Diskrete und Geometrische Algorithmen

1. Übung am 12.10.2020

Richard Weiss Florian Schager Christian Sallinger Fabian Zehetgruber
Paul Winkler Christian Göth

Aufgabe 1. Überlegen Sie sich einen Pseudocode für die folgenden Algorithmen und bestimmen Sie die Anzahl der notwendigen Schritte, die (in Ihrem Pseudocode) nötig sind, um eine n-elementige Menge zu sortieren. Wenden Sie die Algorithmen auf den Datensatz $6, 77, 45, 103, 4, 17$ an.

(a) Selection Sort: der Algorithmus sucht zunächst das kleinste Element und bringt es an die erste Position. Anschließend sucht er das zweitkleinste Element und bringtes an die zweite Position, usw.

(b) Bubble-Sort: Der Algorithmus vergleicht der Reihe nach je zwei benachbarte Zahlen und vertauscht diese, falls sie nicht in der richtigen Reihenfolge angeordnet sind. Dieses Verfahren wird so lange wiederholt, bis alle Zahlen der Eingabe sortiert sind.

Lösung.

(a)

```
1 : Prozedur SELECTION-SORT(A)
2 :   n := A.Länge
3 :   Für i := 1, ..., n − 1
4 :     i_min := i
5 :     Für j := i + 1, ..., n
6 :       Wenn A[j] < A[i_min]
7 :         i_min := j
8 :       Ende Wenn
9 :     Ende Für
10 :    A[i_min], A[i] := A[i], A[i_min]
11 :  Ende Für
12 : Ende Prozedur
```

$Z := \{2, \dots, 7, 10\}$ ist die Menge der Zeilen, die beim Algorithmus wesentliche Operationen ausführen. Wir können also voraussetzen dass für $z \in Z$ eine Konstante c_z existiert, welche die Zeit angibt, die zur Ausführung von der z-ten Zeile benötigt wird. Sei A das Eingabedatenfeld und n die Länge von A. Dann belaufen sich die Kosten der Anwendung von Selection-Sort auf das Datenfeld A auf

$$T(A) = c_2 + \sum_{i=1}^{n-1} \left(c_3 + c_4 + \sum_{j=i+1}^{n} (c_5 + c_6 + c_7 \cdot t_{ij}) + c_{10} \right),$$

wobei t_{ij} mit 1 oder 0 angibt, ob die 7-te Zeile ausgeführt wird oder nicht. Die t_{ij} hängen offensichtlich von A ab. Um die Anzahl der Schritte zu bekommen, setzen wir für alle $z \in Z$ einfach $c_z := 1$.

$$S(A) = 1 + \sum_{i=1}^{n-1} \left(3 + \sum_{j=i+1}^{n} (2 + t_{ij}) \right)$$

Im worst-case gilt die Bedingung in der 6-ten Zeile immer und

$$\forall\, i = 1, \ldots, n-1 : \forall\, j = i+1, \ldots, n : t_{ij} = 1.$$

Wir setzen dies in $S(A)$ ein und erhalten

$$\begin{aligned} S_{\mathrm{s}}(A) = 1 + \sum_{i=1}^{n-1} \left(3 + \sum_{j=i+1}^{n} 3 \right) &= 1 + \sum_{i=1}^{n-1} (3 + 3(n-i)) = 1 + \sum_{i=1}^{n-1} 3 + \sum_{i=1}^{n-1} 3n - \sum_{i=1}^{n-1} 3i \\ &= 1 + 3(n-1) + 3n(n-1) - 3\frac{(n-1)n}{2} = \frac{3n^2}{2} + \frac{3n}{2} - 2 = \mathcal{O}\big(n^2\big). \end{aligned}$$

Im best-case ist die Liste aufsteigend sortiert. Die Bedingung in der 6-ten Zeile hält also nie und

$$\forall\, i = 1, \ldots, n-1 : \forall\, j = i+1, \ldots, n : t_{ij} = 0.$$

Wir setzen dies in $S(A)$ ein und erhalten

$$\begin{aligned} S_{\mathrm{b}}(A) = 1 + \sum_{i=1}^{n-1} \left(3 + \sum_{j=i+1}^{n} 2 \right) &= 1 + \sum_{i=1}^{n-1} (3 + 2(n-i)) = 1 + \sum_{i=1}^{n-1} 3 + \sum_{i=1}^{n-1} 2n - \sum_{i=1}^{n-1} 2i \\ &= 1 + 3(n-1) + 2n(n-1) - 2\frac{(n-1)n}{2} = n^2 + 2n - 2 = \mathcal{O}\big(n^2\big). \end{aligned}$$

(b)

```
1 : Prozedur BUBBLE-SORT(A)
2 :   n := A.Länge
3 :   Für i := 1, ..., n
4 :     Für j := 1, ..., n − i
5 :       Wenn A[j + 1] < A[j]
6 :         A[j], A[j + 1] := A[j + 1], A[j]
7 :       Ende Wenn
8 :     Ende Für
9 :   Ende Für
10 : Ende Prozedur
```

$$\implies T(A) = c_2 + c_3 + \sum_{i=1}^{n} \left(c_3 + c_4 + \sum_{j=1}^{n-i} (c_4 + c_5 + c_6 \cdot t_{ij}) \right)$$

$$\implies S(A) = 2 + \sum_{i=1}^{n} \left(2 + \sum_{j=1}^{n-i} (2 + t_{ij}) \right)$$

$$\implies S_{\mathrm{s}}(A) = 2 + \sum_{i=1}^{n} \left(2 + \sum_{j=1}^{n-i} (2+1) \right) = 2 + \sum_{i=1}^{n} (2 + 3(n-i)) = 2 + \sum_{i=1}^{n} 2 + \sum_{i=1}^{n} 3n - \sum_{i=1}^{n} 3i$$
$$= 2 + 2n + 3n^2 - \frac{3n(n+1)}{2} = \frac{3n^2}{2} + \frac{n}{2} + 2 = \mathcal{O}\big(n^2\big)$$

$$\implies S_{\mathrm{b}}(A) = 2 + \sum_{i=1}^{n} \left(2 + \sum_{j=1}^{n-i} (2+0) \right) = 2 + \sum_{i=1}^{n} (2 + 2(n-i)) = 2 + \sum_{i=1}^{n} 2 + \sum_{i=1}^{n} 2n - \sum_{i=1}^{n} 2i$$
$$= 2 + 2n + 2n^2 - n(n+1) = n^2 + n + 2 = \mathcal{O}\big(n^2\big)$$

Aufgabe 2. Sequentielle Suche: Gegeben sei ein n-elementiger Datenfeld $A[1, \ldots, n]$ und ein Wert x. Überlegen Sie sich einen Pseudocode, der x durch sukzessive Vergleiche mit den Elementen $A[1], A[2], \ldots$ sucht und einen Wert $j \in \{1, \ldots n\}$ ausgibt, falls $x = A[j]$, oder NIL ausgibt, falls x nicht in der Liste A enthalten ist.

(a) Beweisen Sie, dass Ihr Algorithmus ein korrektes Ergebnis liefert (etwa mit Zuhilfenahme einer Schleifeninvariante).

(b) Machen Sie für diesen Algorithmus eine best-case-Analyse, eine worst-case-Analyse und eine average-case-Analyse (für die average-case-Analyse soll das Modell der Zufallspermutationen verwendet werden, d.h. alle Permutationen von $\{1, \ldots, n\}$ sind gleich wahrscheinlich. Weiters soll angenommen werden, dass x im Datensatz enthalten ist).

Lösung.

```
1 : Prozedur SEQUENTIELLE SUCHE(A, x)
2 :   n := A.Länge
3 :   i := NIL
4 :   Für j := 1, ..., n
5 :     Wenn x = A[j]
6 :       i := j
7 :       Abbruch
8 :     Ende Wenn
9 :   Ende Für
10 : Ende Prozedur
```

(a) Bezeichne $\mathtt{n}$ die Länge eines Datenfelds $\mathtt{A}$, $\mathtt{i}$ den (potentiellen) kleinsten Index von einer Variable $\mathtt{x}$ (vielleicht) aus $\mathtt{A}$, und $\mathtt{j}$ die laufende Variable der Schleife der 4-ten Zeile. Unsere Schleifeninvariante lautet wie folgt.

$$I(\mathtt{A}, \mathtt{x}, \mathtt{i}, \mathtt{j}) : \mathtt{j} \leq \mathtt{n} \wedge ((\mathtt{i} = \text{NIL} \wedge \forall\, \mathtt{k} < \mathtt{j} : \mathtt{x} \neq \mathtt{A}[\mathtt{k}]) \vee \mathtt{x} = \mathtt{A}[\mathtt{i}])$$

1. Zu Beginn des ersten Durchlaufs der Schleife ist $\mathtt{i} = \text{NIL}$ und $\mathtt{j} = 1$. $I(\mathtt{A}, \mathtt{x}, \mathtt{i}, \mathtt{j})$ ist hier also wahr.
2. Es gelte $I(\mathtt{A}, \mathtt{x}, \mathtt{i}, \mathtt{j})$. Falls $\mathtt{x} = \mathtt{A}[\mathtt{i}]$, dann hat der Algorithmus bereits terminiert. Weil die Schleife in der 4-ten Zeile nach oben zählt, ist $\mathtt{j}' = \mathtt{j} + 1$. Falls $\mathtt{j} = \mathtt{n}$, dann terminiert der Algorithmus, weil dann $\mathtt{j}' > \mathtt{n}$. Ansonsten gelten $\mathtt{j}' \leq \mathtt{n}$, $\mathtt{i} = \text{NIL}$, und $\forall\, \mathtt{k} < \mathtt{j} : \mathtt{x} \neq \mathtt{A}[\mathtt{k}]$, und die Schleife in der 4-ten Zeile wird ein weiteres mal instanziert. Offensichtlich gilt $\mathtt{A}' = \mathtt{A}$ sowie $\mathtt{x}' = \mathtt{x}$ weil sich $\mathtt{A}$ und $\mathtt{x}$ nie ändern. Für den Wert von $\mathtt{i}'$ machen wir eine Fallunterscheidungs.
   I. Fall ($\mathtt{x}' = \mathtt{A}'[\mathtt{j}']$): Dann wird die 6-te Zeile ausgeführt und $\mathtt{i}' = \mathtt{j}'$. Anschließend terminiert der Algorithmus.
   II. Fall ($\mathtt{x}' \neq \mathtt{A}'[\mathtt{j}']$): Dann wird die 6-te Zeile nicht ausgeführt und es bleibt $\mathtt{i}' = \mathtt{i}$. Wegen $I(\mathtt{A}, \mathtt{x}, \mathtt{i}, \mathtt{j})$ gilt $\forall\, \mathtt{k} < \mathtt{j} : \mathtt{x} \neq \mathtt{A}[\mathtt{k}]$ und wegen der Fallunterscheidungsbedingung gilt $\mathtt{x}' \neq \mathtt{A}'[\mathtt{j}']$. Insgesamt folgt daher $\forall\, \mathtt{k} < \mathtt{j}' : \mathtt{x} \neq \mathtt{A}[\mathtt{k}]$.

   Also gilt auch $I(\mathtt{A}', \mathtt{x}', \mathtt{i}', \mathtt{j}')$.
3. Offensichtlich ist $\mathtt{i}$ für $\mathtt{j} = \mathtt{n}$ das gewünschte Ergebnis.

(b) Sei A ein Datenfeld der Länge n und $x \in A$.

1. Worst-Case:
   x taucht erst im letzten Eintrag von A auf, d.h. $A[n] = x$.

$$\implies T_{\mathrm{s}}(A, x) = c_2 + c_3 + (c_4 + c_5)n + c_6 = \mathcal{O}(n)$$

2. Average-Case:
   Nachdem im Skriptum vorausgesetzt wird (ohne darauf hinzuweisen), dass die Einträge von A verschieden sind und damit o.B.d.A $\{1, \ldots, n\}$, werden wir das hier auch tun. Ansonsten müsste man die Wahrscheinlichkeit berechnen, dass $i = 1, \ldots, n$ der kleinste Index ist, sodass $A[i] = x$.

$$W(A[i] = x \wedge \forall\, j = 1, \ldots, i-1 : A[j] \neq x \mid \exists\, j = 1, \ldots, n : A[j] = x)$$

   Für $\pi \in S_n$ und $\pi(i) = x$ mit $i = 1, \ldots, n$, muss die Wenn-Bedingung in der 5-ten Zeile genau i-Mal überprüft werden. Weil alle Permutationen aus S_n gleich wahrscheinlich sind, erhalten wir für die durchschnittliche Anzahl der Wenn-Bedingungs-Überprüfungen

$$\overline{t} = \sum_{i=1}^{n} \frac{i}{n} = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

$$\implies T_{\mathrm{d}}(A, x) = c_2 + c_3 + (c_4 + c_5)\frac{n+1}{2} + c_6 = \mathcal{O}(n)$$

3. Best-Case:
   x taucht erst im ersten Eintrag von A auf, d.h. $A[1] = x$.

$$T_{\mathrm{b}}(A, x) = c_2 + c_3 + c_4 + c_5 + c_6 = \mathcal{O}(1)$$

Lösung.

1 : **Prozedur** SEQUENTIELLE SUCHE(A, x)
2 : $j := 1$
3 : **Solange** $j \leq A.Länge$ *und* $A[j] \neq x$
4 : $j := j + 1$
5 : **Ende Solange**
6 : **Wenn** $j = A.Länge + 1$
7 : $j := NIL$
8 : **Ende Wenn**
9 : **Ende Prozedur**

Wir wollen nun einen Korrektheitsbeweis führen. Dafür bezeichnen wir mit k den Schleifendurchlauf und definieren die Schleifeninvariante $I(j) : \forall k < j : A[k] \neq x$. Nun gilt

(1) Zu Beginn der Schleife ist $j = 1$ also gibt es gar kein $k \in \mathbb{N}$ das $k < j$ erfüllt, damit ist $I(j)$ erfüllt.

(2) Sei nun $I(j)$ erfüllt. Da die Schleife durchlaufen wird gilt $A[j] \neq x$. Insgesamt zu Beginn des $n+1$-ten Schleifendurchlaufs also aus der Voraussetzung und dem Argument eben $\forall k < j + 1 : A[k] \neq x$.

(3) Bei Beendigung der Schleife ist $j = A.Länge + 1$ oder $A[j] = x$. Im ersten Fall gilt $\forall k < A.Länge + 1 : A[k] \neq x$ also ist x nicht in A enthalten, der Algorithmus endet mit NIL. Sonst gibt der Algorithmus klarerweise ein j aus, für das $A[j] = x$ gilt.

Nun wollen wir noch den Aufwand des Algorithmus analysieren. Sei dazu t die Anzahl der Ausführungen des Schleifenkopfes und s die Anzahl der Ausführungen von Zeile 7. Allgemein gilt nun

$$T(A, x) = (c_3 + c_5)t + c_4(t - 1) + c_7 s + c_2 + c_6 + c_8$$

für den Aufwand.

(1) Im besten Fall ist $A[1] = x$, folglich $s = 0$ und $t = 1$. Damit ergibt sich $T(A, x) = c_2 + c_3 + c_5 + c_6 + c_8 = \mathcal{O}(1)$.

(2) Im schlechtesten Fall ist x nicht in der Liste A enthalten und es ist $s = 1$ und $t = n + 1$. Es ergibt sich ein Aufwand von $T(A, x) = (c_3 + c_5)(n + 1) + nc_4 + c_2 + c_6 + c_7 + c_8 = \mathcal{O}(n)$.

(3) Zur Analyse des durchschnittlichen Falls nehmen wir an, dass x in der Liste A enthalten ist und an jeder Stelle mit der gleichen Wahrscheinlichkeit, nämlich $\frac{1}{n}$ vorzufinden ist. Mit Sicherheit ist also $s = 0$ und wir berechnen

$$\bar{t} = \sum_{i=1}^{n} \frac{i}{n} = \frac{n+1}{2}.$$

und erhalten mit $t = \bar{t}$ den Aufwand $T(A, x) = (c_3 + c_5)\frac{n+1}{2} + c_4\frac{n-1}{2} + c_2 + c_6 + c_8 = \mathcal{O}(n)$.

Aufgabe 3.

(a) Binäre Suche: Gegeben sei ein (aufsteigend) sortiertes Datenfeld $A[1, \ldots, n]$ und ein Wert x. Das sogenannte Suchproblem, also einen Index j mit $x = A[j]$ auszugeben, falls x in A enthalten ist, und einen speziellen Wert NIL auszugeben, falls x nicht in A vorkommt, kann hier mittels Divide-and-Conquer gelöst werden. Man vergleicht x mit dem mittleren Element des Datenfelds und ist nach diesem Vergleichen entweder fündig geworden oder braucht nur noch das halbe Datenfeld mit der gleichen Prozedur zu durchsuchen. Schreiben Sie ein Programm in Pseudocode für die binäre Suche. Begründen Sie, warum die Laufzeit der binären Suche im schlechtesten Fall $\mathcal{O}(\log n)$ ist.

(b) Beim Algorithmus Einfügesortieren wird die sequentielle Suche verwendet, um das bereits sortierte Teilfeld $A[1, \ldots, n]$ (rückwärts) zu durchsuchen. Kann stattdessen die binäre Suche verwendet werden, um die worst-case-Laufzeit von Insertion Sort auf $\mathcal{O}(\log n)$ zu verbessern?

Lösung.

(a)

```
1 : Prozedur DIVIDE-AND-CONQUER-SUCHE(A, x)
2 :   a := 1
3 :   b := A.Länge
4 :   j := ⌊(b − a)/2⌋
5 :   Solange b − a > 0 und A[a + j] ≠ x
6 :     Wenn A[a + j] < x
7 :       a := a + j + 1
8 :     Sonst
9 :       b := a + j − 1
10 :    Ende Wenn
11 :    j := ⌊(b − a)/2⌋
12 :  Ende Solange
13 :  Wenn j = 0
14 :    j := NIL
15 :  Sonst
16 :    j := j + a
17 :  Ende Wenn
18 : Ende Prozedur
```

In jedem Schritt des Solange-Block wird $(b - a)$ mindestens halbiert (oder die Schleife bricht ab):
Fall 1: $A[a + j] < x$

$$b' - a' = b - a - j - 1 = b - a - \lfloor (b - a)/2 \rfloor - 1 \leq \frac{b - a}{2}$$

Fall 2: $A[a + j] > x$

$$b' - a' = a + j - 1 - a = j - 1 \leq \frac{b - a}{2}$$

Fall 3: $A[a + j] = x$
Die Solange-Bedingung ist nicht mehr erfüllt und wir haben den letzten Schleifendurchlauf erreicht.
Insgesamt ist nach $\lceil \log_2(n) \rceil$ in jedem Fall die Solange-Bedingung verletzt und somit wird der Solange-Block maximal $\lceil \log_2(n) \rceil$ oft ausgeführt, was uns insgesamt auf einen Aufwand von $\mathcal{O}(n)$ führt.

```
1 : Prozedur DIVIDE-AND-CONQUER-SUCHE(A, x)
2 :   a := 0
3 :   b := A.Länge + 1
4 :   j := ⌊(b − a)/2⌋
5 :   Solange b − a > 0 und A[a + j] ≠ x
6 :     Wenn A[a + j] < x
7 :       a := a + j + 1
8 :     Sonst
9 :       b := a + j
10 :    Ende Wenn
11 :    j := ⌊(b − a)/2⌋
12 :  Ende Solange
13 :  Wenn j = 0
14 :    j := NIL
15 :  Ende Wenn
16 : Ende Prozedur
```

Es ist nicht ganz klar, was in der Angabe mit begründen gemeint ist. Ein sauberer Beweis oder die Bemerkung, dass die Länge des Datenfeldes sich stets halbiert, die Schleife also sicher nicht öfter als $\lceil \log_2(n) \rceil$ Mal ausgeführt wird?

(b)

Algorithmus 2 Einfügesortieren

```
1:  Prozedur EINFÜGESORTIEREN(A)
2:    Für j := 2, ..., A.Länge
3:      x := A[j]
4:      i := j − 1
5:      Solange i ≥ 1 und A[i] > x
6:        A[i + 1] := A[i]
7:        i := i − 1
8:      Ende Solange
9:      A[i + 1] := x
10:   Ende Für
11: Ende Prozedur
```

Beim Algorithmus Einfügesortieren wird die äußere Schleife stets $(n-1)$-Mal durchlaufen. Ersetzt man die innere Schleife durch ein ähnliches Verfahren wie die Prozedur Divide-and-Conquer-Suche, so wird im schlechtesten Fall die innere Schleife im j-ten Durchlauf der äußeren Schleife $\lceil \log_2 j \rceil$-Mal durchlaufen. Es ergibt sich so folgender Aufwand.

$$\sum_{j=2}^{n} \lceil \log_2 j \rceil \geq \sum_{j=2}^{n} \log_2 j \geq \int_1^n \log_2 x \, \mathrm{d}x = \frac{1}{\log 2} \int_1^n \log x \, \mathrm{d}x$$
$$= \frac{1}{\log 2}((n \log n - n) - (1 \log 1 - 1)) = \frac{1}{\log 2}(n(\log n - 1) + 1) = \mathcal{O}(n \log n) > \mathcal{O}(\log n)$$

Die worst-case-Laufzeit von Insertion Sort lässt sich daher nicht einmal durch anwenden der binären Suche auf $\mathcal{O}(\log n)$ verbessern.

Aufgabe 4. Zeigen Sie, dass die harmonischen Zahlen $H_n = \sum_{k=1}^{n} \frac{1}{k}$ die Abschätzung $H_n = \mathcal{O}(\log n)$ gilt, indem Sie

1. die Summe durch $N = \lfloor \log_2 n \rfloor$ Blöcke der Gestalt $\sum_{j=0}^{2^i-1} \frac{1}{2^i+j}$ (wobei $i = 1, \ldots, N$) abschätzen und anhand dieser Aufteilung $\sum_{k=1}^{n} \frac{1}{k} \leq 1 + \log_2 n$ verifizieren.
2. das Cauchy'sche Integralkriterium verwenden.

Lösung.

1.

$$\{1, \ldots, n\} = \bigcup_{i=0}^{\lfloor \log_2 n \rfloor - 1} \left\{2^i, \ldots, 2^{i+1} - 1\right\} + \left\{2^{\lfloor \log_2 n \rfloor}, \ldots, 2^{\log_2 n}\right\}$$
$$= \bigcup_{i=0}^{\lfloor \log_2 n \rfloor - 1} \left\{2^i + 0, \ldots, 2^i + (2^i - 1)\right\} + \left\{2^{\lfloor \log_2 n \rfloor}, \ldots, n\right\} \subseteq \bigcup_{i=0}^{\lfloor \log_2 n \rfloor} \left\{2^i, \ldots, 2^i + (2^i - 1)\right\}$$

$$\implies H_n = \sum_{k=1}^{n} \frac{1}{k} = \sum_{i=0}^{\lfloor \log_2 n \rfloor - 1} \left(\sum_{j=0}^{2^i-1} \frac{1}{2^i + j} \right) + \sum_{k=2^{\lfloor \log_2 n \rfloor}}^{n} \frac{1}{k} \leq \sum_{i=0}^{\lfloor \log_2 n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i + j} \leq \sum_{i=0}^{\lfloor \log_2 n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i}$$
$$= \sum_{i=0}^{\lfloor \log_2 n \rfloor} \frac{2^i}{2^i} = 1 + \lfloor \log_2 n \rfloor = 1 + \left\lfloor \frac{\log n}{\log 2} \right\rfloor \leq \mathcal{O}(\log n)$$

2. Definiere $f(x) = \frac{1}{x}$. Betrachte die Zerlegung $\mathcal{Z}_n = \{1, 2, 3, \ldots, n\}$ des Intervalls $[1, n]$. Da f monoton fallend ist, ist $\sum_{k=2}^{n} \frac{1}{k}$ die zugehörige Untersumme von f zu $\mathcal{Z}_n$ und es gilt

$$H_n = 1 + \sum_{k=2}^{n} \frac{1}{k} \leq 1 + \int_1^n \frac{1}{x} \, \mathrm{d}x = 1 + \log n - \log 1 = 1 + \log(n) = \mathcal{O}(\log n).$$

Aufgabe 5. Das Horner-Schema dient zur Auswertung von Polynomen. Die Grundidee dahinter ist die Umformung $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \ldots))$.

(i) Wiederholen Sie die Funktionsweise des Horner-Schemas und schreiben sie einen Pseudocode für die Auswertung von Polynomen mittels Horner-Schema.

(ii) Schreiben Sie einen Pseudocode für die direkte Auswerung von Polynomen (Einsetzen).

(iii) Vergleichen Sie die Schrittzahlen beider Codes.

Lösung.

(i) Wir nehmen an, dass Polynome als Arrays übergeben werden mit $p[0] = a_0, \ldots, p[n] = a_n$.

```
1 : Prozedur HORNER-SCHEMA(p, x)
2 :     n := grad p
3 :     y := p[n] · x
4 :     Für i = n − 1, . . . , 1
5 :         y := y + p[i]
6 :         y := x · y
7 :     Ende Für
8 :     y := y + p[0]
9 : Ende Prozedur
```

```
1 : Prozedur HORNER-SCHEMA(p, x)
2 :     n := grad p
3 :     y := p[n]
4 :     Für i = n − 1, . . . , 1
5 :         y := p[i] + x · y
6 :     Ende Für
7 : Ende Prozedur
```

(ii)

```
1 : Prozedur POLYNOM AUSWERTEN(p, x)
2 :     n := grad p
3 :     y := 0
4 :     z := 1
5 :     Für i = 0, . . . , n
6 :         y := y + p[i] · z
7 :         z := z · x
8 :     Ende Für
9 : Ende Prozedur
```

(iii) Wir zählen als Schritte nur reine Rechenschritte und nicht jede ausgeführte Zeile Code: Additionen / Multiplikationen Horner-Schema (in Abhängigkeit vom Grad des Polynoms):

$$A_{\text{Horner-Schema}}(n) = n, \quad M_{\text{Horner-Schema}}(n) = n$$

Additionen / Multiplikationen Polynom-Auswerten (in Abhängigkeit vom Grad des Polynoms):

$$A_{\text{gewöhnlich}}(n) = n + 1, \quad M_{\text{gewöhnlich}}(n) = 2(n + 1).$$

Da Additionen wesentlich billiger sind als Multiplikationen fallen diese nicht so ins Gewicht. Das Horner-Schema spart also fast die Hälfte der Zeit im Vergleich zur gewöhnlichen Auswertung des Polynoms.

Aufgabe 6.

(a) Zeigen Sie mittels vollständiger Induktion, dass ein Algorithmus, dessen Laufzeit $T(n)$ (wobei $n = 2^k$, $k \in \mathbb{Z}^+$) der Rekursion

$$T(n) = \begin{cases} 1 & \text{für } n = 2 \\ 2T(\frac{n}{2}) + 1 & \text{für } n = 2^k, k > 1 \end{cases}$$

genügt, $T(n) = n - 1$ erfüllt.

(b) Zeigen Sie mittels vollständiger Induktion, dass ein Algorithmus, dessen Laufzeit $T(n)$ (wobei $n = 2^k$, $k \in \mathbb{Z}^+$) der Rekursion

$$T(n) = \begin{cases} 2 & \text{für } n = 2 \\ 2T(\frac{n}{2}) + n & \text{für } n = 2^k, k > 1 \end{cases}$$

genügt, $T(n) = n \log_2(n)$ erfüllt.

Lösung.

(a) Wir führen Induktion nach k (und schreiben immer 2^k statt n).
IA($k = 1$):

$$\implies T(2^1) = T(2) = 1 = 2 - 1 = 2^1 - 1$$

IV: $T(2^{k-1}) = 2^{k-1} - 1$
IS($k - 1 \mapsto k$):

$$\implies T(2^k) = 2T(2^{k-1}) + 1 \overset{\text{IV}}{=} 2(2^{k-1} - 1) + 1 = 2^k - 2 + 1 = 2^k - 1$$

(b) Wir führen Induktion nach k (und schreiben immer 2^k statt n).
IA($k = 1$):

$$\implies T(2^1) = T(2) = 2 = 2^1 \log_2 2^1$$

IV: $T(2^{k-1}) = 2^{k-1} \log_2 2^{(k-1)}$
IS($k - 1 \mapsto k$):

$$\implies T(2^k) = 2T(2^{k-1}) + 2^k \overset{\text{IV}}{=} 2(2^{k-1} \log_2 2^{k-1}) + 2^k = 2^k(k-1) + 2^k = 2^k k - 2^k + 2^k = 2^k \log_2(2^k)$$