Universität Karlsruhe
Fakultät für Informatik
ITI Wagner

1. Musterlösung

Problem 1: Average-case-Laufzeit vs. Worst-case-Laufzeit 1pt

(a) Folgender Algorithmus löst das Problem der linearen Suche

Algorithmus 1 : LINEARESUCHE(A, v)

Eingabe : Array A der Länge n, Wert v
Ausgabe : Index i mit $A[i] = v$

```
1 Für i ← 1...n
2 |  Wenn A[i] = v
3 |  |  return i
4 return NIL
```

Wir beweisen die Korrektheit des Algorithmus in zwei Schritten.

Angenommen das linkeste Vorkommen von v in A sei an der Stelle k, also $A[k] = v$. Da die Schleife (Zeile 1) das Array sukzessive für $i = 1, \ldots, n$ durchläuft, gilt für $i = k$, dass die Zeile 3 ausgeführt wird. Es wird also der gesuchte Index k zurückgegeben.

Sei hingegen der Wert v nicht in A vorhanden. Der Vergleich in Zeile 2 schlägt damit immer fehl, und die Zeile 3 kommt nicht zur Ausführung. Es wird also nach Verlassen der Schleife der Wert NIL zurückgegeben. □

(b) Das Array A enthalte den Wert v nicht. Die Schleife durchläuft also für i alle Werte von 1 bis n. In jedem Schleifendurchlauf findet in Zeile 2 genau ein Vergleich statt. Für die Anzahl der Vergleichsoperationen gilt also

$$T(n) = n$$

Da dies die maximale Anzahl an möglichen Schleifendurchläufen ist, ist die Worst–case Laufzeit von LINEARESUCHE in

$$T^{\text{worst}}(n) = T(n) = n \in \Theta(n)$$

Betrachte nun die Average–case Analyse. Für A nehmen wir den Wertebereich aller ganzen Zahlen $\mathbb{Z}$ an. Wegen $|\mathbb{Z}| = \aleph_0 = \infty$ folgt, unter Annahme von Gleichverteilung, für die Wahrscheinlichkeit, dass das Element v in A an der Position k vorhanden ist

$$\Pr(A[k] = v) = 0 \qquad \text{für } 1 \leq k \leq n$$

Damit ergibt sich die gleiche Analyse wie für den Fall, dass v nicht in A ist, und wir erhalten eine Average–case Laufzeit von

$$T^{\text{avg}}(n) = T^{\text{worst}}(n) = n \in \Theta(n)$$

Hinweis: Die Annahme, dass die Elemente in A und v nur Werte der Menge $\{1, \ldots, n\}$ annehmen, liefert ebenfalls eine Average–case Laufzeit von $\Theta(n)$.

(c) Das Problem der linearen Suche erfordert im Worst–case, nämlich wenn der zu suchende Wert nicht im Array ist, immer ein vollständiges Durchlaufen des Arrays. Für ein Array mit n Elementen, folgt daher unmittelbar eine untere Schranke von $\Omega(n)$.

↑

Problem 2: Rekursion 2pt

(a) Wir wenden die allgemeinere Form des Master-Theorems an: $T(n) = \sum_{i=1}^{m} T(\alpha_i n) + f(n)$

(i) $T(n) = 4T(n/2) + n$, d.h. $m = 4$, $\alpha_i = \frac{1}{2}$, $f(n) = n \in \Theta(n)$. Wir vergewissern uns, dass die Vorraussetzungen erfüllt sind, d.h. $0 < \alpha_i < 1$, $m \geq 1$ und $f(n) \in \Theta(n^k), k = 1 \geq 0$, und erhalten $\sum_{i=1}^{m} \alpha_i^k = 4 \cdot \frac{1}{2} = 2 > 1$. Also ist $T(n) \in \Theta(n^c)$ mit c bestimmt durch $\sum_{i=1}^{m} \alpha_i^c = 1$. Es folgt $c = 2$ und somit insgesamt $T(n) \in \Theta(n^2)$.

(ii) $T(n) = 4T(n/2) + n^2$, d.h. $m = 4$, $\alpha_i = \frac{1}{2}$, $f(n) = n^2 \in \Theta(n^2)$. Die Vorraussetzungen sind erfüllt und somit erhalten wir $\sum_{i=1}^{m} \alpha_i^k = 4 \cdot \frac{1}{2}^2 = 1$. Also ist $T(n) \in \Theta(n^k \log n) = \Theta(n^2 \log n)$.

(iii) $T(n) = 4T(n/2) + n^3$, d.h. $m = 4$, $\alpha_i = \frac{1}{2}$, $f(n) = n^3 \in \Theta(n^3)$. Die Vorraussetzungen sind erfüllt und somit erhalten wir $\sum_{i=1}^{m} \alpha_i^k = 4 \cdot \frac{1}{2}^3 = \frac{1}{2} < 1$. Also ist $T(n) \in \Theta(n^k) = \Theta(n^3)$.

(b) Rufe den Algorithmus RecursiveInsertSort(A, n) auf wie in Algorithmus 2.

Algorithmus 2 : RecursiveInsertSort(A, j)

Eingabe : Array A, Index j
Ausgabe : sortiertes Array A

```
1  Wenn j == 1
2      return A
3  sonst
4      RecursiveInsertSort(A, j − 1)
5      i = 1
6      solange i < j tue
7          Wenn A[j] < A[i]
8              Füge A[j] an Stelle i ein und verschiebe Rest des Arrays um eine Stelle nach hinten
9              return A
10         sonst
11             i ← i + 1
12     return A
```

$A[n]$ in das bereits sortierte Array $A[1..n-1]$ einzufügen, braucht im worst-case $\Theta(n)$ Vergleiche, nämlich dann, wenn $A[1..n]$ bereits sortiert war (siehe Zeile 6). Wie man anhand Zeile 2 sieht, dauert das Sortieren eines ein-elementigen Arrays konstante Zeit.
Somit lautet die Rekurrenzgleichung:

$$T(n) \in \begin{cases} \Theta(1), & \text{wenn } n = 1 \\ T(n-1) + \Theta(n), & \text{wenn } n > 1 \end{cases}.$$

Wir lösen iterativ:

$$T(n) \in \Theta(n) + \Theta(n-1) + \cdots + \Theta(1) = \Theta(n + (n-1) + \cdots + 1) = \Theta\left(\frac{(n+1) \cdot (n)}{2}\right) = \Theta(n^2).$$

(c) Rufe den Algorithmus FindMissingNumber$(A, 0)$ auf wie in Algorithmus 3. Der Algorithmus nutzt aus, dass das Fehlen einer Zahl in einer Folge zu einem Unverhältnis von 0- und 1-Bits je Bitstelle führt. Daraus lässt sich dann die fehlende Zahl leicht rekonstruieren.

Laufzeit: Die Zeilen 7- 14 werden (immer, also auch im worst-case) n-mal durchlaufen, verursachen also $c \cdot n$ Kosten, wobei c konstant ist. Dafür muss dann im Rekursionsschritt (immer, also auch im worst-case) nur die kleinere Hälfte der Zahlen betrachtet werden. Die Rekurrenzgleichung lautet also:

Algorithmus 3 : FindMissingNumber(A, j)

Eingabe : Array A, Bitindex j
Ausgabe : Fehlende Zahl

```
1  Wenn A.length == 1
2      kippe das j-te Bit von A[0]
3      return A[0]
4  sonst
5      z_0 ← 0
6      z_1 ← 0
7      Für i ∈ A
8          x ← das j-te Bit von A[i]
9          Wenn x == 0
10             B_0[z_0] = A[i]
11             z_0 ++
12         sonst
13             B_1[z_1] = A[i]
14             z_1 ++
15     j ++
16     Wenn z_0 < z_1
17         FindMissingNumber(B_0, j)
18     sonst
19         FindMissingNumber(B_1, j)
```

$$T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + cn$$

Aus dem Mastertheorem folgt (Anwendung s.o.), dass für die worst-case-Laufzeit gilt:

$$T(n) \in \Theta(n) \subseteq O(n).$$

Hinweis: Es geht auch mit einem oder keinem Hilfsarray.

Problem 3: Amortisierte Analysen 3pt

Ganzheitsmethode. Sei c_i die Kosten der i–ten Operation. Dann ist c_i definiert durch

$$c_i = \begin{cases} i & \text{falls } i \text{ eine Potenz von 2 ist} \\ 1 & \text{sonst} \end{cases}$$

Tabelle 1 zeigt die Werte von c_i für $i = 1, \ldots, 10$.

Die Kosten für n Operationen ergeben sich durch

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j \leq n + (2n - 1) < 3n \tag{1}$$

Die durchschnittlichen Kosten pro Operation berechnen sich nach der Ganzheitsmethode durch

$$\frac{\text{Gesamtkosten}}{\text{Anzahl Operationen}} < \frac{3n}{n} = 3$$

Eine Operation hat also amortisierte Kosten von $O(1)$. Da die Kosten jeder Operation zusätzlich offenbar in $\Omega(1)$ sind, gilt insgesamt also $T^{\text{amort}}(n) \in \Theta(n)$

(a)

Operation i	Kosten c_i
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
10	1
⋮	⋮

Tabelle 1: Die Werte von c_i für $i = 1, \ldots, 10$.

Buchungsmethode. Um die Buchungsmethode anzuwenden, veranschlagen wir für jede Operation Kosten von $\hat{c}_i = 3$. Dann soll gelten

- Falls i keine Potenz von 2 ist, bezahle 1 und erhöhe den Kredit um 2.
- Falls i eine Potenz von 2 ist, bezahle i aus dem Kredit.

Tabelle 2 zeigt den Verlauf des Kredits für die Operationen i von 1 bis 10.

Operation i	Amortisierte Kosten $\hat{c}_i$	Eigentliche Kosten c_i	Restkredit
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9
⋮	⋮	⋮	⋮

Tabelle 2: Amortisierte Kosten, tatsächliche Kosten, sowie die Entwicklung des Kredits für amortisierte Analyse über die Buchungsmethode.

Wir müssen noch zeigen, dass der Kredit niemals negativ wird.

Wir führen eine Induktion. Für $i = 1$ gilt offensichtlich, dass der Kredit nicht negativ ist. Betrachte jetzt ein festes $i > 1$ unter der Annahme, dass der Kredit bis zur Operation $i - 1$ stets nicht negativ geswesen ist. Es ergeben sich die folgenden zwei Fälle:

Fall 1: i ist keine Zweierpotenz. Die i–te Operation hat also Kosten $c_i = 1$. Da $\hat{c}_i = 3$ folgt, dass der Kredit auch weiterhin nicht negativ bleibt.

Fall 2: i ist eine Potenz von 2, also $i = 2^k$ für $k \geq 1$. Die Kosten für die Operation sind $c_i = i$. Da $2^k = 2 \cdot 2^{k-1}$ gilt, gibt es genau $2^{k-1} - 1$ Operationen zwischen der letzten Zweierpotenz und der aktuell betrachteten. Wir haben für jede dieser Operationen j mit $2^{k-1} < j < 2^k$ einen Kreditüberschuss von 2 angesammelt. Zusammen mit der aktuellen Operation i ergibt das einen Kredit von

$$\text{Kredit} = (2^{k-1} - 1) \cdot 2 + \hat{c}_i = (2^{k-1} - 1) \cdot 2 + 3 = 2^k + 1$$

Wir haben also einen Kredit von $2^k + 1$ auf dem Konto, und können damit die i–te Operation bezahlen, ohne dass der Kredit negativ wird.

Insgesamt folgt für alle Operationen

$$\text{Amortisierte Kosten} - \text{Eigentliche Kosten} \geq 0$$

Da die amortisierten Kosten pro Operation auf $O(1)$ veranschlagt sind, und der Kredit niemals negativ wird, haben n Operationen Kosten von $O(n)$. Wie oben gilt somit insgesamt also $T^{\text{amort}}(n) \in \Theta(n)$

(b) Sei A das Bitfeld, das den Zählerstand repräsentiert, wobei das niederwertigste Bit an Position 0 sei. Zusätzlich zu A sei $max[A]$ ein Zeiger, der auf die höchstwertige 1 in A zeigt. Ist beispielsweise $A = [0, 0, 1, 0, 0, 1]$ so wäre $max[A] = 5$. Wir initialisieren A mit lauter Nullen und setzen $max[A]$ auf -1, da es anfangs keine 1 in A gibt.

Der Wert von $max[A]$ wird bei der Ausführung einer INCREMENT–Operation derart aktualisiert, dass bei einer RESET–Operation möglichst wenige Bits zurückgesetzt werden müssen. Dadurch können wir die RESET–Operationen durch angesammelten Kredit von vorangegangenen INCREMENT–Operationen bezahlen.

Algorithmus 4 : INCREMENT(A)

Eingabe : Bitfeld A
Seiteneffekte : Inkrementieren des Bitfelds um 1 und Anpassen von $max[A]$

```
1  i ← 0
2  solange i < |A| und A[i] = 1 tue
3  |  A[i] ← 0
4  |  i ← i + 1
5  Wenn i < |A|
6  |  A[i] ← 1
7  |  Wenn i > max[A]
8  |  |  max[A] ← i
9  sonst
10 |  max[A] ← −1
```

Algorithmus 5 : RESET(A)

Eingabe : Bitfeld A
Seiteneffekte : Zurücksetzen von A und $max[A]$

```
1  Für i ← 0 ... max[A]
2  |  A[i] ← 0
3  max[A] ← −1
```

Die Kosten für das Kippen eines Bits seien 1. Außerdem koste das Aktualisieren von $max[A]$ ebenfalls 1.

Wir zeigen zunächst mit Hilfe der Buchungsmethode, dass INCREMENT einen amortisierten Aufwand von $O(1)$ hat. Wir verbuchen für jeden Bitwechsel von 0 auf 1 Kosten von 2. Den überschüssigen Kredit von 1 assoziieren wir mit dem jeweiligen Bit. Ein Bitwechsel 1 auf 0 kann nur stattfinden, wenn das gleiche Bit zuvor auf 1 gesetzt wurde. Wir bezahlen also den Wechsel 1 auf 0 mit dem gespeicherten Kredit für das Bit, und erhalten keine neuen Kosten. Da $max[A]$ nur konstant oft ausgeführt wird, ergeben sich somit amortisierte Kosten $O(1)$ für INCREMENT.

Betrachten wir nun die Operation RESET. Wir möchten das Zurücksetzen der Bits $A[0, \ldots, max[A]]$ „kostenlos“ durchführen. Wir verbuchen dafür bei INCREMENT für das Aktualisieren von $max[A]$

Kosten von 2[1]. Den überschüssigen Kredit von 1 ordnen wir dem jeweiligen Bit zu, auf das $max[A]$ nach dem Update zeigt. Da RESET nur Bits bis $max[A]$ zurücksetzt, und jedes dieser Bits zuvor mindestens einmal das höchstwertige 1–Bit in A gewesen sein muss (der Zähler springt nicht), hat jedes dieser Bits mindestens einen Kredit von 1 zugewiesen. Wir können also alle Bits bis $max[A]$ kostenlos auf 0 setzen, indem wir den Kredit verbrauchen. Es ergeben sich also keine Kosten, bis auf das Neusetzen von $max[A]$. Die amortisierten Kosten von RESET sind somit auch in $O(1)$.

Insgesamt erhalten wir also, dass Kosten von 4 für INCREMENT und Kosten 1 für RESET ausreichen und der Kredit niemals negativ wird. Eine Folge von n Operationen vom Typ INCREMENT und RESET hat also amortisierte Laufzeit von $O(n)$.

Problem 4: UNION FIND 2pt

(a) Im folgenden bezeichne $Vor[i]$ die (negierte) Höhe des Baumes i. Es wird nun der kleinere Baum stets an den größeren Baum gehängt. Sind die Bäume gleichgroß wächst die Höhe des entstehenden Baumes dabei um Eins.

Algorithmus 6 : H-UNION(i, j)

Wenn $|Vor[i]| < |Vor[j]|$
  $Vor[i] \leftarrow j$
sonst, wenn $|Vor[j]| < |Vor[i]|$
  $Vor[j] \leftarrow i$
sonst
  $Vor[i] \leftarrow j$ und $Vor[j] \leftarrow Vor[j] - 1$

(b) Ja, auch bei h-balancing UNION gilt $h(T) \leq \log_2 |T|$. Beweis durch Induktion über die Anzahl der H-UNION-Operationen:

Solange keine H-UNION-Operation ausgeführt wurde, habe alle Bäume Höhe 0 und einen Knoten, d.h. es gilt für alle T:

$$h(T) = 0 \leq \log_2 1 = 0.$$

Betrachte die n-te H-UNION-Operation, etwa H-UNION(i, j), wobei T_i ein Baum mit der Wurzel i und T_j Baum mit der Wurzel j vor Ausführung der Operation H-UNION(i, j) sei. O.B.d.A. sei $h(T_j) \geq h(T_i)$. Dann wird T_i durch H-UNION(i, j) an die Wurzel j von T_J angehängt und es entsteht der Baum $T_{\text{H-UNION}}(i, j)$ mit

$$h(T_{\text{H-UNION}}(i,j)) = \begin{cases} h(t_j), & \text{wenn } h(T_j) > h(T_i) \\ h(t_j) + 1, & \text{wenn } h(T_j) = h(T_i) \end{cases}.$$

- Falls $h(T_j) > h(T_i)$, dann ist

$$h(T_{\text{H-UNION}}(i,j)) = h(T_j) \overset{IV}{\leq} \log_2(|T_j|) \leq \log_2(|T_{\text{H-UNION}}(i,j)|)$$

- Falls $h(T_j) = h(T_i)$: O.B.d.A. sei $|T_j| \geq |T_i|$, dann ist

$$h(T_{\text{H-UNION}}(i,j)) = h(T_j) + 1 = h(T_i) + 1 \overset{IV}{\leq} \log_2(|T_i|) + 1 = \log_2(2|T_i|) \leq \log_2(|T_{\text{H-UNION}}(i,j)|)$$

□

[1] Offensichtlich hat das keinen Einfluss auf die amortisierten Kosten von INCREMENT.