Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn

Klausur Informatik 2: Algorithmen und Datenstrukturen

Donnerstag, 9. März 21, 2017, 9:00 bis 12.00 Uhr

Name: .....

Matrikel Nr.: .....

Unterschrift: .....

Blättern Sie nicht um bevor Sie dazu aufgefordert werden!

- Schreiben Sie Ihren Namen und Ihre Matrikelnummer **auf alle Blätter**.
- **Unterschreiben Sie das Deckblatt.** Ihre Unterschrift bestätigt, dass Sie alle Fragen ohne nicht erlaubte Hilfsmittel beantwortet haben.
- Schreiben Sie **lesbar** und nur mit **dokumentenechten** Stiften. Benutzen sie **keine rote** oder **grüne** Farbe und **keinen Bleistift**!
- Alle **schriftlichen Hilfsmittel** sind **erlaubt**. **Elektronische Hilfsmittel** sind **nicht** erlaubt.
- Die Klausur besteht aus **8 Aufgaben** (mit jeweils mehreren Teilaufgaben) und **120 Punkten**.
- Zum Bestehen sind **50 Punkte ausreichend**.
- Benutzen Sie für **jede Aufgabe eine eigene Seite**.
- Es wird **nur eine Lösung pro Aufgabe** gewertet. Vergewissern Sie sich, dass Sie zusätzliche Lösungen durchstreichen, andernfalls wird die schlechteste Lösung gewertet.
- Detaillierte Schritte können Ihnen zu **Teilpunkten** verhelfen falls das Endergebnis falsch ist.
- Die Schlüsselwörter **Zeigen Sie...**, **Beweisen Sie...**, **Begründen Sie...** oder **Leiten Sie ... her** zeigen an, dass Sie Ihre Antwort sorgfältig und gegebenenfalls formal begründen müssen.
- Die Schlüsselwörter **Geben sie ... an** zeigen an, dass sie lediglich die geforderte Antwort und keine Begründung liefern müssen.
- Die folgenden Regeln gelten **überall**, außer sie werden explizit außer Kraft gesetzt.
- Bei Laufzeitfragen ist nur die **asymptotische Laufzeit** notwendig.
- Wenn Sie einen Algorithmus angeben sollen, so können Sie Pseudocode angeben. Eine **ausreichend detaillierte** (!) Beschreibung der Funktionsweise Ihres Algorithmus genügt jedoch.
- Algorithmen aus der Vorlesung **können grundsätzlich als Blackbox** verwendet werden.
- Falls Sie Algorithmen entwerfen, welche **Hashtabellen** (als Blackbox) verwenden, dürfen Sie annehmen, dass alle Operationen der Hashtabelle $\mathcal{O}(1)$ Zeit benötigen.
- **Lesen Sie jede Aufgabe sorgfältig** durch und stellen Sie sicher dass Sie diese verstanden haben!

Frage	1	2	3	4	5	6	7	8	**Total**
Punkte									
Maximum	15	15	14	15	15	15	16	15	120

Aufgabe 1: Divide and Conquer (15 Punkte)

Gegeben sei ein **zweidimensionales, quadratisches, 1-basiertes**[1] Array $A[1 \dots n][1 \dots n]$ gefüllt mit **paarweise verschiedenen** Schlüsseln. Außerdem **sei** n **eine Zweierpotenz**, d.h. $n = 2^k$ für ein $k \in \mathbb{N}_0$.

Die Schlüssel in A seien **zeilenweise und spaltenweise aufsteigend sortiert**. Dass heißt für $i < j$ gilt: $A[i,k] < A[j,k]$ und für $k < l$ gilt: $A[i,k] < A[i,l]$. Betrachten Sie den folgenden Algorithmus in Pseudocode der nach dem Prinzip **Divide and Conquer** funktioniert.

Hinweis: A lässt sich als $n \times n$-Matrix interpretieren wobei der Eintrag in der i-ten Zeile und k-ten Spalte durch $A[i,k]$ gegeben ist. Dabei definiert $A[i \dots j][k \dots l]$ die 'Teilmatrix' von der i-ten zur j-ten Zeile in vertikaler und von der k-ten zur l-ten Spalte in horizontaler Richtung (jeweils inklusive).

Algorithm FOOBAR($A[1 \dots n][1 \dots n]$: **2d-array of integers**, key: **integer**) : **boolean**

Require: A ist zeilenweise und spaltenweise aufsteigend sortiert.

```
if n = 1 then                                              ▷ Basisfall
    if A[1,1] = key then
        return True
    else
        return False

p ← A[n/2, n/2]                                            ▷ Pivotelement
if key ≤ p then
    B ← A[1 ... n/2][1 ... n/2]                            ▷ Linke obere 'Teilmatrix'
else
    B ← A[(n/2+1) ... n][(n/2+1) ... n]                    ▷ Rechte untere 'Teilmatrix'

C ← A[(n/2+1) ... n][1 ... n/2]                            ▷ Linke untere 'Teilmatrix'
D ← A[1 ... n/2][(n/2+1) ... n]                            ▷ Rechte obere 'Teilmatrix'

return FOOBAR(B, key) ∨ FOOBAR(C, key) ∨ FOOBAR(D, key)    ▷ Rekursion
```

(a) Beschreiben Sie **im Bezug auf das Array** A **und den Schlüssel** key welche Information der Algorithmus FOOBAR(A, key) liefert *(4 Punkte)*.

(b) Begründen Sie warum der Algorithmus diese Information **korrekt** berechnet *(6 Punkte)*.

(c) Geben Sie die Laufzeitfunktion $T(n)$ in **rekursiver** Form an. *(5 Punkte)*

Hinweis: Gehen sie davon aus, dass das Erstellen der Arrays B, C, D (durch Wiederbenutzen von A und geeignete Indextransformationen) in $\mathcal{O}(1)$ Zeitschritten möglich ist.

[1] D.h. die Indizierung beginnt mit 1 und nicht mit 0.

Aufgabe 2: Landau Notation (15 Punkte)

Geben Sie an ob die folgenden Behauptungen wahr oder falsch sind *(jeweils 1 Punkt)*. Beweisen Sie Ihre Aussage **anhand der Definitionen** der Landau Notation *(3 bzw. 4 bzw. 5 Punkte)*.

(a) $\log_3(n^3 \cdot 3^n) \in \mathcal{O}(n)$ *Sie dürfen benutzen, dass* $\forall n \in \mathbb{N} : \log_3(n) \leq n$.

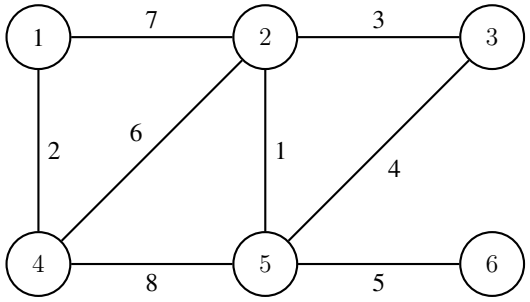
(b) $\sqrt[3]{n} \in \Theta(\sqrt{n})$

(c) $\log_2(n!) \in \Omega(\log_2(n^n))$ *Sie dürfen benutzen, dass* $n! := \prod_{i=1}^{n} i \geq (n/2)^{n/2}$.

Aufgabe 3: Minimaler Spannbaum (14 Punkte)

(a) Führen sie auf dem folgenden gewichteten Graphen den **Algorithmus von Kruskal** aus, indem Sie die **Kanten des minimalen Spannbaumes markieren** und die **Reihenfolge vermerken** in der die Kanten hinzugefügt wurde *(4 Punkte)*.

Hinweis: Sie können die geforderte Reihenfolge direkt neben den Kanten vermerken.



(b) Ein eifriger Fakultätsmitarbeiter schlägt den folgenden Algorithmus im (abstrakten) Pseudocode vor, welcher einen minimalen Spannbaum nach dem Prinzip **Divide and Conquer** finden soll. Begründen Sie dem Mitarbeiter anhand eines geeigneten Beispiels, dass sein Algorithmus **nicht korrekt** ist *(6 Punkte)*.

Algorithm DIVCONMST($G = (V, E)$: **Weighted Graph**) : **Set of Edges**

---

if $V = \emptyset$ oder $|V| = 1$ **then return** $\emptyset$ ▷ *Basisfall*

Teile V in gleichgroße Mengen V_1,V_2 auf
Seien G_1 bzw. G_2 die durch V_1 bzw. V_2 induzierten Teilgraphen.

$M_1 \leftarrow$ DIVCONMST(G_1) ▷ *Rekursive Ermittlung der min. Spannbäume*
$M_2 \leftarrow$ DIVCONMST(G_2)

Ermittle leichteste Kante $e = \{v_1, v_2\} \in E$ mit $v_1 \in V_1, v_2 \in V_2$.
return $M_1 \cup M_2 \cup \{e\}$

(c) Angenommen wir haben einen ungerichteten Graphen mit Gewichten, die positiv oder negativ sein können. Berechnen Prim's und Kruskal's Algorithmus jeweils einen MST für solche Graphen? Begründen Sie ihre Antwort *(4 Punkte)*.

Aufgabe 4: Binäre Suchbäume (15 Punkte)

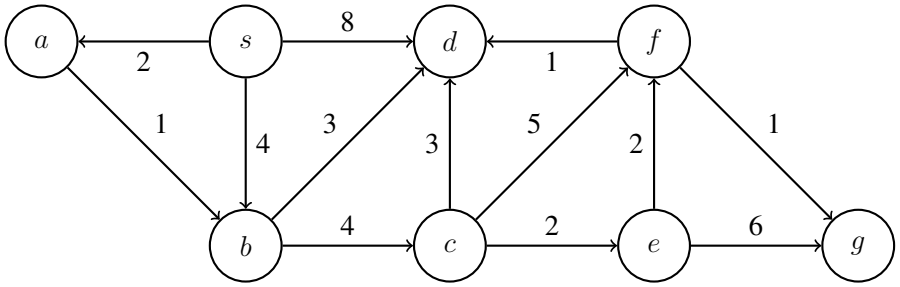
(a) Gegeben sei ein **binärer Baum** T dessen Knoten **paarweise unterschiedliche** Schlüssel enthalten. Geben Sie einen **rekursiven** Algorithmus an, welcher prüft ob T ein binärer **Such**baum ist *(10 Punkte)*.

Hinweis: Ein binärer Suchbaum erfüllt die Eigenschaft, dass der linke Teilbaum eines Knotens im Baum nur Schlüssel enthält, die kleiner sind als der Schlüssel des aktuellen Knotens, während der rechte Teilbaum nur Schlüssel enthält die größer sind.

(b) Gegeben sei ein binärer **Such**baum T dessen Knoten **paarweise unterschiedliche** Schlüssel enthalten und ein beliebiger Knoten k dieses Baumes. Beweisen Sie, dass der Knoten mit dem **kleinsten** Schlüssel im **rechten** Teilbaum an k kein linkes Kind hat *(5 Punkte)*.

Aufgabe 5: Kürzeste Pfade (15 Punkte)

Führen sie **Dijkstras' Algorithmus** auf dem folgenden gewichteten, gerichteten Graphen ausgehend vom Knoten s durch. Die nachfolgende Tabelle soll die gespeicherten Distanzen während der Ausführung angeben. Füllen Sie für jeden Zeilendurchlauf der Hauptschleife, also pro entfernten Knoten aus der Prioritätwarteschlange eine neue Zeile aus.



Initialisierung	s	a	b	c	d	e	f	g
$\delta(s,\cdot) =$	0	∞	∞	∞	∞	∞	∞	∞
1. Schritt $(u = s)$	s	a	b	c	d	e	f	g
$\boldsymbol{\delta(s,\cdot) =}$								
2. Schritt $(\boldsymbol{u =} \quad)$	s	a	b	c	d	e	f	g
$\boldsymbol{\delta(s,\cdot) =}$								
3. Schritt $(\boldsymbol{u =} \quad)$	s	a	b	c	d	e	f	g
$\boldsymbol{\delta(s,\cdot) =}$								
4. Schritt $(\boldsymbol{u =} \quad)$	s	a	b	c	d	e	f	g
$\boldsymbol{\delta(s,\cdot) =}$								
5. Schritt $(\boldsymbol{u =} \quad)$	s	a	b	c	d	e	f	g
$\boldsymbol{\delta(s,\cdot) =}$								
6. Schritt $(\boldsymbol{u =} \quad)$	s	a	b	c	d	e	f	g
$\boldsymbol{\delta(s,\cdot) =}$								
7. Schritt $(\boldsymbol{u =} \quad)$	s	a	b	c	d	e	f	g
$\boldsymbol{\delta(s,\cdot) =}$								
8. Schritt $(\boldsymbol{u =} \quad)$	s	a	b	c	d	e	f	g
$\boldsymbol{\delta(s,\cdot) =}$								

Aufgabe 6: Mystische Funktion (15 Punkte)

Betrachten Sie folgenden Pseudocode der Funktion *myst*, welche als Eingabe ein Array A mit reellen Zahlen als Einträgen erhält.

```
def myst(A):
for i in range(2, A.length) do:
    for j in range(1, i − 1) do:
        for k in range(1, A.length) do:
            if ((|A[i] − A[j]| mod 10) == (A[k] mod 10)) then:
                return true
return false
```

(a) Was berechnet die Funktion *myst* bzw. in welchem Fall gibt sie "true" zurück? (5 Punkte)

(b) Welche asymptotische Laufzeit hat *myst* in Abhängigkeit von n? (4 Punkte)

(c) Geben Sie einen Algorithmus mit gleicher Ausgabe wie *myst* an, dessen asymptotische Laufzeit strikt besser ist als die von *myst*. Was ist die Laufzeit Ihres Algorithmus? (6 Punkte)

Aufgabe 7: Dynamische Programmierung (16 Punkte)

Der aus der Kombinatorik gut bekannte **Binominialkoeffizient** $\binom{n}{k}$ mit $n, k \in \mathbb{N}_0$ und $k \leq n$ lässt sich wie folgt **rekursiv** berechnen.

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

Dabei sind die **Basisfälle** gegeben durch $\binom{n}{0} = \binom{n}{n} = 1$.

a) Geben sie einen Algorithmus nach dem Prinzip der **dynamischen Programmierung** an, welcher den Binominialkoeffizienten $\binom{n}{k}$ in $\mathcal{O}(n \cdot k)$ Zeitschritten berechnet. Vergessen Sie nicht die Laufzeit Ihres Algorithmus' zu begründen. (12 Punkte)

*Hinweis: Sie können dabei **Top-Down** vorgehen und eine Datenstruktur mittels Memoization während der Rekursion füllen oder **Bottom-Up** eine Datenstruktur in einer Vorberechnung mit Teillösungen füllen aus der Sie dann den gewünschten Wert abrufen.*

b) Gegeben sei das Wort $w = abebcababcababdceabab$ sowie das Muster/Pattern $p = ababcabab$. Berechnen Sie das *Verschiebearray* S aus dem Knuth-Morris-Pratt Algorithmus. (4 Punkte)

Aufgabe 8: Zeichenketten Vergleichen (15 Punkte)

Gegeben sei eine **einfach verkettete Liste** L die n (sehr lange) Zeichenketten als Listenelemente enthält. Die Liste L sei **lexikographisch aufsteigend sortiert** (ausgehend vom Listenanfang $L.first$)[2] und die Zeichenketten bestehen aus Zeichen der fünfelementigen Menge $\Sigma := \{a, b, c, d, e\}$. Betrachten sie den folgenden Algorithmus der die Zeichenketten verändert.

Algorithm TRANSFORMLISTDATA(L: **List of Strings**)

```
currentElem ← L.first
while currentElem ≠ null do
    x ← wähle zufälliges Zeichen aus Σ
    s ← currentElem.data
    currentElem.data ← xs          ▷ schreibe x vor Zeichenkette in currentElem
    currentElem ← currentElem.next
```

Geben Sie einen Algorithmus in **Pseudocode** an, der die lexikographische Sortierung der **transformierten** Liste TRANSFORMLISTDATA(L) wieder herstellt und dafür höchstens $\mathcal{O}(n)$ Schritte benötigt. Begründen Sie warum Ihr Algorithmus korrekt ist und höchstens $\mathcal{O}(n)$ Schritte benötigt.

Einschränkung: Gehen Sie davon aus, dass die gespeicherten Zeichenketten **zu lang sind ($\Omega(n)$) um diese in konstanter Zeit zu vergleichen**. Sie können aber einzelne Zeichen zweier Zeichenketten in $\mathcal{O}(1)$ vergleichen.

[2]Die lexikographische Ordnung $\leq_{lex}$ lässt sich rekursiv definieren: Für das leere Wort ϵ und ein beliebiges Wort a gilt $\epsilon \leq_{lex} a$. Für zwei Wörter $a = a_1 a_2 \cdots a_k$ und $b = b_1 b_2 \cdots b_l$ gilt $a \leq_{lex} b$ genau dann, wenn sich entweder a_1 im Alphabet vor b_1 befindet oder a_1 und b_1 identisch sind und für die Restwörter $a_2 \cdots a_k \leq_{lex} b_2 \cdots b_l$ gilt.