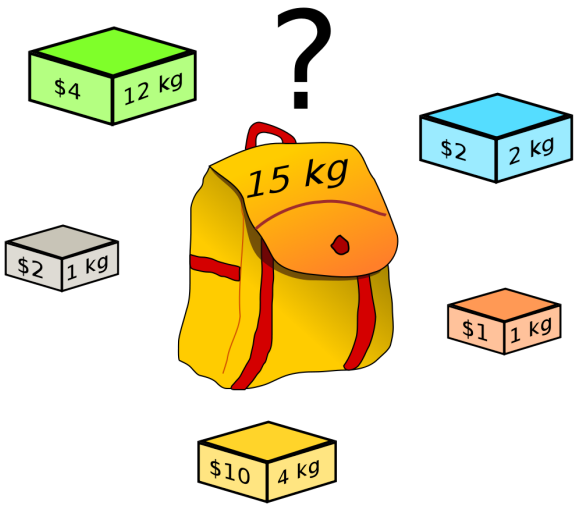
Dynamische Programmierung anhand des Rucksackproblems



David Tyndall

April 2023

Inhaltsverzeichnis

1 Einführung

1.1 Problemstellung

Stelle dir folgende Situation vor: Ein Korbhändler möchte mit seinem Wagen zum Markt fahren. Er hat zu Hause einen grossen Vorrat an verschiedenen Körben, die er verkaufen kann. Er kann recht zuverlässig einschätzen, welchen Gewinn er pro Korb erzielen wird. Leider passen nicht alle Körbe aus dem Lager in den Wagen. Das maximale Zuladegewicht darf nicht überschritten werden. Natürlich möchte er den Gewinn für diesen Markttag optimieren und deshalb eine möglichst gute Auswahl treffen, welche Körbe er an diesem Tag einpackt.

Diese Art von Problem taucht in verschiedenen Formen immer wieder auf. Es soll eine Auswahl aus einer Menge von Objekten getroffen werden. Dabei dürfen die gewählten Objekte eine gewisse Grenze nicht überschreiten, sollen aber nach einem bestimmten Kriterium maximiert werden. Solche Probleme nennt man 0/1-Rucksackproblem.[1] In dieser Lerneinheit wollen wir eine Methode entwickeln, welche diese Art von Optimierungsproblem möglichst effizient lösen kann.

Nachfolgend aufgeführt sind einige Abkürzungen, die in dieser Lerneinheit verwendet werden:

- n: Die Anzahl Waren/Objekte, die zur Auswahl stehen
- i: Der *Index*, ein Bezeichner für die Nummer des betrachteten Objektes. i kann Werte von 1 bis n annehmen.
- g_i: Das Gewicht (oder Volumen, Platzbedarf, etc.) des i-ten Objektes
- p_i: Der Gewinn (oder allgemein: Preis, Wert, etc.) des Objektes
- c oder *cap*: Die Kapazität (engl. capacity) des Wagens, Behälters, etc.

Aufgabe 1: Finde selbst ein anderes Beispiel des 0/1-Rucksackproblems. Überlege dir, was die oben genannten Grössen n, g_i, p_i und c in deinem Beispiel bedeuten.

[1] Der Begriff „Rucksack“ stammt daher, dass man verschiedene Objekte in einen Rucksack mit begrenzter Kapazität füllen soll. 0 steht für „Das Objekt wird nicht eingepackt“, 1 steht für „Das Objekt wird eingepackt“. Werte dazwischen sind nicht erlaubt, da nicht nur ein Bruchteil des Objektes eingepackt werden kann. Diese Variante des Rucksackproblems ist eines der klassischen NP-vollständigen Optimierungsprobleme. Siehe auch `https://de.wikipedia.org/wiki/Rucksackproblem`

1.2 Aufwandsabschätzung Brute Force

Aufgabe 2: Angenommen, der Händler hat sich eine einfache Strategie überlegt, nach welcher er die zu verladenden Körbe auswählt: Im Inventar ist jeder einzelne Korb im Lager aufgelistet. Hinter jeden Inventareintrag schreibt er entweder eine 1, d. h. er packt den Korb ein, oder eine 0, d. h. er packt den Korb nicht ein. Für die Berechnung berücksichtigt er das Gesamtgewicht der bisher einzupackenden Körbe nicht, erst am Schluss bildet er die Summe und schaut, ob diese Kombination in seinem Wagen Platz hat.

Berechne die Anzahl Möglichkeiten, die er betrachten muss, wenn im Lager die folgende Anzahl Körbe n liegt:

n	1	2	3	4	10	100	n (allgemein)
Anz. Möglichkeiten							

Mit zunehmender Anzahl Elemente nimmt die Anzahl zu prüfender Fälle exponentiell zu. Dies ist eine sehr ungünstige Ausgangslage und zeigt, wie schwierig es ist, die exakte Lösung des Problems für grosse Eingaben zu finden.

1.3 Eine andere Strategie?

Aufgabe 3: Findest du eine Strategie, die effizienter ist als diejenige von Aufgabe 2?

- Teste deine Strategie für verschiedene Anzahlen von Körben (1, 2, 3, 5, 10, usw.) mit verschiedenen Gewichten und verschiedenen Kapazitäten. Funktioniert sie immer?
- Findet deine Strategie immer die optimale Lösung – auch dann, wenn du eine ungünstige Ausgangslage wählst?
- Kannst du die Anzahl Schritte abschätzen, die deine Strategie für 1, 2, 3, 4, 10, 100, n Körbe benötigt?

2 Ein einfacheres Problem zum Start

Wir wollen nun eine Methode entwickeln, welche möglichst schnell zum exakten Ziel führt. Hierzu machen wir zuerst zwei Vereinfachungen:

- Wir nehmen an, dass der Gewinn genau gleich ist wie das Gewicht jedes Objektes (sogenanntes *einfaches Rucksackproblem*), also $w_i = p_i$.
- Wir starten zuerst nur mit einem Objekt, erweitern dann auf zwei, drei, usw.

Aufgabe 4: Eine einfache Einstiegsaufgabe: Nehmen wir an, der Händler hat nur einen einzigen Korb auf Lager und dieser hat die Masse 6 kg. Was ist (mit obigen Vereinfachungen) der maximale Gewinn, den der Händler bei einer Transportkapazität von 0 kg, 1 kg, …, 13 kg erzielen kann?

Schreibe deine Resultate in die Tabelle. Finde zudem für jedes „?“ eine geeignete Bezeichnung der entsprechenden Spalten / Zeilen / Zellenwerte. Mit *Index* ist die Nummer des Korbes gemeint.

? \ ?	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Index	?	?													
1	6														

Mit nur einem Korb ist der Tabelleninhalt natürlich schnell vervollständigt. Nun wollen wir uns überlegen, wie eine solche Tabelle mit mehreren Körben funktionieren kann.

Aufgabe 5: Nehmen wir an, der Händler hat zwei Körbe, einen mit der Masse 6 kg und einen mit der Masse 3 kg auf Lager. Die Tabelle aus Aufgabe 4 soll entsprechend erweitert werden, dabei sollen die „?“-Beschriftungen weiterhin gültig bleiben. Die Zeile mit Index 1 soll unverändert bleiben. Füge eine weitere Zeile für den zweiten Korb hinzu. Achtung: In dieser Zeile dürfen beide Körbe zum Beladen des Wagens verwendet werden.

Wenn du fertig bist, erweitere deine Tabelle analog für den Fall, dass der Händler nun drei Körbe auf Lager hat. Die Massen betragen 6 kg, 3 kg und 5 kg.

Bei Aufgabe 5 musstest du schon ein wenig mehr aufpassen beim Einfüllen der Zelleninhalte. Hast du bereits eine Regel entdeckt, mit welcher sich die Werte der einzelnen Zellen zuverlässig füllen lassen?

Aufgabe 6: Die Menge der Körbe auf Lager sei $\{6\,\text{kg}, 3\,\text{kg}, 5\,\text{kg}, 11\,\text{kg}, 7\,\text{kg}\}$ und die Kapazität des Wagens weiterhin 13 kg. Erstelle wiederum eine entsprechende Tabelle. Finde/Verifiziere ein Schema resp. eine Regel, mit der du die Zelleninhalte korrekt einfüllen kannst. Formuliere die gefundene Regel für eine beliebige Zelle in Worten.

Tipp/Trick: Trage den *ersten* Korb erst in der *zweiten* Tabellenzeile ein und stelle in der *ersten* Zeile *keinen* Korb dar. Die erste Zeile ist dann mit lauter Nullen gefüllt. Das hilft, um die gesuchte Regel allgemein zu formulieren.

Objekt \ Kapzität	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Index i / Gewicht g_i	summierter Gewinn													

3 Interpretation und Algorithmus

Die eben gefundene Regel bildet den Kern des Algorithmus, den wir in dieser Unterrichtseinheit betrachten wollen. Zum besseren Überblick sei die Lösung von Aufgabe 6 nochmals abgebildet und ausführlich beschrieben.

Objekt \ Kapzität		0	1	2	3	4	5	6	7	8	9	10	11	12	13
Index i	Gewicht g_i	summierter Gewinn													
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	6	0	0	0	0	0	0	6	6	6	6	6	6	6	6
2	3	0	0	0	3	3	3	6	6	6	9	9	9	9	9
3	5	0	0	0	3	3	5	6	6	8	9	9	11	11	11
4	11	0	0	0	3	3	5	6	6	8	9	9	11	11	11
5	7	0	0	0	3	3	5	6	7	8	9	10	11	12	13

3.1 Bedeutung der Tabelle

Was bedeuten die verschiedenen Bereiche der Tabelle?

Achtung: Im einfachen Rucksackproblem gilt ja, dass Gewinn und Gewicht gleich sind. Demzufolge können diese beiden Begriffe vorläufig als Synonyme verwendet werden.

Kopfspalte „Index“: Der Index i gibt die Nummer des Objektes an, das wir aktuell betrachten. Die zugehörige Tabellenzeile beschreibt, welcher Gewinn mit den bisherigen Objekten (also Indizes $0, \ldots, i$) erreicht werden kann.

Kopfspalte „Gewicht“: Das Gewicht g_i muss nicht unbedingt Teil der Tabelle sein. Für die manuelle Bearbeitung ist es aber hilfreich, die Information gleich am passenden Ort zur Hand zu haben.

Kopfzeile „Kapazität“: Der Zahlenwert beschreibt die Kapazität, welche für jenen Rechenschritt gelten soll. Die letzte Zahl der Zeile ist die reale Kapazität des Gefässes/Wagens. Alle Gewinne, die in der Spalte mit Kapazität k stehen, können erreicht werden mit Objekten, deren Gesamtgewicht k nicht übersteigt.

Beispiel: Betrachten wir die Spalte mit Kapazität 8 (fortan „Spalte 8“ genannt) auf der Zeile mit Index 1 (fortan „Zeile 1“ genannt). Der Wert 6 der Zelle bedeutet, dass der Gewinn 6 erzielt werden kann, ohne dass das Gewicht 8 (=Spaltennummer) überschritten wird.

Tabelleninhalt „Gewinn“: Der Wert einer einzelnen Zelle mit den „Koordinaten“ (Zeile i, Spalte j), oder kurz (i, j), beschreibt den optimalen Gewinn, der mit einer Auswahl aus den Objekten $0, \ldots, i$ erreicht werden kann, wenn dabei das Gesamtgewicht j nicht überschritten werden soll.

Für die rot markierte Zelle (Koordinaten (4, 10)) heisst das beispielsweise, dass höchstens der Gewinn 9 erreicht werden kann, sofern das Gewicht 10 nicht überschritten werden soll, und nur die Objekte 1-4 (Gewichte $\{6\,\text{kg}, 3\,\text{kg}, 5\,\text{kg}, 11\,\text{kg}\}$) zur Auswahl stehen.

Aufgabe 7: Wo in der Tabelle findet man den bestmöglichen Gewinn, den der Händler mit einem Wagen der Kapazität 13 kg erreichen kann?

3.2 Algorithmus

Nun wollen wir den gesamten Algorithmus zur Lösung des einfachen Rucksackproblems in Worten beschreiben und zugleich begründen.

1. Die erste Zeile (Index $i = 0$, d. h. kein Objekt, also auch kein Gewicht) füllt man mit Nullen. Dies ermöglicht, dass wir ab der zweiten Zeile auf die jeweils vorherige Zeile zurückgreifen können.

2. Man arbeitet sich Zeile für Zeile, und innerhalb der Zeile von links nach rechts, vor. Betrachten wir beispielsweise die hellgrün markierte Zelle (5/10). Da wir uns auf der 5. Zeile befinden, ist das Objekt, welches in diesem Schritt neu hinzugekommen ist, dasjenige mit Index 5 und Gewicht (=Gewinn) 7 (dunkelgrün markiert).

   Zur Bestimmung des Wertes der hellgrünen Zelle betrachtet man die darüberliegende Zeile, also Zeile 4. Das ist ja das bisherige Optimum, und nun will man schauen, ob man dieses Optimum unter Verwendung des neuen Objektes noch verbessern kann. Grundsätzlich kommen zwei Varianten in Frage:

   a) Das neue Objekt wird nicht verwendet: Das Gesamtgewicht verändert sich nicht, daher bleibt die Spalte gleich. Wir müssen also die rot markierte Zelle betrachten, welche sich direkt oberhalb der zu berechnenden Zelle befindet. Ihr Gewinn 9 steht unverändert zur Auswahl, da ja nichts hinzugefügt wird.

   b) Das neue Objekt wird verwendet: Weil sich das Gesamtgewicht durch die Verwendung des aktuellen Objektes um 7 kg erhöht, dürfen wir maximal den Wert verwenden, der 7 kg leichter ist als das aktuelle Objekt, also 7 Spalten weiter links liegt. Es handelt sich um die gelb markierte Zelle mit Wert 3. Verwenden wir das aktuelle Objekt, wird also der Gewinn

   $$\text{Neuer Gewinn} = \text{Bisheriger Gewinn} + \text{Gewinn des neuen Objektes} = 3 + 7 = 10$$

   betragen.

   Selbstverständlich funktioniert diese Variante nur, wenn das Gewicht des aktuellen Objektes kleiner oder gleich der betrachteten Spalte ist. Sonst würden wir ja ein Objekt einladen, das zu schwer wäre für diese Spalte.

   Für die hellgrüne Zelle wird der grössere Gewinn aus den beiden Varianten gewählt, in diesem Beispiel also die Variante b) mit Gewinn 10.

   Man fährt nun mit der nächsten Zelle fort, bis die ganze Tabelle ausgefüllt ist.

Aufgabe 8: Erkennst du in der obigen Erklärung das Prinzip der vollständigen Induktion? Nach welcher Grösse geschieht die Induktion? Wo findet die Verankerung statt? Wo geschieht der Induktionsschritt?

Aufgabe 9: Löse unter Verwendung des oben beschriebenen Algorithmus das folgende 0/1-Rucksackproblem, indem du die entsprechende Tabelle von Hand erstellst und ausfüllst:

Der Händler hat Körbe mit den Massen $\{4\,\text{kg}, 3\,\text{kg}, 2\,\text{kg}, 7\,\text{kg}\}$, und sein Wagen hat eine Kapazität von $10\,\text{kg}$.

Das nächste Ziel ist, den Algorithmus formal zu beschreiben.

Aufgabe 10: Folgender Code sei schon gegeben. Formuliere die fehlenden Zeilen in Python oder in Pseudocode, damit `tabelle` gemäss vorher besprochenem Schema gefüllt wird. Gib zum Schluss sowohl die Tabelle als auch den optimalen Gewinn aus.

Implementiere anschliessend deine Lösung am Computer.

Achte auf den Trick, der bei der Definition der Gewichte auf Zeile 3 gemacht wird: Der erste Eintrag der Liste lautet `None`. Die Idee ist, dass das erste Objekt mit `g[1]` angesprochen werden kann. Das ermöglicht eine einfachere Verwendung der Indizes ohne ständige Umrechnung.

```
1 from pprint import pprint   # Ermöglicht mit pprint(tabelle)
2                             # eine schön formatierte Ausgabe
3 # Initalisierung der Variablen
4 g = [None, 6, 3, 5, 11, 7]  # Gewicht=Gewinn der Objekte
5 n = len(g) - 1  # Anzahl der Objekte
6 cap = 13        # Kapazität des Gefässes
7 tabelle = [[0 for j in range(cap + 1)] for i in range(n + 1)]
8     # Kreiert eine Tabelle mit n+1 Zeilen und cap+1 Spalten,
9     # gefüllt mit 0
```

4 Erweiterungen

Wir haben im vorherigen Kapitel einen funktionierenden und einigermassen effizienten Algorithmus gefunden. Nun wollen wir prüfen, ob er sich auch auf andere, allgemeinere Fälle erweitern lässt.

4.1 Erweiterung des Definitionsbereichs: Mehrfach vorhandene Objekte

Der Händler könnte einige Körbe mehrfach auf Lager haben. Streng genommen heisst das sich ergebende Problem nicht mehr 0/1-Rucksackproblem, weil ja ein Korb, der mehrfach vorhanden ist, auch mehrfach gewählt werden kann, also nicht nur 0- oder 1-mal. Ausserdem bilden die zur Auswahl stehenden Körbe keine *Menge*, da sie teilweise mehrfach vorhanden sein können. Man spricht dann von einer *Multimenge*.

Aufgabe 11: Findest du eine Möglichkeit, wie man den in Kapitel 3.2 beschriebenen Algorithmus auf ein Korblager mit folgenden mehrfach vorhandenen Körben {3 kg (2x vorhanden), 5 kg (1x vorhanden), 6 kg, (2x vorhanden)} anwenden kann? Beschreibe deine Lösungsidee in Worten.

Aufgabe 12: Kannst du beweisen, dass mit den Bedingungen von Aufgabe 11 ein Wagen mit der Kapazität 13 kg nicht vollständig gefüllt werden kann?

4.2 Effizienzsteigerung: Schwere Objekte

Was passiert, wenn einzelne Objekte grösser sind als die Kapazität des Wagens?

Aufgabe 13:

a) Funktioniert der unveränderte Algorithmus, wenn Objekte vorhanden sind, deren Gewicht die Kapazitätsgrenze überschreiten?

b) Wie viele Tabellenzeilen sind notwendig, um bei einer Korbmenge von {3 kg, 14 kg, 5 kg, 6 kg, 20 kg} und einer Kapazitätsgrenze von 13 kg die optimale Lösung zu finden?

c) Welchen Schritt kann man dem Algorithmus für diesen Fall hinzufügen, damit er nicht unnötig viel rechnet?

4.3 Verallgemeinerung: Eine Lösung für das allgemeine 0/1-Rucksackproblem

Bisher haben wir uns bei den Lösungen auf das spezielle 0/1-Rucksackproblem konzentriert, bei dem die Gewichte gerade dem Gewinn entsprechen. Im allgemeinen Fall sind die Gewichte jedoch von den Gewinnen/Preisen verschieden. Der Händler könnte also vor folgendem Problem stehen: Er hat Körbe mit den Gewichten {1 kg, 3 kg, 4 kg, 4 kg, 2 kg} und geht von einem Gewinn von {2 Fr, 3 Fr, 3 Fr, 4 Fr, 4 Fr} (gelistet gemäss der Korbreihenfolge) aus. Die Kapazitätsgrenze liegt bei 8 kg.

Aufgabe 14: Grundsätzlich wollen wir die Struktur der Tabelle beibehalten. Der Übersicht halber erweitern wir sie jedoch durch eine Spalte mit dem Gewinn der einzelnen Körbe.

a) Überlege dir, wie du die Tabelle jetzt vervollständigen musst.

Objekt \ Kapzität			0	1	2	3	4	5	6	7	8
Index i	**Gewicht** g_i	**Gewinn** p_i	**?**								
0	0	0									
1	1	2									
2	3	3									
3	4	3									
4	4	4									
5	2	4									

b) Was hat sich an der Interpretation/Bedeutung der Werte im Vergleich mit der Beschreibung auf Seite 5 verändert?

Aufgabe 15: Was bedeutet es, dass in Aufgabe 14 die Zellen (5,7) und (5,8) den gleichen Wert besitzen?

Aufgabe 16: Bei der Implementierung des Algorithmus müssen ebenfalls Anpassungen vorgenommen werden. Nachfolgend ist der Algorithmus abgebildet, der das einfache Rucksackproblem von Aufgabe 10 löst. Allerdings sind die Variablen bereits auf das neue Problem angepasst, beachte insbesondere die neue Liste mit Preisen auf Zeile 5.

Muss sonst noch etwas geändert werden? Wenn ja, wie viele Änderungen benötigst du, um das Problem korrekt zu lösen?

```
1  from pprint import pprint  # Ermöglicht mit pprint(tabelle)
2                             # eine schön formatierte Ausgabe
3  # Initalisierung der Variablen
4  g = [None, 1, 3, 4, 4, 2]  # Gewicht der Objekte
5  p = [None, 2, 3, 3, 4, 4]  # Gewinn/Preis der Objekte
6  n = len(g) - 1  # Anzahl der Objekte
7  cap = 8         # Kapazität des Gefässes
8  tabelle = [[0 for j in range(cap + 1)] for i in range(n + 1)]
9      # Kreiert eine Tabelle mit n+1 Zeilen und cap+1 Spalten,
10     # gefüllt mit 0
11
12 # Tabelle füllen
13 for i in range(1, n + 1): # Zeilen durchlaufen
14     for j in range(1, cap + 1): # Spalten durchlaufen
15         if j < g[i]: # Testen, ob genügend weit rechts
16             tabelle[i][j] = tabelle[i-1][j]
17         else:
18             tabelle[i][j] = max(tabelle[i-1][j],
19                                 tabelle[i-1][j - g[i]] + g[i])
20
21 # Ausgabe
22 pprint(tabelle)
23 print('Optimaler Wert:', tabelle[n][cap])
```

5 Wrapping up

5.1 Aufwandsabschätzung

Wie effizient ist die gefundene Methode?

Hierzu wollen wir uns grob überlegen, wie viele Operationen der eigentliche Berechnungsteil des Algorithmus (Zeilen 12-19 im Code von Aufgabe 16) durchführt.

- In Zeile 15 wird ein einfacher Vergleich und ein Nachschlagen eines Wertes in der Liste durchgeführt: 2 Operationen.
- Entweder wird die Zeile 16 oder es werden die beiden Zeilen 18-19 ausgeführt. Die Zeilen 18-19 sind eine Verallgemeinerung der Zeile 16, daher reicht es, nur diese beiden zu betrachten:
  - Um die beiden Argumente von `max` zu berechnen, finden einige Additionen/Subtraktionen sowie das Nachschlagen in Listen statt. Dies sind alles einfache Operationen, die höchstens eine konstante Anzahl Schritte benötigen.
  - Die Funktion `max` macht ebenfalls nur einen Vergleich: 1 Operation.

Insgesamt findet also im inneren Teil der Schleifen eine konstante (kleine) Anzahl an Operationen statt, um einen einzelnen Tabellenwert zu berechnen.

Aufgabe 17: Wie viele Tabellenwerte müssen berechnet werden? Wie viele Operationen benötigt also eine komplette Tabellenberechnung? Von welchen Eingabegrössen hängt die Laufzeit ab?

Aufgabe 18: Angenommen, die Kapazität betrage jeweils das Doppelte der Anzahl Objekte n. Wie viele Tabellenwerte t müssen dann berechnet werden für folgende n?

Vergleiche deine Resultate mit denen von Aufgabe 2. Unter welchen Umständen lohnt sich der Einsatz der Tabellenmethode?

n	2	3	4	10	100	n (allgemein)
t						

5.2 Der grössere Kontext

Der Algorithmus, den wir hier kennen gelernt haben, ist ein typischer Vertreter des sogenannten *dynamischen Programmierens*. Die dynamische Programmierung umfasst verschiedene Algorithmen/Methoden, bei denen die Zwischenwerte systematisch abgespeichert werden. Dies erlaubt oft grosse Effizienzgewinne. Z. B. ist es auch möglich, bei rekursiven Algorithmen Zwischenresultate abzuspeichern, damit nicht jeder Wert immer wieder neu berechnet werden muss. So können grosse Teile des Rekursionsbaumes übersprungen werden.

Berechnen wir aus den einzelnen Elementen systematisch immer grössere Werte und setzen diese zusammen, bis wir das gesamte Problem gelöst haben, spricht man vom *Bottom-Up-Ansatz*; die hier vorgestellte Tabelle entspricht diesem Ansatz. Wenn man dagegen (z. B. bei der Rekursion) ganz oben beginnt, und immer kleiner werdende Probleme löst, spricht man vom *Top-Down-Ansatz*.

Mit der betrachteten Methode lassen sich nur ganzzahlige Gewichte berechnen. Indem man alle Gewichte mit einem passenden Faktor multipliziert, so dass sie ganzzahlig werden, könnte man auch gewisse Rucksackprobleme mit rationalen Gewichten lösen. Allerdings muss auch die Kapazität mit dem entsprechenden Faktor multipliziert werden. Sie kann dadurch rasch sehr gross werden, was die Tabellenmethode wiederum ineffizient macht.

Es gibt auch noch andere Varianten des Rucksackproblemes, z. B. das fraktionale Rucksackproblem. Hierbei werden auch Bruchteile von Objekten zugelassen. Dieses Problem ist aber algorithmisch viel einfacher lösbar als das 0/1-Rucksackproblem.[2]

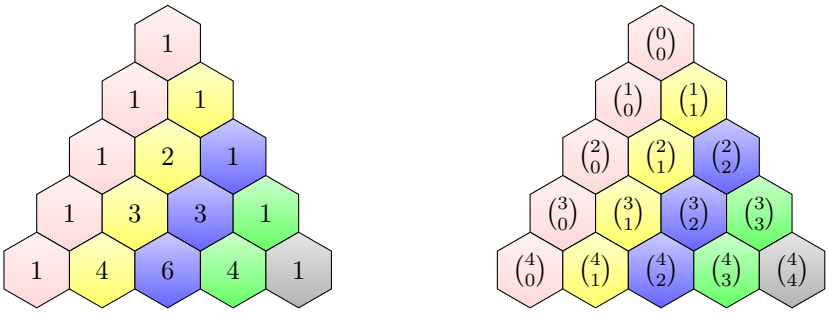
[2] vgl. z. B. `https://www.javatpoint.com/fractional-knapsack-problem`

5.3 Zwei Aufgaben zum Abschluss

Aufgabe 19: Greg liebt Antiquitäten. Er geht beim Antiquitätenhändler einkaufen, wo jedes der verfügbaren 20 Objekte nur einmal vorhanden ist. Er hat 30 Franken dabei. Für dieses Geld möchte er Objekte einkaufen, die ihm möglichst viel Freude bereiten.
Preise der Objekte (in Fr.): {1, 2, 4, 2, 5, 4, 3, 3, 5, 4, 1, 2, 4, 2, 5, 4, 3, 3, 5, 4}
Die Objekte bereiten die Freude: {1, 3, 5, 3, 2, 6, 4, 2, 8, 4, 5, 3, 5, 3, 2, 6, 4, 2, 8, 4}
(beliebig wählbare Einheit, z. B. ng Dopamin pro Liter Blut)

Welche Freude kann Greg maximal erreichen?

Aufgabe 20: Aus dem Mathematikunterricht kennst du sicher das Pascal'sche Dreieck. Eine neue Zeile wird jeweils aus den Werten der darüberliegenden Zeile berechnet, indem die Summe der direkt angrenzenden Werte gebildet wird.



Finde zuerst eine Möglichkeit, das Pascal'sche Dreieck als Tabelle darzustellen. Implementiere anschliessend ein Programm, das diese Tabelle Zeile für Zeile unter Verwendung der darüberliegenden Zeile füllt.

Bemerkung: Mit dem Lösen der Aufgabe 20 hast du einen Algorithmus gefunden, der mit Hilfe dynamischer Programmierung die Binomialkoeffizienten berechnet. Um $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ zu finden, hast du nämlich zwei vorher berechnete und in der Tabelle abgespeicherte Werte verwendet.

6 Quellen

Das Titelbild wurde am 8. März der Seite https://de.wikipedia.org/wiki/Rucksackproblem entnommen.
Das Bild in Kapitel 1.1 wurde am 8. März der Seite `www.alamy.com` entnommen.
Die Bilder in Aufgabe 20 wurden mit Hilfe des LaTeX-Paketes pascaltriangle generiert.

7 Hinweise für die Lehrperson

Ziele

Dieses Unterrichtsmaterial soll in ca. 4 Lektionen Unterstützung bieten, um

- das Rucksackproblem in seinen verschiedenen Varianten kennen zu lernen,
- den Bottom-Up-Ansatz des dynamischen Programmierens (Tabelle) kennen zu lernen und zu verstehen,
- den Bottom-Up-Ansatz auch auf andere Probleme übertragen zu können und kleinere Anpassungen im Algorithmus selber vornehmen zu können.

Notwendige/nützliche Vorkenntnisse

- Die Schüler und Schülerinnen sollten verschachtelte Listen kennen (Werte abspeichern und aufrufen, Umgang mit Indizes).
- Die wenigen vorhandenen Programmausschnitte wurden in Python geschrieben. Die Passagen sind jedoch mit entsprechender Anleitung durch die Lehrperson auch Schülerinnen und Schülern ohne Python-Kenntnisse zugänglich.
- Kenntnis von Rekursion ist von Vorteil, aber nicht zwingend. Explizit erwähnt wird die Rekursion erst in Kapitel „Der grössere Kontext“.
- Dieses Unterrichtsmaterial, das direkt mit dem 2-dimensionalen Rucksackproblem startet, richtet sich an Schülerinnen und Schüler, die schon Berührung mit dem Prinzip der dynamischen Programmierung hatten. Der Einstieg mit einem 1-dimensionalen Problem wie etwa der Berechnung der Fibonacci-Zahlen mit Zwischenspeicherung würde sich beispielsweise als einfacherer Einstieg eignen.
- Für Aufgabe 8 werden Kenntnisse der vollständigen Induktion vorausgesetzt. Falls dieses Thema noch nicht bekannt ist, kann die Aufgabe problemlos weggelassen werden.
- Für Aufgabe 20 sind Kenntnisse des Pascal'schen Dreiecks resp. Binomialkoeffizienten unabdingbar. Auch diese Aufgabe kann problemlos weggelassen werden.
- Für die Laufzeitabschätzung ist die Kombinatorik hilfreich, aber nicht zwingend, genauso wie die $\mathcal{O}$-Notation.

Das Material wurde für Schülerinnen und Schüler im 11./12. Schuljahr entwickelt. Sofern die oben genannten Voraussetzungen erfüllt sind oder die entsprechenden Aufgaben ausgelassen werden, ist das Material auch im 9./10. Schuljahr anwendbar.

Einsatz im Unterricht

Die Aufgaben sind dazu gedacht, wesentliche Inhalte zu vertiefen oder zu entdecken. Sie sollten von den Schülerinnen und Schülern in Einzel- oder Gruppenarbeit gelöst werden. Ein schrittweises Vorgehen ist wünschenswert, jede Lösungsphase sollte einhergehen mit einer anschliessenden Besprechung. Dies gilt in besonderem Masse für die Aufgaben 4, 6 und 17. Da die Lerneinheit schon recht viel Selbststudium und eigenes Entdecken umfasst, sollten die Aufgaben von/mit der Lehrperson besprochen werden, und nicht einfach nur die Lösungen abgegeben werden.

Lösungen der Aufgaben

Aufgabe 1: Beispiele für das 0/1-Rucksackproblem:

- Dieb mit Rucksack (n resp. i: Objekte im Haushalt, g: Gewicht des Objektes, p: Wert des Objektes, c: Kapazität des Rucksacks)
- Team aus Bewerbern zusammenstellen (n resp. i: Bewerber, g: Lohn des einzelnen Bewerbers, p: Effektivität des Bewerbers, c: Lohnsumme aller Bewerber)
- Container mit Items optimal beladen (n resp. i: zu verladende Items, g: Längenmasse der Items, p: Volumen der Items, c: Längenmasse des Containers)
- Portfolio einer Investition (n resp. i: zur Auswahl stehende Aktien, g: zu investierender Betrag pro Aktie, p: erwarteter Gewinn, c: zu investierender Gesamtbetrag)

Aufgabe 2:

n	1	2	3	4	10	100	n (allgemein)
Anz. Möglichkeiten	2	4	8	16	1024	$2^{100} \approx 1.3 \cdot 10^{30}$	2^n

Aufgabe 3: *Hinweis an die Lehrperson:* Diese Aufgabe birgt die Gefahr, dass die Schülerinnen und Schüler die Zeit vergessen; deshalb ist es ratsam, eine zeitliche Beschränkung von etwa 15 Minuten anzusetzen. Die Idee ist nicht, dass die Schüler perfekte Lösungen finden; viel eher soll mit dieser Aufgabe ein Gefühl für die Komplexität des Problems vermittelt werden. Die Diskussion einiger Resultate aus der Klasse kann dies noch unterstützen.

Es sind verschiedene Lösungen denkbar. Als Beispiel sei hier der Greedy-Ansatz gezeigt: Der grösste noch passende Korb wird ausgewählt, bis entweder die Kapazität des Wagens erreicht ist oder kein Korb mehr vorhanden ist, der noch hineinpasst. Es kann einfach gezeigt werden, dass dieser Ansatz nicht immer die optimale Lösung findet; z. B. mit $c = 10\,\text{kg}$, $g = p = \{8\,\text{kg}, 6\,\text{kg}, 3\,\text{kg}\}$. Der Aufwand beträgt $\mathcal{O}(n)$, weil jedes Element maximal einmal betrachtet wird.

Aufgabe 4:

Objekt \ Kapzität		0	1	2	3	4	5	6	7	8	9	10	11	12	13
Index	Gewicht	Gewinn													
1	6	0	0	0	0	0	0	6	6	6	6	6	6	6	6

Da Gewicht und Gewinn beim einfachen Rucksackproblem Synonyme sind, sind die Begriffe im Prinzip austauschbar. Im Hinblick auf das „allgemeine“ Rucksackproblem sind sie in obiger Lösung aber schon korrekt eingesetzt. Der Spaltenbezeichner *Gewinn* könnte auch durch *summierter Gewinn* ersetzt werden.
Mit *Kapazität* ist nicht die maximale Kapazität des Gefässes gemeint, sondern die Kapazität für diesen Berechnungsschritt. In der letzten Spalte fallen diese beiden Bedeutungen zusammen.

Aufgabe 5:

Objekt \ Kapzität		0	1	2	3	4	5	6	7	8	9	10	11	12	13
Index i	Gewicht g_i	summierter Gewinn													
1	6	0	0	0	0	0	0	6	6	6	6	6	6	6	6
2	3	0	0	0	3	3	3	6	6	6	9	9	9	9	9
3	5	0	0	0	3	3	5	6	6	8	9	9	11	11	11

Aufgabe 6:

Tabelle:

Objekt \ Kapzität		0	1	2	3	4	5	6	7	8	9	10	11	12	13
Index i	Gewicht g_i	summierter Gewinn													
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	6	0	0	0	0	0	0	6	6	6	6	6	6	6	6
2	3	0	0	0	3	3	3	6	6	6	9	9	9	9	9
3	5	0	0	0	3	3	5	6	6	8	9	9	11	11	11
4	11	0	0	0	3	3	5	6	6	8	9	9	11	11	11
5	7	0	0	0	3	3	5	6	7	8	9	10	11	12	13

Regel: Kurzfassung ($\approx$ erwartete Antwort der Schülerinnen und Schüler): Um beispielsweise die hellgrüne Zelle zu füllen, wählt man das Maximum aus

a) der unveränderten roten Zelle, oder

b) der gelben Zelle + Gewicht des aktuellen Objektes (dunkelgrün).

Ein ausführlicherer Beschrieb ist in Kapitel 3.2 zu finden.

Aufgabe 7: Den maximalen Gewinn findet man in der Tabelle unten rechts, das heisst in der Zelle (n, c).
Der Grund für die Zeile n ist, dass jede Zeile jeweils das Optimum der bisher verwendeten Objekte speichert, und die letzte Zeile folglich das Optimum aller Objekte.
Grund für die Spalte c ist, dass dort die Kapazität des Gefässes mit der Kapazität des

Berechnungsschrittes übereinstimmt.
Der maximale Gewinn kann aber auch schon in den Zeilen/Spalten davor auftreten.

Aufgabe 8: Die Grösse, nach der die Induktion stattfindet, ist die zur Auswahl stehenden Objekte. Im 0. Schritt steht kein Objekt zur Verfügung, im 1. Schritt verwenden wir ein Objekt, und in jedem weiteren Schritt kommt ein weiteres Objekt hinzu. In jedem Schritt wird der optimale Gewinn mit den zur Verfügung stehenden Objekten berechnet.

Die Verankerung erfolgt in Schritt 1 für $n = 0$. Wenn wir kein Objekt einpacken, dann ist das eingepackte Gewicht logischerweise 0. Dass gleich die ganze Zeile mit Nullen auffüllt wird, hat praktische Gründe beim Algorithmus.

Der Induktionsschritt geschieht in Schritt 2. Die Voraussetzung ist, dass der optimale Gewinn mit den bisher verwendeten $n - 1$ Objekten gefunden wurde. Man berechnet in diesem Schritt den optimalen Gewinn, der unter Verwendung der bisherigen $n - 1$ Objekte vereinigt mit dem n-ten Objekt erreicht werden kann.

Aufgabe 9:

Objekt \ Kapzität		0	1	2	3	4	5	6	7	8	9	10
Index i	Gewicht g_i	summierter Gewinn										
0	0	0	0	0	0	0	0	0	0	0	0	0
1	4	0	0	0	0	4	4	4	4	4	4	4
2	3	0	0	0	3	4	4	4	7	7	7	7
3	2	0	0	2	3	4	5	6	7	7	9	9
4	7	0	0	2	3	4	5	6	7	7	9	10

Aufgabe 10:

```
1  from pprint import pprint   # Ermöglicht mit pprint(tabelle)
2                              # eine schön formatierte Ausgabe
3  # Initalisierung der Variablen
4  g = [None, 6, 3, 5, 11, 7]  # Gewicht=Gewinn der Objekte
5  n = len(g) - 1  # Anzahl der Objekte
6  cap = 13        # Kapazität des Gefässes
7  tabelle = [[0 for j in range(cap + 1)] for i in range(n + 1)]
8      # Kreiert eine Tabelle mit n+1 Zeilen und cap+1 Spalten,
9      # gefüllt mit 0
10
11 # Tabelle füllen
12 for i in range(1, n + 1): # Zeilen durchlaufen
13     for j in range(1, cap + 1): # Spalten durchlaufen
14         if j < g[i]: # Testen, ob genügend weit rechts
15             tabelle[i][j] = tabelle[i-1][j]
16         else:
17             tabelle[i][j] = max(tabelle[i-1][j],
18                                 tabelle[i-1][j - g[i]] + g[i])
19
20 # Backtracking
21 items = []
```

```
22 weight = 0
23 i = n
24 j = cap
25 while i > 0 and j > 0:
26     if tabelle[i-1][j] < tabelle[i][j]:
27         items.append(i)
28         j -= g[i]
29         weight += g[i]
30     i -= 1
31
32 # Ausgabe
33 pprint(tabelle)
34 print('Wert={}, summiertes Gewicht={}, Kapazität={}'.format(
35     tabelle[n][cap], weight, cap))
36 print('Items:', items)
37 print(cap*n,'Operationen.')
```

Im Teil Backtracking (Zeilen 20-30 sowie 34-36) werden die benötigten Objekte aus der Tabelle entnommen. Dies ist nicht Teil der Aufgabe und wird somit auch nicht erwartet. Allenfalls könnte dies eine Challenge für Schülerinnen und Schüler mit fortgeschrittenen Kenntnissen sein.

Aufgabe 11: Idee: Die mehrfach auftretenden Elemente werden entsprechend ihrer Multiplizität in der Liste aufgeführt.

Aufgabe 12: Wenn man den Algorithmus mit entsprechenden Gewichten laufen lässt, sieht man, dass der optimale Gewinn 12 schon in der Spalte 12 erreicht wird. Der Gewinn 13 wird nicht erreicht, auch nicht in Spalte 13. Da Gewicht und Gewinn gleich sind (und der Algorithmus den optimalen Gewinn liefert), bedeutet dies, dass die Kapazität (13 kg) nicht ausgeschöpft werden kann.

Aufgabe 13:

a) Der Algorithmus funktioniert unverändert. Es wird ja mit Hilfe der Verzweigung (Zeile 14 im Code von Aufgabe 10) geprüft, dass zu grosse Gewichte nicht in den Zellenwerten erscheinen.

b) Es sind nur 4 Zeilen notwendig (Indizes 0, 1, 3, 4). Die Zeilen 2 und 5 (mit den Gewichten 14 kg resp. 20 kg) können vor dem Start der Tabellenberechnung aus der Liste entfernt werden.

c) Bevor die Tabelle berechnet wird, sollten aus `g` alle Elemente, deren Gewicht grösser ist als `cap`, entfernt werden. Dies kann z. B. mit `g = [x for x in g if x <= cap]` erzielt werden.

Aufgabe 14:

Objekt \ Kapzität			0	1	2	3	4	5	6	7	8
Index i	Gewicht g_i	Gewinn p_i	summierter Gewinn								
0	0	0	0	0	0	0	0	0	0	0	0
1	1	2	0	2	2	2	2	2	2	2	2
2	3	3	0	2	2	3	5	5	5	5	5
3	4	3	0	2	2	3	5	5	5	6	8
4	4	4	0	2	2	3	5	6	6	7	9
5	2	4	0	2	4	6	6	7	9	10	10

- Der Abschnitt „Achtung“ fällt weg: Gewinn und Gewicht sind nicht mehr gleich.
- Den bisherigen Bereich *Gewinn* könnte man präziser mit *summierter Gewinn* bezeichnen, um ihn vom Gewinn des Objektes abzugrenzen.
- Der Rest bleibt gleich. (Die Formulierung ist schon so gewählt, dass Gewinn und Gewicht korrekt getrennt sind.)

Aufgabe 15: Der optimale Gewinn wird schon bei einer Ladung mit dem Gesamtgewicht 7 erreicht, d. h., es wird nicht die volle Kapazität des Wagens ausgeschöpft.

Aufgabe 16: Es muss nur ein einziger Buchstabe geändert werden, nämlich auf Zeile 19: Addiert werden muss der Preis `p[i]` und nicht mehr das Gewicht `g[i]`.

```
1  from pprint import pprint  # Ermöglicht mit pprint(tabelle)
2                             # eine schön formatierte Ausgabe
3  # Initalisierung der Variablen
4  g = [None, 1, 3, 4, 4, 2]  # Gewicht der Objekte
5  p = [None, 2, 3, 3, 4, 4]  # Gewinn/Preis der Objekte
6  n = len(g) - 1  # Anzahl der Objekte
7  cap = 8         # Kapazität des Gefässes
8  tabelle = [[0 for j in range(cap + 1)] for i in range(n + 1)]
9      # Kreiert eine Tabelle mit n+1 Zeilen und cap+1 Spalten,
10     # gefüllt mit 0
11
12 # Tabelle füllen
13 for i in range(1, n + 1): # Zeilen durchlaufen
14     for j in range(1, cap + 1): # Spalten durchlaufen
15         if j < g[i]: # Testen, ob genügend weit rechts
16             tabelle[i][j] = tabelle[i-1][j]
17         else:
18             tabelle[i][j] = max(tabelle[i-1][j],
19                                 tabelle[i-1][j - g[i]] + p[i])
20
21 # Ausgabe
22 pprint(tabelle)
23 print('Optimaler Wert:', tabelle[n][cap])
```

Aufgabe 17: Die Tabelle umfasst $cap \cdot n$ Einträge. Insgesamt werden also $const \cdot cap \cdot n$ (d. h. $\mathcal{O}(cap \cdot n)$) Operationen durchgeführt. Die Laufzeit hängt also im Gegensatz zur Brute-Force-Lösung nicht nur von der Anzahl der Objekte n, sondern auch von der Kapazität cap ab. Somit hat der Algorithmus eine pseudopolynomielle Laufzeit in cap.

Aufgabe 18:

n	2	3	4	10	100	n (allgemein)
t	8	18	32	200	20 000	$2n \cdot n = 2n^2$

Bei einer kleinen Anzahl von Objekten ist die Tabelle nicht sonderlich effizient. In vorliegender Aufgabe zeigt sie spätestens ab $n = 10$ ihre Vorteile deutlich. $n = 100$ ist mit der Brute-Force-Methode nicht mehr realistisch berechenbar, mit der Tabelle geht es aber problemlos.

Generell ist die Tabelle effizient für vergleichsweise kleine cap und wird ineffizienter mit zunehmendem cap.

Aufgabe 19: Man kann den Algorithmus laufen lassen mit cap = 30, g = Preis, p = „Freude“ und findet den maximal möglichen Wert von 49 „Freude“. Verwendet werden hierzu die Objekte 19, 16, 14, 12, 11, 9, 7, 6, 4 und 2.

Aufgabe 20: Eine mögliche tabellarische Darstellung von $\binom{n}{k}$ respektive dem Pascal'schen Dreieck ist:

n \ k	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

Vorgehen beim Befüllen der Tabelle (Zeile für Zeile, von links nach rechts vorgehen):

a) Die gelb dargestellten Randwerte (jeweils Spalte $k = 0$ und „Diagonalelemente“, d. h., wenn $n = k$) werden jeweils mit dem Wert 1 befüllt.

b) Um die restlichen Werte an den Koordinaten (n, k) (d. h. eigentlich $\binom{n}{k}$) zu berechnen, werden die Werte an den Koordinaten $(n-1, k-1)$ und $(n-1, k)$ addiert. Beispielsweise wird der rote Wert aus der Summe der beiden grünen Werten gebildet.

```
1 from pprint import pprint    # Ermöglicht mit pprint(tabelle)
2                              # eine schön formatierte Ausgabe
3
4 groesse = 6          # Anzahl Zeilen des Pascal'schen Dreiecks
5 tabelle = [[0 for j in range(groesse)] for i in range(groesse)]
```

```
6       # generiert "leere" Tabelle (gefüllt mit 0)
7
8   for n in range(groesse):
9       for k in range(n+1):
10          if (k == 0) or (k == n):
11              tabelle[n][k] = 1
12          else:
13              tabelle[n][k] = tabelle[n-1][k] + tabelle[n-1][k-1]
14
15  pprint(tabelle)
```