Prüfungsteilnehmer	Prüfungstermin	Einzelprüfungsnummer

Kennzahl: _____

Kennwort: _____

Arbeitsplatz-Nr.: _____

Herbst 2017

66115

Erste Staatsprüfung für ein Lehramt an öffentlichen Schulen

— Prüfungsaufgaben —

Fach: **Informatik (vertieft studiert)**

Einzelprüfung: **Theoret. Informatik, Algorithmen**

Anzahl der gestellten Themen (Aufgaben): 2

Anzahl der Druckseiten dieser Vorlage: 14

Bitte wenden!

Thema Nr. 1
(Aufgabengruppe)

Es sind alle Aufgaben dieser Aufgabengruppe zu bearbeiten!

Aufgabe 1:
Myhill Nerode

Für zwei Wörter $x, y \in \Sigma^*$ und eine Sprache $L \subseteq \Sigma^*$ gilt $x \equiv_L y$ genau dann, wenn für jedes Wort $z \in \Sigma^*$ gilt: $(xz \in L \iff yz \in L)$. Die Relation $\equiv_L$ ist eine Äquivalenzrelation.

Der Satz von Myhill und Nerode sagt, dass eine Sprache genau dann regulär ist, wenn $\equiv_L$ endlich viele Äquivalenzklassen hat.

a) Wir bezeichnen mit $\#_a(w)$ die Anzahl Vorkommen des Alphabetsymbols a in der Zeichenkette w. Zeigen Sie mit Hilfe des Satzes von Myhill und Nerode, dass die Sprache

$$L_1 = \{w \mid \#_a(w) = \#_b(w)\}$$

über dem Alphabet $\{a, b\}$ nicht regulär ist.

b) Geben Sie alle Äquivalenzklassen der Äquivalenzrelation $\equiv_{L_2}$ an. Hier ist $L_2 = L((a+b)^*a)$ (über dem Alphabet $\{a, b\}$). Begründen Sie auch, warum es keine anderen Äquivalenzklassen gibt.

c) Ändert sich die Anzahl der Äquivalenzklassen der Sprache L_2, wenn Sie stattdessen Wörter über dem Alphabet $\{a, b, c\}$ betrachten? Begründen Sie Ihre Antwort.

Aufgabe 2:
Kontextfreie Sprachen

Betrachten Sie die Sprache $L_1 = \{a^n b c^n \mid n \in \mathbb{N}\} \cup \{a b^m c^m \mid m \in \mathbb{N}\}$.

a) Geben Sie für L_1 eine kontextfreie Grammatik an.

b) Ist Ihre Grammatik aus a) eindeutig? Begründen Sie Ihre Antwort.

Betrachten Sie die Sprache $L_2 = \{a^{2^n} \mid n \in \mathbb{N}\}$.

c) Zeigen Sie, dass L_2 nicht kontextfrei ist.

Fortsetzung nächste Seite!

Aufgabe 3:
Komplexität

Betrachten Sie die folgenden Probleme:

3SAT

Gegeben: Eine aussagenlogische Formel φ in konjunktiver Normalform (drei Literale pro Klausel).

Frage: Ist φ erfüllbar?

NAE-3SAT

Gegeben: Eine aussagenlogische Formel φ in konjunktiver Normalform (drei Literale pro Klausel).

Frage: Gibt es eine Belegung, die in jeder Klausel mindestens ein Literal wahr und mindestens ein Literal falsch macht?

Wir erlauben, dass NAE-3SAT-Formeln Literale der Form false haben, die immer falsch sind. So ist

$$(x_1 \vee \mathsf{false} \vee \mathsf{false}) \wedge (\neg x_1 \vee x_1 \vee x_1)$$

in NAE-3SAT (setze x_1 wahr).

a) Zeigen Sie, dass sich 3SAT in polynomieller Zeit auf NAE-3SAT reduzieren lässt.
b) Was können Sie aus a) folgern, wenn Sie wissen, dass 3SAT NP-vollständig ist?
c) Was können Sie aus a) folgern, wenn Sie wissen, dass NAE-3SAT NP-vollständig ist?

Aufgabe 4:
Berechenbarkeit

Betrachten Sie die folgenden Sprachen:

- $H = \{w\$x \mid M_w \text{ hält bei Eingabe } x\}$ über Alphabet $\{0, 1, \$\}$
- $H_\varepsilon = \{w \mid M_w \text{ hält bei Eingabe } \varepsilon\}$ über Alphabet $\{0, 1\}$

Dabei sei $x \in \{0,1\}^*$ und bezeichnet M_w die von $w \in \{0,1\}^*$ kodierte Turingmaschine.

a) Zeigen Sie, dass es eine Reduktion von H_ε auf H gibt.
b) Zeigen Sie, dass es eine Reduktion von H auf H_ε gibt.

Diese Reduktionen dürfen mehr als polynomielle Zeit benötigen.

Fortsetzung nächste Seite!

Aufgabe 5:
Die Begründung zählt

Bewerten Sie die folgenden Aussagen und geben Sie eine kurze Begründung (ca. ein bis drei Sätze) Ihrer Bewertung an. Nur Ihre Begründung wird bewertet.

a) Eine universelle Turingmaschine muss ein unendliches Eingabealphabet haben, um beliebige Turingmaschinen simulieren zu können.

b) Für beliebige reguläre Ausdrücke α, β gilt: $L(\alpha \cdot (\alpha + \beta)) = L(\alpha) \cup L(\alpha + \beta)$.

c) Sei L_1 eine Sprache und L_2 eine reguläre Sprache. Dann ist $L_1 \cap L_2$ immer entscheidbar.

d) Es gibt LOOP-Programme, die auch WHILE-Programme sind.

e) Mit dem Satz von RICE kann man nachweisen, dass es unentscheidbar ist, ob eine gegebene Turingmaschine mehr als 10 Zustände hat.

f) Es ist NP-schwer zu entscheiden, ob die Sprache eines endlichen Automaten leer ist.

Aufgabe 6:
Sortieren und das Gegenteil davon

1. Sei $S = (42, 4711, 98, 103, 1001, -13, 8, 98, 5, 13, 45, 64, 42, 98, 0, 17)$. Zeichnen Sie den Rekursionsbaum der Sortierung von S mittels MergeSort. Geben Sie für jeden Knoten sowohl die Ein- als auch die Ausgabe des entsprechenden Aufrufs an.
2. MergeSort benötigt zum Sortieren von n Elementen $O(n \log n)$ Schritte im Worst-Case. Welche Eigenschaft muss eine Eingabe haben, sodass diese Laufzeit tatsächlich erreicht wird?
3. Geben Sie einen Algorithmus in Pseudocode an, der eine Folge von n Elementen aus der Menge $U = \{0, 1\}$ in $O(n)$ Zeit sortiert.

Wir sagen eine Methode $M(S)$ permutiert eine Folge $S = (x_1, \ldots, x_n)$ von n paarweise verschiedenen Elementen, wenn sie die Reihenfolge der Elemente von S potentiell verändert (die ursprüngliche Reihenfolge darf auch beibehalten werden). Eine gute Permutationsmethode erzeugt jede mögliche Permutation von S mit gleicher Wahrscheinlichkeit.

Betrachten Sie die folgende Permutationsmethode:

```
1 Algorithmus : CleverShuffle(S)
  Eingabe : Eine Folge S = (x_1, ..., x_n)
  Ergebnis : S wurde permutiert.
2 begin
3     for i ← 1 to n do
4         k ← Random(i, n);
5         vertausche S[i] ↔ S[k];
```

Die Methode Random(i, j) wählt dabei zufällig eine ganze Zahl aus dem Intervall $[i, j]$, jede mit Wahrscheinlichkeit $\frac{1}{j+1-i}$. Man kann zeigen, dass CleverShuffle jede mögliche Permutation von S erzeugen kann, und jede mögliche Permutation mit gleicher Wahrscheinlichkeit erzeugt.

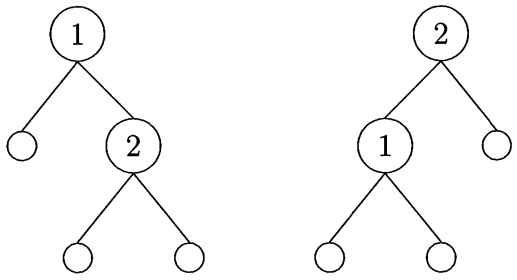
4. Was ist die asymptotische Laufzeit von CleverShuffle, wenn Random eine Laufzeit von $\Theta(1)$ hat?

Fortsetzung nächste Seite!

Aufgabe 7:

Zählen von binären Suchbäumen

Für $n \geq 0$ betrachten wir die Folge $X_n = (1, \ldots, n)$ der ersten n natürlichen Zahlen (X_0 ist also die *leere Folge*). Wir wollen einen effizienten Algorithmus entwerfen, der berechnet, wie viele *verschiedene* binäre Suchbäume für (die Elemente von) X_n existieren. Wir bezeichnen diese Zahl mit $b(n)$. Für $n = 2$ gibt es zwei verschiedene binäre Suchbäume (d.h. $b(2) = 2$):



1. Begründen Sie, dass $b(0) = 1$ und $b(1) = 1$.
2. Warum ist die Anzahl der verschiedenen binären Suchbäume für die Folge $(4, 5)$ genau $b(2)$?
3. Begründen Sie, dass $b(3) = 5$.
4. Welchen (arithmetischen) Zusammenhang erkennen Sie zwischen der Anzahl der verschiedenen binären Suchbäume für X_5 die in der Wurzel den Schlüssel 3 speichern, und der Anzahl der verschiedenen binären Suchbäume für die Folgen $(1, 2)$ bzw. $(4, 5)$? Begründen Sie diesen.
5. Begründen Sie, dass

$$b(n) = \textstyle\sum_{0 \leq l \leq n-1} b(l) \cdot b(n-1-l) \text{ für } n > 1$$
$$\text{und } b(n) = 1 \text{ für } n \leq 1.$$

6. Wie sieht eine Reihenfolge aus, in der die $b(n)$ bottom-up berechnet werden können, die mit der Rekursionsgleichung für $b(n)$ verträglich ist? Begründen Sie Ihre Antwort.
7. Geben Sie ein *dynamisches Programm* in Pseudocode an, das den Wert $b(n)$ iterativ statt rekursiv berechnet.
8. Warum ist Ihr Algorithmus korrekt? Begründen Sie Ihre Antwort.
9. Was ist die asymptotische Laufzeit Ihres Algorithmus? Was ist der asymptotische Speicherbedarf Ihres Algorithmus? Begründen Sie Ihre Antworten.

   Hinweis: Wenn Sie die obige Aufgabe nicht gelöst haben, verwenden Sie folgenden Algorithmus zur Beantwortung dieser Frage (*Achtung:* Das ist nicht die Lösung obiger Aufgabe!):

```
1  Algorithmus : ExAlg(n)
2  begin
3  |  Initialize Array a[0, .., n];
4  |  a[0] ← 1;
5  |  a[1] ← 2;
6  |  for l ← 2 to n do
7  |  |  s ← 1;
8  |  |  for r ← 1 to l do
9  |  |  |  s ← s · (a[r − 1] + a[n − r]);
10 |  |  a[l] ← s;
11 |  return a[n];
```

Fortsetzung nächste Seite!

Aufgabe 8:

Greedy-Färben von Intervallen

Sei $X = \{I_1, I_2, \ldots, I_n\}$ eine Menge von n (geschlossenen) Intervallen über den reellen Zahlen $\mathbb{R}$. Das Intervall I_j sei dabei gegeben durch seine linke Intervallgrenze $l_j \in \mathbb{R}$ sowie seine rechte Intervallgrenze $r_j \in \mathbb{R}$ mit $r_j > l_j$, d.h. $I_j = [l_j, r_j]$.

Wir nehmen in dieser Aufgabe der Einfachheit halber an, dass die Zahlen $l_1, l_2, \ldots, l_n, r_1, r_2, \ldots, r_n$ alle paarweise verschieden sind.

Zwei Intervalle I_j, I_k *überlappen* sich gdw. sie mindestens einen Punkt gemeinsam haben, d.h. gdw. falls für (o.B.d.A.) $l_j < l_k$, auch $l_k < r_j$ gilt. Eine *gültige Färbung* von X mit $c \in \mathbb{N}$ Farben ist eine Funktion $F : X \to \{1, 2, \ldots, c\}$ mit der Eigenschaft, dass für jedes Paar I_j, I_k von überlappenden Intervallen $F(I_j) \neq F(I_k)$ gilt.

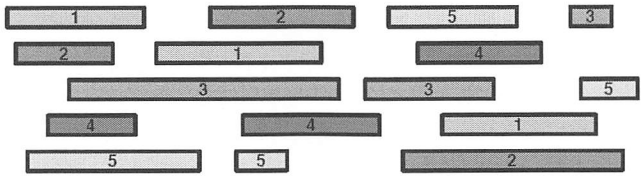


Abbildung 1: Eine gültige Färbung von X

Eine *minimale* gültige Färbung von X ist eine gültige Färbung mit einer minimalen Anzahl an Farben. Die Anzahl von Farben in einer minimalen gültigen Färbung von X bezeichnen wir mit $\chi(X)$.

Wir gehen im Folgenden davon aus, dass für X eine minimale gültige Färbung F^* gefunden wurde.

1. Nehmen wir an, dass aus X alle Intervalle einer bestimmten Farbe von F^* gelöscht werden. Ist die so aus F^* entstandene Färbung der übrigen Intervalle in jedem Fall immer noch eine minimale gültige Färbung? Begründen Sie Ihre Antwort.
2. Nehmen wir an, dass aus X ein beliebiges Intervall gelöscht wird. Ist die so aus F^* entstehende Färbung der übrigen Intervalle in jedem Fall immer noch eine minimale gültige Färbung? Begründen Sie Ihre Antwort.
3. Mit $\omega(X)$ bezeichnen wir die maximale Anzahl von Intervallen in X, die sich paarweise überlappen. Zeigen Sie, dass $\chi(X) \geq \omega(X)$ ist.

Wir betrachten nun folgenden Algorithmus, der die Menge $X = \{I_1, I_2 \ldots, I_n\}$ von n Intervallen einfärbt:

- Zunächst sortieren wir die Intervalle von X aufsteigend nach ihren *linken* Intervallgrenzen. Die Intervalle werden jetzt in *dieser Reihenfolge* nacheinander eingefärbt; ist ein Intervall dabei erst einmal eingefärbt, ändert sich seine Farbe nie wieder. Angenommen die sortierte Reihenfolge der Intervalle sei $I_{\sigma(1)}, \ldots, I_{\sigma(n)}$.
- Das erste Intervall $I_{\sigma(1)}$ erhält die Farbe 1. Für $1 < i \leq n$ verfahren wir im i-ten Schritt zum Färben des i-ten Intervalls $I_{\sigma(i)}$ wie folgt:

  Bestimme die Menge C_i aller Farben der bisher schon eingefärbten Intervalle die $I_{\sigma(i)}$ überlappen. Färbe $I_{\sigma(i)}$ dann mit der Farbe $c_i = \min(\{1, 2, \ldots, n\} \setminus C_i)$.

Fortsetzung nächste Seite!

4. Begründen Sie, warum der Algorithmus immer eine gültige Färbung von X findet (*Hinweis*: Induktion).
5. Zeigen Sie, dass die Anzahl an Farben, die der Algorithmus für das Einfärben benötigt, mindestens $\omega(X)$ ist.
6. Zeigen Sie, dass die Anzahl an Farben, die der Algorithmus für das Einfärben benötigt, höchstens $\omega(X)$ ist.
7. Begründen Sie mit Hilfe der o.g. Eigenschaften, warum der Algorithmus korrekt ist, d.h. immer eine minimale gültige Färbung von X findet.
8. Wir betrachten folgenden Implementierung des Algorithmus in Pseudocode:

```
1  Algorithmus : ColoringNumber(X_L[1,...,n], X_R[1,...,n])
   Eingabe : Felder X_R und X_L mit den rechten und linken Intervallgrenzen.
   Ergebnis : Minimale gültige Färbung der Intervalle.
2  begin
3      sortiere X_L (und passe X_R an);
       /* color[i] ist die Farbe des Intervals i                          */
4      initialisiere Array color[1,..,n];      // mit Nullen
       /* lastintervalofcolor[c] ist der Index des letzten Intervals das mit c gefärbt
          wurde                                                           */
5      initialisiere Array lastintervalofcolor[1,..,n];      // mit Nullen
6      maxcolor ← 1;
7      freecolor ← maxcolor;
8      color[1] ← freecolor;
9      lastintervalofcolor[freecolor] ← 1;
10     for i ← 2 to n do
11         freecolorfound ← false;
12         for c ← 1 to maxcolor do
13             i_c ← lastintervalofcolor[c];
14             if X_L[i] > X_R[i_c] then
                   /* i schneidet kein Interval der Farbe c               */
15                 freecolorfound ← true;
16                 freecolor ← c;
17                 break;
               /* i schneidet ein Interval der Farbe c                    */
18         if !freecolorfound then
19             maxcolor ← maxcolor + 1;
20             freecolor ← maxcolor;
21         color[i] ← freecolor;
22         lastintervalofcolor[freecolor] ← i;
23     return color;
```

Was ist die asymptotische Laufzeit dieses Algorithmus? Was ist der asymptotische Speicherbedarf dieses Algorithmus? Begründen Sie Ihre Antworten.

Thema Nr. 2
(Aufgabengruppe)

Es sind alle Aufgaben dieser Aufgabengruppe zu bearbeiten!

Aufgabe 1:

Zeigen oder widerlegen Sie die folgenden **Aussagen** (die jeweiligen Beweise sind sehr kurz):

a) Bezeichne SAT = $\{\langle\Phi\rangle \mid \Phi$ ist **erfüllbare** Booleschen Formel in konjunktiver Normalform$\}$ das NP-vollständige Erfüllbarkeitsproblem.

Es gibt eine Grammatik G vom Typ Chomsky-0, die genau die $\langle\Phi\rangle$ der **erfüllbaren** Booleschen Formeln Φ in konjunktiver Normalform erzeugt, d.h. $L(G) = \text{SAT}$.

Hinweis: $\langle\Phi\rangle$ bezeichnet lediglich eine Kodierung der Formel Φ.

b) Sei L eine beliebige kontextfreie Sprache über dem Alphabet Σ. Dann ist das Komplement $\overline{L} = \Sigma^* \setminus L$ entscheidbar.

c) Es ist entscheidbar, ob eine durch einen deterministischen endlichen Automaten gegebene Sprache unendlich viele Wörter enthält.

d) Seien L_1 und L_2 beliebige Sprachen über dem Alphabet $\Sigma = \{0, 1\}$ mit $L_1 \neq L_2$. Ist $L_1 \cap L_2$ entscheidbar, dann ist mindestens eine der beiden Sprachen L_1 und L_2 entscheidbar.

Schreiben Sie zuerst zur Aussage „Stimmt“ oder „Stimmt nicht“ und dann Ihre Begründung.

Aufgabe 2:

Sei $M_0, M_1, \ldots$ eine Gödelisierung aller Registermaschinen (RAMs). Das spezielle Halteproblem ist definiert als $K_0 = \{x \in \mathbb{N} \mid M_x$ hält bei Eingabe $x\}$. Gesucht ist eine *totale*, berechenbare Funktion $g : \mathbb{N} \to \mathbb{N}$ mit $W_g = K_0$. Dabei bezeichnet W_g den Wertebereich von g, d.h. $W_g = \{g(x) \mid x \in \mathbb{N}\}$. Gehen Sie wie folgt vor:

a) Definieren Sie eine Funktion $g : \mathbb{N} \to \mathbb{N}$.

b) Beweisen Sie, dass g total und berechenbar ist.

c) Beweisen Sie $W_g \subseteq K_0$.

d) Beweisen Sie $W_g \supseteq K_0$.

Aufgabe 3:

Sei A die durch den regulären Ausdruck $(a+b+c)^* \cdot c \cdot b^* \cdot a \cdot (a+b)^* \cdot c \cdot b^*$ beschriebene Sprache.

a) Geben Sie einen nichtdeterministischen, endlichen Automaten N an, der A akzeptiert.

b) Wandeln Sie N in einen äquivalenten deterministischen, endlichen Automaten um.

Fortsetzung nächste Seite!

Aufgabe 4:

Sei L die durch den regulären Ausdruck $(10 \cup 11)^*((11 \cup \varepsilon)^*1)^*$ beschriebene Sprache. [Alternative Schreibweise: $(10 + 11)^*((11 + \varepsilon)^*1)^*$]

a) Sei SAT das Erfüllbarkeitsproblem (siehe Aufgabe 1 a) zur Definition).
Zeigen Sie: L kann in polynomieller Zeit auf SAT reduziert werden (als Relation: $L \leq_{\mathrm{p}} \mathrm{SAT}$).

b) Angenommen, es wurde für das NP-vollständige Problem SAT gezeigt, dass SAT $\in$ P ist.
Zeigen Sie, dass L dann NP-vollständig ist.

Aufgabe 5:

Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik (V: Variablenmenge; Σ: Menge der Terminalsymbole; S: Startsymbol; P: Menge der Produktionen) in Chomsky-Normalform, und sei $w = w_1 \dots w_n$ ein Wort aus n Zeichen aus Σ. Der Algorithmus von Cocke/Younger/Kasami (CYK-Algorithmus) berechnet für alle $i, j \in \{1, \dots, n\}$, $i \leq j$, die Variablenmenge $V(i,j) = \{A \in V \mid A \overset{*}{\to} w_i \dots w_j\}$.

Sei $G = (V, \Sigma, P, S)$ die kontextfreie Grammatik in Chomsky-Normalform mit $V = \{S, A, B, C\}$, $\Sigma = \{a, b\}$, Startsymbol S und der Menge P der Produktionen:

$$S \to AB \mid BC \qquad B \to CC \mid b$$

$$A \to BA \mid a \qquad C \to AB \mid a$$

Sei $w = baaab$. Folgende Tabelle entsteht durch Anwendung des CYK-Algorithmus. Z. B. bedeutet $B \in V(3,5)$, dass aus der Variablen B das Teilwort $w_3 w_4 w_5 = aab$ hergeleitet werden kann. Drei Einträge wurden weggelassen.

b	*a*	*a*	*a*	*b*
V(1,1) {B}	V(2,2) {A,C}	V(3,3) {A,C}	V(4,4) {A,C}	V(5,5) {B}
V(1,2)	V(2,3) {B}	V(3,4) {B}	V(4,5) {S,C}	
V(1,3)	V(2,4) {S,A,C}	V(3,5) {B}		
V(1,4) {S,A,C}	V(2,5) {S,C}			
V(1,5)				

i) Bestimmen Sie die Mengen $V(1,2)$, $V(1,3)$ und $V(1,5)$.

ii) Wie entnehmen Sie dieser Tabelle, dass $w \in L(G)$ ist?

Fortsetzung nächste Seite!

Aufgabe 6:

Wissen

Ordnen Sie die unten stehenden Aussagen entsprechend ihres Wahrheitsgehalts in einer Tabelle der folgenden Form ein:

Kategorie	*WAHR*	*FALSCH*
X	X1, X3	X2
Y	Y2	Y1
...	...	...

A – Datenstrukturen

A1	Eine Streutabelle, die Kollisionen mit Hilfe einer verketteten Liste auflöst, kann beliebig viele Elemente speichern.
A2	Wird der Datentyp `Set<T>` (zur Darstellung von Mengen) mit einer doppelt verketteten Liste ohne Sortierung implementiert, dann haben das Einfügen eines Wertes in eine bestehende Menge mit n Werten und das anschließende Löschen eines anderen Wertes aus dieser Menge zusammen eine Laufzeitkomplexität von $\mathcal{O}(log_2(n))$.
A3	Eine doppelt verkettete Liste der Länge n ohne wahlfreien Zugriff erlaubt das Löschen des Listenkopfes ebenso wie des Listenendes in $\mathcal{O}(1)$ (konstanter Aufwand, unabhängig von n).
A4	Eine doppelt verkettete Liste ohne wahlfreien Zugriff kann zur Umsetzung von Warteschlangen nach dem FIFO-Prinzip verwendet werden.

B – Laufzeitkomplexität:

Gegeben seien zwei Methoden `f(int n)` und `g(int n)` mit den jeweiligen Laufzeiten $\mathcal{O}_f(log(n))$ bzw. $\mathcal{O}_g(n)$. Bei welchen der folgenden Schleifen stimmt die Laufzeitangabe im Kommentar?

B1	`for (int i = 0; i < n; i++) { f(n); } //` $\mathcal{O}(n)$
B2	`for (int i = 0; i < n; i++) { g(i); } //` $\mathcal{O}(n^2)$
B3	`for (int i = 0; i < n; i *= 2) { f(i); g(i); } //` $\mathcal{O}(n^3)$
B4	`for (int i = n; i > 0; i /= 2) { f(n); } //` $\mathcal{O}(log^2(n))$

Fortsetzung nächste Seite!

C – Graphen:

Ein minimaler Spannbaum eines zusammenhängenden Graphen mit n Knoten ...

C1	... ist stets eindeutig (zu einem Graphen kann es also keine verschiedenen Spannbäume geben).
C2	... muss nicht immer zusammenhängend sein.
C3	... enthält höchstens $n - 2$ Kanten.
C4	... enthält genau $n - 1$ Kanten.

D – Bäume:

D1	Das Einfügen eines Elements in eine *Halde (Heap)* mit n Einträgen hat konstante Laufzeit.
D2	Das Einfügen von n Werten in einen *binären Suchbaum* hat eine Laufzeit von $\mathcal{O}(n)$.
D3	Sowohl das Einfügen und das Löschen, als auch das Suchen haben in einem AVL-Baum einen Laufzeitaufwand von maximal $\mathcal{O}(log_2(n))$.
D4	In einem AVL-Baum entspricht der *Balancefaktor* eines Knotens der Summe der Höhe des linken Unterbaums und der Höhe des rechten Unterbaums.

Aufgabe 7:

Rekursion

Die *Potenzmenge* $\mathcal{P}(n)$ sei die Menge aller Teilmengen der Zahlen von 1 bis n (jeweils einschließlich), wobei die leere Menge $\emptyset$ auch zu den Teilmengen gehört, z. B. $\mathcal{P}(3) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$. Ergänzen Sie die Methode `potenzmenge`, die rekursiv $\mathcal{P}(n)$ bestimmen soll:

Hinweise zur API der Klasse `ArrayList<E>`*:*
▶ `public ArrayList(Collection<? extends E> c)`: *Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.*
▶ `public boolean add(E e)`: *Appends the specified element to the end of this list.*

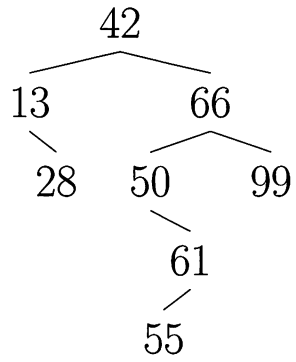
```
static List<List<Long>> potenzmenge(long n) {
        // Rueckgabe pm ist Potenzmenge der Zahlen von 1 bis n
        List<List<Long>> pm = new ArrayList<>();
        ...
```

Fortsetzung nächste Seite!

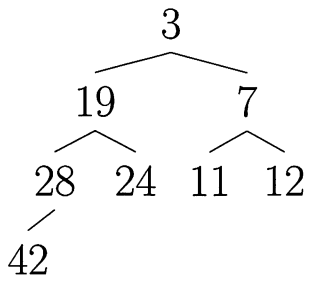
Aufgabe 8:
Bäume

a) Fügen Sie die Zahlen **13, 12, 42, 3, 11** in der gegebenen Reihenfolge in einen zunächst leeren *binären Suchbaum* mit aufsteigender Sortierung ein. Stellen Sie nur das Endergebnis dar.

b) Löschen Sie den Wurzelknoten mit Wert **42** aus dem folgenden *binären Suchbaum* mit aufsteigender Sortierung und ersetzen Sie ihn dabei durch einen geeigneten Wert aus dem **rechten** Teilbaum. Lassen Sie möglichst viele Teilbäume unverändert und erhalten Sie die Suchbaumeigenschaft.



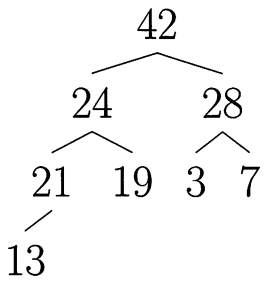
c) Fügen Sie einen neuen Knoten mit dem Wert **13** in die folgende *Min-Halde* ein und stellen Sie anschließend die Halden-Eigenschaft vom neuen Blatt aus beginnend wieder her, wobei möglichst viele Knoten der Halde unverändert bleiben und die Halde zu jedem Zeitpunkt links-vollständig sein soll. Geben Sie nur das Endergebnis an.



d) Geben Sie für die *ursprüngliche Min-Halde* aus Teilaufgabe c) (d.h. **ohne** den neu eingefügten Knoten mit dem Wert 13) die Feld-Einbettung (Array-Darstellung) an.

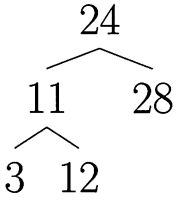
Fortsetzung nächste Seite!

e) Löschen Sie den Wurzelknoten mit Wert **42** aus der folgenden *Max-Halde* und stellen Sie anschließend die Halden-Eigenschaft ausgehend von einer neuen Wurzel wieder her, wobei möglichst viele Knoten der Halde unverändert bleiben und die Halde zu jedem Zeitpunkt links-vollständig sein soll. Geben Sie nur das Endergebnis an.

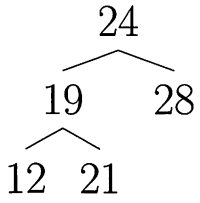


f) Fügen Sie in jeden der folgenden *AVL-Bäume* mit aufsteigender Sortierung jeweils einen neuen Knoten mit dem Wert **13** ein und führen Sie anschließend *bei Bedarf* die erforderliche(n) Rotation(en) durch. Zeichnen Sie den Baum vor und nach den Rotationen.

i) AVL-Baum $\mathcal{A}$:



ii) AVL-Baum $\mathcal{B}$:

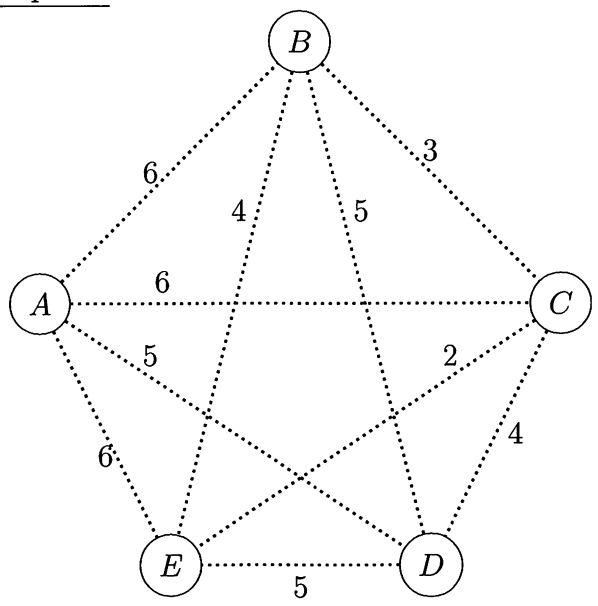


Fortsetzung nächste Seite!

Aufgabe 9:
Graphen

a) Wenden Sie den Algorithmus von *Kruskal* auf den Graphen $\mathcal{F}$ an und geben Sie die Kanten in der Reihenfolge an, in welcher das Verfahren sie *in den Spannbaum aufnimmt*:

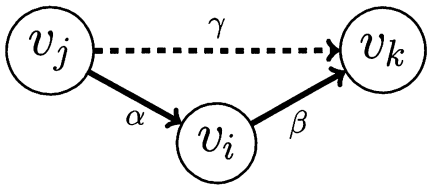
Graph $\mathcal{F}$:



Was Sie in den Graphen zeichnen
*wird **nicht** bewertet!*

b) Für einen beliebigen Graphen $\mathcal{G} = (V, E)$ mit Kantenanzahl $e := |E|$ und Knotenanzahl $v := |V|$, welche Laufzeitkomplexitäten (in Landau-$\mathcal{O}$-Notation) haben effiziente Implementierungen der Algorithmen von *Prim* und *Kruskal* im allgemeinen Fall?

c) Ergänzen Sie den Algorithmus von *Floyd* für Graphen $\mathcal{G}$, dargestellt durch die Adjazenzmatrix g. Der Eintrag `g[j][k]` der Adjazenzmatrix g enthält das (positive) Gewicht der Kante (j, k) bzw. 0, falls es keine solche Kante gibt oder falls $j = k$. Die Implementierung arbeitet hier *in-situ* und ergänzt auch indirekte Kanten direkt in g.




```
// betrachte alle Kanten-Tripel vj => vi => vk,
// vergleiche diese Weglaenge mit vj => vk (sofern vorhanden)
// und aktualisiere ggf. die Weglaenge vj => vk
public void floydify(double[][] g) {
        int n = g.length;
        // durchlaufe alle Knoten vi und vj:
        for (int i = 0; i < n; i++) {
                ...
```