Prüfungsteilnehmer	Prüfungstermin	Einzelprüfungsnummer

Kennzahl: _____

Kennwort: _____

Arbeitsplatz-Nr.: _____

Herbst 2015

66115

Erste Staatsprüfung für ein Lehramt an öffentlichen Schulen

— Prüfungsaufgaben —

Fach: **Informatik (verieft studiert)**

Einzelprüfung: **Theoretische Informatik, Algorithmen**

Anzahl der gestellten Themen (Aufgaben): 2

Anzahl der Druckseiten dieser Vorlage: 10

Bitte wenden!

Thema Nr. 1
(Aufgabengruppe)

Es sind alle Aufgaben dieser Aufgabengruppe zu bearbeiten!

Aufgabe 1:

Die Sprache L über dem Alphabet $\Sigma = \{a, b\}$ enthält alle Wörter, in denen das Wort *bab*, oder das Wort *aba*, oder beide vorkommen. Also ist z.B. $babbba \in L$, aber $bbaabbb \notin L$.

a) Begründen Sie, dass L regulär ist.

b) Geben Sie einen nichtdeterministischen Automaten für L mit 7 Zuständen an. Der Automat soll zu Beginn nichtdeterministisch entscheiden, ob nach *bab* oder *aba* gesucht wird.

c) Führen Sie auf diesem Automaten die Potenzmengenkonstruktion durch und minimieren Sie anschließend den resultierenden deterministischen Automaten.

d) Geben Sie ein Beispiel (mit Begründung!) einer Sprache an, welche nicht durch einen *deterministischen* endlichen Automaten mit nur einem Endzustand erkannt werden kann.

e) Begründen Sie, dass jede Sprache $L \subseteq \{a, b\}^+$ (NB: $\epsilon \notin L$) von einem *nichtdeterministischen* Automaten mit nur einem Endzustand erkannt werden kann. Hinweis: Sie können die Sprachen $U = \{w \mid wa \in L\}$ und $V = \{w \mid wb \in L\}$ verwenden.

Aufgabe 2:

Gegeben ist ein gerichteter Graph $G = (V, E)$, der eine Web-Site repräsentieren soll. Die Knoten des Graphen sind die einzelnen Seiten; eine Kante $(v, v') \in E$ bedeutet, dass die Seite v' auf der Seite v verlinkt ist. Weiterhin sind eine Menge P von "Trails", also typischen Besuchssequenzen gegeben. Jeder solche Trail ist einfach ein Pfad in G, also eine Folge $(v_1, \ldots, v_n)$ von Knoten mit $(v_i, v_{i+1}) \in E$ für $i = 1 \ldots n-1$. Diese Trails werden durch statistische Analyse des Surfverhaltens der Besucher der Web-Site ermittelt.

Nun soll ein Algorithmus gefunden werden, der Werbeanzeigen möglichst effizient platziert; hierzu soll zu gegebener Zahl k, eine Auswahl W von k Seiten (also Knoten von G) ermittelt werden, sodass jeder Trail mindestens eine Seite in W enthält.

Zeigen Sie durch Reduktion von einem geeigneten Problem aus dem Informatikduden oder aus der von Ihnen besuchten Vorlesung, dass es NP-vollständig ist zu entscheiden, ob bei vorgelegten G, P und k eine solche Auswahl W existiert.

Beachten Sie, dass Sie sowohl die Zugehörigkeit zu NP, als auch die NP-Härte (NP-Schwere) zeigen müssen.

Fortsetzung nächste Seite!

Aufgabe 3:

Beim *Postschen Korrespondenzproblem* (PCP) ist bekanntlich eine Liste von Paaren $(u_1, v_1), \ldots, (u_n, v_n)$ mit $u_i, v_i \in \Sigma^*$ für ein Alphabet Σ gegeben. Zum Beispiel $(u_1, v_1) = (b, bab)$ und $(u_2, v_2) = (ba, aa)$ und $(u_3, v_3) = (abb, bb)$, wobei hier also $n = 3$ und $\Sigma = \{a, b\}$ sind. Gefragt ist, ob es eine Indexfolge $i_1, i_2, \ldots, i_N$, wobei $i_k \leq n$ gibt, so dass gilt $u_{i_1} u_{i_2} \ldots u_{i_N} = v_{i_1} v_{i_2} \ldots v_{i_N}$. Im Beispiel wäre $1, 3, 2, 3$ solch eine Lösung, also genauer $N = 4$ und $i_1 = 1, i_2 = 3, i_3 = 2, i_4 = 3$, denn es ist hier $u_1 u_3 u_2 u_3 = b\ abb\ ba\ abb = bab\ bb\ aa\ bb = v_1 v_3 v_2 v_3$.

Es ist bekannt, dass das PCP ein unentscheidbares Problem ist.

Zeigen Sie durch Reduktion von PCP, dass es unentscheidbar ist, von einer gegebenen kontextfreien Grammatik $G = (T, N, S, \rightarrow)$ festzustellen, ob sie eindeutig ist, d.h. ob es für jedes Wort $w \in L(G)$ genau eine Linksableitung $S \rightarrow^* w$ in G gibt. Hinweis: Zu einer gegebenen Instanz des PCP betrachten Sie die (kontextfreie!) Sprache $\{u_{i_1} \ldots u_{i_N} i_N \ldots i_1 \mid N \in \mathbb{N}, i_1, \ldots . i_N \leq n\} \cup \{v_{i_1} \ldots v_{i_N} i_N \ldots i_1 \mid N \in \mathbb{N}, i_1, \ldots . i_N \leq n\}$.

Aufgabe 4:

Sie sollen entlang eines vielbefahrenen Autobahnabschnitts Reklametafeln positionieren. Die möglichen Positionen für solche Tafeln sind $x_1, \ldots, x_n$, wobei x_i den Abstand der Position in Kilometern vom Anfang des Abschnitts bezeichnet. Sie können also eine Tafel "bei Autobahnkilometer" x_i für $i = 1, \ldots, n$ platzieren. Diese Werte x_i sind aufsteigend sortiert. Außerdem sind Werte $r_1, \ldots, r_n$ gegeben, wobei r_i den zu erwartenden Gewinn (in 100TEUR) angibt, wenn Sie eine Tafel bei x_i platzieren.

Allerdings gibt es die Vorschrift, dass zwei aufgestellte Tafeln mehr als 5km Abstand voneinander haben müssen. Platzieren Sie also Tafeln bei x_i und x_j, so muss $|x_i - x_j| > 5$ sein.

Beispiel: $(x_1, x_2, x_3, x_4) = (6, 7, 12, 14)$ und $(r_1, r_2, r_3, r_4) = (5, 6, 5, 1)$. Hier könnten Sie bei km 7 und km 14 werben, was einen Gewinn von 700TEUR liefert, oder aber bei km 6 und 12, was einen Gewinn 1MEUR liefert. Dass das Aufstellen der Tafeln auch Kosten verursacht, bleibt hier außer Betracht.

1. Für jedes $i \leq n$ sei $f(i)$ die größte Zahl unterhalb von i, sodass $x_i - x_{f(i)} > 5$. Der Wert $f(i)$ sei undefiniert, wenn keine solche Zahl existiert. Im Beispiel ist $f(4) = 2, f(3) = 1$ und $f(2), f(1)$ sind undefiniert. Beschreiben Sie, wie die Werte $f(n), f(n-1), \ldots, f(1)$ zusammen in Zeit $O(n)$ bestimmt werden können.
2. Sei W_i der maximal erzielbare Gewinn bei ausschließlicher Verwendung der Positionen $x_1, \ldots, x_i$. Überlegen Sie sich, wie man W_i aus den Werten W_j für $j < i$ berechnen kann.
3. Beschreiben Sie, wie W_n mit dynamischer Programmierung in Zeit $O(n)$ berechnet werden kann.

Thema Nr. 2
(Aufgabengruppe)

Es sind alle Aufgaben dieser Aufgabengruppe zu bearbeiten!

Aufgabe 1:

"Bäume und Rekursion"

Gegeben sei folgende Klasse **Node** für binäre Bäume:

```
class Node implements Comparable<Node> {
	char c;		// irrelevant for inner nodes
	int f;		// frequency (number of occurrences of c)
	Node zero;	// child
	Node one;	// child

	// creates a leaf
	public Node(char c, int f) {
		this.c = c; this.f = f;
	}

	// creates an inner node
	public Node(Node zero, Node one) {
		this.f = zero.f + one.f;
		this.zero = zero; this.one = one;
	}

	// natural ordering based on the frequency
	@Override
	public int compareTo(Node that) {
		return this.f - that.f;
	}
}
```

Aus der Java-API dürfen Sie auch folgende Methoden verwenden:

HashMap<K,V>:
- V put(K key, V value)
- V get(Object key)

Collections:
- static <T extends Comparable<? super T>> void sort(List<T> list)

LinkedList<E>:
- int size()
- E removeFirst()
- void addFirst(E e)

String:
- char charAt(int index)
- String substring(int beginIndex)

a) Ergänzen Sie die Methode **count**, welche die HashMap **map** von Node-Objekten anlegt. Für jedes Zeichen **c** des übergebenen Strings **s** soll **map** ein **Node**-Objekt für dieses Zeichen in **Node.c** enthalten. Am Ende der Schleife gibt **Node.f** jeweils an, wie oft das Zeichen **c** in **s** vorkommt. Das in der letzten Code-Zeile erzeugte Ergebnis von **count** ist die Liste der **Node**-Objekte aus **map**.

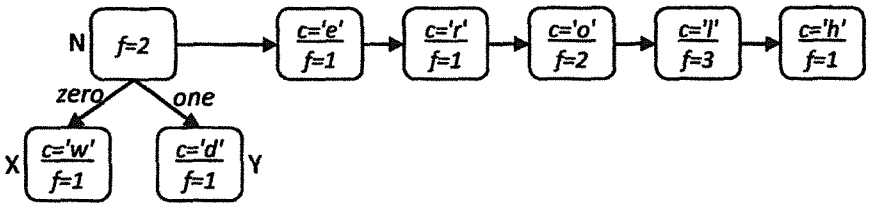
Beispiel: Ergebnis für count("helloworld")


```
LinkedList<Node> count(String s) {
	assert s != null : new IllegalArgumentException();
	HashMap<Character, Node> map = new HashMap<>();
	for (char c : s.toCharArray()) {
		// ToDo: Code hier ergaenzen
	}
	return new LinkedList<>(map.values());
}
```

Fortsetzung nächste Seite!

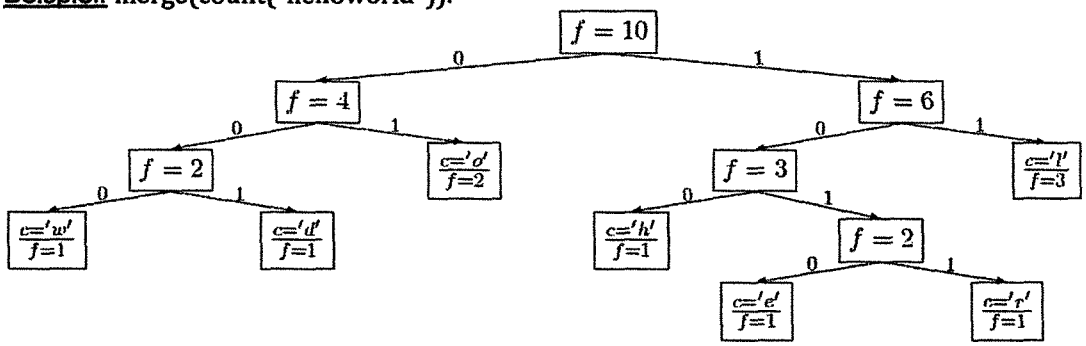
b) Vervollständigen Sie die Methode **merge**, die zuerst die **Node**-Liste mit der API-Methode **Collections.sort** sortiert, damit die *kleinsten* Knoten X und Y (bzgl. **compareTo**) am Anfang der Liste stehen. Dann verschmilzt sie X und Y zu einem neuen inneren Knoten N, wobei X zum **zero**-Kind und Y zum **one**-Kind von N wird. Anschließend werden X und Y aus der Liste entfernt und N hinzugefügt.

Beispiel:
Erster Durchlauf von merge(count(s)) für s = "helloworld"



Schließlich wiederholt **merge** den vorangehend beschriebenen Vorgang, bis nur noch ein Knoten (die Wurzel des Baums) in der Liste übrig bleibt.

Beispiel: merge(count("helloworld")):




```
void merge(LinkedList<Node> nodes) {
      // ToDo: Code hier ergaenzen
```

c) Der Pfad von der Wurzel des resultierenden Baums zu einem Zeichen **c** stellt dessen Huffman-Codierung dar:

$$b \leftrightarrow \overbrace{100}^{h}\overbrace{1010}^{e}\overbrace{11}^{l}\overbrace{11}^{l}\overbrace{01}^{o}\overbrace{000}^{w}\overbrace{01}^{o}\overbrace{1011}^{r}\overbrace{11}^{l}\overbrace{001}^{d}$$

Ergänzen Sie nun die *rekursive* Methode **dec**, die eine als String aus **0** und **1** gespeicherte Bitfolge **b** wieder zur ursprünglichen Zeichenkette dekodiert. Dazu traversiert sie den Baum wiederholt entsprechend der Folge **b**: Sobald ein Blatt erreicht wird, ergänzt sie das Zwischenergebnis mit dem zugehörigen Zeichen und beginnt die Dekodierung der restlichen Folge wieder an der Wurzel des Baums. Falls die Folge vorzeitig (vor Erreichen eines Blatts) endet, dann kann nicht entpackt werden und die Methode muss eine **IllegalArgumentException** werfen.

```
String decode(Node root, String b) {
      assert (root != null && b != null) : new IllegalArgumentException();
      return dec(root, root, b);
}

String dec(Node root, Node node, String b) {
      // ToDo: Code hier ergaenzen
```

Fortsetzung nächste Seite!

Aufgabe 2:

"Haldensortierung"

Gegeben sei folgende Klasse:

```
class W {
      int t;
      String f;
      // ...
```

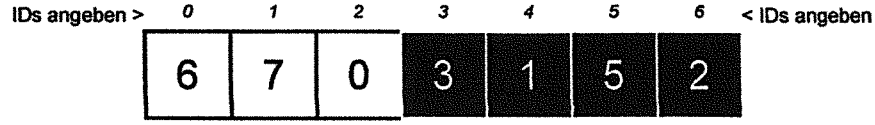
Dazu gibt es verschiedene **Comparatoren**, zum Beispiel:

```
// ascending order for field W.t
class ComparatorAscByFieldT implements Comparator<W> {
      // Returns a negative integer, zero, or a positive integer as the
      // first argument is less than, equal to, or greater than the second.
      @Override
      public int compare(W o1, W o2) { // ...
```

Außerdem steht Ihnen die vorgegebene Methode swap zur Verfügung:

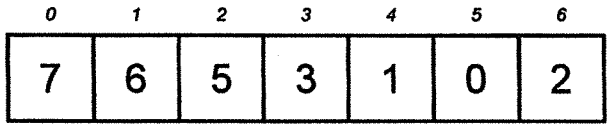
```
void swap(W[] w, int a, int b) { // ...
```

a) Phase 1: Die Haldensortierung beginnt mit der Herstellung der Max-Heap-Eigenschaft von rechts nach links. Diese ist für alle Feldelemente im dunklen Bereich bereits erfüllt. Geben Sie die Positionen (IDs) derjenigen Elemente des Feldes an, die das Verfahren im „Versickerschritt" für das nächste Element mit Hilfe des **ComparatorAscByFieldT** miteinander vergleicht:

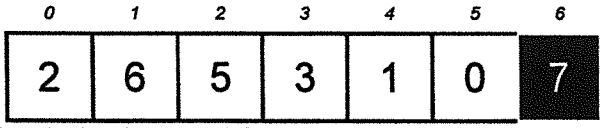


Nach dem Vergleichen werden gegebenenfalls Werte mit **swap** vertauscht. Geben Sie das Resultat (in obiger Array-Darstellung) nach diesem Schritt an.

b) Phase 2: Das folgende Feld enthält den bereits vollständig aufgebauten Max-Heap:



Die Haldensortierung verschiebt das maximale Element in den sortierten (dunklen) Bereich:



Geben Sie das Ergebnis des nachfolgenden „Versickerns" (erneut in derselben Array-Darstellung) an, bei dem die Heap-Eigenschaft wiederhergestellt wird.

c) Ergänzen Sie die rekursive Methode **reheap**, die die Max-Heap-Eigenschaft im Feld **w** zwischen den Indizes **i** und **k** (jeweils einschließlich) in O(log(k–i)) gemäß **Comparator<W> c** wiederherstellt, indem sie das Element **w[i]** „versickert". **k** bezeichnet das Ende des unsortierten Bereichs.

Fortsetzung nächste Seite!

```
// restores the max-heap property in w[i to k] using c
void reheap(W[] w, Comparator<W> c, int i, int k) {
        int leftId = 2 * i + 1;
        int rightId = leftId + 1;
        int kidId;
        // ToDo: Code hier ergaenzen
}
```

d) Implementieren Sie nun die eigentliche Haldensortierung. Sie dürfen hier die Methode **reheap** verwenden.

```
// sorts w in-situ according to the order imposed by c
void heapSort(W[] w, Comparator<W> c) {
        int n = w.length;

        // Phase 1: Max-Heap-Eigenschaft herstellen
        //                  (siehe Teilaufgabe a)
        // ToDo: Code hier ergaenzen


        // Phase 2: jeweils Maximum entnehmen und sortierte Liste am Ende aufbauen
        //                  (siehe Teilaufgabe b)
        // ToDo: Code hier ergaenzen
}
```

Fortsetzung nächste Seite!

Aufgabe 3:

"Laufzeitanalyse mittels Landau-0-Kalkül"

Gegeben seien die folgenden Methoden, in denen bestimmte Stellen mit Kommentaren der Form `/** x **/` markiert sind. Geben Sie zunächst für jede solche Stelle einer Methode eine *geschlossene* Formel an, die *exakt* berechnet, wie oft die jeweiligen Stellen, in Abhängigkeit von den Eingangsgrößen `n` und `m`, durchlaufen werden.
Ermitteln Sie anschließend für diese Quellcode-Fragmente jeweils die kleinste obere Schranke für den Laufzeitaufwand im $\mathcal{O}$-Kalkül (Landau-Notation) abhängig von den Parametern. Skizzieren Sie in wenigen Sätzen, wie Sie die Aufwandklasse abgeschätzt haben – eine Beweisführung ist *nicht* verlangt.

a) matrixVectorMul

```
static int[] matrixVectorMul(int[][] mat, int[] vec) {
      int hoehe = mat.length; /* -> m */
      if (hoehe <= 0)
            return new int[0];
      int breite = mat[0].length; /* -> n */
      int[] erg = new int[hoehe];
      for (int zeile = 0; zeile < hoehe; ++zeile) {
            /** 1 **/
            for (int spalte = 0; spalte < breite /** 2 **/; ++spalte) {
                  erg[zeile] += mat[zeile][spalte] * vec[spalte];
                  /** 3 **/
            }
      }
      return erg;
}
```

b) zaehlen

```
static int zaehlen(int n) {
      int count = 0;
      while (n > 0) {
            if (n % 2 == 1) {
                  ++count;
            }
            n /= 2;
            /** 1 **/
      }
      return count;
}
```

c) f:

```
static int f(int n) {
      if (n <= 0) {
            /** 1 **/
            return 0;
      } else {
            for (int i = 0; i < n; ++i) {
                  /* Operationen mit konstantem Aufwand */
            }
            return f(n / 2);
      }
}
```

Fortsetzung nächste Seite!

Aufgabe 4:

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

a) Sei $L \subseteq \Sigma^*$. Ist L regulär, so ist jede Teilmenge U von L auch regulär.

b) Sei L eine Sprache über dem Alphabet Σ, die rekursiv aufzählbar (= partiell-entscheidbar), aber nicht entscheidbar ist. Sei $\overline{L} = \Sigma^* \setminus L$ das Komplement von L. Dann ist $\overline{L}$ rekursiv aufzählbar.

c) Seien L_1 und L_2 beliebige kontextfreie Sprachen über dem Alphabet Σ. Dann ist $L_1 \cap L_2$ entscheidbar.

d) Alle Probleme in NP sind entscheidbar.

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

Aufgabe 5:

a) Sei $m \in \mathbb{N}_0 = \{0, 1, 2, \ldots\}$. Definieren Sie formal die Menge H_m der Gödelnummern der Turing-Maschinen, die gestartet mit m halten.

b) Gegeben sei das folgende Problem E:

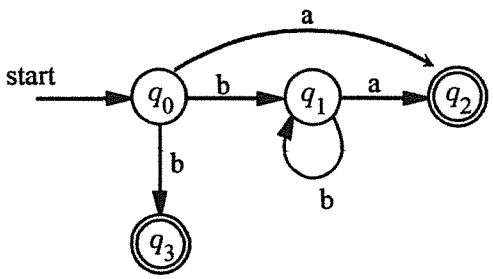
- Entscheide, ob es für die deterministische Turing-Maschine M mit der Gödelnummer $\langle M \rangle$ mindestens zwei Eingaben $w_1, w_2 \in \mathbb{N}_0$, $w_1 \neq w_2$, gibt, so dass die Maschine M gestartet mit w_1 hält und dass M gestartet mit w_2 hält.

Zeigen Sie, dass E nicht entscheidbar ist. Benutzen Sie, dass H_m aus (a) für jedes $m \in \mathbb{N}_0$ nicht entscheidbar ist.

c) Zeigen Sie, dass das Problem E aus (b) partiell-entscheidbar (= rekursiv aufzählbar) ist.

Aufgabe 6:

a) Gegeben sei der folgende nichtdeterministische endliche Automat N:



b) Geben Sie einen regulären Ausdruck $\alpha(N)$ für die Sprache, die der nichtdeterministische endliche Automat N aus (a) akzeptiert, an.

Fortsetzung nächste Seite!

c) Sei $L = \{a^k b^k \mid k \in \mathbb{N}\}$. Jemand behauptet, einen deterministischen endlichen Automaten mit Zustandsmenge $Q = \{q_0, \ldots, q_{n-1}\}$, Startzustand q_0 und Endzustandsmenge F konstruiert zu haben mit $L = L(A)$.

Geben Sie in Abhängigkeit von A ein Wort $z \in L$ an, das folgende Eigenschaft besitzt: Aus einer akzeptierenden Rechnung von A für z können Sie ein Wort $\hat{z}$ konstruieren mit der Eigenschaft: (i) A akzeptiert $\hat{z}$ und (ii) $\hat{z} \notin L$.

Beweisen Sie konkret die Eigenschaften (i) und (ii) für Ihr Wort z.

Nachtrag zur Einzelprüfungnr. 66115, Thema 2, Seite 9, Aufgabe 6, Buchstabe a)

Bitte fügen Sie nach dem abgebildeten nichtdeterministischen endlichen Automaten folgende Fragestellung ein:

„Konstruieren Sie zu *N* mit der Potenzmengen-Konstruktion einen äquivalenten deterministischen endlichen Automaten *A*. Zeichnen Sie die nur vom Startzustand erreichbaren Zustände ein, diese aber *alle*. Die Zustandsnamen von *A* müssen erkennen lassen, wie sie zustande gekommen sind. Führen Sie *keine* „Vereinfachungen" durch!"