Lösungen für Übungsblatt 3

Henning Lehmann

1. November 2022

Aufgabe 3.1: Container beladen

(a)

Damit die Indizes aus der Aufgabenstellung mit denen im Pseudocode übereinstimmen, wird im Folgenden angenommen, dass die Array-Indizes bei 1 beginnen.

GreedyStack(**int**[] v, **int** V)

```
1  sortiere(v);
2  for (int i = 1; i <= v.length; i++) {
3    for (int j = 1; j <= container.length; j++) {
4      if (containerKapazitaet[j] >= v[i]) {
5        container[j].append(v[i]);
6        containerKapazitaet[j] -= container[j];
7        objektPasstInVorhandenenContainer = true;
8        break;
9      }
10   }
11   if (!objektPasstInVorhandenenContainer) {
12     neuerContainerIndex = container.length + 1;
13     container[neuerContainerIndex].append(v[i]);
14     containerKapazitaet[neuerContainerIndex] = V - v[i];
15   }
16 }
17 return container;
```

(b)

In Zeile 1 wird das Array, welches die Objekte enthält, auf folgende Weise sortiert:

$$\text{objekte} = (6, 5, 4, 3, 2, 2, 2, 1)$$

Beim ersten Durchlauf der äußeren Schleife wird die innere Schleife übersprungen, da das Feld `container` noch keine Einträge enthält.

Das Feld `objektPasstInVorhandenenContainer` ist in Zeile 11 dementsprechend `false`, und das Objekt wird in einen neuen Container gepackt. Die verbleibende Kapazität des Containers wird im Feld `containerKapazitaet` an der Stelle 1 gespeichert und enthält den Wert 0, da `container[1]==V` und der Container damit voll ist.

Bei allen folgenden Objekten wird nun zunächst geprüft, ob in den Containern noch Platz ist. Dies ist bei den Objekten an den Stellen 2, 3 und 4 noch nicht der Fall, daher wird für jedes diese Objekte ein weiterer Container hinzugefügt. Die Kapazitäten der Container nach diesen ersten vier Objekten sind:

$$\text{containerKapazitaeten} = (0, 1, 2, 3)$$

Die verbleibenden Objekte werden wie folgt einsortiert:

Das Objekt v_5 wird in den Container an $j = 3$ einsortiert, da dies der erste Container ist mit Kapazität ≥ 2.

v_6 wird in den Container an $j = 4$ einsortiert. Nun gilt:
$\text{containerKapazitaeten} = (0, 1, 0, 1)$.

Da kein Container mehr eine Kapazität ≥ 2 hat, wird v_7 in einen neuen Container sortiert.

Das letzte Element wird in den Container an $j = 2$ einsortiert.

Der Algorithmus gibt so die folgende Lösung zurück:

$$\text{container} = ((6), (5, 1), (4, 2), (3, 2), (2))$$

Dass diese Lösung optimal ist, lässt sich erkennen, indem man die Volumina aller Objekte zusammenrechnet: $\sum_{i=1}^{n} v_n = 25$.

Mit $V = 6$ ergibt sich mit $\lceil 25/6 \rceil = 5$, dass die Objekte nicht auf weniger als 5 Container aufgeteilt werden können. Die vom Algorithmus gefundene Lösung, welche nur 5 Container enthält, ist damit optimal.

(c)

Gegenbeispiel
Gegeben sei eine Probleminstanz mit $V = 7$ und $v = (3, 3, 2, 2, 2, 2)$.
Die Optimale Lösung benötigt 2 Container:

$$\text{container}_{\text{Optimum}} = ((3, 2, 2), (3, 2, 2))$$

Der Greedy-Algorithmus berechnet jedoch eine Lösung mit 3 Containern:

$$\text{container}_{\text{Greedy}} = ((3, 3), (2, 2, 2), (2))$$

Aufgabe 3.2: Bauarbeiter Kontrollieren

(a)

Annahme: Die Arbeitszeiten sind aufsteigend nach ihren Endzeitpunkten sortiert.

MinKontrollen

```
1 Kontrollen = ∅
2 while Arbeitszeiten ≠ ∅ {
3     k = frühester Endzeitpunkt aus Arbeitszeiten
4     Kontrollen = Kontrollen ∪ k
5     Entferne alle Elemente aus Arbeitszeiten, welche sich mit k
  überschneiden
6 }
7 return Kontrollen
```

(b)

Behauptung: Durch die vom Algorithmus bestimmten Kontrollen wird jeder Bauarbeiter mindestens einmal kontrolliert.

Beweis durch Widerspruch: Angenommen, es gibt einen Arbeiter i, welcher nicht durch den Algorithmus kontrolliert wird. Dann muss in jedem Schleifendurchlauf j sein Schichtbeginn b_i hinter dem in Zeile 3 bestimmten k_j liegen, denn: wäre $b_i \leq k_j$, dann würde k_j, da es der Zeitpunkt des frühesten Schichtendes ist, sich mit der Arbeitszeit von i überschneiden und i damit kontrolliert werden - auch dann, wenn $e_i = k_j$.

Es ist jedoch garantiert, dass in jedem Schleifendurchlauf mindestens ein Element aus `Arbeitszeiten` in Zeile 5 entfernt wird, da mindestens die Arbeitszeit mit Endzeitpunkt k mit k kollidiert. Das heißt, bei einer Anzahl von n Arbeitszeiten ist spätestens nach $j = n - 1$ Schleifendurchläufen die Arbeitszeit von i das einzige verbleibende Element in `Arbeitszeiten`.

Das bedeutet, dass spätestens $k_n = e_i$ und damit $b_i < k_n$. ↯ □

(c)

Untenstehend ist eine Konkretisierung des in Teilaufgabe (a) konzipierten Algorithmus. Es gilt weiterhin die Annahme, dass die Arbeitszeiten bereits

nach ihren Endzeitpunkten sortiert sind.

```
MinKontrollen(int[] s, int[] f)
 1 Kontrollen = ∅
 2 iMinEndzeitpunkt = 0;
 3 for (int i = 0; i < length(s); i++) {
 4     if (f[iMinEndzeitpunkt] ≤ s[i]) {
 5         Kontrollen = Kontrollen ∪ f[iMinEndzeitpunkt]
 6         iMinEndzeitpunkt = i;
 7     }
 8 }
 9 return Kontrollen
```

Da es pro Element genau einen Schleifendurchlauf gibt, und alle Operationen innerhalb der Schleife konstante Laufzeit benötigen, liegt dieser Algorithmus in $\Theta(n)$.

Modifiziert man den Algorithmus dahingehend, dass die Arbeitszeiten noch sortiert werden, hängt seine Laufzeit im Wesentlichen von dem Sortieralgorithmus ab - wählt man MergeSort, liegt die Laufzeit in $\Theta(n \log n)$.

(d)

Beweis via vollständiger Induktion.

Induktionsanfang
Für eine Menge von $n = 1$ Bauarbeitern ist die bestmögliche Anzahl an Kontrollen 1. Da durch den Algorithmus dieser Bauarbeiter genau am Ende seiner Schicht kontrolliert wird, ist die Lösung optimal und der Induktionsanfang damit geschafft.

Induktionsannahme
Der Algorithmus findet für ein beliebiges n eine optimale Lösung, unabhängig von der Verteilung der Arbeitszeiten.

Induktionsschluss
Eine beliebige Menge von n Arbeitszeiten kann durch folgende Möglichkeiten auf eine Menge von $n + 1$ Arbeitszeiten erweitert werden kann:

1. Es wird eine Arbeitszeit hinzugefügt, welche disjunkt von allen anderen Arbeitszeiten ist. In diesem Fall muss eine weitere Kontrolle hinzukommen, da dieser Arbeiter nicht zeitgleich mit anderen kontrolliert werden kann. Hier fügt der Algorithmus eine weitere Kontrolle am Ende der neuen Arbeitszeit ein, und die Lösung ist weiterhin optimal.

2. Es wird eine Arbeitszeit hinzugefügt, welche den frühesten Startzeitpunkt aller Arbeitszeiten hat. In diesem Fall ist die erste Kontrolle, welche der Algorithmus veranschlagt, am Ende dieser neuen Arbeitszeit. Alle Arbeitszeiten, welche mit dieser neuen Arbeitszeit kollidieren, werden hierdurch auch kontrolliert, da sie definitionsgemäß sowohl vor der Kontrolle anfangen als auch nach der Kontrolle aufhören. Alle weiteren Arbeitszeiten bilden eine neue Teilmenge mit m Arbeitszeiten ($m < n$), für welche der Algorithmus laut Induktionsannahme ebenfalls eine optimale Lösung findet.

3. Es wird eine Arbeitszeit hinzugefügt, welche mit dem frühesten Endzeitpunkt aller anderen Arbeitszeiten kollidiert. In diesem Fall wird dieser neue Arbeiter bereits kontrolliert und es gibt nichts weiter zu zeigen.

4. Es wird eine Arbeitszeit hinzugefügt, welche weder disjunkt mit allen anderen Aufgaben ist noch sich mit dem frühesten Endzeitpunkt schneidet oder selber den frühesten Endzeitpunkt hat. In diesem Fall wird jedoch die Probleminstanz durch den Algorithmus so lange optimal verkleinert, bis einer dieser drei Fälle für die verbleibenden Arbeitszeiten eintritt. In diesen Fällen löst der Algorithmus, wie oben gezeigt, die Probleminstanz optimal.

□

Aufgabe 3.3: Optimale Auswahl von Algorithmen

(a)

Mit $n = 4$ und $A := \{(s_i, f_i) | i \in \{1, \ldots, n\}\}$:

$$A = \{(1,3), (3,6), (2,4), (4,5)\}$$

Die Methode GreedyEnde findet folgende Lösung:

Prozessor 1: ((1,3),(4,5))
Prozessor 2: ((2,4))
Prozessor 3: ((3,6))

Die optimale Lösung benötigt lediglich 2 Prozessoren:

Prozessor 1: ((1,3),(3,6))
Prozessor 2: ((2,4),(4,5))

(b)

Mit dem Algorithmus GreedyStart aus der Vorlesung kann eine optimale Lösung für jede Instanz gefunden werden.

Dieser Algorithmus ist korrekt, da die untere Schranke $m = \max_{x \in \mathbb{R}} |A(x)|$ gleichzeitig die obere Schranke für die Anzahl der Prozessoren ist: dadurch, dass nach dem Freiwerden eines Prozessors der frühestmögliche Prozess diesem Prozessor zugeordnet wird, ist sichergestellt, dass jeder Prozessor maximal ausgelastet wird. Zu dem Zeitpunkt, an dem m-te gleichzeitige Prozess hinzukommt, ist noch genau ein Prozessor frei.