FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
Professur für Didaktik der Informatik

Lösungs*vorschläge* zu den Staatsexamina:
Theoretische Informatik und Algorithmik

Herbst 2020

1 Thema

Teilaufgabe 1: Theoretische Informatik

Aufgabe 1

(a)

(b)

(c)

(d)

(e)

(f)

(g)

(h)

Aufgabe 2

(a)

(b)

Aufgabe 3

(a)

(b)

Aufgabe 4

(a)

(b)

(c)

Aufgabe 5

Teilaufgabe 2: Algorithmen

Aufgabe 1

(a)

Durchlauf	n	m	x	y
1	3	2	3	0
2	3	2	2	1
3	3	2	1	2

Rückgabewert x = 1

(b) Nein, `countup` wird immer terminieren für jegliche Eingabewerte für `n` und `m`, da die Abbruchbedingung der `while`-Schleife, `y<m`, irgendwann zu `false` ausgewertet wird, denn in jedem Schleifendurchlauf wird `y` inkrementiert.

(c) `n` spielt für die Laufzeit keine Rolle. Es ist bloß der Anfangswert für den Rückgabewert `x`, der in der Schleife herabgezählt wird. Entscheidend ist `m`, das die Obergrenze der Schleifendurchläufe bildet. Die Schleife wird also `m`-mal durchlaufen.
Die restlichen Zuweisungen erfolgen jeweils in O(1).
Folglich beträgt die Laufzeit O(m).

(d)

Durchlauf	n	m	x	y
1	3	2	3	0
2	3	2	2	1
3	3	2	1	2
4	1	2	1	0
5	1	2	0	1
6	1	2	-1	2
7	0	2	-1	2

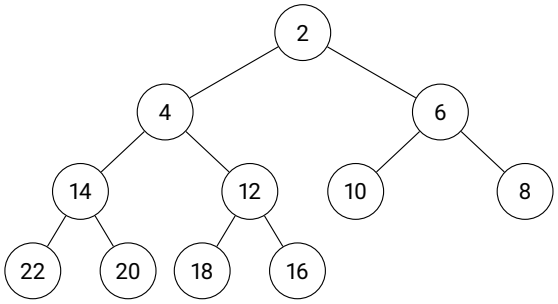
Rückgabewert x = -1

(e) Nein. Das liegt daran, dass die Bedingung der `while`-Schleife, `n<0`, irgendwann zu `false` ausgewertet wird. Bei jeder Iteration der Schleife wird entweder `y` inkrementiert oder `n` halbiert. Beide Operationen werden letzlich dazu führen, dass $n \leq 0$ wird.

(f) Die Laufzeit wird hier bedingt durch die Abbruchbedingung der `while`-Schleife, `n>0`. Innerhalb der Schleife wird jedoch die Verkleinerung (bzw. *Halbierung*) von `n` immer nur ausgeführt nach jeweils `m` Iterationen. Da `n` immer halbiert wird, trägt die Abbruchbedingung `n>0` einen logaritmischen Laufzeitfaktor bei.
Wir kommen also auf eine Laufzeit von: O(m·log(n)).

Aufgabe 2

(a) Der Binärbaum:



(b) `magic` gibt in diesem Falle `true` zurück.

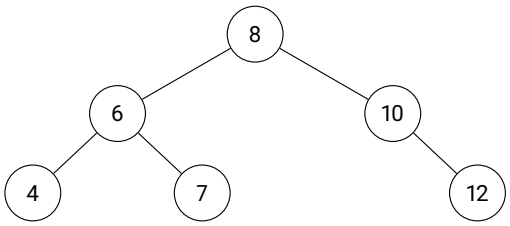
(c) Das Feld `A` repräsentiert einen Baum. `magic` prüft den Teilbaum von `A`, der den Knoten an der Stelle `i` als Wurzelknoten hat, auf Min-Haldeneigenschaft. Das bedeutet, dass die Werte der Kindknoten eines jeden Elternknotens mindestens so groß sind wie der Wert des Elternknotens.

(d) Wenn das Feld aufsteigend sortiert ist, dann ist die Min-Haldeneigenschaft erfüllt, da jeder Sprung zu einem Kindknoten (i * 2 + 1 bzw. 2) einen mindestens ebenso großen Kindknoten offenbart. `magic` prüft die Min-Haldeneigenschaft, und wird somit immer `true` liefern.

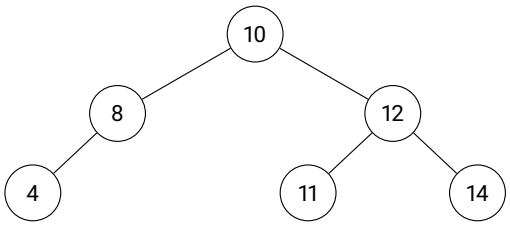
(e) `true:` $\{1,2,3\}$
`true:` $\{1,3,5\}$
`false:` $\{3,2,1\}$

(f) Die entfernte bedingte Anweisung bildet den Basisfall der Rekursion. Er prüft, ob wir an einem Blattknoten angelangt sind. Ein Blattknoten verfügt über keine Kinder mehr, erfüllt also automatisch die Min-Haldeneigenschaft. Damit sind keine weiteren Rekursionsaufrüfe notwendig.
Ohne diesen Basisfall hätte die Rekursion also keine Abbruchbedingung, und würde weiterlaufen bis irgendwann eine `ArrayIndexOutOfBoundsException` einträte, bedingt dadurch, dass mit jedem Aufruf von magic der Wert von `i` erhöht wird, der die Grundlage bildet für den Zugriff auf die Feldelemente.

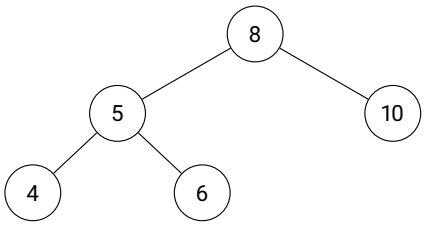
(g) i.) Nach Rechtsrotation:



ii.) Nach Rechts-Links-Rotation:



iii.) Nach Rechts-Links-Rotation:



Aufgabe 3

(a)

A	**B**	**C**	**D**	**E**	**Prioritätswarteschlange**
0	∞	∞	∞	∞	[A]
0	15	5	∞	12	[C,E,B]
0	13	5	10	8	[E,D,B]
0	10	5	9	8	[D,B]
0	10	5	9	8	[B]
0	10	5	9	8	[-]

(b) A → C → E → B

Aufgabe 4

(a) Die anfänglichen Initialisierungen und Wertzuweisungen erfolgen alle in O(1). Die `while`-Schleife hingegen wird `n`-mal ausgeführt, da `i` bei 0 beginnt und so lange läuft, bis `i=n`. Die Operationen innerhalb der Schleife erfolgen ebenfalls alle in O(1). Wir kommen also auf eine Laufzeit von O(n).

Die Frage nach einer „worst-case"-Laufzeit in der Aufgabenstellung ist irreführend, da es nur einen ganzzahligen Eingabewert von `n` gibt. Eine Unterscheidung zwischen einem besten oder schlimmsten Fall ist hier also völlig irrelevant. Sie kann aber bspw. eine Rolle spielen bei Sortierverfahren, deren Eingabewerte Reihungen sind, die unterschiedlich gut vorsortiert sein können.

(b) Die `while`-Schleife wird `n`-mal ausgeführt. Innerhalb der Schleife itereieren wir in der äußeren `for`-Schleife bis `y`, das den Wert `n` zugewiesen bekommt, wobei `n` nach jedem `while`-Schleifendurchlauf dekrementiert wird. Mit jeder while-Iteration vollziehen wir also je um eins weniger Durchläufe der äußeren Schleife (stellt man sich die Anzahl der Iterationen bildlich wie eine Tabelle vor, haben wir hier also kein Quadrat, sondern eine Dreieck). Damit durchlaufen wir die äußere `for`-Schleife (die innerhalb der `while`-Schleife liegt) also $\frac{n}{2}$-mal.

Die *innere* `for`-Schleife durchlaufen wir dagegen `i`-mal. `i` richtet sich nach `y`, das sich wiederum nach `n` richtet, wobei `i` in jedem Durchlauf der äußeren Schleife inkrementiert wird. Die innere Schleife wird also $\frac{n}{2} \cdot \frac{n}{2} = \frac{n}{4}$-mal durchlaufen, wir kommen also auf eine Gesamtlaufzeit von O($n \cdot \frac{n}{2} \cdot \frac{n}{4}$). Das bedeutet also eine Laufzeit von O(n^3).

Auch hier können wir die Frage nach der „worst-case"-Laufzeit ignorieren.

(c) Mit jedem Rekursionsaufruf wird der Eingabewert halbiert. Es sind also log(n) Rekursionsaufrufe erforderlich. Innerhalb eines Rekursionsaufrufes geschieht neben einem Rekursionsaufruf (in O(1)) eine Quadrierung (in O(1)) und eine Addition (in O(1)).

Damit kommen wir auf eine Gesamtlaufzeit von O(log(n)).

Aufgabe 5

(a)

Fach	Schlüssel k
0	E – U
1	?
2	
3	
4	Y – ! – A
5	B
6	
7	D

(b)

Fach	k	Durchlaufene Fächer
0	E	0
1	U	0 → 1
2	D	7 → 0 → 1 → 2
3	?	2 → 3
4	Y	4
5	B	5
6	!	4 → 5 → 6
7	A	4 → 5 → 6 → 7

(c)

Fach	k	Verwendete Streufunktion
0	E	h_0
1	A	h_1
2	U	h_1
3	!	h_1
4	Y	h_0
5	B	h_0
6	?	h_2
7	D	h_0

2 Thema

Teilaufgabe 1: Theoretische Informatik

Aufgabe 1

(a)

(b)

(c)

Aufgabe 2

(a)

(b)

Aufgabe 3

(a)

(b)

(c)

(d)

Aufgabe 4

(a)

(b)

Teilaufgabe 2: Algorithmen

Aufgabe 1

(a)

Schlüssel	Inhalt	Schlüssel	Inhalt
A	Alfons	N	Michael
B	Bert	O	Norbert
C	Christel	P	Matthias
D	Dirk	Q	
E	Adalbert	R	
F	Edith	S	
G	Emil	T	
H		U	
I	Inge	V	
J		W	
K	Kurt	X	
L		Y	
M	Martin	Z	

(b)

Schlüssel	Inhalt	Schlüssel	Inhalt
A		N	Nathan
B	Brigitte	O	
C		P	
D	Dirk, Diana	Q	
E	Elmar, Emanuel	R	
F		S	Sebastian
G		T	Thomas, Torsten
H		U	
I	Inge	V	
J		W	
K	Kurt, Katrin, Karolin	X	
L		Y	
M		Z	

Aufgabe 2

(a) Um die Aussage $f \in O(n^3)$ zu beweisen, müssen wir zeigen, dass es Konstanten c und n_0 gibt, so dass für alle $n \geq n_0$ gilt:
$f(n) \leq c \cdot n^3$

Die quadratischen und logarithmischen Terme in $f(n)$ können wir ignorieren, da sie gegenüber dem kubischen Term vernachlässigbar sind, wenn n groß genug ist. Wir können $f(n)$ also abschätzen durch:
$f(n) \leq 2 \cdot n^3$

Jetzt können wir eine Konstante $c = 2$ wählen und $n_0 = 1$ setzen:
$\Rightarrow \forall n \geq n_0 : f(n) \leq 2 \cdot n^3 \leq c \cdot n^3$
$\Rightarrow f \in O(n^3)$

Für ausreichend große n ist $f(n)$ also beschränkt durch n^3.
f ist also enthalten in der Komplexitätsklasse O(n^3).

(b) Um die Aussage $f(n) = 4^n \notin O(2^n)$ zu beweisen, müssen wir zeigen, dass es *keine* Konstanten c und n_0 gibt, so dass für alle $n \geq n_0$ gilt:
$f(n) \leq c \cdot 2^n$

Wir können dies zeigen durch einen **Widerspruchsbeweis**:
Egal wie groß wir die Konstante c setzen: obige Ungleichung wird niemals gelten für *alle* $n > n_0$, denn allerspätestens sobald $n > c$ ist, wird klar:
$4^n > c \cdot 2^n$
$\Rightarrow f(n) \notin O(2^n)$.

f ist also nicht enthalten in der Komplexitätsklasse O(2^n).

(c) ???

(d) ???

Aufgabe 3

(a)

i	0	1	2	3	4	5
A[i]	10	20	15	40	25	35

(b) 1.

i	0	1	2	3	4	5
A[i]	15	20	35	40	25	-

Knoten 10 wurde gelöscht. Knoten 35 ist an die Spitze gerückt. Anschließend hat er den Platz gewechselt mit Knoten 15, um die Min-Haldeneigenschaft wiederherzustellen.

2.

i	0	1	2	3	4	5
A[i]	20	25	35	40	-	-

Knoten 15 wurde gelöscht.
Knoten 25 ist an die Spitze gerückt. Anschließend hat er den Platz gewechselt mit Knoten 20, um die Min-Haldeneigenschaft wiederherzustellen.

3.

i	0	1	2	3	4	5
A[i]	25	40	35	-	-	-

Knoten 20 wurde gelöscht.
Knoten 40 ist an die Spitze gerückt. Anschließend hat er den Platz gewechselt mit Knoten 25, um die Min-Haldeneigenschaft wiederherzustellen.

4.

i	0	1	2	3	4	5
A[i]	35	40	-	-	-	-

Knoten 25 wurde gelöscht.
Knoten 35 ist an die Spitze gerückt. Anschließend hat er den Platz gewechselt mit Knoten 40, um die Min-Haldeneigenschaft wiederherzustellen.

5.

i	0	1	2	3	4	5
A[i]	40	-	-	-	-	-

Knoten 35 wurde gelöscht.
Knoten 40 ist an die Spitze gerückt.

6.

i	0	1	2	3	4	5
A[i]	-	-	-	-	-	-

Knoten 40 wurde gelöscht.

Aufgabe 4

(a) i.) Ein Gutschein G im Wert von 80 kann eingelöst werden mit $I = \{1,2,3\}$.

ii.) 81.

(b)

```
1   public static boolean[][] tErzeugen(int[] waren,
2                                       int G       ) {
3       int n = waren.length;
4       boolean[][] tabelle = new boolean[n+1][G+1];
5
6       // Vorbefuelle die Eintraege fuer k=0 und i=0:
7       for (int i = 0; i <= n; i++)
8          tabelle[i][0] = true;
9       for (int k = 1; k <= G; k++)
10         tabelle[0][k] = false;
11
12      // Berechne die Eintraege fuer i>0 und k>0:
13      for (int i = 1; i <= n; i++)
14          for (int k = 1; k <= G; k++) {
15              tabelle[i][k] = tabelle[i-1][k];
16              if (k >= waren[i-1])
17                  tabelle[i][k] ||
18                  tabelle[i-1][k-waren[i-1]];
19          }
20      return tabelle;
21  }
```

Das Lösungsverfahren entspricht 1:1 dem so genannten „Rucksack-Problem“, das in der Aufgabenstellung, vermutlich zwecks Ablenkung, einfach umgemünzt wird zum „Gutschein-Problem“. Das Verfahren funktioniert wie folgt:

Zunächst definieren wir die Methodensignatur so, dass sie eine Reihung aus Waren `waren` und einen Gutschein mit Wert `G` erhält, und eine zweidimensionale `boolean`-Reihung zurückgibt. Die Tabelle wird iterativ von oben links nach unten rechts aufgebaut.
Die Spalten stellen die jeweiligen Werte des Gutscheins dar, von 0 bis `G`.
Die Zeilen repräsentieren die Waren, die eingelöst werden mit dem Gutschein.
Zuerst wird die erste Zeile und die erste Spalte vorbefüllt wie folgt:

- Die erste Spalte wird vorbefüllt mit `true`-Werten, denn ein Gutschein mit Wert 0 (erste Spalte) kann vollständig eingelöst werden mit einer leeren Menge an Waren.
- Die erste Zeile wird vorbefüllt mit `false`-Werten, außer der Zelle [0][0]. Denn dort lösen wir ja keine Ware ein. Für einen Gutschein mit Wert 0 mag

das noch zu einem vollständig entwerteten Gutschein führen, nicht mehr jedoch für einen Gutschein mit Wert >0.

Anschließend wird die Tabelle befüllt, indem man für jeden Gutscheinwert `k` und jede Ware i entscheidet, ob man den Gutschein einlösen möchte gegen die `i`-te Ware, oder eben nicht. Wenn der Preis der `i`-ten Ware größer ist als der Gutscheinrestwert `k`, kann man sie nicht mehr einlösen. Man kann allerdings noch prüfen, ob es möglich ist, den Gutscheinwert `k` vollständig zu entwerten mit den vorherigen Waren bis zur `i`-1-ten Ware. Falls das möglich ist, kann man die `i`-te Ware entweder weglassen oder einlösen, um den Gutscheinwert `k` vollständig zu entwerten.

Die Laufzeit dieses Algorithmus beträgt O(n*G), da man die Tabelle iterativ befüllt von oben links nach unten rechts, wobei man für jeden Gutscheinwert und jede Ware eine konstante Anzahl von Operationen ausführt.

Jeder Eintrag in `tabelle`, der hiermit auf `true` gesetzt wurde, löst das Gutschein-Problem.

Aufgabe 5

(a) 3, 9, 11, 20, 10, 8, 4

(b)

```
1   public static boolean enthaelt(int[] r, int s) {
2       int unten = 0;
3       int oben  = r.length;
4       while (unten < oben) {
5           int mitte = unten + (oben - unten) / 2;
6           int wert  = r[mitte];
7           if (wert == s) return true;
8           if (istLinksVon(s, wert))
9               oben  = mitte - 1; // -1, weil wir die
                    Mitte schon ausschliessen koennen
10          else
11              unten = mitte;
12      }
13      return false;
14  }
15
16  public static boolean istLinksvon(int a, int b) {
17      boolean aGerade = (a % 2 == 0);
18      boolean bGerade = (b % 2 == 0);
19      if ( aGerade &&  bGerade) return a > b;
20      if (!aGerade && !bGerade) return a < b;
21      return !aGerade;
22  }
```

Die Methode `enthält` erhält eine Reihung und ein Suchelement.
Um eine Lautzeit zu erhalten, die besser ist als O(n) (Linearzeit), bleibt uns nichts anderes übrig als eine binäre Suche zu implementieren, die in O(log(n)) stattfindet. Wir legen also zunächst den Suchbereich fest über die gesamte Reihung.

Innerhalb der `while`-Schleife wird der Suchbereich dann iterativ verkleinert.
Wir wählen als Suchpunkt die Mitte und prüfen, ob wir das Suchelement dort finden. Ist das nicht der Fall, prüfen wir, ob das Suchelement links von dem gerade angeschauten Prüfwert liegt, damit wir unseren Suchbereich eingrenzen zu können.

Dafür wird hier zwecks Eleganz eine Hilfsfunktion `istLinksVon` definiert:
Sie erhält zwei Werte und prüft zunächst, ob sie gerade sind. Die Frage, ob Wert `a` links von Wert `b` liegt in einer `oddAscendingEvenDescending`-Folge, lässt sich nämlich aufgrund der Totalordnung der Folge beantworten in Abhängigkeit davon, ob sie gerade oder ungerade sind. Weiß die Methode das also, vollzieht sie folgende Fallunterscheidung:

1. Sind beide Werte gerade, fragt sie weiter, ob der erste Wert *größer* ist als der zweite. Denn ist er das, liegt er links von `b`, da der gerade Anteil absteigend sortiert ist.
2. Sind beide Werte ungerade, fragen sie dagegen weiter, ob der erste Wert *kleiner* ist als der zweite. Denn ist er das, liegt er links von `b`, da der ungerade Teil absteigend sortiert ist.
3. In allen anderen Fällen (`a` gerade, `b` ungerade / `a` ungerade, `b` gerade) hängt die Antwort darauf, ob der Suchwert links gelegen ist vom Prüfwert, lediglich ab von der Frage: ist das Suchelement ungerade? Denn ist es das, muss das Prüfelement gerade sein. Und dann müssen wir auf jeden Fall links weitersuchen. Im Gegenfall verhält es sich natürlich umgekehrt.

So können wir mit jeder Iteration den Suchbereich weiter eingrenzen. Finden wir irgendwann das Suchelement, gibt die Suche `true` zurück. Wird der Suchbereich irgendwann 0, da Untergrenze und Obergrenze übereinstimmen (`unten = oben`), endet die Suche erfolglos, und gibt entsprechend `false` zurück.

(c)
1. Suchbereich: 0 bis 6[1]
   Mitte = 3
   Prüflement = 8 → Nicht gefunden!
   Liegt Suchelement 4 links von Prüfelement 8? Nein.
   Suche wird fortgesetzt im Bereich rechts von 8.
2. Suchbereich: 4 bis 6
   Mitte = 5
   Prüfelement = 2 → Nicht gefunden!
   Liegt Suchelement 4 links von Prüfelement 2? Ja.
   Suche wird fortgesetzt im Bereich links von 2.
3. Suchbereich: 4 bis 4
   Mitte = 4
   Prüfelement = 4 → Gefunden!

(d) Wie bereits erwähnt verwenden wir die binäre Suche. Da diese den Suchbereich in jedem Iterationsschritt um die Hälfte verringert, findet sie statt in logarithmischer Zeit, hat also den Aufwand O(log(n)).

[1] Die Obergrenze ist hier bewusst definiert als `r.length`, und nicht `r.length-1`, liegt also außerhalb der Reihung. Das ist aber einerseits unbedenklich, da auf diese Position nicht zugegriffen wird, und andererseits wichtig, da es verhindert, dass bei einer einelementigen Reihung die Suche sofort beendet würde, da Ober- und Untergrenze gleich wären, nämlich: 0.