****

Informatik Grundkurs Q1

Die Bedeutung von NP anhand von Beispielen erklärt

Vorgelegt von:
****

*****, 14. März 2024

Inhaltsverzeichnis

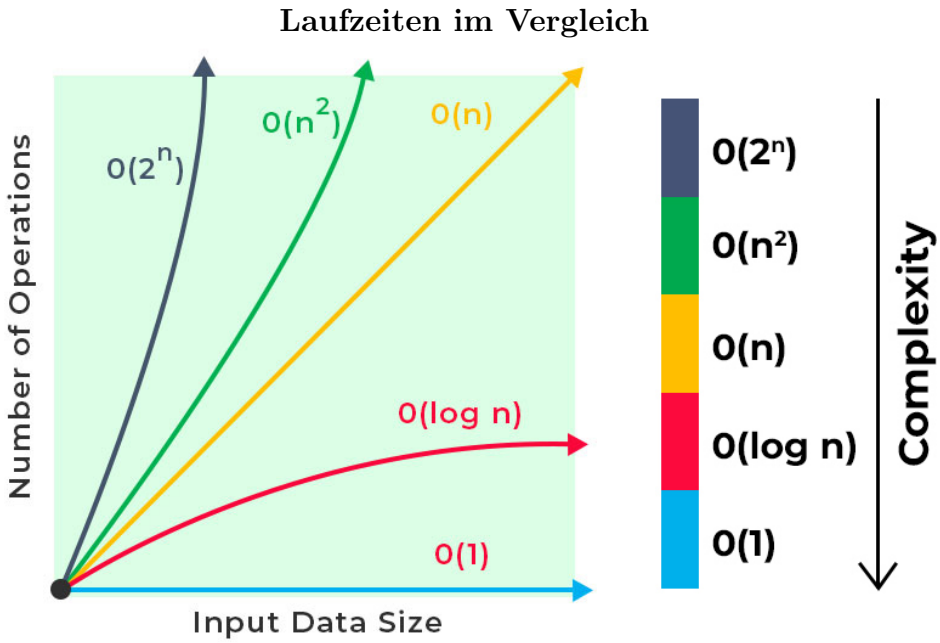
1 Einleitung

In der Welt der theoretischen Informatik ist das Konzept von P und NP eines der grundlegendsten Probleme. Diese beiden Klassen von Problemen sind eng verbunden mit der Frage, wie effizient Computer Probleme lösen können. Die Effizienz solcher Probleme hängt immer von der Lauf- und Speicherkomplexität ab, aber vorallem vom Verständnis des Menschen auf diese Probleme. Denn je weiter sich die theoretische Informatik entwickelt, desto effizienter können Probleme gelöst werden. Sie beschäftigt sich mit der Untersuchung von Algorithmen und Datenstrukturen und ihrer Effizienz. In dieser Facharbeit werden die Komplexitätsklassen P, NP, NP-Schwer und NP-Vollständig genauer betrachtet und es werden verschiedene Beispiele erläutert, die zu verschiedenen Komplexitätsklassen gehören.

2 Was heißt P und NP?

2.1 P

P ist eine von mehreren Komplexitätsklassen P heißt 'Polynomiell' oder auch 'Polynomialzeit'.[1] Alle Probleme der Klasse P können deterministisch[2] in polynomieller Laufzeit gelöst werden (Variante 1: Optimierungsproblem). Zudem können sie auch in polynomieller Zeit auf Richtigkeit geprüft werden (Variante 2: Entscheidungsproblem). Diese Probleme werden als 'einfach' bezeichnet, weil sie in einer angemessenen Zeit gelöst werden können. Polynomialzeit hat eine maximale Laufzeit von $\mathcal{O}(n^k)$, wobei n die Eingabe ist und k eine unabhängig von der Eingabe Konstante.



Quelle: What is Logarithmic Time Complexity? (GeeksforGeeks), 2024

Beispiele für Algorithmen, die in dieser Klasse liegen sind beispielsweise Quicksort, Mergesort oder der Dijkstra-Algorithmus.

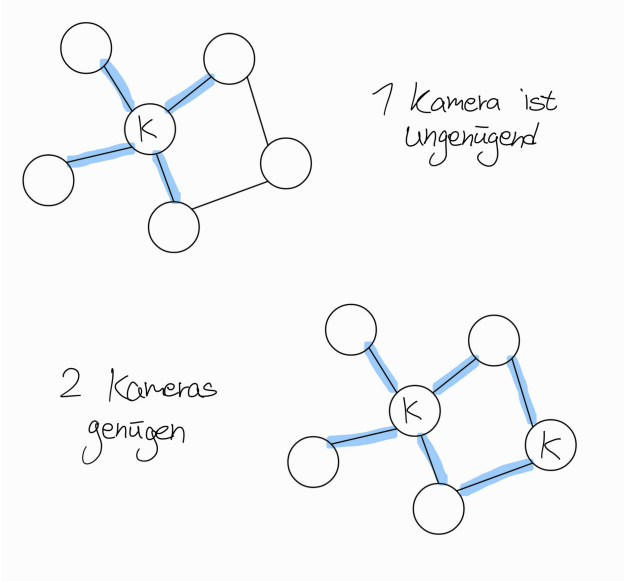
[1] Vgl.: P Komplexitätsklasse (studysmarter.de), o.J.
[2] Die Bedeutung von 'deterministisch' wird später erklärt

2.2 NP

NP heißt 'Nichtdeterministisch Polynomiell'.[1] Nichtdeterministisch bedeutet, dass es keinen bekannten deterministischen Algorithmus gibt, der das Problem in polynomieller Zeit löst. Es gibt nur eine 'magische Methode', also eine Hypothese wie man das Problem in polynomieller Zeit lösen **könnte**. Dennoch ist es möglich die Probleme in dieser Komplexitätsklasse in polynomieller Zeit auf Richtigkeit zu überprüfen.

2.2.1 Knotenüberdeckungsproblem

Das Knotenüberdeckungsproblem (engl.: Vertex Cover) wird im Folgenden als Entscheidungsproblem betrachtet. Es sei ein Graph $G = (V, E)$ mit n Knoten V und m Kanten E gegeben.[2] Das Ziel ist es, einen Algorithmus zu finden, der entscheidet ob k Kameras reichen um alle Kanten im Graph zu überwachen.



Quelle: Eigenes Bild, abgeleitet von: Was ist NP-schwer? (Algorithmen und Datenstrukturen), 2021

Ein einfacher Ansatz zur Lösung ist **Brute Force**. Das heißt, man testet alle möglichen Kombinationen aus und überprüft ob alle Kanten überwacht

[1] Vgl.:Was ist NP-schwer? (Algorithmen und Datenstrukturen), 2021

[2] Beispiel entnommen von: Was ist NP-schwer? (Algorithmen und Datenstrukturen), 2021

sind. Falls es genauso viele oder mehr Kameras als Knoten gibt, dann ist auch der komplette Graph überwacht.

Algorithm 1 Knotenüberdeckungsproblem (G, k)

```
1: if k ≥ n then
2:     return true
3: end if
4: for Kombination X von k Kameras auf V ∈ G verteilt do
5:     if G ist komplett überwacht then
6:         return true
7:     end if
8: end for
9: return false
```

Mit k Kameras und n Knoten gibt es $\binom{n}{k}$ Kombinationen. Abgeschätzt wird dies auf n^k. Dabei wird für jede Kombination geprüft, ob jede Kante überwacht wird. Also wird das Ganze m-mal wiederholt. Somit ergibt sich eine Laufzeit von $\mathcal{O}(n^k \cdot m)$ und ist somit exponentiell. Ob man die Laufzeit auf Polynomialzeit minimieren kann, weiß man nicht (Stand: 8. Mai 2024). Deswegen überlegt man sich eine 'magische Methode' aus, die bestimmt wo die Kameras gesetzt werden müssen um den kompletten Graphen zu überwachen. Jetzt müsste nur noch auf Richtigkeit geprüft werden und man erhält eine Laufzeit von maximal $\mathcal{O}(n^k)$.

Algorithm 2 Knotenüberdeckungsproblem (G, k)

```
1: if k ≥ n then
2:     return true
3: end if
4: // 'Magische Methode'
5: X ← Kombination von k Kameras auf V ∈ G verteilt
6: if X ist komplett überwacht then
7:     return true
8: end if
9: return false
```

Ein Algorithmus hat einen eindeutigen Ablauf mit endlichen Schritten. Die 'magische Methode' macht das Problem Nichtdeterministisch, weil eben

keine 'magische Methode' determiniert werden kann - sie ist unbekannt, also gibt es keinen eindeutigen Ablauf.

2.3 P = NP oder P ≠ NP?

Die große Frage, die sich stellt ist ob Probleme, die in NP liegen auch in P liegen. Sozusagen ist die Frage ob es die 'magische Methode' für Probleme in NP existiert. Bis heute (Stand: 8. Mai 2024) konnte nicht bewiesen werden ob P = NP oder P ≠ NP ist.

2.4 Warum ist das wichtig?

Eine Welt, wo P = NP bewiesen wäre ist komplett anders als unsere jetzige. Krebs würde geheilt werden (Proteinfaltung) und Verschlüsselungen mit RSA-Verfahren wären nicht mehr sicher (Kryptologie).[1] Viele Algorithmen wären viel effizienter, also hätten eine polynomielle Laufzeit. Es würde die Welt verändern. Das wichtigste ist jedoch logisch zu verstehen, was genau diese Klassen ausmacht und warum man Probleme voneinander differenziert.

2.5 NP-Vollständig

Sie sind 'schwerer' als alle anderen NP-Probleme. Die Komplexitätsklasse enthält Entscheidungsprobleme, die das Problem in polynomieller Zeit auf Richtigkeit überprüfen kann und es gibt keine Lösung dieser Probleme in polynomieller Zeit. Diese Probleme liegen in NP und NP-Schwer. Wenn nur eines der NP-vollständigen Probleme in polynomieller Zeit gelöst werden kann, dann bedeutet das, dass **alle** Probleme in NP gelöst werden können. Ein Problem X gehört zu dieser Klasse wenn jedes Problem Y in NP reduzierbar auf das Problem X in polynomieller Zeit ist.[2]

2.6 NP-Schwer

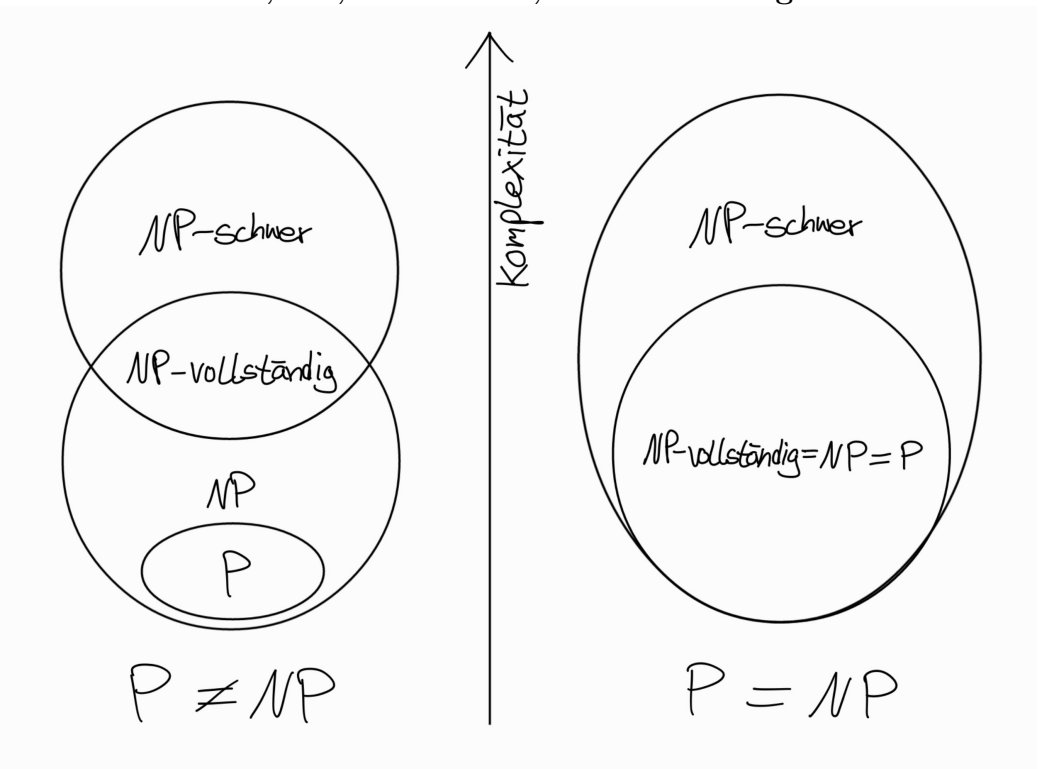
Für NP-schwere Probleme ist kein Algorithmus bekannt, der das Problem in polynomieller Zeit lösen kann. Sie sind 'schwerer' als alle anderen NP-

[1] Beispiele entnommen von: What would be the Impact of P=NP? (Mason Wheeler, vinaykola), 2012

[2] Vgl.: Difference between NP hard and NP complete problem (sugandha18bcs3001), 2023 und Klasse NP-Vollständigkeit (studysmarter.de), o.J.

vollständigen Probleme. Wenn nur eines der Probleme in der Klasse NP-Schwer in polynomieller Zeit gelöst werden kann, dann können **alle** Probleme in NP in polynomieller Zeit gelöst werden. Diese Probleme gehören zu den 'schwierigsten' Probleme, weil es nur wenige Ansätze gibt sie zu lösen. Dazu gehören auch Probleme, die überhaupt nicht lösbar sind. Es ist ebenfalls nicht garantiert, dass sie in polynomieller Zeit gelöst werden können. Ein Problem X gehört zu dieser Klasse wenn ein Problem Y in NP-Vollständig reduzierbar auf das Problem X in polynomieller Zeit ist.[1]

P, NP, NP-Schwer, NP-Vollständig



Quelle: Eigenes Bild, abgeleitet von: If You Solve This Math Problem, You Could Steal All the Bitcoin in the World (Ryan F. Mandelbaum), 2019

Anmerkung: Es gibt viele weitere kleinere Komplexitätsklassen, die jedoch hier nicht besprochen werden.

[1]Vgl.: Difference between NP hard and NP complete problem (sugandha18bcs3001), 2023 und P, NP, CoNP, NP hard and NP complete | Complexity Classes (umng01), 2023

2.7 SAT-Problem

Das Erfüllbarkeitsproblem (engl.: Satisfiability problem) ist ein Entscheidungsproblem für aussagenlogische Formeln.[1] Es ist das erste Problem, das auf NP-Vollständigkeit bewiesen wurde.[2] Gegeben ist eine boolsche Formel, die auf Erfüllbarkeit geprüft werden müssen. Die Formel ist erfüllbar wenn bei einer Belegung der Variablen die Formel wahr ist.

$$f(x_1, x_2) = (x_1 \vee x_2) \wedge (\neg x_1 \wedge x_2)$$

$f(x_1, x_2) = true$ wenn $x_1 = false$ und $x_2 = true$

$$\begin{aligned} f(false, true) &= (false \vee true) \wedge (\neg false \wedge true) \\ &= (false \vee true) \wedge (true \wedge true) \\ &= true \wedge true \\ &= true \end{aligned}$$

Mit dem Cook-Levin-Theorem kann das Problem auf NP-Vollständigkeit bewiesen werden.[3] Zudem lässt sich damit beweisen, dass alle NP-Probleme auf das SAT-Problem reduzieren lassen.

2.8 Reduktionen

Mit Reduktionen kann bewiesen werden ob ein Problem NP-vollständig oder NP-schwer ist. Es geht darum ein Problem X auf ein Problem Y zu übertragen damit eine Lösung für Y eine Lösung für X bietet.[4] Eine formale Reduktion zwei bekannter/komplexer Probleme würde den Rahmen dieser Facharbeit sprengen, weswegen im Folgenden ein kleines Beispiel erklärt wird, das das Konzept näher bringt.[5]

Die Methode 'maximalesElement' soll aus einer Liste von Elementen des Typ *int* den größten Wert zurückgeben. Beispielsweise möchte man das Problem 'maximalesElement' auf das Problem 'sortiereElemente' reduzieren. Es

[1] Vgl.: SAT Problem (studysmarter.de), o.J.
[2] Bewiesen von Steve Cook in 1971
[3] Vgl.: Cook–Levin theorem or Cook's theorem (soubhikmitra98), 2021
[4] Vgl.: Reduktionen: Theoretische Informatik (Niklas Steenfatt), 2020
[5] Beispiel von: Reduktionen: Theoretische Informatik (Niklas Steenfatt), 2020

wird davon ausgegangen, dass die Methode 'sortiereElemente' bereits durch eine Bibliothek zur Verfügung steht.

Diese Methode hilft nun um die Methode 'maximalesElement' zu konstruieren. Um das maximale Element aus einer (aufsteigend) sortierten Liste zu ermitteln, nimmt man das letzte Element.

Algorithm 3 maximalesElement(liste)

```
1: import bib42
2: sortierteListe = bib42.quicksort(liste)
3: letztesElement = sortierteListe.getElement(liste.size()-1)
4: return letztesElement
```

Somit wurde 'bewiesen', dass 'maximalesElement' **höchstens** so schwierig wie 'sortiereListe' ist.

$$maximalesElement \leq sortiereElemente$$

Da 'maximalesElement' **höchstens** so schwierig ist, bedeutet es, dass es eine bessere Methode für 'maximalesElement' geben **kann**. In dem Fall gibt es sie wirklich. Quicksort hat eine Laufzeit von $\mathcal{O}(nlogn)$. Da diese Methode für 'maximalesElement' genutzt wird und sonst keine weitere aussagekräftige Komplexität entsteht, beträgt sie auch $\mathcal{O}(nlogn)$. Doch es gibt einen einfacheren Algorithmus mit einer Laufzeitkomplexität von $\mathcal{O}(n)$:

Algorithm 4 maximalesElement(liste)

```
1: maximalesElement = -∞
2: for element in liste do
3:     if element > maximalesElement then
4:         maximalesElement = element
5:     end if
6: end for
7: return maximalesElement
```

3 Das Rucksackproblem

Beim Rucksackproblem (engl.: Knapsack problem)[1] sind n Gegenstände und die Mengen $w = \{w_1, w_2, ...w_n\}$ und $g = \{g_1, g_2, ...g_n\}$ gegeben. w_i beschreibt den Wert und g_i das Gewicht des Gegenstandes i. Die Variable t beschreibt die Tragkraft[2] eines Rucksacks. Das Ziel ist es, so viele Gegenstände wie möglich in den Rucksack zu packen, wobei die Tragkraft t des Rucksacks nicht übersteigt und der Profit[3] maximal ist.

$$max \sum_{i=0}^{n} w_i$$
$$\sum_{i=0}^{n} g_i \leq t$$

Man stellt sich einen Einbrecher vor, der möglichst viel mitnehmen will. Da er nur einen Rucksack hat, muss er **entscheiden**, welche Gegenstände er mitnimmt um möglichst viel Profit zu erlangen und die Gegenstände im Darknet zu verkaufen. Natürlich muss er sich auch möglichst schnell entscheiden, da die Polizei hinter ihm her ist.

3.1 Brute Force

Es ist möglich jede einzelne Kombination von möglichen Rucksäcken zu berechnen, und dann den Rucksack mit höchstem Profit als Ergebnis zurückzugeben. Das ist jedoch sehr ineffizient, da eine Laufzeitkomplexität von $\mathcal{O}(2^n)$ entsteht.

Somit ist es für den Einbrecher unoptimal. Bis er alle Kombinationen von Gegenständen ausprobiert hat, wurde er bereits von der Polizei festgenommen.

[1] Genauer gesagt 0/1 Knapsack, da das Problem eine weitere Variante hat, bei der auch Fraktionen (Teile) von Gegenständen genommen werden können

[2] Maximales Gewicht

[3] Summe aller Werte der Gegenstände im Rucksack

3.2 Dynamische Programmierung

Mit dynamischer Programmierung (DP) kann ein optimales Ergebnis gefunden werden.[1] Man betrachte zunächst diese Tabellen:

w	g
7	2
6	3
8	5
5	4
6	6

0	1	2	3	4	5	6	7	8
0								
0								
0								
0								
0								

In der linken Tabelle sind die Gegenstände und ihre entsprechenden Gewichte und Werte. Typisch für DP ist es ein Problem in mehrere Unterprobleme aufzuteilen, die Lösungen für die jeweiligen Unterprobleme zu speichern, und dann aus diesen Lösungen das Hauptproblem zu lösen. Deswegen sind in der rechten Tabelle mehrere mögliche Maxima für t (0 bis 8). Also wird das Problem für $t = 1, t = 2, ...t = 8$ gelöst. Jedes mal wenn entschieden wird ob ein Gegenstand in den Rucksack gelegt werden soll oder nicht, werden nur die vorherigen Gegenstände mit einbezogen. Wenn $t = 0$ ist, dann kann kein Gegenstand reinpassen. Deswegen kann auch kein Profit erlangen werden.
Man betrachte die erste Zeile (erster Gegenstand). Das Gewicht von Gegenstand 1 ist 2. Das heißt in einen Rucksack mit $t = 1$, passt dieser Gegenstand mit $g_1 = 2$ genau 0 mal rein. Somit ist der Profit 0. In einen Rucksack mit $t = 2$, passt dieser Gegenstand mit $g_1 = 2$ genau 1 mal rein. Somit ist der Profit 7. Da es jeden Gegenstand immer nur einmal gibt, können somit alle Rucksäcke mit höherer Tragkraft auch maximal nur 1mal diesen Gegenstand tragen.

w	g
7	2
6	3
8	5
5	4
6	6

0	1	2	3	4	5	6	7	8
0	0	7	7	7	7	7	7	7
0								
0								
0								
0								

[1] Gerichtet nach: 4.5 0/1 Knapsack - Two Methods - Dynamic Programming (Abdul Bari), 2018 und 0/1 Knapsack Problem Dynamic Programming (Tushar Roy), 2015

Für die nächste Zeile/den nächsten Gegenstand, werden alle vorherigen Zeilen/Gegenstände mit einbezogen. Gegenstand 2 hat ein Gewicht von $g_2 = 3$. Das heißt alle Rucksäcke mit $t < 3$ behalten den gleichen maximalen Profit von vorher. Jetzt wird geprüft wie viel Profit man mehr erlangen kann. Entweder es wird der Profit von vorher genommen (Gegenstand 1) oder man nimmt den Wert von Gegenstand 2 ($w_2 = 6$).

$$max(7, 6) = 7$$

In einen Rucksack mit $t = 4$ kann entweder Gegenstand 1 gelegt werden oder es wird Gegenstand 2 eingelegt. Wichtig: Wenn der Gegenstand 2 mit $g_2 = 3$ in einen Rucksack mit $t = 4$ eingelegt wird, dann bleibt noch ein Gewicht von genau 1 übrig (4-3=1). Deswegen wird geprüft was der maximale Profit von vorher im Rucksack mit $t = 1$ war[1]. Es ist 0.

$$max(7, 6 + 0) = 7$$

In einen Rucksack mit $t = 5$ kann entweder Gegenstand 1 gelegt werden oder es wird Gegenstand 2 eingelegt. Wenn der Gegenstand 2 mit $g_2 = 3$ in einen Rucksack mit $t = 5$ eingelegt wird, dann bleibt noch ein Gewicht von genau 2 übrig (5-3=2). Deswegen wird geprüft, was der maximale Profit von vorher im Rucksack mit $t = 2$ war. Es ist 7.

$$max(7, 6 + 7) = 13$$

Ein höherer Profit als 13 kann für diese zwei Gegenstände nicht erlangt werden, weil beide Gegenstände zusammen ein Gewicht von 5 bilden. In einen Rucksack mit mehr Tragkraft bleibt der Profit somit gleich.

w	g	0	1	2	3	4	5	6	7	8
7	2	0	0	7	7	7	7	7	7	7
6	3	0	0	7	7	7	13	13	13	13
8	5	0								
5	4	0								
6	6	0								

Es wird noch für eine Zeile die Spalten gefüllt. Danach wird eine Formel hergeleitet.

[1]In der Tabelle: 3 Spalten nach hinten (Übriges Gewicht) und 1 Zeile nach oben (Maximaler Profit vorheriger Gegenstände)

Zunächst werden alle Spalten von $t = 0$ bis $t = 4$ in Zeile 3 mit den vorherigen maximalen Profiten gefüllt (da $g_3 = 5$).

$$t = 5 : max(13, 8 + 0) = 13$$
$$t = 6 : max(13, 8 + 0) = 13$$
$$t = 7 : max(13, 8 + 7) = 15$$
$$t = 8 : max(13, 8 + 7) = 15$$

w	g	0	1	2	3	4	5	6	7	8
7	2	0	0	7	7	7	7	7	7	7
6	3	0	0	7	7	7	13	13	13	13
8	5	0	0	7	7	7	7	7	15	15
5	4	0								
6	6	0								

...

w	g	0	1	2	3	4	5	6	7	8
7	2	0	0	7	7	7	7	7	7	7
6	3	0	0	7	7	7	13	13	13	13
8	5	0	0	7	7	7	7	7	15	15
5	4	0	0	7	7	7	7	12	15	15
6	6	0	0	7	7	7	7	12	15	15

Das maximale Profit für einen Rucksack mit $t = 8$, $w = \{7, 6, 8, 5, 6\}$ und $g = \{2, 3, 5, 4, 6\}$ ist 15 (siehe letzte Zeile und Spalte).

Die Formel

$$dp_{[i][j]} = max(dp_{[i-1][j]}, dp_{[i-1][j-g_i]} + w_i)$$

lässt sich herleiten.[1]

[1] i=Zeile=Gegenstand, j=Spalte=Tragkraft (Tragkräfte von 0 bis t)

Algorithm 5 rucksackproblem_dp(n, g, w, t)

```
1: dp ← [n][t + 1] //Tabelle
2: for i ← 0; i < t + 1; i + + do //Fuellung der ersten Zeile
3:     if i > g_0 then
4:         dp[0][i] = w_0
5:     end if
6: end for
7: for i ← 1; i < n; i + + do
8:     for j ← 0; j < t + 1; j + + do
9:         //Passt der Gegenstand in den Rucksack mit Tragkraft j rein?
10:        if j − g_i >= 0 then
11:            dp[i][j] = max(dp[i − 1][j], dp[i − 1][j − g_i] + w_i)
12:        else
13:            dp[i][j] = dp[i − 1][j]
14:        end if
15:    end for
16: end for
17: return ← dp[n − 1][t]
```

3.3 Laufzeitanalyse

Es wird durch die Tabelle dp mit n Zeilen und t Spalten iteriert (Zwei for-Schleifen). Somit entsteht eine Laufzeit von $\mathcal{O}(n \cdot t)$. Doch in Wirklichkeit ist die Laufzeit nur 'pseudo-polynomiell', weil es einen Unterschied zwischen Eingabegröße und Eingabelänge (gemessen in Bits) gibt. Deshalb ist die Laufzeit des Rucksackproblems exponentiell.
Genauere Gründe liegen bei der (De-)Kodierungslaufzeit.[1]

3.4 Speicheranalyse

Es werden nur konstant Variablen erstellt, also beträgt der Speicherbedarf $\mathcal{O}(1)$.

[1] Zum nachlesen/nachschauen: The Knapsack Problem (MIT), 2011 und Was ist NP-schwer? (Algorithmen und Datenstrukturen), 2021

3.5 NP-Vollständigkeit

Das Problem ist prinzipiell NP-vollständig.[1] Aufgrund der pseudo-polynomiellen Laufzeit des Algorithmus ist das Problem nur 'schwach' NP-vollständig. Wenn die Gewichte der Gegenstände in Binär angegeben werden, dann ist das Problem 'stark' NP-vollständig.

3.6 Andere Lösungsansätze

Hoffentlich kann der Einbrecher jetzt mit dem bestmöglichen Profit nach Hause gehen und die Gegenstände im Rucksack verkaufen. Jedoch hat der Lösungsansatz mit DP viel zu lange gedauert, weswegen sein Fluchtfahrzeug draußen abgeschleppt wurde. Deswegen überlegt der Einbrecher sich noch andere Lösungsansätze.

Tatsächlich gibt es zu diesem Problem noch viele andere Lösungsansätze. Wie zum Beispiel der gierige Ansatz (engl.: Greedy), wobei in $\mathcal{O}(nlogn)$ Laufzeit eine Lösung gefunden wird, die jedoch nicht optimal ist, **aber** 'gut'.[2] Es gibt dann auch Lösungsansätze mit sogenannten genetischen Algorithmen.

[1] Beweis mit mehreren Zwischenreduktionen aus SAT, Hamiltonian Path und dem Teilsummenproblem: NP-Complete Problems (Markus Krötzsch), 2017

[2] im Englischen sagt man oft zu derartigen Algorithmen 'reasonable' (dt.: angemessen; vernünftig)

3.7 Quellcode

Implementiert wird das Programm mit der Programmiersprache Java mit Hilfe der Entwicklungsumgebung IntelliJ IDEA (Community Edition). Die Eingabe der benötigten Werte werden vor Ausführung des Algorithmus abegfragt.

```
1
import java.util.Scanner;
3
public class Rucksackproblem_DP {
5
    public static void main(String[] args) {
7
        Rucksackproblem_DP rucksackproblem = new Rucksackproblem_DP();
9       Scanner scanner = new Scanner(System.in);

11      System.out.println("Anzahl Gegenstaende angeben:");

13      int anzahl = scanner.nextInt();

15      int[] gewichte = new int[anzahl];
        for (int i = 0; i < anzahl; i++) {
17          System.out.println("Gewicht von Gegenstand " + (i+1) +":");
            gewichte[i] = scanner.nextInt();
19      }

21      int[] werte = new int[anzahl];
        for (int i = 0; i < anzahl; i++) {
23          System.out.println("Wert von Gegenstand " + (i+1) +":");
            werte[i] = scanner.nextInt();
25      }

27      System.out.println("Tragkraft des Rucksacks angeben:");
        int tragkraft = scanner.nextInt();
29
        int ergebnis = rucksackproblem.rucksackproblem_dp(anzahl, gewichte, werte, tragkraft);
31      System.out.println("Maximaler Profit: " + ergebnis);
    }
33
    int rucksackproblem_dp(int anzahl, int[] gewichte, int[] werte, int tragkraft) {
35
        int[][] dp = new int[anzahl][tragkraft+1];
37
        for (int i = 0; i < tragkraft+1; i++) {
39          if(i >= gewichte[0])
                dp[0][i] = werte[0];
41      }

43      for (int i = 1; i < anzahl; i++) {
            for (int j = 0; j < tragkraft+1; j++) {
45              if(j-gewichte[i] >= 0)
                    dp[i][j] = Math.max(dp[i-1][j], dp[i-1][j-gewichte[i]]+werte[i]);
47              else
                    dp[i][j] = dp[i-1][j];
49          }
        }
51
        return dp[anzahl-1][tragkraft];
53  }
}
```

4 Literaturverzeichnis

P Komplexitätsklasse (studysmarter.de), o.J.
(Letzter Aufruf: 08.03.2024 19:36)
https://www.studysmarter.de/schule/informatik/technische-informatik/p-komplexitaetsklasse/

Was ist NP-schwer? (Algorithmen und Datenstrukturen), 2021
(Letzter Aufruf: 08.03.2024 19:36)
https://www.youtube.com/watch?v=spUTwAbcw9o

What would be the Impact of P=NP? (Mason Wheeler, vinaykola), 2012 (Letzter Aufruf: 08.03.2024 19:36)
https://softwareengineering.stackexchange.com/questions/148836/what-would-be-the-impact-of-p-np

Difference between NP hard and NP complete problem (sugandha18bcs3001), 2023 (Letzter Aufruf: 08.03.2024 19:36)
https://www.geeksforgeeks.org/difference-between-np-hard-and-np-complete-problem/

Reduktionen: Theoretische Informatik (Niklas Steenfatt), 2020
(Letzter Aufruf: 08.03.2024 19:44)
https://www.youtube.com/watch?v=cMCWJHeKmr0

NP-Complete Problems (Markus Krötzsch), 2017 (Letzter Aufruf: 12.03.2024 19:38)
https://iccl.inf.tu-dresden.de/w/images/3/32/CT2017-Lecture-08-overlay.pdf

SAT Problem (studysmarter.de), o.J. (Letzter Aufruf: 08.03.2024 19:36)
https://www.studysmarter.de/schule/informatik/theoretische-informatik/erfuellbarkeitsproblem/

Cook–Levin theorem or Cook's theorem (soubhikmitra98), 2021
(Letzter Aufruf: 08.03.2024 19:30)
https://www.geeksforgeeks.org/cook-levin-theorem-or-cooks-theorem/

4.5 0/1 Knapsack - Two Methods - Dynamic Programming (Abdul Bari), 2018 (Letzter Aufruf: 08.03.2024 19:32)
https://www.youtube.com/watch?v=nLmhmB6NzcM

0/1 Knapsack Problem Dynamic Programming (Tushar Roy), 2015 (Letzter Aufruf: 08.03.2024 19:36)
https://www.youtube.com/watch?v=8LusJS5-AGo

The Knapsack Problem (MIT), 2011 (Letzter Aufruf: 12.03.2024 19:06)
https://courses.csail.mit.edu/6.006/fall11/rec/rec21_knapsack.pdf

What is Logarithmic Time Complexity? (GeeksforGeeks), 2024 (Letzter Aufruf: 13.03.2024 16:53)
https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/

If You Solve This Math Problem, You Could Steal All the Bitcoin in the World (Ryan F. Mandelbaum), 2019 (Letzter Aufruf: 13.03.2024 16:55)
https://gizmodo.com/if-you-solve-this-math-problem-you-could-steal-all-the-1836047131

Klasse NP-Vollständigkeit (studysmarter.de), o.J. (Letzter Aufruf: 08.03.2024 19:34)
https://www.studysmarter.de/schule/informatik/theoretische-informatik/klasse-np-vollstaendigkeit/

P, NP, CoNP, NP hard and NP complete | Complexity Classes (umng01), 2023 (Letzter Aufruf: 08.03.2024 19:37)
https://www.geeksforgeeks.org/types-of-complexity-classes-p-np-conp-np-hard-and-np-complete/

5 Eidesstattliche Erklärung

'Ich erkläre, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.'

Unterschritft:

Unbekannt :)