *top-down* bei Bedarf oder
- *bottom-up* auf Vorrat durchgeführt werden.
- *top-down* ist günstig, wenn nur manche Teillösungen gebraucht werden und im voraus nicht klar ist, welche das sein werden.
- *bottom-up* kann die Speichereffizienz steigern, wenn die Teillösungen nur 'schichtweise' gebraucht werden, also Teillösungen überschrieben werden können.

Dynamisches Programmieren

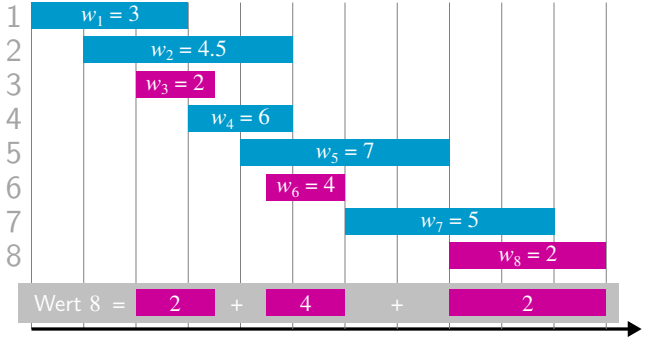
- Redundante Berechnung durch Speicherung von Teillösungen vermeiden!
- Rekursive Formel für Optimierungsproblem aufstellen:
- Optimale Lösung in Abhängigkeit von Teillösungen darstellen (Substrukturanalyse)
- Wenn Anzahl *gleicher* Teillösungen groß ist, kommt der Vorteil der dynamischen Programmierung zum Tragen.
- Andernfalls benötigt dynamische Programmieren meist zu viel Speicher und der Laufzeitgewinn ist nicht so groß.

Dynamisches Programmieren

- Strategie zum Aufstellen der Rekursionsformel:
  - Optimale Lösung als gegeben nehmen und dann ihren Wert in Abhängigkeit von Teillösungen darstellen.
  - Dabei wird oft eine Fallunterscheidung gemacht, sowie Maximum/Min. gebildet.
- Formel über Zwischenspeicherung (*bottom-up* oder *top-down*) implementieren
- Dies ergibt nur den optimalen Wert, nicht die zugehörige Lösung.
  - Manchmal kann die Lösung bei der Bestimmung des optimalen Wertes mit gespeichert werden.
  - Meist ist allerdings ein zweiter Durchlauf erforderlich.

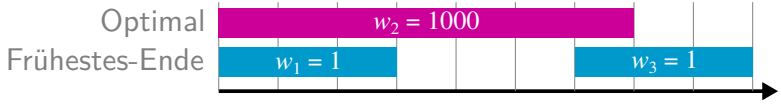
Beispiel 1: Gewichtete Intervallauswahl

- Erweiterung: Auswahl gewichteter Intervalle.
- Es gibt Anfragen $1, \ldots, n$ zur Nutzung einer Ressource (z.B. Raum, Prozessor, Messgerät) in dem Zeitintervall $I_k = [s_k,\ f_k)$ und mit dem Gewicht w_k.
- Ziel: Wähle kompatible Intervalle $S \subseteq \{1, \ldots, n\}$, 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 Frühstes-Ende optimal.
- Bei der gewichteten Intervallauswahl kann er grandios scheitern:

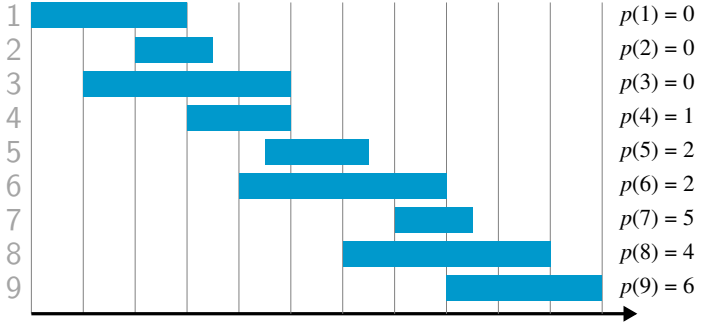


- Neuer Ansatz mit dynamischer Programmierung
- Wie kann die optimale Lösungen aus Lösungen von Teilproblemen ableiten werden?
- Wir nehmen die Intervalle als aufsteigend nach Endzeiten sortiert an.

Bester Vorgänger

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

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



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

Ansatz für Dynamisches Programmieren: Rekursionsgleichung

- **Genereller Ansatz:** Funktion $\textsc{Opt}$ = Wert einer optimalen Lösung
- Hier: $\textsc{Opt}(k)$ = optimaler Wert (maximales Gewicht) für Anfragen $1, \ldots, k$.
- **Substrukturanalyse:** Durch Überlegungen zur Optimalität von Teillösungen stellen wir eine Rekursionsgleichung für $\textsc{Opt}$ auf:
- Ansatzpunkt für die Lösung per **dynamischer Programmierung**
- Mit Fallunterscheidung in der Defintion von $\textsc{Opt}(k)$:
  - Ist Anfrage k in optimaler Lösung enthalten oder nicht?

Rekursionsgleichung für optimale, gewichtete Intervallauswahl

- **Definition:** Sei $\textsc{Opt}(k)$ der optimale Wert für Anfragen $1, \ldots, k$.
- **1. Fall:** Anfrage k ist in Lösung $\textsc{Opt}(k)$ enthalten.
  - Anfragen $p(k)+1, \ldots, k-1$ sind mit der Lösung nicht kompatibel.
  - Die optimale Lösung muss die optimale Lösung für $1, \ldots, p(k)$ umfasssen
  - und beinhaltet zusätzlich das Gewicht w_k.
- **2. Fall:** Anfrage k ist in Lösung $\textsc{Opt}(k)$ nicht enthalten.
  - Die optimale Lösung muss die optimale Lösung für $1, \ldots, k-1$ umfasssen.
- Das ergibt folgende Rekursionsgleichung:

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

Gewichtete Intervallauswahl – Erster Ansatz

- Die Rekursionsgleichung kann wie folgt umgesetzt werden:

```
1  Sortiere Anfragen nach Endzeit      // f_1 ≤ ··· ≤ f_n
2  Berechne p_1, ..., p_n
3  opt(n)
4
5  procedure opt(k)
6   if k = 0
7     return 0
8   else
9     return max( w_k + opt(p_k), opt(k − 1) )
10  end
```

Kritische Betrachtung des Implementationsansatzes

- Die Laufzeit ist schlecht.
- Wie bei der rekursiven Berechnung der Fibonacci Zahlen werden die Lösungen für Teilprobleme vielfach neuberechnet.
- Die Rekursionsformel für $\textsc{Opt}$ liefert **nur die Basis** für dynamische Programmierung
- Darauf aufbauend wird die Speicherung der Zwischenlösungen implementiert.
- Soll *top-down* oder *bottom-up* vorgegangen werden?
- Aus der Rekusionsformel für $\textsc{Opt}$ sehen wir, dass alle Teillösungen benötigt werden, da $\textsc{Opt}(k)$ von $\textsc{Opt}(k-1)$ abhängt.
- Daher ist der *Bottom-up* Ansatz vorzuziehen.

Gewichtete Intervallauswahl mit Dynamischer Programmierung

- 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 ≤ ··· ≤ f_n
2  Berechne p_1, ..., p_n
3
4  m[0] ← 0
5  for k = 1 to n
6    m[k] ← max( w_k + m[p_k], m[k−1] )
7  return m[n]
```

Laufzeit für gewichtete Intervallauswahl

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 $\boldsymbol{O}(n \log n)$ und mit Speicherbedarf in $\boldsymbol{O}(n)$.

- Sortieren der Aufträge nach Endzeiten: $\boldsymbol{O}(n \log n)$
- Berechnung von p_k durch Sortierung nach Anfangszeiten: $\boldsymbol{O}(n \log n)$
- Initialisierung und Berechnung des Feldes $m[]$ jeweils: $\boldsymbol{O}(n)$
- Die Vorbereitung hat also insgesamt eine Laufzeit in $\boldsymbol{O}(n \log n)$ und die eigentliche Intervallauswahl bei gegebenen Sortiertungen $\boldsymbol{O}(n)$.
- Insgesamt ist die Laufzeit also, wie behauptet, in $\boldsymbol{O}(n \log n)$.
- Der Speicherbedarf für Feld $m[]$ lässt sich nicht (wie im Fibonacci Beispiel S. 13) reduzieren, da die Rekursionsformel mit $p(k)$ auf unterschiedliche Vorlösungen zurückgreift. □

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 ∅
4 else if w_k + m[p_k] > m[k−1]
5   return {k} ∪ 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 Gesmtwert maximal ist und ihr Gesamtgewicht die Kapazität nicht überschreitet.

- Formal ist das Ziel $S \subseteq \{1, \ldots, n\}$ 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

- **Komplexitätsklasse P**: Probleme, für die es einen Algorithmus mit Laufzeit in $\boldsymbol{O}(p(n))$ für ein Polynom $p(n)$ in der Eingabegröße n gibt.
- Ganz grob: P $\sim$ halbwegs effizient lösbar, bzw.
- Probleme außerhalb von P sind für größere Eingaben 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.
- **Komplexitätsklasse NP**: 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 $\mathcal{X}$ heißt **NP-schwer** (*NP-hard*), wenn ein beliebiges Problem aus NP in polynomieller Zeit auf eine Lösung von $\mathcal{X}$ zurückgeführt werden kann.
- Ein Problem heißt **NP-vollständig**, wenn es zu NP gehört und NP-schwer ist.

Interlude: NP-Vollständigkeit

- Offensichtlich gilt P⊆NP.
- Es wird von den allermeisten vermutet, dass P≠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 einizges 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).

- **Hamiltonpfad:** Finde einen Pfad, der jeden Knoten eines gegebenen Graphen genau einmal besucht, falls es ihn gibt. (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.)

Erster Ansatz für das 0/1-Rucksack Problem

- **Definition:** Sei $\textsc{Opt}(k)$ der Wert einer optimalen Lösung für Objekte $1, \ldots, k$.
  (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 $1, \ldots, k-1$.
- **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 $\textsc{Opt}$ mitberücksichtigen.

Richtiger Ansatz für das 0/1-Rucksack Problem

- **Definition:** Sei $\textsc{Opt}(k, W)$ der Wert einer optimalen Lösung $\boldsymbol{O}$ für Objekte $1, \ldots, k$ mit Maximalgewicht W. (Reihenfolge der Objekte in $\boldsymbol{O}$ ist beliebig.)
- Falls $w_k > W$ kann Objekt k nicht Teil der Lösung sein. Andernfalls:
- **1. Fall:** Objekt k ist in der Lösung $\boldsymbol{O}$ enthalten.
- Dann besteht die Lösung $\boldsymbol{O}$ in der optimalen Lösung für Teilproblem $1, \ldots, k-1$ mit Maximalgewicht W.
- **2. Fall:** Objekt k ist in der Lösung $\boldsymbol{O}$ nicht enthalten.
- Dann besteht die Lösung $\boldsymbol{O}$ in der optimalen Lösung für Teilproblem $1, \ldots, k-1$ mit Maximalgewicht $W - w_k$.

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

Weitere Entscheidungen zur Implementation

- $\textsc{Opt}$ greift in der ersten Dimension nur auf Vorgänger zu, d.h. $\textsc{Opt}(k, \cdot)$ hängt nur von $\textsc{Opt}(k-1, \cdot)$ ab.
  - Daher könnte bei dem *bottom-up* Ansatz der Speicherbedarf auf zwei Spalten der Matrix M beschränkt werden, analog zu dem Fibonacci Beispiel S. 13.
  - Allerdings kann dann die eigentliche Lösung (welche Objekte ausgewählt werden) nicht ausgelesen werden kann.
  - Denn dazu wird die Matrix der gespeicherten Teillösungen ein zweites Mal durchlaufen, siehe Seite 26.
- Bei dieser Rekursionsformel für $\textsc{Opt}$ werden nicht alle Teillösungen benötigt.
  - Daher benutzen wir hier den *top-down* Ansatz.
    Der *bottom-up* Ansatz ist leichter zu implementieren, aber 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  }
```

Knapsack Beispiel Client

```
  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 $\boldsymbol{O}(KW)$, wobei K die Anzahl der Objekte und W die Kapazität des Rucksacks ist. Der Speicherbedarf ist ebenfalls in $\boldsymbol{O}(KW)$.

Beweis.

- Die Methode *opt()* berechnet den Wert durch die Abfrage in Zeile 23 für jedes Paar (k, w) nur einmal. Dies geht jeweils in $\boldsymbol{O}(1)$.
- Somit ist die Laufzeit von *opt()* und von der Initialisierung in $\boldsymbol{O}(KW)$. □

- 0/1-Knapsack ist NP-vollständig und die Laufzeit ist in $\boldsymbol{O}(KW)$??
- Auflösung: Diese Abschätzung 'zählt nicht', da sie nicht nur von der Anzahl der Eingabeobjekte, sondern auch von einem Eingabewert abhängt.

Laufzeit der Knapsack Implementierung

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 $\boldsymbol{O}(K2^n)$, 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.

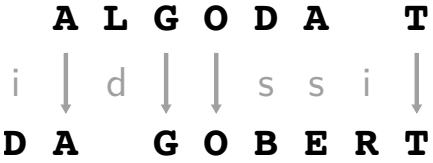
- Die elementaren Buchstabenoperationen sind:
  - **D:** Einen Buchstaben löschen (*delete*)
  - **I:** Einen Buchstaben einfügen (*insert*)
  - **S:** Einen Buchstaben ersetzen (*substitute*)

- 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.

Editieroperationen in Action

- 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?

Editierdistanz durch Dynamisches Programmieren

- Definiere $\textsc{Opt}$ gemäß dem Ansatz der dynamischen Programmierung.
- Zunächst einige Schreibweisen:
- 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

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

Editierdistanz durch Dynamisches Programmieren

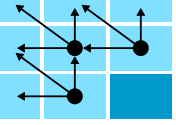
- $\textsc{Opt}(i, j)$ sei die Editierdistanz zwischen den Substrings $a[1 : i]$ und $b[1 : j]$.
- Für $j = 0$ soll String $a[1 : i]$ in einen leeren String umgewandelt werden: i-mal Löschen (Operation **D**): $\textsc{Opt}(i, 0) = i$.
- Für $i = 0$ soll ein leerer String in den String $b[1 : j]$ umgewandelt werden: j-mal Einfügen (Operation **I**): $\textsc{Opt}(0, j) = j$.
- 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:

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

Matrix der gespeicherten Teillösungen

- Wir betrachten die Matrix der Teillösungen für a = ALGODAT und b = 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.
- Wie bei unseren vorigen Beispielen, 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;
  }
```

Implementation der Editierdistanz

```
  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**

Literatur

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/~algorith/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