TECHNISCHE UNIVERSITÄT DARMSTADT
FACHGEBIET THEORETISCHE INFORMATIK

PROF. JOHANNES BUCHMANN
NABIL ALKEILANI ALKADRI
NINA BINDEL
PATRICK STRUCK

# Algorithmen und Datenstrukturen

SoSe 2018

TECHNISCHE UNIVERSITÄT DARMSTADT

**12. Lösungsblatt** — 02.07.2018
v1.1

## P1 Rucksack-Problem

Ein Rucksack soll mit $n$ Gegenständen gefüllt werden. Die Gegenstände haben positive Nutzwerte $w_1, ..., w_n$ sowie positive Gewicht $g_1, ..., g_n$. Im Rucksack kann ein Maximalgewicht $G_{max}$ transportiert werden. Die Gegenstände können beliebig zerlegt werden und anteilig mitgenommen werden (z.B. ein halber Regenschirm). Der Nutzwert soll maximiert werden.

Gesucht ist also ein Vektor $(\alpha_1, ...\alpha_n) \in [0,1]^n$, sodass

$$\begin{array}{rcl} \alpha_1 g_1 + ... + \alpha_n g_n & \leq & G_{max} \\ \alpha_1 w_1 + ... + \alpha_n w_n & & maximal \end{array}$$

a) Implementieren Sie in Pseudocode einen gierigen Algorithmus (greedy algorithm), der dieses Problem effizient löst. Sie dürfen Hilfsfunktionen, wie z.B. eine Sortierfunktion, verwenden ohne diese zu beschreiben.

b) Welche Laufzeit hat der Algorithmus?

Angenommen, die Gegenstände dürfen nicht zerlegt werden, d.h. entweder man packt den ganzen Gegenstand in den Rucksack oder man lässt ihn draußen.

c) Zeigen oder widerlegen Sie, dass ein gieriger Algorithmus wie in a) dieses Problem effizient lösen kann.

**Lösung.**

a) siehe Algorithm 1.

b) Der Laufzeit des Algorithmus befindet sich in $O(n \log n)$. Diese resultiert aus dem sortieren in Zeile 6. Das ëinpacken"verschlechtert die Laufzeit nicht mehr.

c) Diese Version ist bekannt als (0,1)-Rucksackproblem. Sie ist NP-vollständig und besitzt wahrscheinlich keinen effizienten Algorithmus. Ein Gegenbeispiel ist daher gegeben mit $G_{max} = 50$ und

| **Gegenstand** $k$ | **Wert** $w_k$ | **Gewicht** $g_k$ | **Profitabilitätsindex** $d_k$ |
|---|---|---|---|
| 1 | 60 | 10 | 6 |
| 2 | 100 | 20 | 5 |
| 3 | 120 | 30 | 4 |

Der Greedy-Algorithmus würde die Gegenstände mit den höchsten Profitabilitätsindizes, also Gegenstand 1 und 2, wählen, was zu einem Gewicht von $G = 30$ und einem Gesamtwert von $W = 160$ resultiert. Da $G_{max} = 50$ passt der dritte Gegenstand nicht mehr rein. Die optimale Lösung würde allerdings Gegenstand 2 und 3 beinhalten ($G = 50, W = 220$) und würde den ersten Gegenstand trotz seines hohen Profitabilitätsindex außen vor lassen.

**Algorithm 1**

```
1:  procedure RUCKSACK(g = (g_1, ..., g_n), w = (w_1, ..., w_n))
2:      for i = 1..n do
3:          d_i ← w_i / g_i
4:          α_i ← 0
5:      end for
6:      SortDescending(d_i)          ▷ Sortiert Gegenstände absteigend nach Wert pro Gewicht
7:      i ← 0
8:      r ← G_max
9:      while r > 0 do
10:         i ← i + 1
11:         if g_i ≤ r then          ▷ Gegenstand passt komplett in den Rucksack
12:             α_i ← 1
13:             r ← r − g_i
14:         else                     ▷ Gegenstand passt nur teilweise in den Rucksack
15:             α_i ← r / g_i
16:             r ← 0
17:         end if
18:     end while
19: end procedure
```

## P2 String matching

Gegeben sei der Text $T = jvjggjvjggvjjvjjvjvjg$ und das Pattern $P = jvjggv$.

a) Verwenden Sie den naiven String Matching Algorithmus um alle Vorkommen von $P$ in $T$ zu finden. Vervollständigen Sie dazu folgende Tabelle:

| Iteration | I | R |
|---|---|---|
| 0 | 0 | - |
| 1 | 0 | - |
| 2 | 0; 2 | - |
| 3 | 0 | - |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |
| 10 | | |
| 11 | | |
| 12 | | |
| 13 | | |
| 14 | | |
| 15 | | |
| 16 | | |
| 17 | | |
| 18 | | |
| 19 | | |
| 20 | | |

Dabei enthält die Menge $I$ Indizes $i$ mit $T[i] = P[0]$, also die Startindizes potentieller Matches (diese werden aus der Tabelle entfernt, sobald ein Zeichen des potentiellen Matches nicht mehr korrekt ist). Die Menge $R$ enhält die Startindizes erfolgreicher (also vollständiger) Matches.

b) Bei welchen Eingabeparametern läuft der Algorithmus besonders ineffizient?

c) Welche Laufzeit hat der Algorithmus?

**Lösung.**

a) Die Tabelle resultiert zu:

| Iteration | I | R |
|---|---|---|
| 0 | 0 | - |
| 1 | 0 | - |
| 2 | 0; 2 | - |
| 3 | 0 | - |
| 4 | 0 | - |
| 5 | 5 | - |
| 6 | 5 | - |
| 7 | 5;7 | - |
| 8 | 5 | - |
| 9 | 5 | - |
| 10 | - | 5 |
| 11 | 11 | 5 |
| 12 | 12 | 5 |
| 13 | 12 | 5 |
| 14 | 12;14 | 5 |
| 15 | 15 | 5 |
| 16 | 15 | 5 |
| 17 | 15;17 | 5 |
| 18 | 17 | 5 |
| 19 | 17;19 | 5 |
| 20 | 17 | 5 |

Das Muster kommt also vollständig einmal vor, beginnend bei Index $i = 5$ des Textes.

b) Der Algorithmus läuft insbesondere ineffizient, wenn eine große Anzahl gleicher Zeichen hintereinander stehen. Z.B.:
$T = AAAAAAAAAAAAAAAAAAAAAAAAAB$
$P = AAAAAAAAAAAAB$

c) Der naive String Matching Algorithmus liegt in $O((n-m+1)\cdot m)$, wobei $m$ die Länge des Musters und $n$ die Länge des Textes sind.

## P3 Rabin-Karp-Algorithmus

Gegeben seien die Integer-Arrays $T = [1,3,1,5,7,8,2,6,4,9,5,7,8,7,5]$ und $P = [5,7,8]$, sowie $q = 11$. Wenden Sie den Rabin-Karp-Algorithmus an, um alle Vorkommen von $P$ in $T$ zu finden, indem Sie folgende Tabelle ausfüllen.

| $i$ | $t_i$ | $h(t_i)$ | $h(t_i) == h(P)$ | $t_i == P$ | hit/miss |
|---|---|---|---|---|---|
| 0 | | | | | |
| 1 | | | | | |
| 2 | | | | | |
| 3 | | | | | |
| 4 | | | | | |
| 5 | | | | | |
| 6 | | | | | |
| 7 | | | | | |
| 8 | | | | | |
| 9 | | | | | |
| 10 | | | | | |
| 11 | | | | | |
| 12 | | | | | |

Verwenden Sie dabei die Hashfunktion:

$$h(k) = (k[0]10^2 + k[1]10^1 + k[2]10^0) \mod q.$$

**Lösung.** Wir teilen $T$ zunächst in die Subarrays $t_i$ auf und berechnen die Hashfunktion. Ist dieser Wert gleich $h(P) = 6$, überprüfen wir den potentiellen Match auf Richtigkeit.

| $i$ | $t_i$ | $h(t_i)$ | $h(t_i) == h(P)$ | $t_i == P$ | hit/miss |
|---|---|---|---|---|---|
| 0 | [1,3,1] | 10 | false | | miss |
| 1 | [3,1,5] | 7 | false | | miss |
| 2 | [1,5,7] | 3 | false | | miss |
| 3 | [5,7,8] | 6 | **true** | **true** | hit |
| 4 | [7,8,2] | 1 | false | | miss |
| 5 | [8,2,6] | 1 | false | | miss |
| 6 | [2,6,4] | 0 | false | | miss |
| 7 | [6,4,9] | 0 | false | | miss |
| 8 | [4,9,5] | 0 | false | | miss |
| 9 | [9,5,7] | 0 | false | | miss |
| 10 | [5,7,8] | 6 | **true** | **true** | hit |
| 11 | [7,8,7] | 6 | **true** | false | spurious hit |
| 12 | [8,7,5] | 6 | **true** | false | spurious hit |

## H1 Kruskal-Algorithmus

Der Kruskal-Algorithmus, welcher einen minimal aufspannenden Baum aus einen Graphen G erstellt, zählt ebenfalls zu den gierigen Algorithmen.

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3       do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
5  for each edge (u, v) ∈ E, taken in nondecreasing order by weight
6       do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

a) Überlegen Sie sich eine Methode, wie der zweitleichteste aufspannende Baum ermittelt werden kann.

b) Zeigen Sie folgende Aussagen für einen ausgegebenen, minimal aufspannenden Baum:

1. Der Algorithmus ist gierig.
2. Der Algorithmus terminiert.
3. T ist ein aufspannender Teilgraph von G.
4. T ist azyklisch.

**Lösung.**

a) Ausgehend von einem minimal aufspannenden Baum $T = (V, E')$ wird über alle Kanten $e \in E \setminus E'$ iteriert, wodurch $T \cup \{e\}$ zyklisch wird. Aus diesem Zyklus entfernen wir die schwerste von $e$ verschiedene Kante $d_e$. Dieser aufspannende Baum $T_e$ hat nun ein Gewicht von $w(T)+w(e)-w(d_e)$. Die Kante $e^* \in E \setminus E'$ mit $w(e^*)-w(d_e{}^*)$ minimal ergibt den zweitleichtesten aufspannenden Baum.

b) Die vier Aussagen stimmen:

1. Dies folgt aus der Sortierung in Zeile 4. Nach jeder Iteration wird immer die kleinste Kantengewichtung gewählt. Dies entspricht dem optimal Iterationszustand der charakteristisch für greedy algorithms sind.
2. Dies folgt aus den terminierenden for-each-Schleifen, da nicht über die Grenzen hinaus iteriert wird.
3. Die Menge der Knoten von T und G sind gleich. Zudem gilt in Zeile 4 $(u, v, ) \in E$.
4. Siehe Zeile 6.