186.866 Algorithmen und Datenstrukturen VU

Lösung für das Aufgabenblatt Algorithmenanalyse

PDF erstellt am: 18. Mai 2022

1 Einleitende Worte

Mit dieser Beispielsammlung können Sie das Berechnen von Laufzeiten und Rückgabewerten von Algorithmen üben und Ihre Lösungen mit ausführlich beschriebenen Musterlösungen vergleichen.

1.1 Differenzengleichungen

Um asymptotische Laufzeiten zu berechnen, kann man auch formal mathematisch die Übungen lösen. Eine Hilfestellung dabei sind Differenzengleichungen, kennengelernt in Algebra und Diskrete Mathematik, um die Entwicklung von Folgen zu beschreiben:

Addition $\longrightarrow$ Lineares Wachstum

$$y_{n+1} = y_n \pm c \quad \longmapsto \quad y_0 \pm k \cdot c = y_n$$

Multiplikation $\longrightarrow$ Exponentielles Wachstum

$$y_{n+1} = y_n \cdot c \quad \longmapsto \quad y_0 \cdot c^k = y_n$$

Exponenzierung

$$y_{n+1} = y_n^c \quad \longmapsto \quad (y_0)^{(c^k)} = y_n$$

Wichtig dabei zu beachten ist: $(y_0)^{(c^k)} \neq \left(y_0{}^c\right)^k = y_0^{c \cdot k}$.

1.2 Schleifen: Unterschiedliche Syntax, gleiche Semantik

Es gibt viele Möglichkeiten und Notationsweisen eine Schleife auszudrücken. Die beliebtesten unter ihnen sind *for*- und *while*-Schleifen.

Anbei mehrere semantisch äquivalente Schleifen (nicht alle gültiges Pseudo-Code nach der vereinbarten Notation):

(a)
```
for (x = 0; x < n; x++){
    ...
}
```

(b)
```
for x ← 0 bis (n − 1)
    ...
```

(c)
```
x = 0;
while (x < n){
    ...
    x++;
}
```

(d)
```
x ← 0
while x < n
    ...
    x ← x + 1
```

(e)
```
do {
   ...
   x = x + 1
} while(x < n)
```

(f)
```
do
   ...
   x ← x + 1
while x < n
```

(g)
```
repeat {
   ...
   x = x + 1
} until(x ≥ n)
```

(h)
```
repeat
   ...
   x ← x + 1
until x ≥ n
```

Die rechte Spalte entspricht der in dieser Lehrveranstaltung vereinbarten Notation für Pseudo-Code und die linke Spalte enthält der Syntax aus der Programmiersprache Java (es wurden keine Typen dazugeschrieben).

Über die do-while & repeat-until Schleife: Die *do-while-* oder *repeat-until*-Schleife unterscheidet sich von einer gewöhnlichen for- oder *while*-Schleife nur dadurch, dass der Schleifenkörper immer ausgeführt wird, bevor die Bedingung überprüft wird. Effektiv bedeutet das nur, dass im Falle, dass die Bedingung am Anfang nicht zutrifft, wir trotzdem einmal iterieren - jedoch bleibt alles andere gleich.
Der Unterschied zwischen einer $do - while$ und $repeat - until$-Schleife besteht darin, dass während die $do - while$-Schleife die Schleifenbedingung verlangt (welche erfüllt sein muss), letzteres die Abbruchsbedingung verlangt (welche nicht erfüllt sein darf).

Um die Anzahl der Iterationen all dieser Schleifen zu ermitteln, könnte man einerseits zählen und intuitiv zur Lösung gelangen, andererseits arithmetisch.

Wir gelangen in diesem Fall anhand der Aufstellung einer Differenzengleichung auf arithmetischerweise zur exakten Lösung:

1. Wir bestimmen die Abbruchsbedingung der Schleife. Das ist die Bedingung, ab die die Schleife terminiert.
   In unserem Fall ist das $\neg(x < n) = (x \geq n)$.

2. Wir bestimmen den initialen Wert von x, auch x_0 oder x_{Initial}.
   In unserem Fall: $x_0 = 0$.

3. Wir bestimmen den Typen der Differenzengleichung und den Summanden, Faktor oder Exponenten welcher x manipuliert.
   In unserem Fall handelt es sich um lineares Wachstum, da x jede Iteration um 1 inkrementiert wird.

$$x_0 = 0$$

$$c = 1$$

4. Wir stellen die Gleichung auf, um die unbekannte Anzahl der k Iterationen zu bestimmen, ab die die Abbruchsbedingung erfüllt ist:

$$x_0 + k \cdot c \geq n$$

$$0 + k \cdot 1 \geq n$$

$$k \geq n$$

   Dadurch, dass die Anzahl der Iterationen nur positiv und wachsend sein kann und wir nicht weiter iterieren können wenn einmal die Abbruchsbedingung erfüllt ist, wissen wir, dass die erste Zahl, welche die Ungleichung erfüllt (hier n) unsere Lösung ist.

2 Lineare Laufzeit

Aufgabe 1. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionB(n):
    x ← 1
    for j ← 1 bis 5n
        x ← 4 · x
        z ← j
        x ← ⌊x/2⌋
    for j ← 1 bis 2n
        z ← z − 1
    return z
```

Lösung

Die erste Zuweisung beansprucht eine Laufzeit von $\Theta(1)$.

Danach iterieren wir in der ersten *for*-Schleife so oft wir der Schleifenvariable j einen Wert im Intervall $[1, 5n]$ zuordnen können: also $5n$-Mal.
In diesen $5n\rfloor$ Iterationen vervierfachen und halbieren wir (also verdoppeln wir effektiv) x und weisen z dem aktuellsten Wert von j zu.

Das bedeutet wir haben in der ersten Schleife eine Laufzeit von $\Theta(5n) = \Theta(n)$, weil wir multiplikative Konstanten in der asymptotischen Abschätzung ignorieren können. Danach hat z den selben Wert wie die höchste Zahl des Intervalls von j.
Weiters hat x nach der Ausführung der ersten Schleife den Wert:

$$x := x_{\text{Initial}} \cdot 2^{5n}$$

$$x := 2^{5n}$$

Auf analoger Weise lässt es sich die Laufzeit für die zweite Schleife feststellen. Diese Schleife wird $2n$ mal betreten, somit ebenfalls in $\Theta(n)$. Wir dekrementieren in der zweiten Schleife z bei jeder Iteration um 1, und können uns mit der folgenden Gleichung den Rückgabewert der Funktion berechnen:

$$z = z_{\text{initial}} - 2n \cdot 1$$

$$z = 5n - 2n = 3n$$

Die Gesamtlaufzeit ergibt sich aus der Addition beider Schleifen, wobei wir alle einzelnen Wertzuweisungen außer Acht lassen, da sie $\Theta(1)$ betragen und weder als Faktor noch als Summand asymptotisch einen Einfluss haben.
Daraus folgt die Gesamtlaufzeit $\Theta(5n + 2n) = \Theta(7n) = \Theta(n)$.

Aufgabe 2. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionB(n):
    n ← 2n
    x ← 1
    a ← 1
    for j ← 1 bis n
        x ← x + ax
        z ← j
        a ← −a
    while 1 ≤ x
        z ← z + 1
    return z
```

Lösung

Die ersten 3 Wertzuweisungen haben eine Laufzeit von $\Theta(1)$.

Danach folgt eine *for*-Schleife, welche so oft iteriert wie es Werte (ganze Zahlen) in dem Intervall $[1, n]$ gibt. Das entspricht n Iterationen, also $\Theta(n)$.

Im Körper der *for*-Schleife wird wie folgt manipuliert:

$$x := x + ax = x \cdot (a + 1)$$

Der Wert von a während dieser Operation -1 wenn j gerade ist und 1 sonst:

$$x := x \cdot (a + 1) = x \cdot ((-1)^{j+1} + 1)$$

Die folgende Reihe in der x_0 für den initialen Wert 1 von x steht, zeigt eine konstante Entwicklung unabhängig von der Anzahl der Iterationen $0 \leq i \leq n$ in x_i, ab $i \geq 2$.

$$x_0 := 1$$

$$\begin{aligned}
x_1 &:= x_0 \cdot ((-1)^{1+1} + 1) = x_0 \cdot 2 = 1 \cdot 2 = 2 \\
x_2 &:= x_1 \cdot ((-1)^{2+1} + 1) = x_1 \cdot 0 = 2 \cdot 0 = 0 \\
x_3 &:= x_2 \cdot ((-1)^{3+1} + 1) = x_2 \cdot 2 = 0 \cdot 2 = 0 \\
x_4 &:= x_3 \cdot ((-1)^{4+1} + 1) = x_3 \cdot 0 = 0 \cdot 0 = 0 \\
&\dots
\end{aligned}$$

Das bedeutet der finale Wert von x beträgt nach der Ausführung aller Anweisungen des Programmes 0 falls $n \geq 2$ und sonst 2.

Weiters weisen wir in der Schleife der Variable z den aktuellen Wert von j zu, wodurch der finale Wert von z auch der von j in der letzten Iteration sein muss, also n.

Wir können den finalen Wert von a nach der Ausführung der Schleife mit der folgenden Formel ermitteln:

$$a := a_{\text{Initial}} \cdot (-1)^n = 1 \cdot (-1)^n$$

Anschließend werten wir in der letzten *while*-Schleife die Bedingung $1 \leq x$ zu einem Wahrheitswert aus. Wir wissen aus den vorherigen Absätzen, dass:

$$x := \begin{cases} 0, & \text{falls } n \geq 2 \\ 2, & \text{sonst} \end{cases}$$

Dadurch, dass wir innerhalb der *while*-Schleife mit keiner Instruktion den Wert von x manipulieren, folgt in dem Fall, dass $n < 2$ ist, dass $1 \leq x$ zu wahr ausgewertet und die Schleife nie terminiert (z wird ewig inkrementiert).

Ansonsten wird die *while*-Schleife erst gar nicht betreten und das Programm terminiert sofort nach der *for*-Schleife. Das entspricht einer Laufzeit von $\Theta(1)$, da was bei den ersten endlichen vielen n passiert irrelevant ist.

Es ergibt sich dadurch eine Gesamtlaufzeit von $\Theta(1+1+1+n \cdot (1+1+1)+1) = \Theta(n)$.

Aufgabe 3. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionA(n):
    x ← 50000
    while x > 1
        for j ← 1 bis ⌊n/50⌋
            z ← 2j
        x ← x/5
    return z
```

Lösung

Die erste Instruktion ist eine Wertzuweisung und hat eine Laufzeit von $\Theta(1)$.

Die äußere *while*-Schleife iteriert so oft, bis die Abbruchsbedingung $x \leq 1$ erfüllt ist. Die Variable x (welche wir am Anfang der Funktion initialisiert haben) wird bei jeder Iteration gefünftelt. Deshalb können wir die Anzahl an Iterationen k mit einer Ungleichung folgender Form herleiten:

$$50000 \cdot \left(\frac{1}{5}\right)^k \leq 1$$

$$\left(\frac{1}{5}\right)^k \leq \frac{1}{50000}$$

$$\log\left(\left(\frac{1}{5}\right)^k\right) \leq \log\left(\frac{1}{50000}\right)$$

$$k \cdot \log\left(\frac{1}{5}\right) \leq \log\left(\frac{1}{50000}\right)$$

$$k \cdot \log 5 \geq \log 50000$$

$$k \cdot \log 5 \geq \log 50000$$

$$k \geq \frac{\log 50000}{\log 5}$$

$$k \geq \log_5 50000$$

Daraus folgt, dass wir $\lceil \log_5 50000 \rceil = 7$ Iterationen benötigen, also eine Laufzeit von $\Theta(7) = \Theta(1)$.

In der inneren, verschachtelten for-Schleife iterieren wir $\lfloor \frac{n}{50} \rfloor$-Mal und weisen jedes Mal das doppelte der Schleifenvariable j der Variable z zu. Dabei haben wir eine Laufzeit von $\Theta(\lfloor \frac{n}{50} \rfloor) = \Theta(n)$ und belegen z immer mit dem finalen Wert $2 \cdot \lfloor \frac{n}{50} \rfloor$ (der letzte Wert der Schleifenvariable).
Dadurch, dass wir beim zuweisen eines Wertes auf z auch nicht den alten Zustand von z referenzieren und auch keine Variable aus der äußeren Schleife, haben Iterationen der

äußeren Schleife keine Wirkung auf die innere und z wird mit dem Wert $2 \cdot \lfloor \frac{n}{50} \rfloor$ von der Funktion zurückgegeben.

Die Gesamtlaufzeit beträgt: $\Theta(1 + 6(\cdot n \cdot 1 + 1)) = \Theta(n)$

---

3 Logarithmische Laufzeit

Aufgabe 4. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionLOG(n):
    x ← n
    z ← 0
    while x > 1
        x ← ⌊x/2⌋
        z ← z + 1
    return z
```

Lösung Das ist ein klassisches Beispiel für eine logarithmische Laufzeit. Wenn in einer Schleife ein Wert so lange durch eine Konstante dividiert wird, solange sie größer als 1 ist, spricht das sehr für eine logartihmische Laufzeit. Sehen wir uns an, was bei einer Eingabe $n = 16$ passiert. Die Werte für x sind jene, bevor die Anweisung in der Schleife durchgeführt werden:

1. Iteration: $x = 16, z = 0$
2. Iteration: $x = 8, z = 1$
3. Iteration: $x = 4, z = 2$
4. Iteration: $x = 2, z = 3$
5. Iteration: $x = 1, z = 4 \Rightarrow$ Bedingung $x > 1$ ist nicht mehr erfüllt und wir geben $z = 4$ zurück.

Es wird somit insgesamt vier mal die Schleife betreten und außerdem ist $16 = 2^4 \Leftrightarrow \log_2 16 = 4$. Es wird die Schleife maximal nur $\lceil \log_2 x \rceil$ betreten, weil nach dem $\lceil \log_2 x \rceil$ Schleifendurchlauf muss x kleiner als 1 sein. Das geht aus der Definition des Logarithmus hervor. Die Laufzeit der Funktion liegt somit in $\Theta(\log_2 n)$.

Falls statt $x \leftarrow \lfloor \frac{x}{2} \rfloor$ diese Zeile $x \leftarrow \lfloor \frac{x}{5} \rfloor$ verwendet, liegt die Laufzeit in $\Theta(\log_5 n)$.
Kann man somit sagen, dass die Laufzeit mit $\Theta(\log_5 n)$ asymptotisch schneller ist als $\Theta(\log_2 n)$?

Nein, weil Logarithmen mit verschiedenen Basen sich nur um einen konstanten Faktor unterscheiden.

$$\log_5 n = \log_2 n \cdot \underbrace{\frac{1}{\log_2 5}}_{\text{ist konstanter Wert}} \qquad \text{allgemein: } \log_a n = \frac{\log_b n}{\log_b a}$$

Somit macht es asmptotisch keinen Unterschied, welche Basis der Logarithmus verwendet.

Aufgabe 5. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionB(n):
    x ← 1
    for j ← 1 bis ⌊log_5 n⌋
        x ← 4 · x
        z ← j
        x ← ⌊x/2⌋
    for j ← 1 bis ⌊log_2 n⌋
        z ← z − 1
    return z
```

Lösung

Diese Aufgabe ähnelt sehr der ersten Aufgabe, es gibt aber kleine Unterschiede.

Die erste Zuweisung beansprucht eine Laufzeit von $\Theta(1)$.

Danach iterieren wir in der ersten *for*-Schleife so oft wir der Schleifenvariable j einen Wert im Intervall $[1, \lfloor \log_5 n \rfloor]$ zuordnen können: also $\lfloor \log_5 n \rfloor$-Mal.
In diesen $\lfloor \log_5 n \rfloor$ Iterationen vervierfachen und halbieren wir (also verdoppeln wir effektiv) x und weisen z dem aktuellsten Wert von j zu.

Das bedeutet wir haben in der ersten Schleife eine Laufzeit von $\Theta(\lfloor \log_5 n \rfloor)$ und danach hat z den selben Wert wie die höchste Zahl des Intervalls von j.
Weiters hat x nach der Ausführung der ersten Schleife den Wert:

$$x := x_{\text{Initial}} \cdot 2^{\lfloor \log_5 n \rfloor}$$

$$x := 1 \cdot 2^{\lfloor \log_5 n \rfloor}$$

Auf analoger Weise lässt es sich die Laufzeit $\Theta(\lfloor \log_2 n \rfloor)$ für die zweite Schleife feststellen. Wir dekrementieren in der zweiten Schleife z bei jeder Iteration um 1, und können uns mit der folgenden Gleichung den Rückgabewert der Funktion berechnen:

$$z = z_{\text{initial}} - \lfloor \log_2 n \rfloor \cdot 1$$

$$z = \lfloor \log_5 n \rfloor - \lfloor \log_2 n \rfloor$$

Die Gesamtlaufzeit ergibt sich aus der Addition beider Schleifen, wobei wir alle einzelnen Wertzuweisungen außer Acht lassen, da sie $\Theta(1)$ betragen und weder als Faktor noch als Summand asymptotisch einen Einfluss haben.
Daraus folgt, die Gesamtlaufzeit beträgt $\Theta(\lfloor \log_5 n \rfloor + \lfloor \log_2 n \rfloor) = \Theta(\log_2 n) = \Theta(\log_5 n) = \Theta(\log n)$, da sich Logarithmen mit verschiedenen Basen sich nur um einen konstanten Faktor unterscheiden. Dies gilt aber nur, falls die Basis b des Logarithmus $b > 1$ ist.
Zur Erinnerung: $\log_a n = \dfrac{\log_b n}{\log_b a}$

Aufgabe 6. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionE(n):
    a ← 10
    b ← n
    for i = 1 bis a
        b ← b * b
    d ← b
    while d > 1
        d ← ⌊2d/3⌋
    return b
```

Lösung Der Wert von b wird in der for-Schleife mit sich selbst multipliziert. Da a den fixen Wert 10 hat, besitzt b am Ende den Wert $n^{2^{10}} = n^{1024}$. Somit liegt der Rückgabewert in $\Theta(n^{1024})$.

Die ersten Zuweisungen laufen in konstanter Zeit ab. Auch die erste for-Schleife liegt in konstanter Zeit, da a den fixen Wert 10 hat und sich in der Schleife nur eine Zuweisung befindet.

In der unteren while-Schleife wird der Wert von $d = b$, welcher sich in $\Theta(n^{1024})$ befindet, mit $\frac{2}{3}$ multipliziert.

$$\frac{2}{3} = \frac{1}{\frac{3}{2}} = \frac{1}{1.5}$$

Dadurch erhält man eine logarithmische Laufzeit für die while-Schleife. Es werden $\lceil \log_{1.5}(d) \rceil$ Schleifendurchläufe benötigt, damit d kleiner als 1 wird. Dadurch ergibt sich die Laufzeit:

$$\Theta(\log_{1.5}(n^{1024})) = \Theta(\log n^{1024}) = \Theta(1024 \cdot \log n) = \Theta(\log n)$$

Da die Basis und Exponenten in Logarithmen für asymptotischen Abschätzungen irrelevant sind (da sie sich nur von einem konstanten Faktor unterscheiden), ist die Laufzeit für die while-Schleife $\Theta(\log n)$.

Insgesamt ergibt sich somit folgende Laufzeit: $\Theta(1 + \log n) = \Theta(\log n)$

Aufgabe 7. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionA(n):
    a ← n/10
    if a ≤ 10 then
        return a
    if a > 10 then
        FunktionA(a)
```

Lösung

Sei dazu $T(n)$ die Laufzeit von dem Algorithmus. Die Laufzeit von einer Ausführung besteht aus den nicht-rekursiven Anweisungen und der Rekursion. Daher gilt:

$$T(n) = T(n)_{\text{nicht-rekursiv}} + T(n)_{\text{Rekursion}}.$$

Die nicht-rekursiven Teile benötigen nur konstante Zeit und lassen sich daher mit einer Konstante c abschätzen. Die Rekursion wird mit $a = \frac{n}{10}$ aufgerufen und benötig daher höchstens $T(\frac{n}{10})$ zeit. Sie wird jedoch nur aufgerufen, wenn $a > 10$. Wir erhalten also folgende Ungleichung:

$$T(n) \leq \left\{ \begin{array}{ll} c, & \frac{n}{10} \leq 10 \\ c + T(\frac{n}{10}), & \frac{n}{10} > 10 \end{array} \right\}$$

Nun können wir das gleiche Spiel für $T(\frac{n}{10})$ machen. Wir erhalten im Endeffekt

$$T(n) \leq \underbrace{c + \cdots + c}_{k \text{ mal}} + T(n \cdot 10^{-k}),$$

falls $n \cdot 10^{-k} > 10$, also wir nach $k-1$ Schritten noch weitere rekurisive Aufrufe haben. Sobald $n \cdot 10^{-(k+1)} \leq 10$ gilt ersetzten wir $T(n \cdot 10^{-k})$ nur noch ein letztes mal mit c. Bestimmen wir das kleinst mögliche k:

$$n \cdot 10^{-(k+1)} \leq 10$$

$$n \leq 10^{k+2}$$
$$\log n \leq (k+2)\log 10$$
$$\log_{10} n - 2 \leq k.$$

Daraus schließen wir:

$$T(n) \leq \underbrace{c + \cdots + c}_{\lceil \log_{10} n \rceil - 2 \text{ mal}} + T(n \cdot 10^{-\lceil \log_{10} n \rceil + 2}) \leq \underbrace{c + \cdots + c}_{\lceil \log_{10} n \rceil - 1 \text{ mal}} = (\lceil \log_{10} n \rceil - 1)c = \Theta(\log n)$$

Weiterer Lösungsweg mit Umformung des Algorithmus:
Da bei jedem rekursiven Aufruf der Parameter n nur durch 10 dividiert wird, und überprüft wird, ob es entweder kleiner oder größer als zehn ist, kann man die Funktion auch in einem iterativen Algorithmus umschreiben. Es handelt sich um einen linearrekursiven Algorithmus, dessen rekursiver Funktionsaufruf einfach durch eine *while*-Schleife ersetzt werden kann:

```
FunktionA(n):
    a ← n/10
    while a > 10
        a ← a/10
    return a
```

Der Algorithmus ist semantisch äquivalent zu davor und hat zusätzlich die selbe asymptotische Laufzeit. Genaugenommen müsste dies natürlich nun gezeigt werden, aber für dieses Beispiel nehmen wir an, dass dies gilt. Betrachten wir also nun den umgeformten Algorithmus.

Die erste Wertzuweisung hat eine Laufzeit von $\Theta(1)$.

Danach folgt die *while*-Schleife, mit der Abbruchsbedingung $a \leq 10$.
Innerhalb dieser, in jeder der k Iterationen teilen wir a durch 10 und weisen diesen neuen Wert sich selbst zu.
Mit der folgenden Ungleichung lassen sich die Anzahl der Iterationen ermitteln (unter der Annahme $n \geq 0$):

$$a_{\text{Initial}} \cdot \left(\frac{1}{10}\right)^k \leq 10$$
$$\frac{n}{10} \cdot \left(\frac{1}{10}\right)^k \leq 10$$
$$n \leq 10^{k+2}$$
$$\log(n) \leq \log(10^{k+2})$$
$$\log(n) \leq (k+2) \cdot \log(10)$$
$$\frac{\log(n)}{\log(10)} - 2 \leq k$$

$$\log_{10}(n) - 2 \leq k$$

Damit können wir auch den Rückgabewert a bestimmen:

$$a := a_{\text{Initial}} \cdot \left(\frac{1}{10}\right)^{\lceil \log_{10}(n)-2 \rceil}$$

Die Gesamtlaufzeit des Algorithmus lautet:
$\Theta(1 + \lceil \log_{10}(n) - 2 \rceil \cdot 1) = \Theta(\lceil \log_{10}(n) - 2 \rceil) = \Theta(\log n)$.

Aufgabe 8. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

Hinweis: *Es gilt* $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$.

```
FunktionA(n):
Input: n ≥ 1
    z ← 0
    for i = 1 bis n
        z ← z + i + 1
        l ← 0
        while l ≤ n
            l ← l + i
    while z > 0
        z ← ⌊z/2⌋
```

Lösung

Fangen wir mit z an: Anfangs ist $z = 0$, wird dann in der *for*-Scheife n mal erhöht und in der *while*-Schleife so lange halbiert, bis es wieder 0 ist. Berechnen wir uns nun den Wert von z nach der *for*-schleife: wir addieren n mal 1 und alle Zahlen zwischen 1 und n, also:

$$z = \sum_{i=1}^{n}(i+1) = \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 = \frac{n(n+1)}{2} + n = \frac{n(n+3)}{2}.$$

Nun zu dem halbieren: Da die *for*-Schleife klarerweise schon $\Omega(n)$ Zeit brauch, ist im Prinzip die Erkenntniss, dass man nur weniger n mal halbieren muss ausreichen, da $\Omega(n) + O(n) = \Omega(n)$. Vollständigkeitshalber zeigen wir jedoch die etwas stärkere Behauptung, nämlich dass wir nur $O(\log(n))$ Halbierungen benötigen. Dazu sollte man beobachten, dass abrunden den Prozess nur beschleunigen kann (wir ziehen etwas extra

ab) und, dass bei $z = 1$ nur noch eine Halbierung notwedig ist. Daher berechnen wir die Anzahl an benötige (exacten) Halbierungen ab, bis $z = 1$ gilt.

$$z_{\text{initial}} \cdot \frac{1}{2}^k = 1$$

$$\log_2(\frac{n(n+3)}{2}) = k$$

Daher reichen $\lceil \log_2(\frac{n(n+3)}{2}) \rceil = O(\log(\frac{n(n+3)}{2})) = O(\log(n^2)) = O(\log n)$ Halbierungen. Also dominiert die *for*-Schleife und wir können die *while*-Schleife, sowie z, vernachlässigen.

Betrachten wir nun also l: Hier braucht die *while*-Schleife $\lceil \frac{n}{i} \rceil$ Iteration, da l nach k iteration den Wert $i \cdot k$ hat. Die Laufzeit von der *for*-Schleife zusammen mit der *while*-Schleife ist dann gegeben durch:

$$\Theta(\sum_{i=1}^{n} \lceil \frac{n}{i} \rceil) = \Theta(n \sum_{i=1}^{n} \frac{1}{i}) = \Theta(n \log(n))$$

Damit ist die gesamte Laufzeit von dem Algorithmus $\Theta(n \log n)$.

---

4 Polynomielle Laufzeit

Aufgabe 9. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionE(n):
    sum ← 0
    for i ← 1, ..., n
        for j ← 1 ..., i
            sum ← sum + 1
    return sum
```

Lösung Wir haben nun eine verschachtelte *for*-Schleife vor uns. In der inneren Schleife wird j bis i (die äußere Schleifenvariable) iteriert. Somit wird die innere Schleife i mal betreten. In der inneren Schleife befindet sich nur eine Zuweisung, somit vereinfacht sich unsere Rechnung. Stellen wir nun fest, wie oft die Zeile $sum \leftarrow sum + 1$ ausgeführt wird:

Innere Schleife:

$$\sum_{j=1}^{i} 1 = i$$

Mit äußerer Schleife:

$$\sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

Die zweite Summe $\sum_{i=1}^{n} i$ wird mittels der Gaußschen Summenformel umgeformt und wir kommen zu dem Ergebnis, dass die Zeile $sum \leftarrow sum + 1$ insgesamt $(n^2+n)/2$-mal ausgeführt wird. Somit liegt die Laufzeit in $\Theta(n^2)$

Aufgabe 10. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionB(n):
    x ← 1
    y ← 0
    z ← 0
    while x < n^2
        while y ≤ x
            y ← y + 1
            z ← z + n
        y ← 0
        x ← x + 1
    return z
```

Lösung

Die ersten 3 Wertzuweisungen haben eine Laufzeit von $\Theta(3) = \Theta(1)$.

Die äußere *while*-Schleife iteriert so oft, bis die Abbruchsbedingung ($x \geq n^2$) zutrifft und inkrementiert bei jeder Iteration x um 1. Wir können die Anzahl der k Iterationen mit der folgenden Ungleichung ermitteln:

$$x_{\text{Initial}} + k \cdot 1 \geq n^2$$

$$1 + k \geq n^2$$

$$k \geq n^2 - 1$$

Und stellen damit fest die Ungleichung ist bei (n^2-1) Iterationen erfüllt und die innere Schleife wird (n^2-1)-Mal insgesamt angefangen.

Die innere *while*-Schleife hat die Abbruchsbedingung ($y > x$) und inkrementiert bei jeder Iteration y um 1.

Bei jedem der n^2-Aufrufe der inneren Schleife, hat x aufsteigend die Werte aus dem Intervall $[1, n^2 - 1]$.

Das bedeutet bei der ersten Ausführung der inneren Schleife ist die Abbruchsbedingung $(y > 1)$, danach $(y > 2)$ und so weiter bis $(y > n^2 - 1)$.

Sei $a \in [1, n^2 - 1]$ und die aktuelle Abbruchsbedingung der inneren Schleife $(y > a)$. Dann können wir die Anzahl der k Iterationen dieser mit der folgenden Ungleichung ermitteln:

$$y_{\text{Initial}} + k \cdot 1 > a$$

$$0 + k > a$$

Diese Ungleichung ist bei $a + 1$ Iterationen erfüllt.

Das bedeutet die Anzahl der *Iterationen* der inneren Schleife liegen im Intervall $[1 + 1, n^2 - 1 + 1] = [2, n^2]$.
Nun wollen wir wissen wie oft die Instruktion, welche z um n inkrementiert ausgeführt wird.
Dafür müssen wir die Anzahl aller Iterationen im Intervall welche wir festgestellt haben summieren:

$$k_{\text{Total}} = \sum_{i=2}^{n^2} i = \frac{n^2(n^2+1)}{2} - 1 = \frac{n^4 + n^2 - 2}{2} \in \Theta(n^4)$$

Der Algoritmus hat somit eine asymptotische Laufzeit von $\Theta(n^4)$.

Aufgabe 11. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionB(n):
    x ← 1
    y ← 0
    z ← 0
    while x < n^2
        while y ≤ x
            y ← y + 1
            z ← z + n
        x ← x + 1
    return z
```

Lösung

Diese Aufgabe ähnelt sehr der vorherigen, es gibt einen kleinen (aber feinen) Unterschied. Nämlich die Variable y wird nach der inneren *while*-Schelife nicht mehr zurückgesetzt. Das hat zur Folge, dass die innere Schleife nur mehr 1x (außer wenn $x = 1$ zweimal, da $y = 0$ am Start) betreten wird.

Nach der inneren Schleife gilt: $y = x + 1$. Dann wird x um eines erhöht und die äußere Schleife beginnt wieder. Somit kann die innere Schleife maximal nur noch einmal durchlaufen werden.

Da x immer nur um 1 erhöht wird, und die äußere Schleife als Bedingung $x < n^2$ hat, wird die äußere Schleife $\Theta(n^2)$ betreten, aber der Body der Schleife liegt in konstanter Zeit $\Theta(1)$.

Somit ist die gesamte Laufzeit in $\Theta(n^2)$.

Aufgabe 12. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionA(n):
    a ← 2
    b ← 2n²
    while b > 1
        a ← a + 1
        c ← n
        while c ≥ n
            b ← b − a
            a ← ⌊a/2⌋
            c ← ⌊2n/c⌋
```

Lösung

Die ersten beiden Wertzuweisungen haben beide eine Laufzeit von je $\Theta(1)$.

Die äußere *while*-Schleife hat die Abbruchbedingung $b \leq 1$. Um die Anzahl der Iterationen dieser zu berechnen, müssen wir uns die innere verschachtelte *while*-Schleife ansehen, da darin b um a dekrementiert wird und dadurch bestimmt wann die äußere Schleife abbricht und das Programm terminiert.

Innerhalb der äußeren *while*-Schleife inkrementieren wir a um 1 und weisen c den Wert n zu.

Die innere *while*-Schleife hat die Abbruchbedingung $c < n$.
Darin weisen wir c den Wert $\lfloor \frac{2n}{c} \rfloor = \lfloor 2n \cdot c^{-1} \rfloor$ zu.

Wenn wir nun den initialen Wert von c als c_0, den Wert von c nach der ersten Zuweisung als c_1 und den Wert von c nach der zweiten Zuweisung als c_2 und so weiter bezeichnen, dann lässt sich wenn man diese berechnet ein Muster erkennen:

$$c_0 = n$$
$$c_1 = \lfloor\frac{2n}{c_0}\rfloor = \lfloor\frac{2n}{n}\rfloor = 2$$
$$c_2 = \lfloor\frac{2n}{c_1}\rfloor = \lfloor\frac{2n}{2}\rfloor = n$$
$$c_3 = \lfloor\frac{2n}{c_2}\rfloor = \lfloor\frac{2n}{n}\rfloor = 2$$
$$c_4 = \lfloor\frac{2n}{c_3}\rfloor = \lfloor\frac{2n}{2}\rfloor = n$$
$$\dots$$

Wie leicht ersichtlich alterniert der Wert von c_i zwischen 2 für gerade und n für ungerade i. Es handelt sich (unter bestimmten Bedingungen die gleich näher erläutert werden) um eine Endlosschleife. Dann terminiert das Programm gar nicht und wir haben theoretisch unendliche Laufzeit.

Nun möchten wir herausfinden ab welchem c_i die Schleife hält – also $c_i < n$ erfüllt ist. Dadurch, dass das Argument der Funktion n die Variable c bestimmt, bestimmt sie auch ob wir die innere Schleife verlassen können.
Bei $c_i = n$ kann $c_i < n$ nicht erfüllt sein und bei $c_i = 2$ kann $c_i < n$ nur erfüllt sein, wenn $n \geq 3$.

Daraus folgt: Das Programm terminiert nicht wenn die Vorbedingung $n \geq 3$ nicht erfüllt ist.
Wenn die genannte Vorbedingung jedoch erfüllt ist, dann iteriert die innere Schleife nur 1-Mal (die innere Schleife hat eine asymptotische Laufzeit von $\Theta(1)$) und der Algorithmus ist semantisch äquivalent zu [1]:

`FunktionA-V2(`n`):`
Input: $n \geq 3$
    $a \leftarrow 2$
    $b \leftarrow 2n^2$
    while $b > 1$
        $a \leftarrow a + 1$
        $c \leftarrow n$
        $b \leftarrow b - a$
        $a \leftarrow \lfloor\frac{a}{2}\rfloor$
        $c \leftarrow \lfloor\frac{2n}{c}\rfloor$

[1] Semantisch äquivalente Algorithmen liefern das selbe Ergebnis, jedoch haben sie in der Praxis nicht notwendigerweise die selbe Laufzeit. In unserem Fall haben wir die redundante Überprüfung einer Bedingung entfernt und mehrere Wertzuweisungen und Berechnungen mit $\Theta(1)$ entfernt. Hier bleibt die asymptotische Laufzeit daher auch nach der Umwandlung gleich zum urprünglichen Algorithmus.

Wir können noch weiter vereinfachen:

```
FunktionA-V3(n):
Input: n ≥ 3
    a ← 2
    b ← 2n²
    while b > 1
        b ← b − (a + 1)
        a ← ⌊(a+1)/2⌋
        c ← 2
```

Die *while*-Schleife der neuen, semantisch äquivalenten Funktion iteriert so lange, bis die Abbruchbedingung $b \leq 1$ erfüllt ist.

Wir schauen uns die Folge der Variable a genauer an:

$$a_0 = 2$$

$$a_1 = \lfloor \frac{(a_0 + 1)}{2} \rfloor = \lfloor \frac{(2 + 1)}{2} \rfloor = 1$$

$$a_2 = \lfloor \frac{(a_1 + 1)}{2} \rfloor = \lfloor \frac{(1 + 1)}{2} \rfloor = 1$$

$$a_3 = \lfloor \frac{(a_2 + 1)}{2} \rfloor = \lfloor \frac{(1 + 1)}{2} \rfloor = 1$$

$$\dots$$

Daraus folgt, dass angenommen werden kann, dass innerhalb der *while*-Schleife, a immer den Wert 1 hat:

```
FunktionA-V4(n):
Input: n ≥ 3
    a ← 2
    b ← 2n²
    while b > 1
        b ← b − 2
        a ← 1
        c ← 2
```

Deshalb können wir die Anzahl der Iterationen der *while*-Schleife mit folgender Ungleichung berechnen:

$$b_{\text{Initial}} - k \cdot 2 \leq 1$$

$$2n^2 - 2k \leq 1$$

$$-2k \leq 1 - 2n^2$$

$$-k \leq \frac{1}{2} - n^2$$

$$k \geq n^2 - \frac{1}{2}$$

Daraus folgt, wir iterieren die innere Schleife mindestens $\lceil n^2 - \frac{1}{2} \rceil$ Mal. Das entspricht einer asymptotischen Laufzeit von $\Theta(n^2)$.

Dadurch dass alle Operationen innerhalb und außerhalb unserer einzigen Schleife nur $\Theta(1)$ Zeit benötigen zugehören, erhalten wir die Gesamtlaufzeit $\Theta(n^2)$.

---

5 Exponentielle Laufzeit

Aufgabe 13. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionE(n):
    sum ← 0
    for i ← 1, ..., n
        z ← 1
        for j ← 2 ..., i
            z ← z · 2
        sum ← sum + z
    while sum > 0
        sum ← sum − 1
```

Lösung Die erste verschachtelte for-Schleife liegt in $\Theta(n^2)$. (siehe Polynomielle Laufzeit). Die Frage ist nun, was im ersten Teil der Funktion berechnet wird. Wenn wir das berechnet haben, können wir sehr einfach die Laufzeit bestimmen, da in der unteren while-Schleife das Ergebnis immer um 1 dekrementiert wird, bis 0 erreicht wird.

Gehen wir zurück zur verschachtelten for-Schleife. Für ein beliebiges i wird in z die Zahl 2^{i-1} berechnet und dann zur Variable *sum* addiert. Somit kommen wir für die Variable *sum* zu folgender Reihe:

$$sum = \sum_{i=1}^{n} 2^{i-1} = \sum_{i=0}^{n-1} 2^i$$

Das ist, wie aus Analysis bekannt, die geometrische Reihe. Wenn wir uns die Partialsummen der geometrischen Reihe ansehen, wird diese wie folgt berechnet (für $q \neq 1$):

$$\sum_{i=0}^{n} q^i = \frac{1 - q^{n+1}}{1 - q}$$

$$\sum_{i=0}^{n-1} 2^i = \frac{1 - 2^n}{1 - 2} = 2^n - 1$$

Das bedeutet, die Variable *sum* liegt in $\Theta(2^n)$ und daraus folgt, dass natürlich auch die Laufzeit der Funktion in $\Theta(2^n)$ ist.

Was würde bedeuten, wenn anstatt $z \leftarrow z \cdot 2$ die Zeile $z \leftarrow z \cdot 3$ verwendet wird?

Das bedeutet, dass die Funktion in $\Theta(3^n)$ liegt. Anders als bei der Logarithmischen Laufzeit, macht es sehr wohl einen Unterschied, welche Basis verwendet wird.

Für $a, b, n \in \mathbb{N}, a > 1, b > 1, a > b$, gilt $a^n \gg b^n$.

Somit ist auch $\Theta(3^n) \neq \Theta(2^n)$.

Aufgabe 14. *Bestimmen Sie für die Funktion die Laufzeit in Abhängigkeit von n in Θ-Notation.*

```
FunktionB(n):
    for i ← 1, ..., n
        A[i] ← 0
    while A[n] = 0
        i ← 1
        while A[i] = 1
            A[i] ← 0
            i ← i + 1
        A[i] ← 1
```

Lösung

Die erste *for*-Schleife initialisiert die Werte von dem Array $A[1], \ldots, A[n]$ auf 0. Das benötig klarerweise $\Theta(n)$ Zeit.

Die verschachtelten *while*-Schleifen realisieren nun einen binären counter. Hierzu kann die Bitfolge $A[1], \ldots, A[n]$ also n-bit lange binäre Zahl aufgefasst werden. Die Bedingung der äußeren *while*-Scheife ist dann nicht mehr erfüllt, wenn die Zahl größer-gleich 2^{n-1} wird. Wir iterieren daher die äußeren *while*-Scheife 2^{n-1} mal.

Betrachten wir nun die innere *while*-Schleife genauer. Wir können ihre Laufzeit bestimmen, indem wir zählen wie oft man in einem binären counter bits auf 0 setzten muss. Hierzu: das least significant bit $A[n]$ wird jedes zweite mal auf 0 gestezt (wenn der counter gerade wird), das bit $A[n-1]$ jedes vierte mal (wenn der counter durch 4 teilbar ist), das bit $A[n-2]$ jedes achte mal, ... Addieren wir das auf kommen wir auf:

$$\frac{2^{n-1}}{2} + \frac{2^{n-1}}{4} + \frac{2^{n-1}}{8} + \cdots + \frac{2^{n-1}}{2^{n-1}} = \sum_{i=1}^{n-1} \frac{2^{n-1}}{2^i} \leq 2^{n-1} \sum_{i=1}^{\infty} \frac{1}{2^i} = 2^{n-1}.$$

Also wird der Körper der inneren *while*-Scheife während des ganze Zählvorgang $\Theta(2^{n-1}) = \Theta(2^n)$ mal ausgeführt.

Insgesammt ergibt sich also eine Laufzeit von $\Theta(n + 2^n + 2^n) = \Theta(2^n)$ für den Algorithmus Es ist an dieser Stelle auch wichtig zu erwähnen, dass $\Theta(n \cdot 2^n) \neq \Theta(n + 2^n)$ gilt.

---

6 Laufzeit & Rückgabewert

Aufgabe 15. *Bestimmen Sie für die Funktion die Laufzeit und Rückgabewert in Abhängigkeit von n jeweils in Θ-Notation.*

`FunktionB(n):`
Input: $n \geq 1$
$i \leftarrow 1$
$a \leftarrow 1$
repeat
$i \leftarrow i + 1$
$a \leftarrow a \cdot 2$
until $i \geq \log_2 n$
$z \leftarrow 0$
for $j \leftarrow 0, \ldots, a$
$z \leftarrow z + j + n$
return z

Lösung

Die ersten zwei Wertzuweisungsanweisungen für i, a haben eine Laufzeit von $\Theta(1)$.

Die Abbruchsbedingung der darauf folgenden *repeat*-Schleife lautet $i \geq \log_2(n)$.
Wir berechnen dir k Iterationen der *repeat*-Schleife mit folgender Formel, da i in jeder Iteration um 1 inkrementiert wird:

$$i_{\text{Initial}} + k \cdot 1 \geq \log_2(n)$$

$$1 + k \geq \log_2(n)$$

$$k \geq \log_2(n) - 1$$

Wir wissen nun dass die Schleife mindestens $(\log_2(n) - 1)$-Mal iteriert.

Dadurch hat die *repeat*-Schleife, inklusive den 2 darin enthaltenen Operationen eine Laufzeit von $\Theta((\log_2(n) - 1) \cdot (1 + 1)) = \Theta(\log(n))$.

Als nächstes berechnen wir den Wert von a nach der *repeat*-Schleife, wobei es in jeder Iteration verdoppelt wird. Wir nutzen dafür die folgende Gleichung:

$$a := a_{\text{Initial}} \cdot 2^{\log_2(n)-1}$$

$$a := a_{\text{Initial}} \cdot \left(\frac{2^{\log_2(n)}}{2} \right)$$

Wir setzen den initialen Wert 1 von a ein:

$$a := 1 \cdot \frac{2^{\log_2(n)}}{2} = \frac{1}{2} \cdot 2^{\log_2(n)} = \frac{1}{2} \cdot n$$

Nach der *repeat*-Schleife setzen wir z auf 0.

In der darauf folgenden *for*-Schleife weisen wir der Schleifenvariable j alle Werte $[0; a]$ zu. Es sind $a+1$ bzw. $(0.5n+1)$ Werte im gegebenen Intervall wenn wir den Wert von a einsetzen.

Wir berechnen den Rückgabewert z mit der folgenden Formel:

$$z = \underbrace{z_{\text{Initial}}}_{=0} + \left(\sum_{j=0}^{0.5n+1} j + n \right)$$

$$z = \sum_{j=0}^{0.5n+1} j + n$$

$$z = (0.5n+1) \cdot n + \sum_{j=0}^{0.5n+1} j$$

$$z = 0.5n^2 + n + \sum_{j=0}^{0.5n+1} j$$

$$z = 0.5n^2 + n + 0.125n^2 + 0.75n + 1$$

$$z = 0.625n^2 + 1.75n + 1 \in \Theta(n^2)$$

Der Rückgabewert z ist deshalb in $\Theta(n^2)$.

Aufgabe 16. *Bestimmen Sie für die Funktion die Laufzeit und Rückgabewert in Abhängigkeit von n jeweils in Θ-Notation.*

```
FunktionD(n):
    c ← 1000
    b ← n
    for i = 1 bis 2b
        if i > 10^4 then
            c ← c + 1
        for j = 1 bis b
            c ← c + 1
    d ← c
    while d > 1
        d ← ⌊d/4⌋
    return c
```

Lösung Lösung: Laufzeit: $\Theta(n^2 + \log n) = \Theta(n^2)$, Rückgabewert: $\Theta(n^2 + 2n - 10^4) = \Theta(n^2)$

Rückgabewert. Betrachten wir zuerst den Rückgabewert der Funktion, da wir diesen bei der folgenden Laufzeitabschätzung für die untere while-Schleife verwenden werden. Zu Beginn wird c der Wert 1000 zugewiesen, für die asymptomatische Abschätzung kann man diesen Wert als konstant ($\Theta(1)$) betrachten.

Dann folgen verschachtelte for-Schleifen, in der der Wert von c jeweils inkrementiert wird. In der äußeren for-Schleife wird der Wert nur dann inkrementiert, falls die Zählvariable i größer ist als 10^4. Da wir den asymptotischen Wert suchen, wird der Wert um $\Theta(n)$ erhöht. Wir interessieren uns besonders für große n, falls bspw. n im Bereich 10^6 liegt, machen die ersten 10^4 Iterationen nur einen kleinen Teil der Gesamtanzahl aus. Insgesamt wird c für große n ($n > 10^4$) $2n - 10^4$-mal inkrementiert, somit $\Theta(n)$.

In der inneren for-Schleife wird c auch inkrementiert. Da b den Wert n behält, wird c ebenfalls n mal inkrementiert. Die innere Schleife wird $2n$-mal ausgeführt, somit wird die Variable c insgesamt $2n - 10^4 + 2n \cdot n = 2n^2 + 2n - 10^4 = \Theta(n^2)$ mal inkrementiert. Im weiteren Verlauf wird c nicht mehr verändert.

Laufzeit. Wir haben zu Beginn zwei einfache Zuweisungen, die in konstanter Zeit $\Theta(1)$ erfolgen. Dann folgt eine for-Schleife, welche $2b = 2n$ Iterationen durchläuft. Die Variable b wird auch nicht im Inneren der for-Schleife verändert, somit können wir davon ausgehen, dass die äußere for-Schleife $\Theta(n)$ Iterationen durchläuft. Jetzt müssen wir betrachten, welche Laufzeit der Body der Schleife hat. Die Laufzeit der if-Verzweigung ist konstant. Die innere for-Schleife hängt auch von b ab, da aber b im Verlauf nicht verändert wurde, besitzt diese for-Schleife n-Iterationen. In dieser befindet sich nur eine einzelne Zuweisung, somit haben wir in dieser Schleife n Operationen. Die Anzahl der Operationen in der äußeren for-Schleife befinden sich somit in $\Theta(n)$, die wiederum $\Theta(n)$ durchgeführt werden. Das ergibt eine asymptotische Laufzeit von $\Theta(n^2)$ für die verschachtelten for-Schleifen.

Die Laufzeit der while-Schleife hängt von c ab, der Wert von c ist in $\Theta(n^2)$. Im Body der Schleife wird d jedesmal geviertelt, man braucht somit $\lceil \log_4(d) \rceil$ Iterationen, damit d kleiner-gleich als 1 ist.

$$\log_4(n^2) = 2 \cdot \log_4(n) = \Theta(\log(n))$$

Da sich die verschiedenen Logarithmen sich nur um eine konstante unterscheiden, wird die while-Schleife $\Theta(\log(n))$ ausgeführt.

Insgesamt haben wir somit $\Theta(n^2)$ für die verschachtelte for-Schleife und $\Theta(\log(n))$ für die while-Schleife, es gilt $\Theta(n^2) \gg \Theta(\log n)$.

Laufzeit: $\Theta(n^2)$

Aufgabe 17. *Bestimmen Sie für die Funktion die Laufzeit und Rückgabewert in Abhängigkeit von n jeweils in Θ-Notation.*

```
FunktionA(n):
    x ← n²
    z ← 1
    while x > 1
        x ← x/3
        z ← 3 · z
    return z
```

Lösung

Laufzeit. Jede Zuweisungs-Instruktion beansprucht nur $\Theta(1)$ Zeit. Für die ersten zwei Instruktionen bedeutet das deshalb eine Laufzeit von: $\Theta(1+1) = \Theta(1)$.

Anschließend beobachten wir die *while*-Schleife mit der Abbruchsbedingung $x > 1$ genauer: Wir iterieren so lange bis sie *nicht* mehr erfüllt ist (oder wir betreten sie unter Umständen erst gar nicht).

Die Gültigkeit der Bedingung ist abhängig von der Zuweisung $x \leftarrow \frac{x}{3}$.
Daraus folgt: Wir dritteln x so oft, bis das Gegenteil der Bedingung, also $x \leq 1$ zutrifft.

Sei k die Anzahl der Iterationen, können wir sie mit folgender Formel genau herleiten:

$$x \cdot \left(\frac{1}{3}\right)^k \leq 1$$

Zur Berechnung der Formel müssen wir den initialen Wert von x noch einsetzen:

$$n^2 \cdot \left(\frac{1}{3}\right)^k \leq 1$$

$$n^2 \cdot \frac{1}{3^k} \leq 1$$

$$n^2 \leq 3^k$$

$$\log_3(n^2) \leq \log_3(3^k)$$

$$2 \cdot \log_3(n) \leq k \cdot \log_3(3)$$

$$2 \cdot \log_3(n) \leq k$$

Nun müssen wir die Anzahl der Iterationen k, mit der Laufzeit der Instruktionen in der *while*-Schleife multiplizieren: $\Theta(1+1)$.

Zusammengefasst: $\Theta(1+1+2 \cdot \log_3(n) \cdot (1+1)) = \Theta(\log_3(n)) = \Theta(\log_2(n)) = \Theta(\log(n))$

Rückgabewert. Bei jeder Iteration, multiplizieren wir z mit dem Dreifachen des vorherigen Wertes und weisen es sich selbst zu.
Daraus folgt, zur Ermittlung des finalen Zustandes von z, benötigen wir eine Gleichung der Form:

$$z := \text{Initial} \cdot 3^{\text{Iterationen}}$$

Wir wissen bereits, wie oft wir durch die Schleife iterieren müssen: $k = 2 \cdot \log_3(n)$ (wir vernachlässigen die Rundung auf eine ganze Zahl).

Daraus folgt:

$$z := 1 \cdot 3^{2 \cdot \log_3(n)} = 3^{\cdot \log_3(n^2)} = n^2$$

Deshalb ist: $z \in \Theta(n^2)$

Anmerkungen:

Das ist einer der „wenigen“ Fälle wo der Faktor 2 relevant für die Größenordnung ist. Hätten wir $k = c \cdot \log_3(n)$ Schleifendurchläufe, wäre der Rückgabewert $\Theta(n^c)$.

Außerdem ist es im Exponenten auch notwendig, auf die Basis des Logarithmus zu achten. Für alle $c, a, b > 1$ und $a \neq b$ gilt:

$$\Theta(c^{\log_a(n)}) \neq \Theta(c^{\log_b(n)})$$

Dasselbe gilt auch für konstanten im Exponenten. Als Beispiel:

$$\Theta(3^{2n}) \neq \Theta(3^n)$$

da $\Theta(3^{2n}) = \Theta(9^n)$.

Aufgabe 18. *Bestimmen Sie für die Funktion die Laufzeit und Rückgabewert in Abhängigkeit von n jeweils in Θ-Notation.*

```
FunktionB(n):
    j ← 0
    z ← 1
    for i = 1 bis 3n
        if i mod 3 = 0 then
            j ← j + i
    j ← 2/3 · j − n
    while j > 0
        z ← n · z
        j ← j − 1
    return z
```

Lösung

Laufzeit. Die ersten zwei Zuweisungen des Pseudocodes haben eine Laufzeit von $\Theta(1)$.

Die darauf folgende for-Schleife iteriert $3n$-Mal, jedoch wird die darin enthaltene if-Verzweigung nur jedes Mal betreten wenn i durch 3 teilbar ist.

Im Intervall $[1, 3n]$ gibt es n Zahlen, die durch 3 teilbar sind: $\frac{3n}{3} = n$.

Daraus folgt, die enthaltene if-Verzweigung wird genau n-Mal betreten und daraufhin j inkrementiert. Der Wert von j lässt sich wie folgt berechenen:

$$j := 3 + 6 + \cdots + 3n = \sum_{i=1}^{n} 3i = 3 \sum_{i=1}^{n} i = 3 \cdot \frac{n(n+1)}{2}$$

Die for-Schleife hat inklusive der enthaltenen Zuweisung eine Laufzeit von $\Theta(n \cdot 1)$.

Die darauf folgende Zuweisung hat eine Laufzeit von $\Theta(1)$.
Damit wird j der folgende Wert zugewiesen:

$$j := \frac{2}{3} \cdot \left(3 \cdot \frac{n(n+1)}{2}\right) - n$$

$$j := n(n+1) - n = n^2$$

In der darauf folgenden *while*-Schleife, wird so oft iteriert bis die Bedingung $j \leq 0$ erfüllt ist (beziehungsweise $j > 0$ nicht erfüllt ist).

Dadurch, dass bei jeder Iteration von j genau 1 abgezogen wird und j anfangs positiv ist, können wir mit einer Gleichung der folgenden Form ermitteln, wie viele k Iterationen es geben wird:

$$j + (k \cdot (-1)) = 0$$

Wir setzen in der Ungleichung den Wert von j vor der Schleife ein:

$$n^2 + (k \cdot (-1)) = 0$$

$$n^2 = k$$

Dadurch stellen wir fest, dass wir nach n^2 Iterationen den Scheifenkörper verlassen müssen, wodurch wir für die letzte Schleife eine Laufzeit von $\Theta(n^2)$ feststellen.

Zusammenfassend gelangen wir zur Laufzeit $\Theta(1+1+n \cdot 1+1+(n^2) \cdot (1+1)) = \Theta(n^2)$ dadurch dass, $\Theta(n^2) \gg \Theta(n)$.

Rückgabewert. Die Variable, welche zurückgegeben wird, wird erst in der letzten Schleife manipuliert und lässt sich nur Anhand der Anzahl der Iterationen dieser feststellen: $k = n^2$.

Wir konstruieren eine Gleichung der Form:

$$z := \text{Initial} \cdot n^{\text{Iterationen}}$$

Und gelangen dadurch zu:

$$z := 1 \cdot n^{(n^2)} = n^{(n^2)}$$

Damit erhalten wir $\Theta(n^{(n^2)})$ (Wichtig: $n^{(n^2)} \neq (n^n)^2 = n^{2n}$).

---

7 Unterschiede zwischen untere/obere Schranke und Best-/Worst-Case

Im Zuge der Algorithmenanalyse wird man häufig mit den Konzepten „Best-Case“ und „Worst-Case“ konfrontiert. Ein häufiger Fehler ist es, anzunehmen, dass diese Begriffe austauschbar mit unterer bzw. oberer Schranke sind.

Dies lässt sich jedoch nicht so sagen. Im folgenden ein einfacher Algorithmus:

```
f(n):
  x ← 0
  for i ← 0 . . . , n
    for j ← 0, . . . , n
      x ← x + 1
  return x
```
(1)

Der Algorithmus ist offensichtlich in $\Theta(n^2)$. Offensichtlich gilt $f(n) = \Omega(n)$, sowie weiters eine ungenauere untere Schranke $f(n) = \Omega(1)$. Ebenso gilt $f(n) = O(n^2)$ und $(n) = O(n!)$ sowie $f(n) = O(n^3)$.
Dies gilt, da die Laufzeit von f durch die angegebenen Laufzeiten von unten bzw. von oben beschränkt wird. Es ist so z.B. eine richtige Aussage, dass unser Algorithmus mindestens konstante, höchstens faktorielle Laufzeit benötigen wird. Können wir also sagen, dass der Algorithmus eine Worst-Case Laufzeit von n^3 hat, oder eine konstante Best-Case Laufzeit hat?
Nein. Da der Algorithmus verhältnismäßig simpel ist, zeigt sich auch ohne große Rechenschritte, dass die Laufzeit **immer** in n^2 ist. Gleichgültig welche Annahme getroffen werden, haben beide for-Schleifen n Durchläufe. Der Best-Case ist also ident mit dem Worst-Case, beide liegen in $O(n^2)$. Best- und Worst-Case sind die **schärfsten** Schranken des Algorithmus. **Es kann beliebig viele obere und untere Schranken geben, jedoch sind nur die scharfen oberen und unteren Schranken relevant zum bestimmen eines Best-Case und eines Worst-Case**.

Aufgabe 19. *Ermitteln Sie die engsten Schranken für die Worst-Case und Best-Case Laufzeit in Abhänigkeit von n:*

```
FunktionF(n):
    a ← n
    if a mod 4 = 0 then
        b ← 0
        for i = 1 bis n
            for j = 1 bis i
                b ← b + 1
        return b
    else if a mod 4 = 1 then
        return 42
    else
        for i = 1 bis n
            a ← a/n
        return a
```

Lösung Sei $T(n)$ die Laufzeit von dem Algorithmus. Wenn wir Konstanten vernachlässigen ist die Laufzeit wie folgt:

$$T(n) := \begin{cases} n^2, & n = 0 \mod 4 \\ 1, & n = 1 \mod 4 \\ n, & n = 2 \mod 4 \\ n, & n = 3 \mod 4 \end{cases}$$

Die Laufzeit $T(n)$ von dem Algorithmus liegt also in: $O(n^2)$ und $\Omega(1)$ (das gilt sowohl für den Best-case als auch für den Worst-case).

Aufgabe 20. *Ermitteln Sie die engsten Schranken für die Worst-Case und Best-Case Laufzeit in Abhänigkeit von Array L und Länge n:*

```
FunktionF(L,n):
    a ← n
    do {
      swapped ← false
      for i = 0 bis a − 1
       if L[i] > L[i + 1] then
           copy ← L[i + 1]
           L[i + 1] ← L[i]
           L[i] ← copy
           swapped ← true
      a ← a − 1
    } while(swapped)
```

Lösung Der hier aufgeführte Algorithmus beschäftigt ist eine Implementation von Bubble-Sort. Anhand dieses Beispiels lassen sich Best-Case und Worst-Case sehr gut erläutern, da hier unterschiedliche Best-Case und Worst-Case Laufzeiten existieren, abhängig von dem Array, das dem Algorithmus übergeben wird. Zusätzlich Faktoren wie eben ein solches Array oder ähnliche strukturierte Elemente, können häufig dafür sorgen, dass der gleiche Algorithmus drastisch unterschiedliche Laufzeiten haben kann.

Der Algorithmus setzt zunächst die Länge des Arrays (n) in eine Variable a ab, und durchläuft im Anschluss das Array. Sollte hier ein Wertepaar gefunden werden, welches nicht in der korrekten Reihenfolge ist, so werden die Werte vertauscht. Zuletzt wird die obere Grenze der Schleife (a) verkleinert, da nach jedem Schleifendurchlauf der größte Wert am Ende des Arrays steht, da er durch die if-Bedingung „nach oben gereicht“ wird.

Es ist offensichtlich, dass die Schleife nur fortgeführt wird, sofern innerhalb der Schleife ein Wertetausch stattgefunden hat. Ist dies nicht der Fall, so verbleibt die Variable *swapped* auf $false$ und die Schleife bricht ab. Der Best-Case ist also, dass kein Austausch stattfindet, d.h. das Array, welches übergeben wurde, ist bereits sortiert. In diesem Falle hat der Algorithmus eine Laufzeit von $\Theta(n)$, da er die Schleife ein einziges Mal komplett durchläuft.

Im Worst-Case terminiert die do-while Schleife erst, wenn a auf 0 reduziert wurde, d.h. die for-Schleife überhaupt nicht mehr betreten wird. Dies ist genau dann der Fall, wenn das Array genau umgekehrt sortiert ist, da hier n Schleifendurchläufe benötigt werden, um das Array „umzudrehen“. Als Beispiel betrachten wir das Array $\{\ 8, 7, 6, 5\ \}$, dessen korrekte Sortierung die genau umgekehrte Reihenfolge wäre. Im ersten Durchlauf der For-Schleife würde die if-Bedingung jedes Mal betreten werden, da 8 als größte Zahl an die rechts äußerste Stelle geschoben werden muss. Nach diesem Durchlauf wäre unser Array also $\{\ 7, 6, 5, 8\ \}$. Im nächsten Durchlauf würden durch $a \leftarrow a - 1$ nur noch die ersten drei Werte betrachtet werden, für jeden davon wiederrum ein weiterer Schleifendurchlauf notwendig wäre, um ihn an die richtige Stelle zu bringen. Insgesamt werden im Falle der umgekehrt sortierten Liste maximal $\frac{n \cdot (n-1)}{2}$ Vertauschungen durchgeführt ($n - 1$ für die erste, $n - 2$ für die zweite, etc.). Im Worst-Case befindet sich der Algorithmus also asymptotisch in $\Theta(n^2)$.
