186.866 Algorithmen und Datenstrukturen VU

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 $(x = 0;\ x < n;\ x{+}{+})\{$
$\quad \ldots$
$\}$

(b)
for $x \leftarrow 0$ `bis` $(n-1)$
$\quad \ldots$

(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
```

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
```

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
```

---

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
```

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
```

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
```

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)
```

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⌋
```

---

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
```

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
```

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
```

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⌋
```

---

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
```

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
```

---

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 ≥ 1
    i ← 1
    a ← 1
    repeat
      i ← i + 1
      a ← a · 2
    until i ≥ log_2 n
    z ← 0
    for j ← 0, ..., a
        z ← z + j + n
    return z
```

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
```

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

```
FunktionA(n):
    x ← n^2
    z ← 1
    while x > 1
        x ← x/3
        z ← 3 · z
    return z
```

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
```

---

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
```

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)
```
