Algorithmen und Datenstrukturen
Musterlösung 8

Martin Avanzini <martin.avanzini@uibk.ac.at>
Thomas Bauereiß <thomas.bauereiss@uibk.ac.at>
Herbert Jordan <herbert@dps.uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>

17. Mai, zur Besprechung am 24. Mai

Aufgabe 1) Dynamische Programmierung

1. Sei $e(b, i)$ der maximale Ertrag bei einem Gewichtslimit von b unter der Berücksichtigung der Sorten $0, \ldots, i-1$:

$$e(b, 0) = 0$$
$$e(b, i+1) = \begin{cases} e(b,i) & b < m[i] \\ max\,(e(b,i), e(b-m[i], i) + v[i]) & sonst \end{cases}$$

   Daher: solange noch keine Sorte berücksichtigt wurde, ist der Ertrag 0. Bei der Berücksichtigung jeder zusätzlichen Sorte muss entschieden werden ob die zusätzliche Masse das Gewichtslimit nicht übersteigt (Fallunterscheidung) und zum anderen, ob der maximale Ertrag des um die Masse der aktuelle Sorte reduzierten Gewichtslimits zuzüglich des Wertes der aktuellen Sorte größer ist als der Ertrag der erreicht werden kann ohne die aktuelle Sorte aufzunehmen.

2. Für eine Implementierung müssen die Gewichte der einzelnen Sorten auf sowie das Gewichtslimit auf einen endlichen Bereich der natürlichen Zahlen reduziert werden (um einen endlichen Array zur Zwischenspeicherung der Ergebnisse verwenden zu können) – im Beispiel durch Multiplikation aller Gewichte mit 2 (auch das Limit).

```
1  import java.util.BitSet;
2
3  public class Knapsack {
4
5      /**
6       * Solves the knapsack problem.
7       *
8       * @param limit
9       *             the total weight limitation
10      * @param m
```

```
11          *            an array describing the mass of the individual items
12          * @param v
13          *            the value of the individual items
14          * @return a bit set describing the indices of the items to be
                picked.
15          */
16         public static BitSet solve(int limit, int m[], int v[]) {
17             assert (m.length == v.length);
18
19             int n = m.length;
20
21             // the evaluation of the recursive function e
22             int e[][] = new int[limit + 1][n + 1];
23
24             // initialize e(b,0)
25             for (int b = 0; b < limit; b++) {
26                 e[b][0] = 0;
27             }
28
29             // process recursive part
30             for (int i = 0; i < n; i++) {
31                 // investigate insertion of item n ...
32                 for (int b = 0; b <= limit; b++) {
33                     if (b < m[i]) {
34                         // first case: b < m[i] => do not add item
35                         e[b][i + 1] = e[b][i];
36                     } else {
37                         // second case: it would fit
38
39                         // compute costs of alternatives
40                         int without = e[b][i];
41                         int with = v[i] + e[b - m[i]][i];
42
43                         // pick better option
44                         if (with > without) {
45                             e[b][i + 1] = with;
46                         } else {
47                             e[b][i + 1] = without;
48                         }
49                     }
50                 }
51             }
52
53             // reconstruct solution by walking once through array
54             BitSet res = new BitSet();
55             int b = limit;
56             for (int i = n; i > 0; i--) {
57
58                 // determine whether item i-1 has been picked
59                 boolean picked = e[b][i] != e[b][i-1];
60                 if (picked) {
61                     // the item was picked => add item to res ...
62                     res.set(i - 1);
63
64                     // ... and continue walk with smaller limit
65                     b -= m[i - 1];
66                 }
67             }
68
69             return res;
70         }
71
```

```
72      public static void main(String[] args) {
73
74          // masses * 2 => to get integers
75          int m[] = { 1, 4, 2, 3, 3, 1 };
76          int v[] = { 250, 400, 300, 350, 280, 240 };
77
78          int limit = 5*2; // also limit * 2
79
80          BitSet res = solve(limit, m, v);
81
82          // compute total sum
83          int sum =0;
84          for(int i=0; i<v.length; i++) {
85              if(res.get(i)) {
86                  sum += v[i];
87              }
88          }
89
90          System.out.println("Selected Items: " + res);
91          System.out.println("Total Value:    " + sum);
92
93      }
94
95  }
```

Als Resultat-Datenstruktur wurde ein BitSet verwendet (Aufgrund der einfachen Darstellung von Teilmengen). Das Programm ermittelt die optimale Lösung A, C, D, E, F mit einem Gesamtwert von 1420 GE.

3. Laufzeit: $\mathcal{O}(b \cdot |S|)$

Aufgabe 2) Greedy-Verfahren: Huffman Codierung

1. Huffman Codierung - Gegeben: ein Alphabeth Σ sowie die Häufigkeit $f : \Sigma \to \mathbb{N}$ welche jedem Zeichen $c \in \Sigma$ die Anzahl der Verwendung innerhalb eines zu komprimierenden Strings zuordnet. Der folgende Algorithmus erzeugt einen Baum B aus dem eine minimale Präfix-Codierung abgeleitet werden kann:

Algorithm 1 Huffman Codierung

1: $F = \{Node(c, f(c), \perp, \perp) | c \in \Sigma\}$
2: **while** $|F| > 1$ **do**
3: $\quad n_1, n_2 :=$ die zwei Bäume in F mit den kleinsten Häufigkeiten in deren Wurzeln
4: $\quad F := (F \setminus \{n_1, n_2\}) \uplus \{Node(\_, n_1.h + n_2.h, n_1, n_2)\}$
5: **end while**
6: **return** verbliebenen Baum B in F

Im Algorithmus wird die Datenstruktur $Node(x, y, z, w)$ zur Darstellung eines Knoten in einem binären Baum verwendet, welcher dem Zeichen $x \in \Sigma \cup \{\_\}$ die Häufigkeit $y \in \mathbb{N}$ zuordnet und dessen linker / rechter Teilbaum durch die Strukturen z und w definiert sind. Die Konstante $\perp$ representiert den leeren Teilbaum. Der operator $.h$ erlaubt die Häufigkeit aus einem Knoten auszulesen.

Im resultierenden Baum B finden sich die Zeichen des Alphabeth Σ ausschließlich in den Blättern. Durch zuordnen einer 0/1 zu jeder Kante die einen Knoten mit dessen lin-

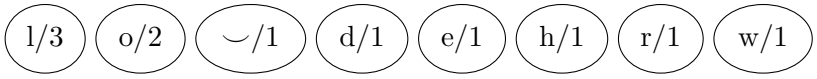
ken/rechten Teilbaum verbindet kann die Codierung eines Zeichens $c \in \Sigma$ entlang des Pfades von der Wurzel zum entsprechenden Blattknoten abgelesen werden.

2. Der Algorithmus ist gierig da er versucht auf basis einer Reihe von lokalen Entscheidungen (durch jeweils der Auswahl der 2 kleinsten Bäume n_1 und n_2 im aktuellen Wald F) eine optimalen Lösung entwickelt. Für diese Entscheidung müssen keine Teilprobleme gelöst werden.

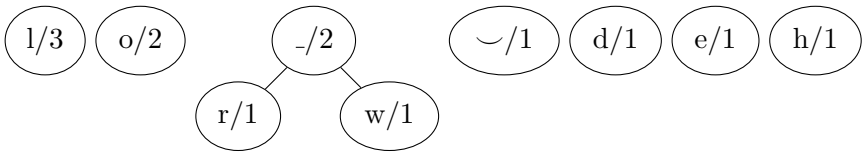
3. Häufigkeitstabelle:

Zeichen c	h	e	l	o	‿	w	r	d
$f(c)$	1	1	3	2	1	1	1	1

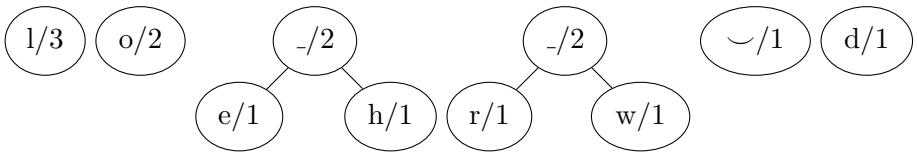
Jedes Zeichen ist ein Knoten (schon sortiert nach Häufigkeit und Alphabeth)



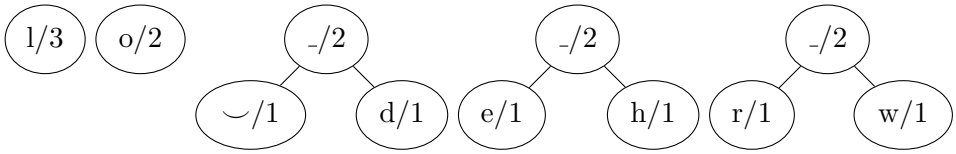
Die 2 Sub-Bäume mit der geringsten Häufigkeit werden verschmolzen und einsortiert:



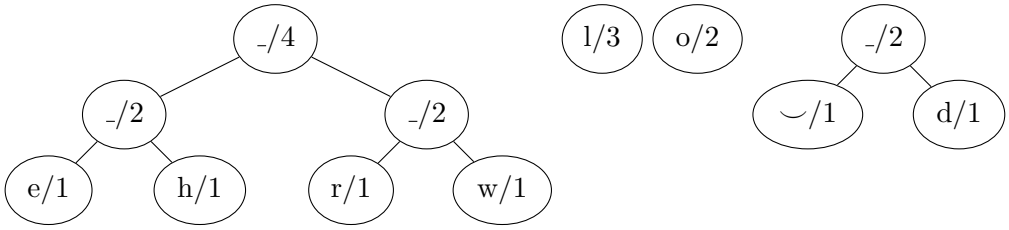
Dies wird wiederholt bis nur noch ein Baum übrig ist.



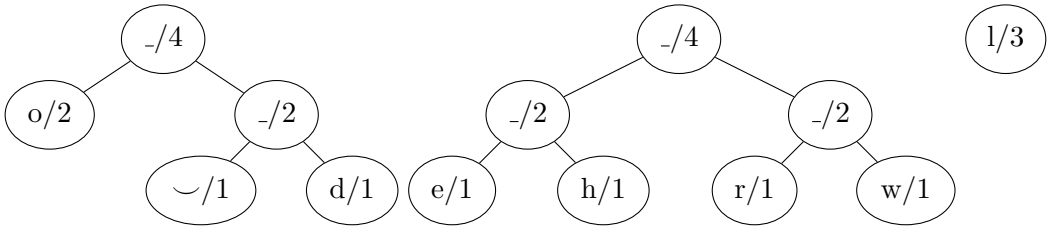
Verschmelzung von ‿ und d:



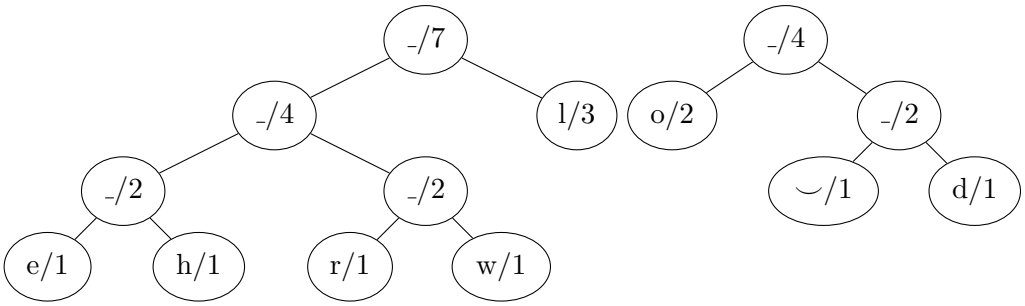
Verschmelzung von e/h und r/w:



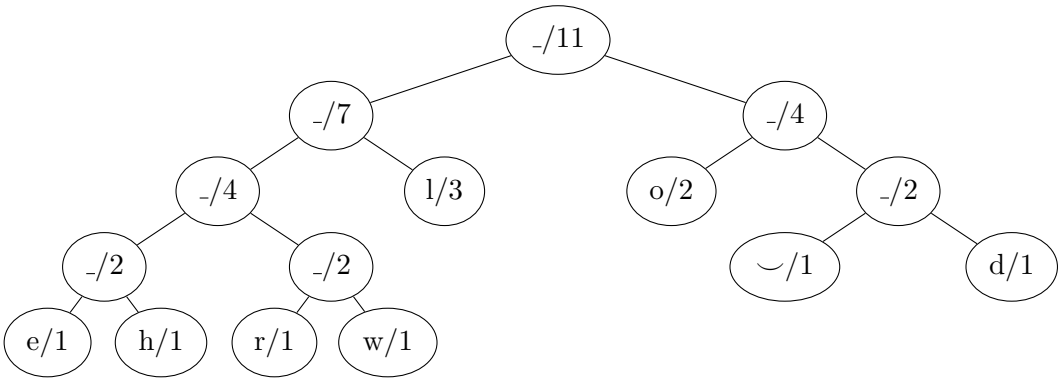
Verschmelzung von o und ‿/d:



Verschmelzung von e/h/r/w und l:



Letzter Schritt:



Aus dem Baum lassen sich folgende Codewörter für die einzelnen Zeichen ableiten:

Zeichen c	h	e	l	o	⌣	w	r	d
Code	0001	0000	01	10	110	0011	0010	111

Der String "hello world"benötigt regulär codiert $8 \times 11 = 88$ bits. Mit der gegeben Codierung kann der String mittels $3 \times 2 + 2 \times 2 + 4 + 4 + 4 + 4 + 3 + 3 = 32$ bit codiert werden.

Aufgabe 3) Entwicklung eines Greedy-Verfahren

1. Greedy: immer die größte Münze wählen, die den Restwert nicht übersteigt.
2. €1.48 = €1 + €0.20 + €0.20 + €0.05 + €0.02 + €0.01
3. Das Kriterium führt für die Euromünzen immer zu einer minimalen Anzahl an Münzen. Beweis: Für Euromünzen gilt: in einer Optimalen Lösung können €0.02 und €0.20 Münzen höchstens 2x vorkommen, da 3 Stück jeweils durch €0.05 + €0.01 bzw. €0.50 + €0.10 ersetzt werden können um eine besser Lösung zu erhalten. Alle übrigen Münzen (ausgenommen die €2.00 Münze) können nur 1x vorkommen, da sonst jeweils 2 Stück durch eine größere Münze ersetzt werden könnten. Sei f eine Funktion die jeder Münze ihre maximale Häufigkeit in einer Optimalen Lösung zuordnet. Für jede Euromünze i gilt $\sum_{1 \leq j < i} f(j) * w_j \leq w_i$. Die Summe aller möglichen kleineren Münzen in einer optimalen Lösung ist kleiner gleich einer gegebenen Münze. Daher, ohne das hinzufügen der größten Münze die den Restwert nicht übersteigt kann der Restwert nur dann erreicht werden, sollte er exakt einem Münzwert entsprechen. In diesem Fall ist die Wahl der passenden Münze trivialerweise optimal. Andernfalls folgt, dass die größte Münze die den Restwert nicht übersteigt in jedem Fall notwendig ist um den Zielbetrag zu erreichen. Des Weiteren folgt, dass die optimale Lösung eindeutig ist und durch das angegebene Greedy Kriterium berechnet werden kann.

   Dies gilt jedoch nicht für alle möglichen Münzbeträge W. So würde bei einer Einführung einer 99 Cent Münze dieses Greedy-Kriterium nicht mehr zu einer optimalen Lösung führen.

Z.B. würde der Betrag von € 1.49 mit € 1.00 + 2x€ 0.20 + € 0.05 + 2x€ 0.02 beglichen - daher, mit 6 Münzen. Hingegen könnte man mit € 0.99 + € 0.50 den Betrag mit lediglich 2 Münzen abdecken.