Serie 10

Freitag, 6. Mai 2022 16:07

Theoretische Aufgaben

1. Nicht jede Greedy-Methode für das Aktivitäten-Auswahl-Problem erzeugt eine maximale Menge paarweise zueinander kompatibler Aktivitäten. Geben Sie Beispiele für die folgenden Strategien, die zeigen, dass die Strategien nicht zu optimalen Lösungen führen. Skizzieren Sie Ihre Beispiele grafisch.

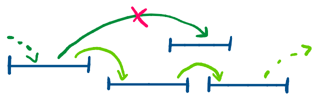
   (a) Man wählt immer die Aktivität mit der geringsten Dauer, die zu den vorher ausgewählten Aktivitäten kompatibel ist.

   (b) Man wählt immer die kompatible Aktivität, die sich mit den wenigsten anderen noch verbliebenen Aktivitäten überlappt.

   (c) Man wählt immer die Aktivität mit der frühesten Startzeit, die zu den vorher ausgewählten Aktivitäten kompatibel ist.

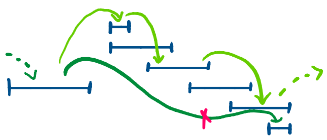
   (1 Punkt)

a)



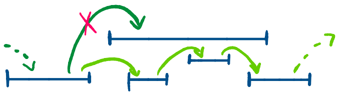
Hier kann die kürzeste Aktivität zwei aufeinanderfolgende Aktivitäten "überlappen", wodurch nicht optimale Lösungen entstehen.

b)



Wenn zwei Aktivitäten gleich wenig Überlappungen haben, jedoch eine zuerst auftritt, können durch das auswählen der zweiten Aktivität vorherige Aktivitäten ausgelassen werden.

c)



Wenn die erstbeginnende Aktivität ausgewählt wird, kann diese mehrere kürzere Aktivitäten "überschatten", wodurch nicht optimale Lösungen produziert werden.

2. Gegeben sei folgende Zeichenkette bestehend aus den Zeichen a, b, d, o, p, q, r:

$$opabqprqpdbqpdopq$$

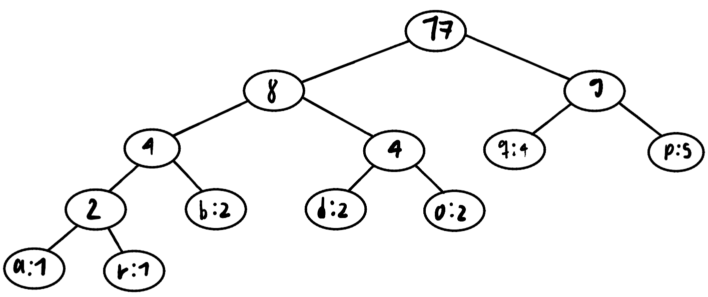
   Konstruieren Sie einen Huffman-Code für diese Zeichenkette. Stellen Sie den binären Codierungsbaum dar und geben Sie für jedes Zeichen den Binärcode an. Wie lang ist die Huffman-codierte Zeichenkette? Sie müssen nicht den gesamten Code für die Zeichenkette aufschreiben.

   (1 Punkt)

i	a	b	d	o	p	q	r
n_i	1	2	2	2	5	4	1

$\Rightarrow$

i	p	q	b	d	o	a	r
b_i	0	101	100	1110	1111	1100	1101



3. In dieser Aufgabe betrachten wir das Problem, Wechselgeld für n Cent zusammenzustellen, indem wir so wenige Münzen wie möglich verwenden.

   a) Entwerfen Sie einen Greedy-Algorithmus, der das Wechselgeld aus Münzen mit Nennwerten aus der Menge $N = \{25, 10, 5, 1\}$ Cent zusammenstellt und beweisen Sie, dass Ihr Algorithmus zu einer optimalen Lösung führt. Der Beweis soll die folgenden Schritte beinhalten:

      – Zeigen Sie, dass das Wechselgeld-Problem eine *optimale Teilstruktur* hat.
      – Zeigen Sie, dass Ihr Algorithmus die *Gierige-Auswahl-Eigenschaft* hat. Das bedeutet, dass es immer eine optimale Lösung gibt die durch Ihre *gierige* Auswahlvorschrift gefunden werden kann (die lokale optimale Wahl führt zu einer globalen optimalen Lösung).

   (1 Punkt)

Zunächst beweisen wir die optimale Teilstruktur. Nehmen wir an, dass wir für n Cent Wechselgeld eine optimale Lösung mit t Anzahl Münzen haben. Sei $\{m_1, —, m_t\}$ das Set von diesen Münzen. Es gilt somit $v(m_1) + v(m_2) + — + v(m_t) = n$.

v ist eine Funktion, die von einer Münze den Wert gibt.

Fixieren wir nun die Münze m_1. Wir möchten zeigen, dass das Set $\{m_2, -, m_t\}$ eine optimale Lösung für $n - v(m_1)$ ist (per Widerspruch). Nehmen wir an, dass dieses nicht die optimale Lösung ist. Dann könnten wir eine Münze m_i weglassen und hätten immer noch den Wert $n - v(m_1)$. Dann könnten wir jedoch diese Kombination brauchen für die Lösung von n cent. Dies gibt uns einen Widerspruch!

Nun zeigen wir die Gierige-Auswahl-Eigenschaft. Hierfür machen wir eine Fallunterscheidung für den Wert n.

(i) $1 \leq n < 5$: Hier kann Lösung nur aus 1er-Münzen bestehen und somit führt die gierige Auswahl (hier 1er Münze) zur optimalen Lösung.

(ii) $5 \leq n < 10$: analog wie oben, da am besten zuerst eine 5er zu wählen und dann fallen wir in Fall (i)

(iii) $10 \leq n < 25$: Wähle eine 10er. Falls dann Wert immer noch zwischen 10 und 25 liegt wähle nochmals eine 10er (sonst jetzt schon im Fall (ii)) und komme danach direkt in den Fall (ii).

(iv) $25 < n$: Wähle 25er solange, bis wir in den Fall (iii) rutschen

Ausserdem bemerken wir, dass in allen Fällen immer die beste Auswahl getroffen wurde, da falls wir die Münzen weiter aufspalten, nur noch mehr Münzen hinzukommen und somit die optimale Eigenschaft verloren geht.

Unser Greedy-Algorithmus funktioniert analog zum Beweis der gierigen Auswahl-Eigenschaft.

b) Geben Sie eine Menge N von (fiktiven) Münzenwerten an, für die Ihr Greedy-Algorithmus nicht zur optimalen Lösung führt. Ihre Menge sollte 1-Cent Münzen enthalten, damit es für jeden Wert von n eine Lösung gibt. Nehmen Sie an, dass der Nennwert jeder Münze eine ganze Zahl ist. Geben Sie ein Beispiel an, wo der Greedy-Algorithmus versagt. **(1 Punkt)**

c) Geben Sie einen auf dynamischer Programmierung basierenden Algorithmus an, der das Wechselgeld für eine beliebige Menge N von k verschiedenen Münzwerten und einen beliebigen Betrag von n Cent erstellt, wobei in N immer 1-Cent Münzen enthalten sind. Ihr Algorithmus sollte in Zeit $\mathcal{O}(n \cdot k)$ ablaufen. *Tipp: Bauen sie eine Tabelle* `c[j]` *auf, welche für jeden Betrag j die optimale Anzahl Münzen enthält. Bauen Sie parallel dazu eine Tabelle* `denom[j]` *auf, welche den Wert einer beliebigen Münze angibt, die in der optimalen Lösung für j Cent vorkommt. Im letzten Schritt soll der Algorithmus mittels der Lösungstabelle* `denom[j]` *eine optimale Lösung konstruieren.* **(1 Punkt)**

(b) $N = \{50, 20, 1\}$. Falls nun $n = 60$ ist, würde der Greedy-Algorithmus die 50er wählen und danach 10 mal eine 1er-Münze. Damit bräuchten wir insgesamt 11 Münzen. Jedoch würde die Lösung aus nur 3 Münzen ($3 \times 20\text{er} = 60$) bestehen.

(c) Sei $N = \{n_1, \ldots, n_k\}$ das gegebene Set von Münzen mit $v(n_i) \neq v(n_j)\ \forall i,j \in \{1, \ldots, k\}$
Ausserdem gelte $v(n_1) > \ldots > v(n_k)$ mit $v(n_k) = 1$.

<u>OptimalChange Dynamic (N, n)</u>:

```
c, denom = new Array von Länge n
    c[1] = 1
    for i = 2 to n:
        c[i] = ∞
    for i = 2 to n:
        for j = 1 to k:
            if v(n_j) ≤ i and c[i - v(n_j)] < c[i]:
                c[i] = c[i - n_j]
                denom[i] = n_j
return denom
```

$\Theta(k)$ $\Theta(k)$

und somit erhalten wir eine gesamte Laufzeit $\Theta(n \cdot k)$.

⇒ Beachte, dass denom nun die Münzen der optimalen Lösung enthält.

Implementieren Sie eine Java-Programm *HuffmanCode*, welche folgende Funktionalität bietet:

1. Eine Methode `void prefixCode(String s)`, welche für eine gegebene Zeichenkette einen optimalen Präfix-Code generiert. Der Präfix-Code soll als Binärbaum repräsentiert werden. Schreiben Sie dazu eine Klasse `Node` die einen Knoten im Baum darstellt. Verwenden Sie die Klasse `java.util.PriorityQueue` als Prioritätswarteschlange für die Konstruktion des Codes. **(2 Punkte)**
2. Eine Methode `void printCode(String s)`, welche die codierte Darstellung einer Zeichenkette als Folge von Nullen und Einsen ausgibt. **(2 Punkte)**

Diese beiden Aufgaben sind in den Abgaben Node.java und HuffmanCode.java enthalten. Dabei haben wir in der Klasse Node den Knoten modelliert und in der Klasse HuffmanCode den Rest der Programmlogik aufgebaut. Dabei ruft die Methode printCode die Methode prefixCode auf.

3. Demonstrieren Sie Ihr Programm anhand der folgenden Eingaben und geben Sie die durchschnittliche Anzahl Bits pro Zeichen an:
   - *Jobs launched into a sermon about how the Macintosh and its software would be so easy to use that there would be no manuals.*
   - *An academic career in which a person is forced to produce scientific writings in great amounts creates a danger of intellectual superficiality, Einstein said.*

Die Ausgabe für den ersten Satz sieht wie folgt aus:

```
010000111110011101000101101101010101111000010110110010001000100101111110111110011010
010101100101111101110111101110011011001111110101111000011011111001000111001101100000
010001111011000010100101111110111110100110001101011110001000100111101010001010111111
000000111010010110110111111000010010111101011011010001001001111000010101111001100011
011010100000100111011110001011010110000111001101101111000111001101100101111110000010
01011110101101101000100100111100000111111100101110110101110101110110110101001000010
```

Die durchschnittliche Anzahl an Bits pro Zeichen ist **4.008**.

Die Ausgabe für den zweiten Satz sieht wie folgt aus:

```
001010110111101110101011101001000000100101010101101010111001110000000111110010101111
101111101001000010101000100011011101101111000000111011010011101111001001101101111111
100110111101000010010110100010011110111100011110011100100011010100001100110101001000
001011100001011111101010101101111101011101010000101011111101011011001010111101111101
011100011101000110111000100110011001101011100001101101010011100011101000000001101101
110110100101110101111110100001111101001111111111001010111000000000111001110001010100
000110111000111110011000110111100000011111111010101001011100011101010001111100001 0
1001100010110010101101101000000010101111001101110010100100010111
```

Die durchschnittliche Anzahl an Bits pro Zeichen ist **4.082**.