Stand: September 2023

Teil 2: Erläuterungen und Lösungsvorschläge

Die Illustrierenden Prüfungsaufgaben (Teil 1: Beispielaufgaben, Teil 2: Erläuterungen und Lösungsvorschläge) dienen der einmaligen exemplarischen Veranschaulichung von Struktur, Anspruch und Niveau der Abiturprüfung auf grundlegendem bzw. erhöhtem Anforderungsniveau im neunjährigen Gymnasium in Bayern.

Informatik

erhöhtes Anforderungsniveau

Die Bewertung der erbrachten Prüfungsleistungen hat sich für jede Aufgabe nach der jeweils am rechten Rand der Aufgabenstellung vermerkten, maximal erreichbaren Anzahl von Bewertungseinheiten (BE) zu richten.

Der Erwartungshorizont stellt für jede Teilaufgabe eine mögliche Lösung dar; als objektorientierte Programmiersprachen werden Java und Python verwendet. Nicht dargestellte korrekte Lösungen sind als gleichwertig zu akzeptieren.

Die von einem Prüfling insgesamt erreichten Bewertungseinheiten werden gemäß folgender Tabelle in Notenpunkte umgesetzt:

mind. zu erreichender Anteil an den insgesamt zu erreichenden Bewertungseinheiten (in %)	Notenpunkte	Notenstufe
95	15	+1
90	14	1
85	13	1−
80	12	+2
75	11	2
70	10	2−
65	9	+3
60	8	3
55	7	3−
50	6	+4
45	5	4
40	4	4−
33	3	+5
27	2	5
20	1	5−
0	0	6

I

Nr.		BE
1a	• Das Wasserfallmodell ist sequenziell. Eine neue Phase beginnt erst, wenn die davorliegende vollständig abgeschlossen ist. Agile Modelle sind iterativ. Phasen werden mehrfach durchlaufen. • Beim Wasserfallmodell liegen zu Beginn Lasten- und Pflichtenheft vor, bei agilen Modellen können Änderungsanforderungen unkompliziert übernommen werden. • Beim Wasserfallmodell ist der Auftraggeber nur zu Beginn und am Ende eingebunden, bei agilen Modellen möglicherweise kontinuierlich.	3
b	Ein balancierter Binärbaum hat möglichst wenig Ebenen. Abschätzung der maximalen Anzahl an notwendigen Vergleichen: Liste: 3500 im schlechtesten Fall Baum: $\log_2(3500+1) = 11{,}77\ldots$, d. h. 12 im schlechtesten Fall Bei diesem Baum sind also erheblich weniger Vergleiche nötig.	3

c	Mögliche Lösung in Python:
	`def pruefen(self, eingabe: str):`
	`  ergebnis: bool = istRichtig(eingabe)`
	`  self.anzahlAufrufe += 1`
	`  if not ergebnis:`
	`  self.anzahlFehler += 1`
	`  return ergebnis`
	
	Mögliche Lösung in Java:
	`boolean pruefen(String eingabe) {`
	`  boolean ergebnis = istRichtig(eingabe);`
	`  anzahlAufrufe++;`
	`  if (!ergebnis) {`
	`  anzahlFehler++;`
	`  }`
	`  return ergebnis;`
	`}`
		3

d Mögliche Lösung in Python: 7

in der Klasse VOKABELBAUM:

```
def einfuegen(self, vokabelNeu: Vokabel):
    self.wurzel = self.wurzel.einfuegen(vokabelNeu)
```

in der Klasse BAUMELEMENT:

```
@abc.abstractmethod
def einfuegen(self, vokabelNeu: Vokabel) -> Baumelement:
    pass
```

in der Klasse KNOTEN:

```
def __init__(self, vokabelNeu: Vokabel):
    self.inhalt = vokabelNeu
    self.nachfolgerLi = Abschluss()
    self.nachfolgerRe = Abschluss()

def einfuegen(self, vokabelNeu: Vokabel) -> Baumelement:
    if not self.inhalt.istGleich(vokabelNeu):
        if self.inhalt.istKleiner(vokabelNeu):
            self.nachfolgerRe = self.nachfolgerRe.einfuegen(vokabelNeu)
        else:
            self.nachfolgerLi = self.nachfolgerLi.einfuegen(vokabelNeu)
    return self
```

in der Klasse ABSCHLUSS:

```
def einfuegen(self,vokabelNeu: Vokabel) -> Baumelement:
    return Knoten(vokabelNeu)
```

Mögliche Lösung in Java:

in der Klasse VOKABELBAUM:

```
void einfuegen(Vokabel vokabelNeu) {
    wurzel = wurzel.einfuegen(vokabelNeu);
}
```

in der Klasse BAUMELEMENT:

```
abstract Baumelement einfuegen(Vokabel vokabelNeu);
```

in der Klasse KNOTEN:

```
Knoten(Vokabel vokabelNeu) {
    inhalt = vokabelNeu;
    nachfolgerLi = new Abschluss();
    nachfolgerRe = new Abschluss();
}

Baumelement einfuegen(Vokabel vokabelNeu) {
    if (!inhalt.istGleich(vokabelNeu)) {
        if (inhalt.istKleiner(vokabelNeu)) {
            nachfolgerRe = nachfolgerRe.einfuegen(vokabelNeu);
        } else {
            nachfolgerLi = nachfolgerLi.einfuegen(vokabelNeu);
        }
    }
    return this;
}
```

in der Klasse ABSCHLUSS:

```
Baumelement einfuegen(Vokabel vokabelNeu) {
    return new Knoten(vokabelNeu);
}
```

e	Kirsche, Garten, Auto, gehen, Mann, reden Mögliches Wort: Liebe	2
f	Die Vokabeln mit den Schlüsselwerten „Garten“ und „Kirsche“ können vom Algorithmus nicht ausgewählt werden, da jeweils beide Nachfolger Knoten sind und daher die Variable *vokabel* nicht die leere Referenz enthalten kann. Deswegen kann nie der Inhalt dieser beiden Vokabeln zurückgeben werden.	2
g	Methode listeErstellen(jahrgangsstufe, lektion, liste) nachfolgerLi.listeErstellen(jahrgangsstufe, lektion, liste) wenn jahrgangsstufe gleich inhalt.jahrgangsstufeGeben() und lektion gleich inhalt.lektionGeben() dann liste.hintenEinfuegen(inhalt) endeWenn nachfolgerRe.listeErstellen(jahrgangsstufe, lektion, liste) endeMethode	4

h	Mögliche Lösung in Python:
	`class Vokabeltrainer:`
	`  def __init__(self):`
	`  self.vokabeln: Vokabelbaum = Vokabelbaum()`
	`  self.listeAktuell: Liste = Liste()`
	
	Mögliche Lösung in Java:
	`class Vokabeltrainer {`
	`  private Vokabelbaum vokabeln;`
	`  private Liste listeAktuell;`
	`  Vokabeltrainer() {`
	`  vokabeln = new Vokabelbaum();`
	`  listeAktuell = new Liste();`
	`  }`
	`}`
		2

i	• Erzeugung einer leeren Liste, die *listeAktuell* zugewiesen wird (oder Leeren der Liste *listeAktuell*) • Aufruf von *vokabeln.listeErstellen(jahrgangsstufe, lektion, listeAktuell)*	2
j	Ein Ordnungskriterium sinnvoll zu implementieren, ist mit großem Aufwand verbunden. Der Vorteil von weniger Ebenen geht dadurch verloren, dass während der Suche innerhalb eines Knotens meist mehrere Vergleiche notwendig sind.	2

2a	Mögliche Lösung in Python:
	`def durchlaufen(self, startknoten: Knoten):`
	`  i:int`
	`  for i in range(0, len(self.knoten)):`
	`  self.knoten[i].markierungSetzen(False)`
	`  self.tiefensuche(self.indexGeben(startknoten))`
	
	`def tiefensuche(self, i: int):`
	`  self.knoten[i].markierungSetzen(True)`
	`  j:int`
	`  for j in range(0, len(self.knoten)):`
	`  if self.matrix[i][j] and not`
	`  self.knoten[j].markierungGeben():`
	`  self.tiefensuche(j)`
	
	`def testen(self)-> bool:`
	`  i:int`
	`  for i in range(0, len(self.knoten):`
	`  self.durchlaufen(self.knoten[i])`
	`  j:int`
	`  for j in range(0, len(self.knoten)):`
	`  if not self.knoten[j].markierungGeben():`
	`  return False`
	`  return True`
	
	Mögliche Lösung in Java:
	`void durchlaufen(Knoten startknoten) {`
	`  for (int i=0; i<knoten.length; i++) {`
	`  knoten[i].markierungSetzen(false);`
	`  }`
	`  tiefensuche(indexGeben(startknoten));`
	`}`
	
	`void tiefensuche(int i) {`
	`  knoten[i].markierungSetzen(true);`
	`  for (int j=0; j<knoten.length; j++) {`
	`  if (matrix[i][j] && !knoten[j].markierungGeben()) {`
	`  tiefensuche(j);`
	`  }`
	`  }`
	`}`
	
	`boolean testen() {`
	`  for (int i=0; i<knoten.length; i++) {`
	`  durchlaufen(knoten[i]);`
	`  for (int j=0; j<knoten.length; j++) {`
	`  if (!knoten[j].markierungGeben()) {`
	`  return false;`
	`  }`
	`  }`
	`  }`
	`  return true;`
	`}`
		9

b	Die beiden Einbahnstraßen zwischen der Polizei und der benachbarten Kreuzung würden durch dieselbe Kante repräsentiert, sodass aus dem Graphen nicht hervorginge, dass es sich um zwei Straßen handelt.	1
		40

1a (2 BE)

	rest	zahlBin	zahlDez	stelle
vor der Wiederholung	0	0	13	1
nach dem 1. Durchlauf	1	1	6	10
nach dem 2. Durchlauf	0	1	3	100
nach dem 3. Durchlauf	1	101	1	1000
nach dem 4. Durchlauf	1	1101	0	10000

b (2 BE)

Wenn der Wert in *zahlDez* gerade ist, dann steht nach ganzzahliger Division durch 2 und anschließender Multiplikation mit 2 dieser wieder im Akkumulator; nach Subtraktion von *zahlDez* und Multiplikation mit −1 ergibt sich also 0.

Wenn der Wert in *zahlDez* ungerade ist, dann steht nach ganzzahliger Division durch 2 und anschließender Multiplikation mit 2 der um 1 verminderte Wert von *zahlDez* im Akkumulator; nach Subtraktion von *zahlDez* und Multiplikation mit −1 ergibt sich also 1.

In beiden Fällen ergibt sich der Rest bei der Division durch 2, der in *rest* gespeichert wird.

c (4 BE)

```
            loadi 0
            store rest
            store zahlBin
            loadi 1
            store stelle
wiederhole: load zahlDez
            jmpnp ende
            divi 2
            muli 2
            sub zahlDez
            muli −1
            store rest          // rest = zahlDez mod 2
            mul stelle
            add zahlBin
            store zahlBin       // zahlBin = zahlBin + stelle*rest
            load zahlDez
            divi 2
            store zahlDez       // zahlDez = zahlDez/2
            load stelle
            muli 10
            store stelle        // stelle = stelle*10
            jmp wiederhole
ende:       hold
```

2a (2 BE)

Man erkennt am führenden Bit, ob eine Zahl positiv (Wert 0) oder negativ (Wert 1) ist.

$00010110_2 = 2 + 4 + 16 = 22$ (positiv) und

$10100010_2 = -2^7 + 2^5 + 2 = -128 + 32 + 2 = -94$ (negativ)

b kleinste: $1000\ 0000\ 0000\ 0000_2$ — 1

größte: $0111\ 1111\ 1111\ 1111_2$

Bereich von −32 768 bis 32 767

c — 4

i $10110101_2 + 11111101_2 = (1)10110010_2 = 10110010_2$; Übertrag wird gestrichen
Dezimaldarstellung: −78

ii $1101_2 + 1010_2 = 10111_2$; dezimal: $-3 + (-6) = -9$
Darstellung des Ergebnisses im Zweierkomplement: $0111_2 = +7$

Reicht die Anzahl der Bits aus, um das Ergebnis darzustellen (wie bei i), so spricht man von einem Übertrag, der vernachlässigt werden kann.

Reicht hingegen die Anzahl der Bits nicht aus, um das Ergebnis darzustellen (wie bei ii), so spricht man von einem Überlauf. Die Berechnung im Zweierkomplement ist mit der gegebenen festen Anzahl an Bits nicht korrekt möglich.

Man erkennt, dass ein Überlauf stattgefunden hat, wenn die beiden zu addierenden Zahlen das gleiche Vorzeichen, das Ergebnis der Addition jedoch das andere Vorzeichen hat.

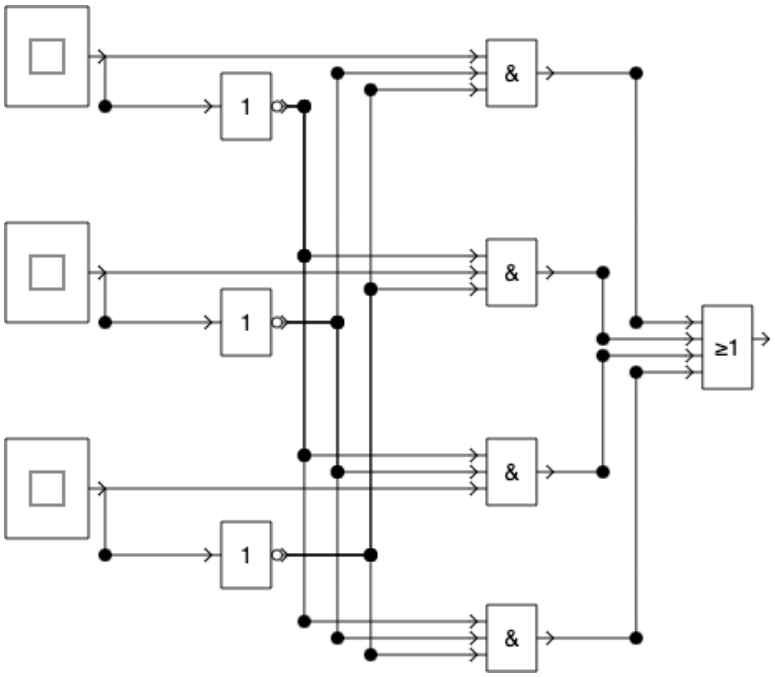
3a — 2

In der folgenden Lösung repräsentiert der Eingabewert 1 die Zustimmung.

x	y	z	R	G
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
1	0	0	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

$G(x, y, z) = \neg x \wedge y \wedge z \vee x \wedge \neg y \wedge z \vee x \wedge y \wedge \neg z \vee x \wedge y \wedge z$

b — 4

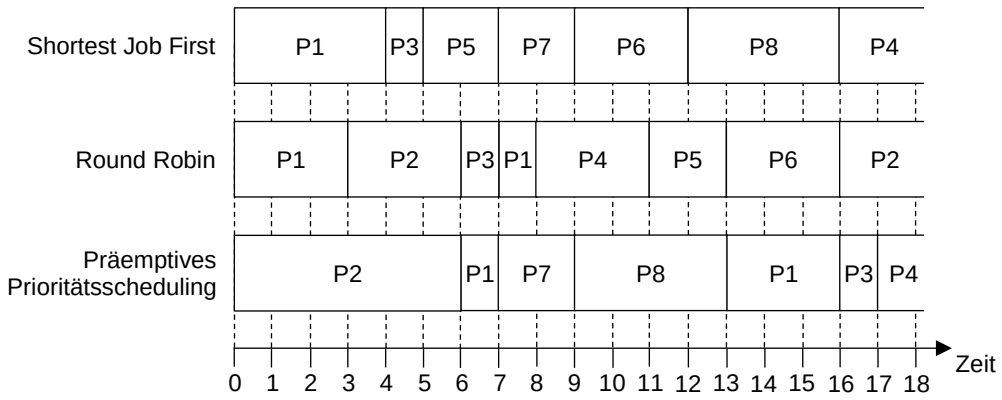


Logikgatter:

NOT:
Negiert den Eingabewert.

AND:
Liefert genau dann 1, wenn alle Eingaben 1 sind.

OR:
Liefert genau dann 1, wenn mindestens eine Eingabe 1 ist.

c	Der gegebene Schaltterm liefert genau dann den Wert 1, wenn mindestens zwei Jurymitglieder für den Kandidaten bzw. die Kandidatin gestimmt haben, wodurch er bzw. sie für die Finalrunde qualifiziert ist.	1
4a	Eine Warteschlange für Prozesse, die zur Ausführung anstehen, steht zur Verfügung. *First Come First Serve*: • Nicht präemptiv, d. h. ein zur Ausführung ausgewählter Prozess wird nicht mehr unterbrochen, sondern zu Ende ausgeführt. • Ein neu eintreffender Prozess reiht sich hinten in die Warteschlage ein; bei freiwerdendem Prozessor wird der am längsten wartende Prozess aktiv. *Round Robin*: • Präemptiv, d. h. die Ausführung eines Prozesses kann unterbrochen werden, sodass ein anderer Prozess zur Ausführung kommt. • Nach einer bestimmten Zeitdauer q wird die Abarbeitung eines aktiven Prozesses unterbrochen, auch wenn dieser noch nicht fertig abgearbeitet ist. Der Prozess wird hinten in die Warteschlange erneut eingereiht und der erste Prozess in der Warteschlange wird aktiviert. Ist ein Prozess vor Ablauf der Zeitdauer q fertig, wird der Prozessor sofort freigegeben und der nächste wartende Prozess aktiviert.	4
b		4
c	Antwortzeit eines Prozesses = Wartezeit + Bedienzeit z. B.: • *Shortest Job First*: beste mittlere Antwortzeit der nicht-präemptiven Strategien, aber nicht fair; lange Prozesse „verhungern“. • *First Come First Serve*: fair, aber nicht ideal bezüglich mittlerer Antwortzeit; auch kurze Prozesse müssen u. U. lange auf ihre Ausführung warten.	2
5a	• Der Verbraucher V möchte Daten entnehmen, wenn der Speicher leer ist. V blockiert den Speicher. • Der Erzeuger E möchte Daten im Speicher ablegen, wenn dieser voll ist. E blockiert den Speicher. • E legt Daten im Speicher ab und gleichzeitig entnimmt V Daten aus dem Speicher, was zu einer Inkonsistenz führen kann.	2

b s: Anzahl an Elementen im Speicher — 3

Erzeuger:	**Verbraucher:**

wiederhole immer
  erzeuge Element
  wiederhole
  lies s
  warte 10 ms
  solange s gleich max
  lege Element im Speicher ab
  erhöhe s um 1
  schreibe s
endeWiederhole
	wiederhole immer
	  wiederhole
	  lies s
	  warte 10 ms
	  solange s gleich 0
	  entnimm Element aus Speicher
	  verringere s um 1
	  schreibe s
	  verbrauche Element
	endeWiederhole

Möglicher Ablauf, der zu einer Inkonsistenz führt (mit max = 10, s = 4):

Erzeuger:	**Verbraucher:**
	V liest Wert von s
E liest Wert von s, *[4<10]*	
E legt Element im Speicher ab	
E erhöht s auf 5	
E schreibt s	
	[4>0], V entnimmt Element aus Speicher
	V verringert s auf 3
	V schreibt s

Die Anzahl der Elemente im Speicher wurde nicht geändert, dennoch wurde s dekrementiert.

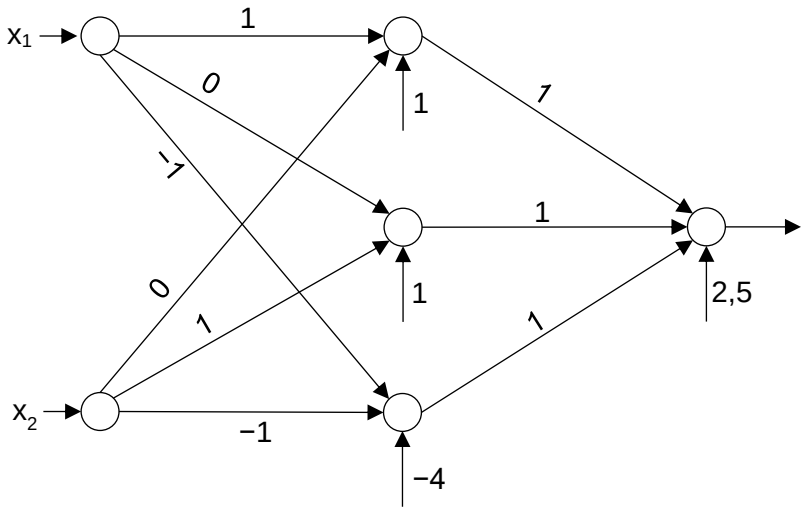
c — 3

```
init(S; 1)     //exklusiver Speicherzugriff
init(b; 0)     // Bestand, Anzahl an Elementen im Speicher
init(f; max)   // Anzahl an freien Speicherplätzen
```

Erzeuger:	**Verbraucher:**

wiederhole immer
  erzeuge Element
  wait(f)
  wait(S)
  lege Element im Speicher ab
  signal(S)
  signal(b)
endeWiederhole
	wiederhole immer
	  wait(b)
	  wait(S)
	  entnimm Element aus Speicher
	  signal(S)
	  signal(f)
	  verbrauche Element
	endeWiederhole

40

Nr.		BE
1a	Auf der Eingabeschicht benötigt man jeweils ein Eingabeneuron für den x_1- und den x_2-Wert, das den Wert unverändert weiterleitet; als Aktivierungsfunktion wählt man deshalb die Identität. In der verborgenen Schicht dient jeweils ein Neuron mit Heavisidefunktion als linearer Separator für eine Dreiecksseite. $x_1 \geq 1 \Rightarrow 1 \cdot x_1 + 0 \cdot x_2 - 1 \geq 0$ $x_2 \geq 1 \Rightarrow 0 \cdot x_1 + 1 \cdot x_2 - 1 \geq 0$ $x_2 \leq -1 \cdot x_1 + 4 \Rightarrow -1 \cdot x_1 - 1 \cdot x_2 + 4 \geq 0$ Auf der Ausgabeschicht benötigt man ein weiteres Neuron zur Realisierung der UND-Funktion. 	6
b	Eindringlinge könnten bei unverschlüsselter Kommunikation über das Internet die Zugangsdaten von Mitarbeitenden ausspähen und sich darüber von außen Zugriff auf das System verschaffen; in der Folge könnten sie Systemparameter so manipulieren, dass der Produktionsprozess unterbrochen wird oder es zu einer fehlerhaften Produktion bis hin zur Zerstörung des Systems kommt. Mögliche Maßnahmen sind beispielsweise: • Zugriffsschutz: z. B. durch hinreichend sichere Passwörter (12 Zeichen, Alphabet mit Groß- und Kleinbuchstaben, Sonderzeichen, Ziffern), Zwei-Faktor-Authentifizierung oder Biometriedaten. • Technische Maßnahmen: z. B. Verschlüsselung, VPN • Vieraugenprinzip: Mitarbeitende sind niemals allein verantwortlich für die Steuerung der Anlage.	2
2a	In der Lernphase erhält das künstliche neuronale Netz, das zu Beginn mit zufälligen Gewichten versehen ist, Trainingsdaten mit korrekt zugeordneten Sollwerten der Ausgabe. Nacheinander wird für jedes Trainingsdatum der Ausgabewert des künstlichen neuronalen Netzes berechnet. Entspricht der berechnete Ausgabewert nicht dem Sollwert, erfolgt die Anpassung der Gewichte (Backpropagation).	2
b	Sensoren werden zur Messung von Temperatur und pH-Wert benötigt. Zur Steuerung der Konzentration dienen als Aktoren Ventile, Pumpen o. ä., für die Temperatur z. B. Heizung/Kühlung.	1

c (4)

x_1	x_2	**Istwert**	Sollwert
1	0	**0**	1
0	0	**0**	0
0	1	**0**	1
1	1	**1**	1
2	1	**1**	0

Die Kostenfunktion dient zur Quantifizierung des Modellfehlers. Ein Beispiel für eine Kostenfunktion ist der mittlere quadratische Fehler, der sich nach der Formel

$MSE = \frac{1}{n}\sum(Istwert - Sollwert)^2$

berechnet. Im Beispiel ergibt sich:

$MSE = \frac{1}{5}((-1)^2 + 0^2 + (-1)^2 + 0^2 + 1^2) = \frac{3}{5}$

d (1)

Z. B. Variation der Modellparameter: Erhöhung der Anzahl der Neuronen pro Schicht, Erhöhung der Anzahl der Schichten

Weitere Möglichkeiten wären etwa die Wahl eines geeigneteren Modells, die Variation der Lerndaten oder die mehrfache Verwendung der Lerndaten.

e (2)

In der Sandboxing-Phase wird überprüft, ob die neue Software korrekt arbeitet. Der Mitarbeiter, der bisher die Systemparameter überwacht hat, macht dies weiterhin und überprüft zusätzlich, ob die Software genauso handeln würde wie er. Die Software selbst wirkt sich in dieser Phase noch nicht auf das System aus. Nach erfolgreichen Tests kann die Software in den Produktivbetrieb gehen.

3a (4)

Das k-Means-Verfahren ist ein Clustering-Verfahren. Zunächst wird der Wert von k und damit die Anzahl der Cluster festgelegt. In der gegebenen Aufgabe ist k die Anzahl der unterschiedlichen Hosengrößen, also k = 3. Zu Beginn werden k zufällige Wertepaare aus Weite und Länge als Clusterzentren gewählt. Anschließend wird jedes Wertepaar der Umfrage dem nächstgelegenen Clusterzentrum zugeordnet. Für jedes Clusterzentrum wird nun der geometrische Mittelpunkt seiner Messwerte ermittelt und das Clusterzentrum dorthin gesetzt. Das Verfahren wird so lange wiederholt, bis sich die Cluster nicht mehr verändern. Die Wertepaare der Clusterzentren, die sich am Ende ergeben, entsprechen den Weiten und Längen für die Hosengrößen S, M und L. Das Clusterzentrum, zu dem das jeweilige Wertepaar eines Mitarbeiters bzw. einer Mitarbeiterin gehört, legt damit die Größe der Hose fest, die der Mitarbeiter bzw. die Mitarbeiterin erhält.

Dabei handelt es sich um ein unüberwachtes Lernverfahren, da das Verfahren selbständig Muster aus den Daten erkennt. Die gewünschten Ausgabedaten sind nicht vorgegeben.

b (4)

Name	Curie	Dalton	Nobel	Pauling	Mariotte	Bunsen	Hahn	Haber
Weite	35	35	38	33	40	42	31	28
Länge	32	31	35	31	32	32	35	30
Zentrum	A	A	B	A	B	B	C	A

Neue Clusterzentren:

$x_A = (35 + 35 + 33 + 28) / 4 = 32{,}75$

$y_A = (32 + 31 + 31 + 30) / 4 = 31$

$x_B = (38 + 40 + 42) / 3 = 40$

$y_B = (35 + 32 + 32) / 3 = 33$

A (32,75 | 31); B (40 | 33); C (31 | 35)

c (4)

Mögliche Aspekte:

	Unternehmen	Beschäftigte
Chancen	+ Günstige Produktion + Daten aller Mitarbeitenden werden einbezogen + Corporate Identity + Daten können mit anderen (Gesundheit!) zusammengeführt werden.	+ Bereitgestellte Kleidung
Risiken	- Beschäftigte unzufrieden	- passt nicht jedem, oft zu weit oder eng - Zweckentfremdung der Daten für betriebliche Entscheidungen

Je größer k gewählt wird, desto mehr unterschiedliche Hosengrößen müssen produziert werden und desto teurer wird die Produktion der Kleidung, aber desto besser passt sie in der Regel auch den Mitarbeitenden.

Als Datenschutzbeauftragter sollte man Bedenken angesichts der nichtanonymisierten Datensammlung äußern. Es gilt der Grundsatz der Datensparsamkeit. Auch sollte der Zweck der Datenverarbeitung klar geregelt werden, um beispielsweise weitergehende Analysen sowie die Zusammenführung mit anderen Daten zu verhindern.

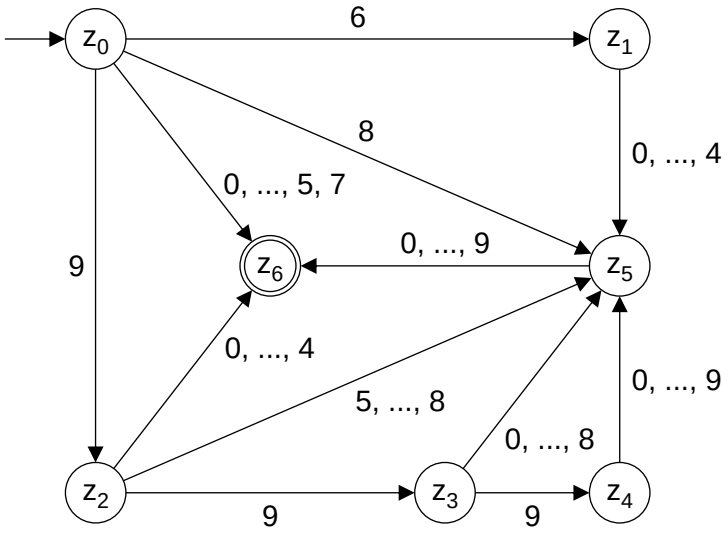
4a (3)

```
leitet(curie, zentralmanagement).
leitet(dalton, fertigung).
leitet(nobel, entwicklung).
leitet(pauling, buchhaltung).
mitarbeiter(boyle, fertigung).
direkt_übergeordnet(zentralmanagement, fertigung).
direkt_übergeordnet(zentralmanagement, entwicklung).
direkt_übergeordnet(zentralmanagement, buchhaltung).
```

b	übergeordnet(X, Y) :-
	    direkt_übergeordnet(X, Y);
	    direkt_übergeordnet(X, Z),
	    übergeordnet(Z, Y).
	
	vorgesetzte(X, Y) :-
	    leitet(X, Z),
	    mitarbeiter(Y, Z);
	    leitet(X, Z),
	    übergeordnet(Z, A),
	    leitet(Y, A);
	    leitet(X, Z),
	    übergeordnet(Z, A),
	    mitarbeiter(Y, A).
		6

c	vorgesetzte(curie, Y).	1
		40

IV

Nr.		BE
1a		5
b	Gruppennummer = Ziffer0bis4 | '5' | '7' | '8' Ziffer | '9' Ziffer0bis4 | '6' Ziffer0bis4 Ziffer | '9' Ziffer5bis8 Ziffer | '9' '9' Ziffer0bis8 Ziffer | '9' '9' '9' Ziffer Ziffer Ziffer0bis4 = '0' | '1' | '2' | '3' | '4'; Ziffer5bis8 = '5' | '6' | '7' | '8'; Ziffer0bis8 = Ziffer0bis4 | Ziffer5bis8; Ziffer = Ziffer0bis8 | '9';	3

c | 9

Eingabealphabet: $X = \{0; 1; 2; \ldots; 9; \_\}$

Bandalphabet $B = \{0; 1; 2; \ldots; 9; \_; x; \square\}$

Zustandsmenge $Z = \{z_1; z_2; z_3; z_E\}$

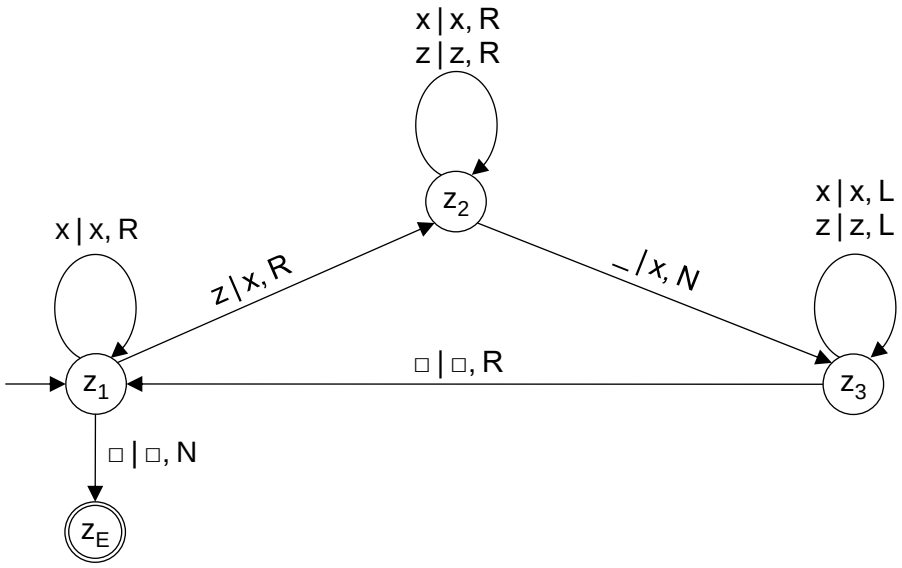
Startzustand z_1

Endzustand z_E

Leerzeichen $\square$

Zustandsübergangsfunktion δ kann mit $z \in \{0; 1; 2; \ldots; 9\}$ auf eine der drei folgenden Arten angegeben werden:

- Als Zustandsübergangsdiagramm:



- Als Tabelle:

	x	z	□	_
z_1	(z_1, x, R)	(z_2, x, R)	$(z_E, \square, N)$	
z_2	(z_2, x, R)	(z_2, z, R)		(z_3, x, N)
z_3	(z_3, x, L)	(z_3, z, L)	$(z_1, \square, R)$	
z_E				

- Mit Funktionswerten:

$\delta(z_1, x) = (z_1, x, R)$

$\delta(z_1, z) = (z_2, x, R)$

$\delta(z_2, x) = (z_2, x, R)$

$\delta(z_2, z) = (z_2, z, R)$

$\delta(z_2, \_) = (z_3, x, N)$

$\delta(z_3, x) = (z_3, x, L)$

$\delta(z_3, z) = (z_3, z, L)$

$\delta(z_3, \square) = (z_1, \square, R)$

$\delta(z_1, \square) = (z_E, \square, N)$

2a	zweiHoch(6) = zweiHoch(3) * zweiHoch(3) = 2 * zweiHoch(1) * zweiHoch(1) * 2 * zweiHoch(1) * zweiHoch(1) = 2 * 2 * zweiHoch(0) * zweiHoch(0) * 2 * zweiHoch(0) * zweiHoch(0) * 2 * 2 * zweiHoch(0) * zweiHoch(0) * 2 * zweiHoch(0) * zweiHoch(0) = 2 * 2 * 1 * 1 * 2 * 1 * 1 * 2 * 2 * 1 * 1 * 2 * 1 * 1 = 64	2
b	zweiHoch(0): 1 Aufruf zweiHoch(1): 3 Aufrufe zweiHoch(3): 7 Aufrufe zweiHoch(6): 15 Aufrufe zweiHoch(12): 31 Aufrufe Bei Verdoppelung des Eingabewerts verdoppelt sich die Anzahl der Methodenaufrufe und erhöht sich zusätzlich um 1. Begründung: *zweiHoch(2n)* ruft zweimal *zweiHoch(n)* auf. Da der Aufruf von *zweiHoch(2n)* mitgezählt wird, ergibt sich der genannte Zusammenhang. Laufzeitaufwand in Landau-Notation: O(n).	4
c	Der in der Methode *zweiHochB* gegebene Algorithmus weist einen linearen Laufzeitaufwand auf: O(n). Der Vermutung aus Teilaufgabe 2b zufolge würden demnach beide Algorithmen denselben Laufzeitaufwand aufweisen. Dennoch ist der Methode *zweiHochB* der Vorzug zu geben, da sie im Vergleich zur Methode *zweiHoch* bei gleicher Eingabe mit weniger rekursiven Aufrufen, weniger Berechnungen und weniger Vergleichen auskommt.	3
3a	Für $b = 40$ ergibt sich folgende Lösung: 13 + 13 + 14 = 40 Für $b = 43$ betrachtet man die Summen: Mit drei Zahlen maximal bildbar: 14 + 14 + 14 < 43 Mit vier Zahlen minimal bildbar: 13 + 13 + 13 + 14 > 43 Alternativ kann durch Betrachtung aller relevanten Fälle ausgeschlossen werden, dass es eine Lösung gibt: 13 < 43 13 + 13 < 43 13 + 13 + 13 < 43 13 + 13 + 13 + 14 > 43 14 + 13 < 43 14 + 13 + 13 < 43 14 + 13 + 13 + 13 > 43 14 + 14 < 43 14 + 14 + 13 < 43 14+14+13+13 > 43 14+14+14 < 43 14+14+14+13 > 43	2

b	b eine ganze Zahl a ein Feld positiver ganzer Zahlen n die Länge des Feldes Iterativ: Methode rucksack_iterativ erstelle ein zweidimensionales boolesches Feld w der Größe $2^n \times n$ belege jede Zeile mit einer der 2^n möglichen Verteilungen von wahr und falsch auf n Plätze zähle von i = 0 bis $2^n - 1$ belege Variable sum mit 0 zähle von j = 0 bis n − 1 wenn w[i][j] wahr dann addiere zu sum den Wert von a[j] endeWenn endeZähle wenn sum und b gleich dann gib wahr zurück endeWenn endeZähle gib falsch zurück endeMethode Rekursiv: Methode rucksack_rekursiv(b, a, n) wenn n gleich 1 dann wenn b gleich a[0] dann gib wahr zurück sonst gib falsch zurück endeWenn sonst gib rucksack_rekursiv(b − a[n − 1], a, n − 1) oder rucksack_rekursiv(b, a, n − 1) zurück endeWenn endeMethode	8
c	Man wählt beispielsweise: $a_1 = a_2 = \ldots = a_b = 13$, $a_{b+1} = a_{b+2} = \ldots = a_{2b} = 14$ *und* $a_{2b+1} = a_{2b+2} = \ldots = a_{3b} = 17$. Hat man einen Lösungsalgorithmus für das beschriebene Rucksackproblem, dann kann man überprüfen, ob mit einer Auswahl aus $a_1, a_2, \ldots, a_{3b}$ eine Summe gebildet werden kann, deren Wert *b* ist.	2
d	Die Komplexitätsklasse P enthält alle Entscheidungsprobleme, die in polynomialer Laufzeit für deterministische Turingmaschinen lösbar sind. Die Probleme der Komplexitätsklasse NP sind zwar in polynomialer Laufzeit entscheidbar, jedoch wird hierfür ein (nicht realisierbares) nichtdeterministisches Maschinenmodell eingesetzt.	2
		40