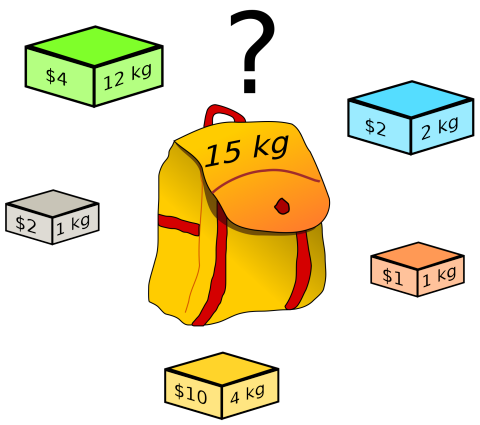
28. Dezember 2017

Das Rucksackproblem

Das Rucksackproblem ist ein Optimierungsproblem nicht nur in der Informatik.
Ein Ganove findet in einem Tresor verschiedene Gegenstände vor, die in Abhängigkeit ihres Typs ein bestimmtes Volumen besitzen und auf dem Schwarzmarkt einen fixen Geldbetrag einbringen. Der Ganove hat nun das Problem, seinen Rucksack so mit Diebesgut zu füllen, dass der erbeutete Wert maximal wird.
Dieses Rucksackproblem gibt es in verschiedensten Varianten. Wir betrachten hier das Problem unter der Annahme, dass Wert und Volumen als ganzzahlige Größe beschreiben werden und die Gegenstände nicht teilbar sind.



Quelle: https://commons.wikimedia.org/wiki/File:Knapsack.svg, Autor: Drake

Definition 1: Rucksackproblem

Gegeben sie eine Zahl V (Volumen des Rucksacks) mit $V \in \mathbb{N}$ sowie eine Menge von (Wert, Volumen) - Paaren $(w_1, v_1), \cdots, (w_1, v_1)$ mit
$w_i, v_i \in \mathbb{N}^+, 1 \leq i \leq n$.

Gesucht ist eine Folge ganzzahliger Werte $a_1, \cdots, a_n \in \mathbb{N}$, so dass $\sum_{i=1}^{n} a_i \cdot v_i \leq V$ und $\sum_{i=1}^{n} a_i \cdot w_i$ maximal wird.

Man kann dieses Problem auch so verstehen, dass für alle Gegenstände entschieden werden muss, ob sie in den Rucksack kommen oder nicht. Gesucht ist also die 0-1-Folge, die die Auswahl der Gegenstände beschriebt, so dass der Wert maximal wird.
Diese Betrachtungsweise verdeutlicht schnell, dass dieses Problem eine exponentielle Komplexität besitzt.
Neben der Brute-Force-Lösung bietet sich hier ein Greedy-Ansatz an. Löse das Problem durch die Idee: nimm in jedem Schritt das Objekt mit der höchsten Gewinnrate (Verhältnis von Gewicht zu Gewinn). Diese Lösung ist schnell, garantiert aber nicht immer die optimale Lösung.
Sicherer ist dagegen die Vorgehensweise nach dem *Bellmann'schen Optimalitätsprinzip*: Das Gesamtproblem in Teilprobleme zerlegen, diese separat optimal lösen. Wenn immer optimale Teillösungen zusammengesetzt werden, dann ist die Gesamtlösung auch optimal.

Und nun noch Theos Beispielprogramm vom Rucksack, welches sehr schön alle diese Konzepte verpackt.

Listing 1: Rucksackproblem

```
1  import Data.List
2  import Data.Function (on)
3
4  type Index = Int
5  type Value = Int
6  type Weight = Int
7  type Item = (Weight, Value)
8  type ItemGroup = [Index]
9  type Solution = (ItemGroup, Value)
10
11 example1:: [Item]
12 example1 = [(2,6),(3,11),(4,15),(5,23),(6,25),(7,27),(8,35),(9,40)
       ,(10,45)]
13
14 knapsack:: [Item] -> Weight -> Solution
15 knapsack objects size = head $ partials objects size
16
17 partials:: [Item] -> Weight -> [Solution]
18 partials _ 0 = [([], 0)]
19 partials objects size = bestsolution : previous
20     where bestsolution = maximumBy (compare `on` solutionValue) [s1,
           s2]
21           s1 = bestwithsize objects size
22           s2 = bestcombination previous size
23           previous = partials objects (size-1)
24
25 bestwithsize:: [Item] -> Weight -> Solution
26 bestwithsize objects size
27     | null fittingitems = ([], 0)
28     | otherwise = ([fst bestitem], itemValue$snd bestitem)
29     where bestitem = maximumBy (compare `on` (itemValue.snd))
          fittingitems
30           fittingitems = filter ((==size).itemWeight.snd) $ zip
                [0..] objects
31
```

```
32 bestcombination :: [Solution] -> Weight -> Solution
33 bestcombination _ 1 = ([], 0)
34 bestcombination solutions size = maximumBy (compare `on`
       solutionValue) $ [solutionConcat (solutions !! (size - i - 1),
       solutions !! (i - 1)) | i <- takeWhile ((<=size) . (*2)) [1..]]
35
36 solutionConcat :: (Solution, Solution) -> Solution
37 solutionConcat (s1, s2) = (solutionItems s1 ++ solutionItems s2,
       solutionValue s1 + solutionValue s2)
38
39 solutionItems :: Solution -> ItemGroup
40 solutionItems = fst
41
42 solutionValue :: Solution -> Value
43 solutionValue = snd
44
45 itemWeight :: Item -> Weight
46 itemWeight = fst
47
48 itemValue :: Item -> Value
49 itemValue = snd
```