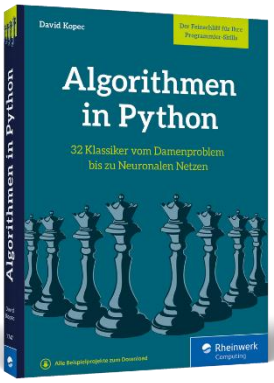
Modul Algorithmen

Rucksackproblem mit dynamischer Programmierung lösen



Dem nebenstehenden, empfehlenswerten Buch mit den treffenden Titel „Algorithmen in Python" ist nachfolgendes Beispiel ab Seite 245 entnommen.

9.1 Das Rucksackproblem

Das Rucksackproblem ist ein Optimierungsproblem, das aus einer gängigen Berechnungsanforderung – dem Finden der besten Einsatzmöglichkeit beschränkter Ressourcen mit einer endlichen Menge von Kombinationsmöglichkeiten – eine amüsante Geschichte spinnt. Ein Dieb betritt eine Wohnung mit der Absicht, zu stehlen. Er hat einen Rucksack, und die Kapazität des Rucksacks begrenzt, was er stehlen kann. Wie entscheidet er, was er in den Rucksack packen soll? Das Problem wird in Abbildung 9.1 veranschaulicht.

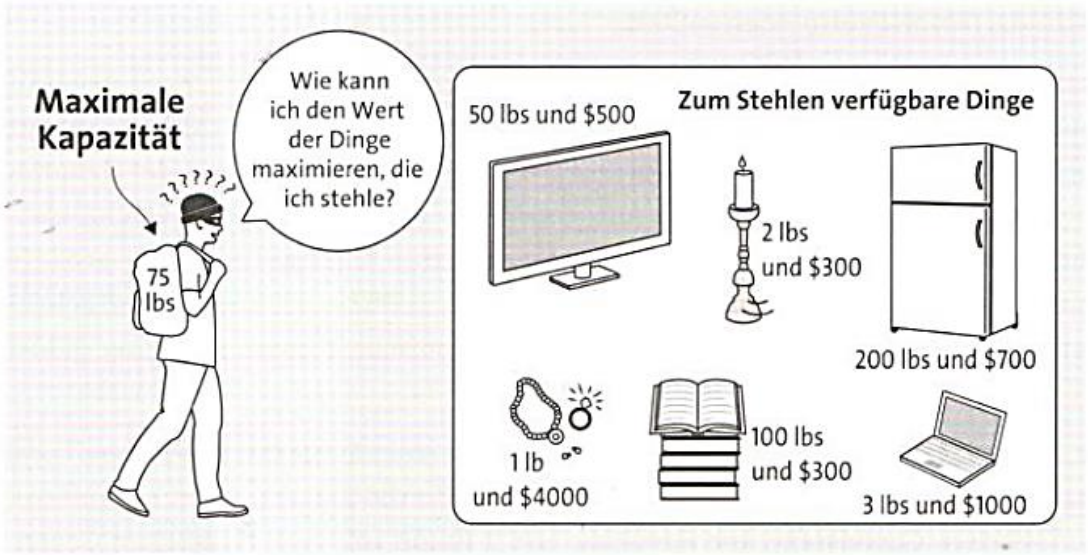


Abbildung 9.1 Der Einbrecher muss entscheiden, welche Gegenstände er stehlen soll, weil die Kapazität des Rucksacks begrenzt ist.

245

Wenn der Dieb jede beliebige Menge jedes Gegenstands mitnehmen könnte, dann könnte er einfach den Wert jedes Gegenstands durch dessen Gewicht teilen, um die wertvollsten Gegenstände für die verfügbare Kapazität zu finden. Aber um das Szenario realistischer zu machen, sagen wir, der Dieb kann nicht die Hälfte eines Gegenstands (wie etwa 2,5 Fernseher) mitnehmen. Stattdessen finden wir einen Lösungsweg für die 0/1-Variante der Aufgabe, die so genannt wird, weil sie eine weitere Regel erzwingt: Der Dieb kann entweder ein Exemplar jedes Gegenstands mitnehmen oder keins.

Definieren wir zuerst ein `NamedTuple`, um unsere Gegenstände zu speichern.

```
from typing import NamedTuple, List
class Item(NamedTuple):
    name: str
    weight: int
    value: float
```

Listing 9.1 knapsack.py

Wenn wir versuchen würden, dieses Problem mit einem Brute-Force-Ansatz zu lösen, würden wir uns jede Kombination von Gegenständen anschauen, die für das Stecken in den Rucksack verfügbar sind. Für mathematisch Interessierte: Es handelt sich um eine *Potenzmenge*, und die Potenzmenge einer Menge (in unserem Fall der Menge von Gegenständen) hat 2^*N* verschiedene mögliche Teilmengen, wobei *N* die Anzahl der Gegenstände ist. Deshalb müssten wir 2^*N* Kombinationen analysieren (O(2^N)). Das geht für eine kleine Anzahl von Elementen in Ordnung, ist aber für eine große Anzahl untragbar. Jeder Ansatz, der ein Problem mithilfe einer exponentiellen Anzahl von Schritten löst, ist ein Ansatz, den wir vermeiden sollten.

Stattdessen verwenden wir ein Verfahren, das *dynamische Programmierung* genannt wird und konzeptionell der Memoisation (Kapitel 1, »Kleine Aufgaben«) ähnelt. Anstatt ein Problem direkt mit einem Brute-Force-Ansatz zu lösen, löst man bei der dynamischen Programmierung Teilprobleme, aus denen das größere Problem besteht, speichert diese Ergebnisse und verwendet diese gespeicherten Ergebnisse, um das größere Problem zu lösen. Solange die Kapazität des Rucksacks in diskreten (unstetigen) Schritten betrachtet wird, kann das Problem durch dynamische Programmierung gelöst werden.

Um das Problem beispielsweise für einen Rucksack mit einer Kapazität von 3 Pfund[1] und drei Gegenständen zu lösen, können wir es zunächst für die Kapazität von 1 Pfund mit einem möglichen Gegenstand, die Kapazität von 2 Pfund mit zwei möglichen Gegen-

1 [englisches Pfund, entspricht ca. 454 g]

ständen und die Kapazität von 3 Pfund mit zwei Möglichkeiten lösen. Schließlich können wir es für alle drei möglichen Gegenstände lösen.

Auf dem Weg befüllen wir eine Tabelle, die uns die bestmögliche Lösung für jede Kombination aus Gegenständen und Kapazität verrät. Unsere Funktion befüllt zuerst die Tabelle und findet dann anhand der Tabelle die Lösung heraus.[2]

```
def knapsack(items: List[Item], max_capacity: int) -> List[Item]:
    # Eine Tabelle für dynamische Programmierung aufbauen
    table: List[List[float]] = [[0.0 for _ in range(max_capacity + 1)] for _ in
     range(len(items) + 1)]
    for i, item in enumerate(items):
        for capacity in range(1, max_capacity + 1):
            previous_items_value: float = table[i][capacity]
            if capacity >= item.weight: # Gegenstand passt in Rucksack
                value_freeing_weight_for_item: float = table[i][capacity -
                 item.weight]
                # Nur nehmen, wenn wertvoller als voriger Gegenstand
                table[i + 1][capacity] = max(value_freeing_weight_for_
                 item + item.value, previous_items_value)
            else: # Kein Platz für diesen Gegenstand
                table[i + 1][capacity] = previous_items_value
    # Lösung aus der Tabelle heraussuchen
    solution: List[Item] = []
    capacity = max_capacity
    for i in range(len(items), 0, -1): # Rückwärts arbeiten
        # Wurde dieser Gegenstand verwendet?
        if table[i - 1][capacity] != table[i][capacity]:
            solution.append(items[i - 1])
            # Wenn dieser Gegenstand verwendet wurde, sein Gewicht abziehen
            capacity -= items[i - 1].weight
    return solution
```

Listing 9.2 knapsack.py (Fortsetzung)

2 Ich habe diverse Ressourcen studiert, um diese Lösung zu schreiben. Die maßgeblichste von ihnen war *Algorithms* (Addison-Wesley, 1988), 2. Auflage, von Robert Sedgewick (Seite 596). Ich habe mir mehrere Beispiele des 0/1-Rucksackproblems auf Rosetta Code angeschaut, besonders die Python-Lösung mit dynamischer Programmierung (*http://mng.bz/kx8C*), von der diese Lösung größtenteils ein Backport aus der Swift-Version dieses Buches ist. (Ich bin von Python zu Swift und wieder zurück zu Python gegangen.)

Die innere Schleife des ersten Teils dieser Funktion wird *N* * *C* Mal ausgeführt, wobei *N* die Anzahl der Gegenstände und *C* die maximale Kapazität des Rucksacks ist. Deshalb wird der Algorithmus in der Zeit O(*N* * C) ausgeführt – eine signifikante Verbesserung gegenüber dem Brute-Force-Ansatz für eine große Anzahl von Gegenständen. Für die folgenden 11 Gegenstände müsste ein Brute-Force-Algorithmus beispielsweise 2^11 oder 2.048 Kombinationen untersuchen. Die gerade gezeigte Funktion mit dynamischer Programmierung wird 825-mal ausgeführt, weil die maximale Kapazität des Rucksacks 75 (beliebige) Maßeinheiten groß ist (11 * 75). Dieser Unterschied würde für mehr Gegenstände exponentiell wachsen.

Schauen wir uns die Lösung in Aktion an.

```
if __name__ == "__main__":
    items: List[Item] = [Item("Fernseher", 50, 500),
                         Item("Kerzenhalter", 2, 300),
                         Item("Stereoanlage", 35, 400),
                         Item("Laptop", 3, 1000),
                         Item("Essen", 15, 50),
                         Item("Kleidung", 20, 800),
                         Item("Schmuck", 1, 4000),
                         Item("Bücher", 100, 300),
                         Item("Drucker", 18, 30),
                         Item("Kühlschrank", 200, 700),
                         Item("Gemälde", 10, 1000)]
    print(knapsack(items, 75))
```

Listing 9.3 knapsack.py (Fortsetzung)

Wenn Sie sich die auf der Konsole ausgegebenen Ergebnisse anschauen, sehen Sie, dass die optimalen Gegenstände zum Stehlen das Gemälde, der Schmuck, die Kleidung, der Laptop, die Stereoanlage und die Kerzenhalter sind. Hier eine Beispielausgabe, die die wertvollsten Gegenstände zeigt, die der Dieb angesichts der beschränkten Kapazität des Rucksacks stehlen kann:

```
[Item(name='Gemälde', weight=10, value=1000), Item(name='Schmuck', weight=1,
 value=4000), Item(name='Kleidung', weight=20, value=800), Item(name='Laptop',
 weight=3, value=1000), Item(name='Stereoanlage', weight=35, value=400),
 Item(name='Kerzenhalter', weight=2, value=300)]
```

Um ein besseres Verständnis dafür zu bekommen, wie das alles funktioniert, lassen Sie uns einige Besonderheiten der Funktion betrachten:

```
for i, item in enumerate(items):
    for capacity in range(1, max_capacity + 1):
```

Für jede mögliche Anzahl von Gegenständen gehen wir alle Kapazitäten bis zur maximalen Kapazität des Rucksacks auf lineare Weise in einer Schleife durch. Beachten Sie, dass ich »jede mögliche Anzahl von Gegenständen« und nicht »jeden Gegenstand« gesagt habe. Wenn `i` gleich 2 ist, steht es nicht für den zweiten Gegenstand. Es steht für die mögliche Kombination der ersten beiden Gegenstände für jede untersuchte Kapazität. `item` ist der nächste Gegenstand, den wir zu stehlen erwägen:

```
previous_items_value: float = table[i][capacity]
if capacity >= item.weight: # Gegenstand passt in Rucksack
```

`previous_items_value` ist der Wert der letzten Kombination von Gegenständen, die für die aktuelle `capacity` untersucht wird. Für jede mögliche Kombination von Gegenständen überprüfen wir, ob das Hinzufügen des letzten »neuen« Gegenstandes überhaupt möglich ist.

Wenn der Gegenstand mehr wiegt als die Rucksackkapazität, die wir prüfen, kopieren wir einfach den Wert der letzten Kombination von Gegenständen, die wir für die fragliche Kapazität geprüft haben.

```
else: # Kein Platz für diesen Gegenstand
    table[i + 1][capacity] = previous_items_value
```

Andernfalls prüfen wir, ob das Hinzufügen des »neuen« Gegenstandes einen höheren Wert erzielen wird als die letzte Kombination von Gegenständen, die wir für diese Kapazität geprüft haben. Das tun wir, indem wir den Wert des Gegenstandes zu dem Wert addieren, der in der Tabelle bereits für frühere Kombinationen von Gegenständen berechnet wurde, deren Kapazität dem Gewicht des Gegenstandes entspricht, abgezogen von der aktuellen Kapazität, die wir prüfen. Wenn dieser Wert höher ist als die letzte Kombination von Gegenständen für die aktuelle Kapazität, fügen wir ihn ein; andernfalls fügen wir den letzten Wert ein:

```
value_freeing_weight_for_item: float = table[i][capacity - item.weight]
# Nur nehmen, wenn wertvoller als voriger Gegenstand
table[i + 1][capacity] = max(value_freeing_weight_for_item + item.value,
 previous_items_value)
```

Das schließt den Aufbau der Tabelle ab. Um jedoch tatsächlich herauszufinden, welche Gegenstände zur Lösung gehören, müssen wir von der höchsten Kapazität und der zuletzt erforschten Kombination von Gegenständen aus rückwärts arbeiten.

```
for i in range(len(items), 0, -1): # Rückwärts arbeiten
    # Wurde dieser Gegenstand verwendet?
    if table[i - 1][capacity] != table[i][capacity]:
```

Wir beginnen am Ende, durchlaufen unsere Tabelle von rechts nach links in einer Schleife und überprüfen bei jedem Schritt, ob es eine Änderung an dem in die Tabelle eingefügten Wert gab. Wenn das der Fall war, bedeutet dies, dass wir den neuen Gegenstand, der in einer bestimmten Kombination berücksichtigt wurde, hinzugefügt haben, weil die Kombination wertvoller war als die vorherige. Deshalb fügen wir diesen Gegenstand zur Lösung hinzu. Außerdem wird die Kapazität um das Gewicht des Gegenstandes vermindert, was man sich als Hochklettern in der Tabelle vorstellen kann:

```
solution.append(items[i - 1])
# Wenn dieser Gegenstand verwendet wurde, sein Gewicht abziehen
capacity -= items[i - 1].weight
```

Hinweis

Sowohl während des Aufbaus der Tabelle als auch während der Lösungssuche haben Sie vielleicht einige Manipulationen von Iteratoren und Tabellengröße um 1 gesehen. Das geschieht aus Komfortgründen von einer Programmierperspektive aus betrachtet. Denken Sie daran, wie das Problem von unten auf konstruiert wird. Wenn das Problem beginnt, haben wir es mit einem Rucksack mit der Kapazität null zu tun. Wenn Sie in einer Tabelle von unten nach oben arbeiten, wird klar, wofür Sie die zusätzliche Zeile und Spalte benötigen.

Sind Sie immer noch verwirrt? Tabelle 9.1 ist die Tabelle, die die Funktion `knapsack()` aufbaut. Es wäre eine ziemlich große Tabelle für die Aufgabe von eben, also schauen wir uns stattdessen eine Tabelle für einen Rucksack mit 3 Pfund Kapazität und drei Gegenständen an: Streichhölzern (1 Pfund), Taschenlampe (2 Pfund) und Buck (1 Pfund). Angenommen, diese Gegenstände sind 5 $, 10 $ beziehungsweise 15 $ wert.

	0 lb.	1 lb.	2 lb.	3 lb.
Streichhölzer (1 lb., 5 $)	0	5	5	5
Taschenlampe (2 lbs., 10 $)	0	5	10	15
Buch (1 lb., 15 $)	0	15	20	25

Tabelle 9.1 Ein Beispiel für ein Rucksackproblem mit drei Gegenständen

Wenn Sie die Tabelle von links nach rechts betrachten, steigt der Wert (wie viel Sie versuchen, in den Rucksack zu packen). Wenn Sie die Tabelle von oben nach unten betrachten, steigt die Anzahl der Gegenstände, die Sie einzupacken versuchen. In der ersten Zeile versuchen Sie nur die Streichhölzer einzupacken. In der zweiten Zeile versuchen Sie, die wertvollste Kombination aus Streichhölzern und Taschenlampe einzupacken, die in den Rucksack passt. In der dritten Reihe fügen Sie die wertvollste Kombination aller drei Gegenstände hinzu.

Als Übung, um Ihr Verständnis zu verbessern, versuchen Sie selbst, eine leere Version dieser Tabelle auszufüllen, indem Sie den in der Funktion `knapsack()` beschriebenen Algorithmus auf diese drei Gegenstände anwenden. Verwenden Sie dann den Algorithmus am Ende der Funktion, um wieder die richtigen Gegenstände aus der Tabelle auszulesen. Diese Tabelle entspricht der Variablen `table` in der Funktion.