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

Klausur Algorithmen und Datenstrukturen

24. August 2022, 14:00 -17:00

Name: ....

Matrikel Nr.: ....

Unterschrift: ....

Erst öffnen wenn die Klausuraufsicht die Erlaubnis erteilt!

- Legen Sie Ihren **Studentenausweis** sichtbar vor sich.
- Schreiben Sie Ihren **Namen** und Ihre **Matrikelnummer** an die vorgesehenen Stellen.
- **Unterschreiben Sie** diese Seite um zu bestätigen, dass Sie die Fragen ohne unerlaubte Hilfsmittel beantwortet haben und die Klausuraufsicht über (gesundheitliche) Probleme informiert haben.
- Dies ist eine **Open Book** Klausur weshalb alle gedruckten und handgeschriebenen Materialien erlaubt sind. **Elektronische Hilfsmittel** sind **nicht** erlaubt.
- Schreiben Sie **lesbar** und nur mit **dokumentenechten** Stiften. Benutzen Sie **keine rote** Farbe und **keinen Bleistift**!
- 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 als Blackbox** verwendet werden.
- **Lesen Sie jede Aufgabe sorgfältig** durch und stellen Sie sicher dass Sie diese verstanden haben!
- Falls Sie eine Frage zur Aufgabenstellung haben, geben Sie der Klausuraufsicht ein **Handzeichen**.
- Schreiben Sie Ihren Namen auf **alle Blätter**!
- 50 Punkte sind ausreichend zum **Bestehen der Klausur**.
- 100 Punkte sind ausreichend für die **beste Note**.

Aufgabe	1	2	3	4	5	6	7	8	**Total**
Maximum	22	12	12	15	15	12	12	20	120
Punkte									

Aufgabe 1: Kurze Fragen *(22 Punkte)*

(a) Im folgenden geht es darum die Schlüssel $k_1 = 12, k_2 = 15$ und $k_3 = 8$ mittels der in der Vorlesung beschriebenen Variante von *Cuckoo Hashing* in einer Hashtabelle der Größe $m = 7$ zu speichern. Geben seien hierzu die beiden Hashfunktionen $h_1(x) = 2\cdot x+1 \mod m$ und $h_2(x) = x+3 \mod m$. Geben Sie in der linken Tabelle den Zustand der Hashtabelle nachdem k_1 und k_2 eingefügt wurden an. In der rechten Tabelle soll der finale Zustand - also nach hinzufügen von k_3 - zu sehen sein. *(4 Punkte)*

0	1	2	3	4	5	6

0	1	2	3	4	5	6

(b) Sei $f : \mathbb{R} \to \mathbb{R}$ eine stetige Funktion mit $f(0) = -1$ und $f(1) = 1$. Beschreiben Sie einen Algorithmus, welcher in $O(\log n)$ Zeit eine Nullstelle von f bis auf $1/n$ genau findet. Ihr Algorithmus soll also eine Zahl x ausgeben, so dass f im Intervall $(x - \frac{1}{n}, x + \frac{1}{n})$ eine Nullstelle hat. Wir nehmen dabei an, dass das Auswerten von f an einer Stelle x (d.h. die Berechnung von $f(x)$) konstante Zeit beansprucht.

Begründen Sie die Korrektheit Ihres Algorithmus. *(6 Punkte)*

(c) Angenommen es gäbe eine Prioritätswarteschlange H deren Operationen folgende Laufzeit hätten: Sowohl `create` wie auch `insert`, `getMin` und `decreaseKey` haben konstante Laufzeit $O(1)$, während ein Aufruf von `deleteMin` eine Laufzeit von $O(\log n)$ hat (wenn n Elemente in der Datenstruktur gespeichert sind).

Welche asymptotische Laufzeit hätte Dijkstras Algorithmus (in Abhängigkeit der Kantenzahl m und der Knotenzahl n) wenn man H als zugrundeliegende Prioritätswarteschlange benutzt. Begründen Sie. *(5 Punkte)*

(d) Beweisen oder Widerlegen Sie folgende Aussage: Heapsort ist *stabil*. *(7 Punkte)*

Hinweise: Gehen Sie davon aus, dass Heapsort die Array-Implementierung aus der Vorlesung für den binären Heap benutzt. Ein Sortieralgorithmus heißt stabil, falls die ursprüngliche Reihenfolge von Elementen mit gleichem Schlüssel beibehalten bleibt. Bsp: Besteht das Array initial aus folgenden key-value-Paaren $[(3, a), (1, r), (1, b)]$, *ist* $[(1, r), (1, b), (3, a)]$ *eine stabile Sortierung, jedoch* $[(1, b), (1, r), (3, a)]$ *nicht.*

Musterlösung

(a)

			15	12		
0	1	2	3	4	5	6

	12		8	15		
0	1	2	3	4	5	6

(b)
```
a = 0
b = 1
for i in {1, ..., log n} do
    x = (a+b)/2
    if f(x) > 0 then
        b = x
    else if f(x) < 0 then
        a = x
    else
        return x
return (a+b)/2
```


Wir haben als Schleifeninvariante, dass $a < b$ und $f(a) < 0$ und $f(b) > 0$. Am Ende hat f also eine Nullstelle in (a, b). Das Intervall (a, b) schrumpft in jeder Iteration um die Hälfte. Am Ende gilt also $b - a < 1/2^{\log n} = 1/n$.

(c) Dijkstra ruft n mal insert/getMin/delMin auf, decreaseKey wird höchstens m mal aufgerufen (und 1 create, das aber für die Laufzeit keine Rolle spielt). Es folgt also folgende Laufzeit: $O(n \cdot \log n + m)$.

(d) Die Aussage ist falsch. Gegenbeispiel wäre das Array $[2, 1, 2]$. Der binäre Heap hätte nach dem Einfügen aller Werte die 1 in der Wurzel, die erste 2 als linkes Kind und die zweite 2 als rechtes Kind. Nun wird die 1 mittels delMin gelöscht. Beim Löschen wird das Wurzelelement mit dem letzten Eintrag, sprich der rechten 2, getauscht und anschließend das neue letzte Element gelöscht. Nun ist diejenige 2 im Wurzelknoten die ursprünglich die rechte 2 war, da die MinHeap Eigenschaft gilt, muss nicht repariert werden. Somit wird diese 2 später vor der anderen im (sortierten) Array stehen, was nicht der ursprünglichen Reihenfolge entspricht.

Aufgabe 2: Landau-Notation *(12 Punkte)*

(a) Gegeben folgende 5 Funktionen in Abhängigkeit von $n \in \mathbb{N}$. Geben Sie eine Sortierung der Funktionen bezüglich der $\mathcal{O}$-Notation an, sprich, für zwei aufeinanderfolgende Funktionen f, g in dieser Sortierung gilt jeweils $f(n) \in \mathcal{O}\big(g(n)\big)$. Die angegebenen Beziehungen müssen weder bewiesen noch begründet werden. *(4 Punkte)*

- $a(n) = 3 \cdot n^2 \cdot \log_2(n) + 4$
- $b(n) = 2^{2\log_2(n)}$
- $c(n) = a(n)/\sqrt{n+1}$
- $d(n) = \log_2(n!)$
- $e(n) = \sqrt{n^5}$

(b) Beweisen oder Widerlegen Sie anhand der Definition der Landau-Notation: $10 \cdot n^9 \in \Omega(n^{10})$ *(4 Punkte)*

(c) Beweisen oder Widerlegen Sie entweder anhand der Definition der Landau-Notation oder anhand der Grenzwert-Charakterisierung: $\sum_{i=0}^{n} (2 \cdot (i + 1)) \in o(n^3)$ *(4 Punkte)*

Musterlösung

(a) $d(n), c(n), b(n), a(n), e(n)$

(b) Die Aussage gilt nicht. Angenommen es gäbe ein c und n_0 so dass die Aussage gilt, dann

$$\begin{aligned} 10n^9 &\geq cn^{10} \\ 10 &\geq cn \\ \frac{10}{c} &\geq n \end{aligned}$$

Wähle also z.B. $n = \max\{10/c, n_0\} + 1$ um einen Widerspruch zur Annahme zu konstruieren.

(c) Die Aussage stimmt:

$$\sum_{i=0}^{n} (2 \cdot (i+1)) = 2\sum_{i=0}^{n+1} i = 2\frac{(n+1)(n+2)}{2} = n^2 + 3n + 2 \leq 6n^2$$

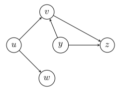
Da $6n^2 \leq cn^3 \Leftrightarrow n \geq 6/c$ (für beliebiges c), wähle $n_0 = \lceil 6/c \rceil$, dann gilt für alle $n \geq n_0$: $\sum_{i=0}^{n} (2 \cdot (i + 1)) \leq 6n^2 \leq cn^3$.

Alternativ: Berechne Summe wie oben, dann:

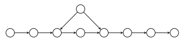
$$\lim_{n\to\infty} \frac{\sum_{i=0}^{n} (2 \cdot (i+1))}{n^3} = \lim_{n\to\infty} \frac{n^2 + 3n + 2}{n^3} = \lim_{n\to\infty} \frac{1}{n} + \frac{3}{n^2} + \frac{2}{n^3} = 0$$

Aufgabe 3: Topologische Sortierung *(12 Punkte)*

(a) Geben Sie eine topologische Sortierung des folgenden Graphen an. *(3 Punkte)*



(b) Wie viele topologische Sortierungen gibt es für den folgenden Graphen? *(3 Punkte)*



(c) Gegeben seien zwei gerichtete Graphen ohne Zyklen (DAGs) $G = (V, E)$ und $G' = (V, E')$ mit gleicher Knotenmenge V, so dass G ein Teilgraph von G' ist, d.h. $E \subseteq E'$. Auf welchem Graphen gibt es mehr Möglichkeiten, die Knotenmenge topologisch zu sortieren? Begründen Sie Ihre Antwort. *(6 Punkte)*

Musterlösung

(a) Z.B. u, w, y, v, z

(b) 3

(c) Eine topologische Sortierung eines Graphen bleibt gültig, wenn man Kanten aus dem Graph entfernt. Daher ist jede topologische Sortierung in G' auch eine in G und es gibt mehr Möglichkeiten, G topologisch zu sortieren.

Aufgabe 4: Rabin-Karp mit Quersummen *(15 Punkte)*

In dieser Aufgabe arbeiten wir mit einer modifizierten Variante des aus der Vorlesung bekannten Rabin-Karp Algorithmus. Sei $x = x_1x_2x_3 \dots x_r$ die Dezimaldarstellung (also zur Basis 10) von x. Dann definieren wir die Quersumme $\overline{x} := \sum_{i=1}^{r} x_i$ (Beispiel: $\overline{1927} = 1 + 9 + 2 + 7 = 19$).

Der hier verwendete Rabin-Karp benutzt als Hashfunktion $h(x) = \overline{x} \mod M$, wobei M ein beliebiger Modulus sein kann. Stimmen Hashwert von Muster und Textausschnitt überein, wird wie auch in der Vorlesung, mithilfe von `TestPosition` auf tatsächliche Gleichheit geprüft.

(a) Sei der Suchtext $T = 12518230051251$ der Länge $n = 14$ und das Muster $P = 251$ der Länge $m = 3$. Die zu benutzende Hashfunktion sei $h(x) = \overline{x} \mod 5$. Geben Sie für alle $s \in \{0, 1, \dots, 11\}$ den Hashwert $h(T[s, s+1, s+2])$ an. *(9 Punkte)*

Hinweis: $T[s, s+1, s+2]$ *ist das Teilwort aus* T *an Positionen* s, $s+1$ *und* $s+2$. *Bsp.* $T[3, 4, 5] = 182$.

(b) Wie oft muss bei der Ausführung von Rabin-Karp die Funktion `TestPosition` für das Beispiel aus der a) aufgerufen werden? *(2 Punkte)*

(c) Geben Sie eine Möglichkeit an um $h(T[s + 1, s + 2, \dots, s + m])$ mit der Hashfunktion $h(x) = \overline{x} \mod M$ in konstanter Zeit zu berechnen, unter der Bedingung, dass der Hashwert des vorherigen Teilstrings $h(T[s, s + 1, \dots, s + m - 1])$ bekannt ist. *(4 Punkte)*

Hinweis: Die Laufzeit einer einzelnen Addition, Subtraktion, Multiplikation, Division oder Modulorechnung wird als konstant angenommen.

Musterlösung

(a) Tabelle:

s	$T[s, s+1, s+2]$	$h(T[s, s+1, s+2])$
0	125	3
1	251	3
2	518	4
3	182	1
4	823	3
5	230	0
6	300	3
7	005	0
8	051	1
9	512	3
10	125	3
11	251	3

(b) Da $h(P) = 3$, folgt dass es 7 `TestPosition` aufrufe gibt.

(c) Da $h(T[s, s + 1, \dots, s + m - 1]) = (T[s] + T[s + 1] + ... + T[s + m - 1]) \mod M$, muss man nur $T[s]$ subtrahieren und $T[s + m]$ addieren, sprich:

$$h(T[s + 1, s + 1, \dots, s + m]) = (T[s + m] - T[s] + h(T[s, s + 1, \dots, s + m - 1])) \mod M$$

Aufgabe 5: Längster Gemeinsamer Teilstring *(15 Punkte)*

Gegeben seien zwei Strings A, B der Länge n über den Zeichen $\{0, 1\}$. Wir möchten die Länge des längsten *gemeinsamen* Teilstrings von A und B berechnen. Ein Teilstring ist dabei eine zusammenhängende Teilfolge von Zeichen von A beziehungsweise B.

Geben Sie einen Algorithmus an der dieses Problem in Zeit $\mathcal{O}(n^2)$ löst und begründen Sie die Laufzeit. *(15 Punkte)*

Alternativ können Sei einen Algorithmus angeben der dieses Problem in Zeit $\mathcal{O}(n^3)$ löst. Dafür erhalten Sie maximal 5 Punkte.

Hinweise: Es gibt eine Lösung die Dynamisches Programmieren Bottom-Up benutzt. Merken Sie Sich für zwei Indizes i, j *nur die maximale Länge eines gemeinsamen Teilstrings von* A, B *der jeweils beim Index* i, j *in* A *bzw.* B *endet.*

Musterlösung

(a) Für jeden zusammenhängenden Teilstring in A führen wir eine Suche nach diesem Teilstring in B aus. Es gibt $\mathcal{O}(n^2)$ solche Teilstrings in A (denn jeder zusammenhängende Teilstring ist durch einen Start- und End-index definiert) und jede Suche nach einem Teilstring dauert $\mathcal{O}(n)$. Insgesamt also $\mathcal{O}(n^3)$.

(b) Sei $\ell(i, j)$ die Länge des längsten gemeinsamen Teilstrings von $A[0..i]$, $B[0..j]$, der bei $A[i]$ und $B[j]$ endet. Im Basisfall setzen wir $\ell(i, 0) = 1$ falls $A[i] = B[0]$ sonst $\ell(i, 0) = 0$. Analog setzen wir $\ell(1, j) = 1$ falls $A[0] = B[j]$ sonst $\ell(0, j) = 0$. Rekursiv setzen wir für $i, j \geq 1$

$$\ell(i,j) := \begin{cases} \ell(i-1, j-1) + 1, & \text{falls } A[i] = B[j] \\ 0, & \text{sonst.} \end{cases}$$

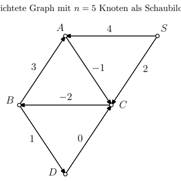
Wir berechnen nun diese Funktion komplett, bottom-up mittels dynamischer Progammierung. Wir speichern die Ergebnisse in einem 2-dimensionalen Array $L[i, j]$.

Algorithm 1 `compute-bottom-up`(A, B)
```
L ← Neues 2-dimensionales n × n-Array
for k = 0 to n − 1 do                               ▷ Basisfälle
    L[0, k] ← (A[0] = B[k])
    L[k, 0] ← (A[k] = B[0])
for i = 1 to n − 1 do                               ▷ Bottom Up Berechnung von L
    for j = 1 to n − 1 do
        if A[i] = B[j] then
            L[i, j] ← L[i−1, j−1] + 1
        else
            L[i, j] ← 0
m ← 0
for i = 1 to n − 1 do                               ▷ grössten Wert berechnen
    for j = 1 to n − 1 do
        if L[i, j] > m then m ← L[i, j]
return m
```

Wir berechnen L gemäss der obigen Rekursion. Zum Schluss iterieren wir durch das gesamte Array L und merken uns den grössten Wert den wir antreffen und geben ihn aus. Beide Schritte dauern jeweils $\mathcal{O}(n^2)$.

Aufgabe 6: Kürzeste Pfade *(12 Punkte)*

Gegeben sei der folgende gerichtete Graph mit $n = 5$ Knoten als Schaubild.



(a) Hat dieser Graph einen negativen Zyklus? Falls ja markieren Sie die Kanten von einem negativen Zyklus (Umkreisung oder Färbung der Kanten). *(2 Punkte)*

(b) Führen Sie auf diesem Graph den *Bellman-Ford* Algorithmus mit Startknoten S aus und tragen Sie *nach jeder* Iteration der äußeren Schleife die bisher berechneten Distanzen von S *in der vorgegeben Tabelle* ein (siehe Lösungsblatt). *(10 Punkte)*

Wichtig: Iterieren Sie in der inneren Schleife des Bellman-Ford Algorithmus die Kanten aufsteigend nach Größe der Gewichte, d.h., stets in der Reihenfolge $-2, -1, 0, 1, 2, 3, 4$ *!*

Musterlösung

(a) Die Kanten $(C, B), (B, D), (D, C)$ bilden einen negativen Zyklus.

(b) Lösungstabelle:

	$\delta(S,S)$	$\delta(S,A)$	$\delta(S,B)$	$\delta(S,C)$	$\delta(S,D)$
Initial	0	∞	∞	∞	∞
$i = 1$	0	4	∞	2	∞
$i = 2$	0	3	0	2	1
$i = 3$	0	3	0	1	1
$i = 4$	0	2	-1	1	0

Aufgabe 7: Mystischer Algorithmus *(12 Punkte)*

Sei $G = (V, E)$ ein *ungerichteter* Graph mit n Knoten. Betrachten Sie den folgenden Algorithmus `Mystery` gegeben als Pseudocode, welcher als Eingabe G sowie einen Knoten $s \in V$ erhält.

Algorithm 2 `Mystery`(G, s)
```
D ← neues Dictionary
D[s] ← 0
for jeden Knoten u ∈ V \ {s} do
    D[u] ← −1
u ← s; i ← 1
while Solange es eine Kante {u, v} ∈ E gibt, so dass D[v] = −1 ist do
    v ← beliebiger Nachbar von u mit D[v] = −1
    D[v] ← D[u] + i
    u ← v; i ← i + 1
```

(a) Wie viele Iterationen kann die while Schleife in Abhängigkeit von n maximal durchlaufen? *(6 Punkte)*

(b) Was ist der größte mögliche Wert in D nach Ausführung von `Mystery`(G, s) asymptotisch in Abhängigkeit von n? *(6 Punkte)*

Musterlösung

(a) $n - 1$

(b) $\Theta(n^2)$

Aufgabe 8: Minimale Spannbäume *(20 Punkte)*

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph, der durch Adjazenzlisten gegeben ist. Angenommen wir wissen, dass G genau einen Zyklus enthält.

(a) Beschreiben Sie einen Algorithmus mit Laufzeit $O(|V|)$, welcher den Zyklus von G ausgibt. Die Ausgabe sollte also die Form $v_1, v_2, \ldots, v_k, v_1$ haben, wobei $v_1, v_2, \ldots, v_k$ paarweise verschieden sind mit $\{v_i, v_{i+1}\} \in E$ für $i = 1, \ldots, k-1$ und $\{v_1, v_k\} \in E$. Begründen Sie die Laufzeit. *(8 Punkte)*

Sei nun $G = (V, E, w)$ ein zusammenhängender, ungerichteter, gewichteter Graph, C ein Zyklus in G und $e \in C$ eine schwerste Kante im Zyklus, d.h. $w(e) \geq w(e')$ für alle $e' \in C$.

(b) Zeigen Sie, dass es einen minimalen Spannbaum von G gibt, welcher e nicht enthält. *(8 Punkte)*

Sei nun $G = (V, E, w)$ ein zusammenhängender, ungerichteter, gewichteter Graph, der durch Adjazenzlisten gegeben ist. Angenommen wir wissen, dass G genau einen Zyklus enthält.

(c) Geben Sie einen Algorithmus an, der in Laufzeit $O(|V|)$ einen minimalen Spannbaum von G berechnet. *(4 Punkte)*

Hinweis: Sie dürfen Teil (a) und (b) auch dann nutzen, wenn Sie sie nicht gelöst haben. Wenn es hilfreich für Sie ist, dürfen Sie annehmen, dass zusätzlich eine Adjazenzmatrix (mit den Gewichten der Kanten als Einträge) gegeben ist.

Musterlösung

(a) Man startet ausgehend von einem beliebigen Knoten s ein DFS. Sei P der Pfad grauer Knoten beginnend bei s, gespeichert als doppelt verkettete Liste. Wir updaten P nach jedem Schritt, d.h. wir hängen einen Knoten hinten an, wenn er grau gefärbt wird und löschen den letzten Knoten, wenn er schwarz gefärbt wird. Sobald man eine Rückwärtskante entdeckt, d.h. man von einem Knoten u ausgehend einen grauen Knoten v entdeckt, geben wir zunächst v und danach P von hinten nach vorne aus, bis wir in P wieder auf v treffen.

Mit Breitensuche: Jeder Knoten merkt sich, von welchem Knoten aus er markiert wurde. Wenn man dann von u aus auf einen Knoten v stößt, der bereits von w aus markiert wurde, geht man von v aus sowohl über u als auch über w den Pfad der Vorgänger zurück Richtung Wurzel s, bis man auf einen gemeinsamen Knoten x stößt. Der gesuchte Kreis ist dann der Pfad von x über u nach v zurück über w nach x.

Die Laufzeit wird dominiert durch den DFS/BFS welcher Laufzeit $O(m + n)$ hat. Da $m = n$ gilt, ist die Laufzeit $O(n)$.

(b) Sei T ein MST von G welcher $e = \{u, v\}$ enthält. $T \setminus \{e\}$ besteht dann aus zwei Zusammenhangskomponenten A (alle Knoten die von u aus erreichbar sind) und B (alle Knoten die von v aus erreichbar sind). Es gibt eine Kante $e' \in C$, welche eine Schnittkante von (A, B) ist, d.h. beide Komponenten verbindet. $(T \setminus \{e\}) \cup \{e'\}$ ist also ein Spannbaum und ist wegen $w(e') \leq w(e)$ nicht schwerer als T.

(c) Wir nutzen den Algorithmus von (a) um den Zyklus auszugeben. Dann gehen wir durch den Zyklus und löschen eine schwerste Kante. Der MST besteht dann aus allen übrigen Kanten.

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

Klausur Algorithmen und Datenstrukturen

25. Februar 2021, 9:00 -12:00

Name: ....

Matrikel Nr.: ....

Unterschrift: ....

Erst öffnen, wenn die Klausuraufsicht die Erlaubnis erteilt!

- Schreiben Sie Ihren **Namen** und Ihre **Matrikelnummer** an die vorgesehenen Stellen.
- **Unterschreiben Sie** diese Seite um zu bestätigen, dass Sie die Fragen ohne unerlaubte Hilfsmittel beantwortet haben und die Klausuraufsicht über (gesundheitliche) Probleme informiert haben.
- Dies ist eine **Open Book** Klausur, weshalb alle gedruckten und handgeschriebenen Materialien erlaubt sind. **Elektronische Hilfsmittel** sind **nicht** erlaubt.
- Schreiben Sie **lesbar** und nur mit **dokumentenechten** Stiften. Benutzen Sie **keine rote** Farbe und **keinen Bleistift**!
- 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, Zeichnen Sie ...** zeigen an, dass Sie lediglich die geforderte Antwort oder Zeichnung und keine Begründung liefern müssen (falls nicht explizit verlangt).
- 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 als Blackbox** verwendet werden.
- **Lesen Sie jede Aufgabe sorgfältig** durch und stellen Sie sicher, dass Sie diese verstanden haben!
- Falls Sie eine Frage zur Aufgabenstellung haben, geben Sie der Klausuraufsicht ein **Handzeichen**.
- Schreiben Sie Ihren Namen auf **alle Blätter**!
- Füllen Sie die **Erklärung über Ihren gesundheitlichen Status** aus.

Aufgabe	1	2	3	4	5	6	7	8	**Total**
Maximum	21	17	18	15	10	14	16	9	120
Punkte									

Aufgabe 1: Kurze Fragen *(21 Punkte)*

(a) Sei $h(x, i) := x + i \mod 7$ eine Hashfunktion mit linearem Sondieren zur Kollisionsauflösung. Fügen Sie die Schlüssel $44, 45, 79, 55, 91, 18$ mittels h in die Tabelle ein. *(3 Punkte)*

0	1	2	3	4	5	6

(b) Geben Sie einen *gewichteten, ungerichteten* Graphen G mit *positiven* Gewichten an und markieren Sie einen Knoten v und einen Shortest Path Tree von v in G, so dass *(3 Punkte)*

- G *kein* Baum ist,
- der minimale Spannbaum von G eindeutig ist und
- ihr markierter Shortest Path Tree dem minimalen Spannbaum von G entspricht.

(c) Gegeben seien zwei Arrays A und B mit $|A| = m$ und $|B| = n$ und $m \leq n$. Die Einträge der Arrays seien natürliche Zahlen. Man möchte herausfinden, ob es eine Zahl gibt, die sowohl in A als auch in B vorkommt. Geben Sie einen möglichst effizienten Algorithmus für dieses Problem an ...

(i) unter der Annahme, dass finden und einfügen in einer Hashtabelle $O(1)$ Zeitschritte benötigt, solange der Load der Hashtabelle $O(1)$ ist. *(4 Punkte)*

(ii) ohne Nutzung von Hashing. *(5 Punkte)*

Analysieren Sie jeweils die Laufzeit als Funktion von m und n.

(d) Gegeben seien zwei Rot-Schwarz Bäume T_1 und T_2 mit gleicher *Schwarztiefe* h. Zudem seien alle Schlüssel in T_1 *echt kleiner* als alle Schlüssel in T_2.

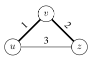
Geben Sie einen Algorithmus an welcher T_1 und T_2 in $\mathcal{O}(h)$ Zeitschritten in einen *gültigen* Rot-Schwarz Baum überführt der *genau* die Schlüssel aus T_1 und T_2 enthält. Erklären Sie kurz die Laufzeit und warum der Algorithmus einen gültigen Rot-Schwarz Baum zurück gibt. *(6 Punkte)*

Musterlösung

(a) keys $i = \{44, 45, 79, 55, 91, 18\}$

$h(44) = 44 \mod 7 = 2$, $h(45) = 45 \mod 7 = 3$, $h(79) = 79 \mod 7 = 2$, aber 2 ist schon mit 44 belegt, nun wird Feld 3 geprüft, welches ebenfalls belegt ist. Entsprechend wird 79 in Feld 4 gespeichert. $h(55) = 55 \mod 7 = 6$, $h(91) = 91 \mod 7 = 0$, $h(18) = 18 \mod 7 = 4$, da 4 besetzt ist gehen wir weiter zu 5.

91	-	44	45	79	18	55
0	1	2	3	4	5	6

(b) 

Die fett gedruckten Kanten zeigen den MST und den Shortest Path Tree von v.

(c) (i) Man alloziiert eine Hashtabelle der Größe n und hasht die Werte von A in diese Tabelle. Danach testet man für jede Zahl in B, ob sie in der Hashtabelle ist. Die Gesamtkosten betragen $O(m + n)$.

(ii) Man sortiert A in Zeit $O(m \log m)$. Nun testet man mittels binärer Suche für jede Zahl in B, ob sie in A liegt. Dies kostet $O(n \log m)$. Insgesamt beträgt die Laufzeit also $O(m \log m + n \log m) = O(n \log m)$.

(d) Zuerst konstruieren wir uns einen Rot-Schwarz Baum T', welcher einen neu erzeugten (schwarzen) Wurzelknoten v bekomtmt. Dieser Wurzelknoten führt T_1 als linken und T_2 als rechten Teilbaum. Also, $\mathsf{root}(T') = v$, $\mathsf{LeftChild}(v) = \mathsf{root}(T_1)$ und $\mathsf{RightChild}(v) = \mathsf{root}(T_2)$. Der Schlüssel von v spielt hierbei keine Rolle. Somit haben wir einen Rot-Schwarz Baum mit folgenden Eigenschaften erstellt (wir ignorieren den Schlüssel von v hier):

- Der Wurzelkoten v ist schwarz.
- Alle Schlüssel links vom Wurzelknoten sind kleiner als die Schlüssel rechts vom Wurzelknoten.
- Da T_1 und T_2 valide Rot-Schwarz Bäume mit Schwarztiefe h sind, hat T' die Schwarztiefe $h + 1$.

Nachem wir T' erstellt haben, löschen wir den Wurzelknoten v um nur noch Schlüssel aus T_1 und T_2 zu haben. Daraus ensteht ein Rot-Schwarz Baum T mit den gewünschten eigenschaften.

Da die Löschoperation in $O(h)$ Schritten ausgeführt werden kann, ergibt sich eine Gesamtlaufzeit von $O(h)$.

Aufgabe 2: O-Notation *(17 Punkte)*

(a) Gegeben sei der folgende Pseudocode.

Algorithm 1 `myst-div(n)`

```
1: while n > 1 do
2:     n = n/3                ▷ Dieser Schritt kostet eine Zeiteinheit
3: return n
```

Nehmen Sie an, dass die Laufzeit $T(n)$ der Anzahl der Divisionen in Zeile 2 entspricht. Geben Sie die *exakte* Laufzeit $T(n)$ der Funktion `myst-div(n)` an. Zeigen Sie dann anhand der Definition der O-Notation (d.h. ohne Grenzwerte), dass $T(n) \in O(\log_{100} n)$. *(5 Punkte)*

Hinweis: Sie dürfen den Definitionsbereich von `myst-div` *auf die Menge* $\{3^k \mid k \in \mathbb{N}\}$ *beschränken.*

(b) Geben Sie an, ob die folgenden Aussagen wahr oder falsch sind. Beweisen oder widerlegen Sie die Aussagen anhand der Definition der O-Notation oder anhand der Grenzwert-Charakterisierung.

(i) $2\sqrt{n} + \log(n) \in o(n)$ *(4 Punkte)*

(ii) $2^{2n} \in \Theta(2^n)$ *(3 Punkte)*

(iii) $a^n \in \omega(n^k)$, for every real $a > 1$ and integer $k \geq 1$ *(5 Punkte)*

Musterlösung

1. Für $n = 3^k$ führt der Algorithmus genau $k = \log_3 n$ Divisionen aus, d.h. $T(n) = \log_3 n$. Es gilt $\log_3 n = \frac{\log_{100} n}{\log_{100} 3}$, d.h. mit $c = (\log_{100} 3)^{-1}$ und $n_0 = 1$ gilt $\log_3 n \leq c \log_{100} n$ für alle $n \geq n_0$. Also ist $\log_3 n \in O(\log_{100} n)$.

2. (a) Wahr. Mit L'Hospital erhält man

$$\lim_{n\to\infty} \frac{2\sqrt{n} + \log(n)}{n} = \lim_{n\to\infty} \frac{(2\sqrt{n} + \log(n))'}{(n)'} = \lim_{n\to\infty} \frac{1}{\sqrt{n}} + \frac{1}{n \ln 2} = 0\,.$$

(b) Falsch. Es gilt

$$\lim_{n\to\infty} \frac{2^{2n}}{2^n} = \lim_{n\to\infty} 2^n = \infty\,.$$

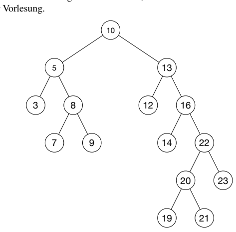
Daraus folgt $2^{2n} \notin O(2^n)$ und deshalb $2^{2n} \notin \Theta(2^n)$.

(c) Wahr. Bezeichne $f^{(k)}$ die k-te Ableitung einer Funktion f. Mit L'Hospital erhält man

$$\lim_{n\to\infty} \frac{a^n}{n^k} = \lim_{n\to\infty} \frac{(a^n)^{(k)}}{(n^k)^{(k)}} = \lim_{n\to\infty} \frac{(\ln a)^k a^n}{k!} = \infty\,.$$

Aufgabe 3: Traversierung Binärer Suchbäume *(18 Punkte)*

(a) Betrachten Sie den folgenden binären Suchbaum. Nummerieren Sie die Knoten entsprechend der Besuchsreihenfolge einer *Pre-Order*, *In-Order* und *Post-Order DFS-Traversierung*, analog zur Vorlesung. *(6 Punkte)*



(b) Zeigen Sie, dass es (wenn man die Suchschlüssel ignoriert) *genau einen* nicht-leeren binären Suchbaum gibt bei dem die Besuchsreihenfolgen der *Pre-Order*, *In-Order* und *Post-Order DFS-Traversierungen* gleich sind. *(5 Punkte)*

Hinweis: Teilpunkte falls Sie diesen Baum angeben, aber nicht die Eindeutigkeit beweisen.

(c) Sei T ein Binärbaum in dem jeder Knoten *entweder* 2 Kindknoten *oder* keine Kindknoten hat. Zudem speichern wir zu jedem *jedem* Knoten v von T einen Wert $\mathsf{Post}(v)$, der die Position von v in der Besuchsreihenfolge einer *Post-Order* DFS-Traversierung enthält.

Geben Sie einen Algorithmus an welcher für einen Knoten v die Größe des *rechten Teilbaumes* von v in *konstant* vielen Zeitschritten bestimmt. Erklären Sie Ihren Algorithmus.

Beispiel: 5 ist die richtige Ausgabe für den Knoten mit Schlüssel 16. *(7 Punkte)*

Musterlösung

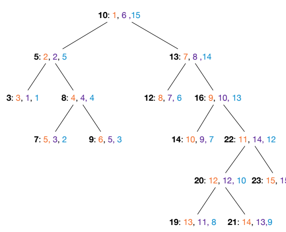


Abbildung 1: Lösungsbaum Teil (a)

(a) Abbildung 1 gibt die Lösung an. Die pre-order, in-order, and post-order Besuchsreihenfolgen sind jeweils in Orange, Lila , und Blau.

(b) Ein binärer Suchbaum mit einem einzigen Knoten erfüllt die Anforderung. Ein binärer Suchbaum mit $n > 1$ Knoten kann diese Anforderung nicht erfüllen da für die Wurzel r gilt $\mathsf{Pre}(r) = 1 \neq n = \mathsf{Post}(r)$.

(c) Wenn v keine Kinder hat, gib 0 aus. Sonst sei u das linke Kind von v. Gib $\mathsf{Post}(v) - \mathsf{Post}(u) - 1$ aus.

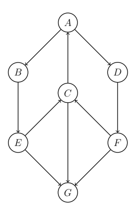
Der Algorithmus ist offensichtlich korrekt wenn v keine Kindknoten hat. Ansonsten basiert die Korrektheit auf der Post-order einer DFS-Traversierung. Das linke Kind u von v bekommt seine Post-order Nummer nachdem alle Knoten im linken Teilbaum von der DFS abgearbeitet sind. Der Knoten v erhält seine Post-order Nummer nachdem der linke *und* der rechte Teilbaum von v abgearbeitet sind. Wenn r die Größe des rechten Teilbaumes ist erhält v seine Nummer $r + 1$ Schritte nachdem u seine Nummer erhält, nämlich nachdem zusätzlich alle r Knoten des rechten Teilbaumes und v selbst abgearbeitet sind. Damit ist

$$\begin{aligned} & \mathsf{Post}(v) = \mathsf{Post}(u) + r + 1 \\ \iff \quad & r = \mathsf{Post}(v) - \mathsf{Post}(u) - 1. \end{aligned}$$

Aufgabe 4: Graph-Traversierung *(15 Punkte)*

(a) Betrachten Sie den unten stehenden Graph. Führen Sie eine *Tiefensuche* auf diesen Graph mit dem Startknoten A aus. Beschriften Sie die Kanten jeweils mit B, V, R beziehungsweise Q wenn es sich um eine Baum-, Vorwärts-, Rückwärts- beziehungsweise Querkante handelt.

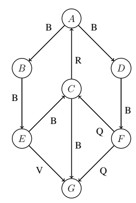
Wichtiger Hinweis: Falls Sie die Wahl zwischen mehreren Knoten haben, wählen Sie als Nachfolgeknoten jeweils den alphabetisch kleinsten. *(6 Punkte)*



(b) Sei $G = (V, E)$ ein *zusammenhängender, ungerichteter* Graph. Sei $u \in V$. Wenn wir eine Tiefensuche mit Startknoten u ausführen erhalten wir einen Baum T. Angenommen eine Breitensuche mit gleichem Startknoten u produziert exakt den gleichen Baum T. Beweisen Sie, dass dann $G = T$ gilt. *(9 Punkte)*

Musterlösung

(a)



(b) Sei T der Baum welcher sowohl von der Breitensuche (BFS) als auch der Tiefensuche (DFS) zurückgegebenen wird. T ist also *sowohl* BFS-Baum *als auch* DFS-Baum, was wir im folgenden ausnutzen.

Formulierungsmöglichkeit A: Für einen Widerspruchsbeweis nehmen wir an, dass G eine Kante $\{x, y\}$ hat die nicht Teil von T ist. Eine Eigenschaft der DFS in ungerichteten Graphen ist, dass entweder x ein Vorgänger von y in T sein muss oder umgekehrt, denn die DFS hat in ungerichteten Graphen nur Baum-kanten oder Rückwärts-kanten. Sei o.B.d.A. x der Vorgänger von y in T (1).

Betrachten wir nun die Eigenschaften von T als BFS-Baum. Aufgrund von (1) können x und y nicht in der gleichen BFS Schicht sein. Da x und y aber eine Kante teilen, sind diese auf jeden Fall in *aufeinanderfolgenden* BFS-Schichten (2). Aus den beiden Eigenschaften (1),(2) folgt, dass die Kante $\{x, y\}$ in T sein müsste, ein Widerspruch.

Formulierungsmöglichkeit B: Sei $\{x, y\} \in E$ eine beliebige Kante von G. In *ungerichteten* Graphen gibt es bzgl. der DFS keine Querkanten. Damit ist entweder x ein Vorgänger von y in T oder umgekehrt (1). Nun ist T aber insbesondere ein BFS-Baum, d.h., x und y sind in aufeinanderfolgenden Schichten in T (2). Aufgrund von (1) und (2) muss $\{x, y\}$ eine Baumkante sein. Damit sind alle Kanten von G Baumkanten und damit muss $G = T$ sein.

Formulierungsmöglichkeit C: Angenommen G ist kein Baum. Damit hat G mindestens einen Zyklus. Sei o.B.d.A. v_1 der Knoten auf einem solchen Zyklus in G der am nähesten zur Wurzel von T ist. Sei $v_1, \ldots, v_k$ für $k \geq 3$ die Folge von Knoten des Zyklus, d.h., $\{v_i, v_{i+1}\} \in E$ und $\{v_k, v_1\} \in E$. Da v_k und v_2 jeweils voneinander erreichbar sind, selbst ohne die Kanten $\{v_1, v_2\}$ und $\{v_1, v_k\}$, enthält T als DFS-Baum nur *entweder* $\{v_1, v_2\}$ *oder* $\{v_1, v_k\}$, der jeweils andere Knoten wird über den Kreis exploriert.

Der Baum T gesehen als BFS-Baum müsste sowohl $\{v_1, v_2\}$ *als auch* $\{v_1, v_k\}$ enthalten (andernfalls gäbe es einen Zyklusknoten der näher an der Wurzel von T ist). Damit kann T

nicht sowohl BFS Baum als auch DFS Baum sein, ein Widerspruch zur Vorraussetzung.

Aufgabe 5: Kürzeste Pfade *(10 Punkte)*

(a) Geben Sie einen *gewichteten, gerichteten* Graphen G und markieren Sie einen Knoten v von G, so dass

- G kein Baum ist,
- es mindestens eine Kante mit negativem Gewicht gibt,
- G keine negativen Kreise hat,
- man durch Anwendung von Dijkstra auf G mit v als Wurzel einen Shortest Path Tree von v in G erhält.

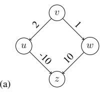
Markieren Sie den Shortest-Path-Tree in G, den Dijkstras' Algorithmus ausgibt. *(4 Punkte)*

(b) Geben Sie einen *gewichteten, gerichteten* Graphen G und markieren Sie einen Knoten v von G, so dass

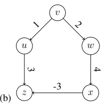
- G kein Baum ist,
- es mindestens eine Kante mit negativem Gewicht gibt,
- G keine negativen Kreise hat,
- man durch Anwendung von Dijkstra auf G mit v als Wurzel *keinen* Shortest Path Tree von v in G erhält.

Markieren Sie den Baum in G, den Dijkstras' Algorithmus ausgibt. Markieren Sie den Knoten, für den die berechnete Distanz nicht minimal ist. Geben Sie die kürzeste Distanz von v zu diesem Knoten sowie die von Dijkstra berechnete Distanz. *(6 Punkte)*

Musterlösung



(a)



(b)

Dijkstra would start by putting the edge (v, u) in the shortest path tree. Then (v, w). Then (u, z). But (u, z) should not be in the shortest path tree, since the path (v, w, x, z) is shorter than (v, u, z).

Aufgabe 6: Mystischer Algorithmus *(14 Punkte)*

Betrachten Sie den folgenden Algorithmus, welcher als Input ein Array A der Länge n sowie eine natürliche Zahl m erhält mit der Eigenschaft, dass die Einträge in A natürliche Zahlen zwischen 0 und $m-1$ sind.

Algorithm 2 `myst`

```
1: B = [0] · m                          ▷ create an array of length m with each entry 0.
2: for i = 0 to n − 1 do
3:     B[A[i]] = B[A[i]] + 1
4: ℓ = 0
5: for j = 0 to m − 1 do
6:     if B[j] > 0 then
7:         for k = 0 to B[j] − 1 do
8:             A[ℓ + k] = j
9:     ℓ = ℓ + B[j]
```

(a) Welche Bedeutung hat der Wert $B[j]$ (nach Zeile 3) für ein $j \in \{0, \ldots, m-1\}$? *(3 Punkte)*

(b) Beschreiben Sie in einem kurzen Satz, was `myst` tut. *(3 Punkte)*

(c) Geben Sie die asymptotische Laufzeit von `myst` als Funktion von n und m an und begründen Sie diese. *(4 Punkte)*

(d) Beschreiben Sie die Vor- und Nachteile von `myst` gegenüber anderen Ihnen bekannten Algorithmen, welche die gleiche Funktion haben. *(4 Punkte)*

Musterlösung

(a) Der Wert $B[j]$ entspricht der Anzahl an Vorkommen des Wertes j in A.

(b) `myst` ist ein Sortieralgorithmus (entspricht Counting Sort).

(c) Zeile 1 benötigt Laufzeit $O(m)$, Zeilen 2-3 Laufzeit $O(n)$. Die Schleife in Zeile 5 wird m mal durchlaufen und die innere Schleife (Zeile 7) *insgesamt* n mal. Zeilen 5-9 benötigen also Laufzeit $O(m+n)$. Die Gesamtlaufzeit ist also $O(m+n)$.

(d) Die beste Sortierlaufzeit für allgemeine Zahlbereiche ist $O(n \log n)$ (z.B. Mergesort). Für $m = o(n \log n)$ ist Counting Sort daher besser, für $m = \omega(n \log n)$ (bzw. wenn nichts über eine obere Schranke von m bekannt ist) ist Mergesort besser. Counting Sort hat die gleiche Laufzeit wie Bucket Sort. Allerdings ist Counting Sort in dieser Form nur für das Sortieren von Zahlen geeignet, nicht für Sortieren von allgemeinen Dateien nach Sortierschlüssel.

Aufgabe 7: Dynamische Programmierung *(16 Punkte)*

Gegeben sei eine Folge ganzer Zahlen $S = (s_1, \ldots, s_n)$.

(a) Geben Sie einen Algorithmus an (nach dem Prinzip des Dynamischen Programmierens) der *die Länge* einer *maximalen aufsteigenden Teilfolge* von S in $\mathcal{O}(n^2)$ Zeitschritten ausgibt. Dass heißt, die Länge k einer längsten Folge $(s_{i_1}, \ldots, s_{i_k})$ sodass $s_{i_1} \leq \cdots \leq s_{i_k}$ und $i_1 < \cdots < i_k$. Begründen Sie die Laufzeit. *(10 Punkte)*

Beispiel: Eine max. aufsteigende Teilfolge von $S = (3,6,9,4,2,1,5,7,8)$ ist $(3,4,5,7,8)$ und die richtige Ausgabe für S ist somit 5.

(b) Geben Sie einen Algorithmus an, der *die Länge* einer *maximalen auf- und wieder absteigenden Teilfolge* von S in $\mathcal{O}(n^2)$ Zeit ausgibt. Dass heißt, die Länge k einer längsten Folge $(s_{j_1}, \ldots, s_{j_k})$ sodass $j_1 < \cdots < j_k$ und für ein $\ell \in \{1, \ldots, k\}$ gilt $s_{j_1} \leq \cdots \leq s_{j_\ell} \geq \cdots \geq s_{j_k}$. Begründen Sie die Laufzeit. *(6 Punkte)*

Beispiel: Eine maximale auf- und absteigende Teilfolge von $S = (3,6,9,4,2,1,5,7,8)$ ist $(3,6,9,4,2,1)$ und die richtige Ausgabe für S ist somit 6.

Musterlösung

(a) Sei $k(j)$ die Länge eine längsten Teilfolge von $S_j := (s_1, \ldots, s_j)$ die mit dem Wert s_j endet. Wir können $k(j)$ rekursiv wie folgt bestimmen. Im Basisfall haben wir $k(1) = 1$. Für $j > 1$ setzen wir

$$k(j) := 1 + \max_{0 \leq \ell < j \text{ und } s_\ell \leq s_j} k(\ell).$$

(Formal müssten wir noch $k(0) = 0$ und $s_0 := -\infty$ setzen aber im Pseudocode unten lösen wir den Randfall anders). Um die längste Teilfolge von $S_j := (s_1, \ldots, s_j)$ die auf den Wert s_j endet zu finden, prüft obige Rekursion die Längen $1+k(\ell)$ für alle passenden $s_\ell \leq s_j$ und wählt diejenige Option die $k(j)$ maximiert. Wir wenden unser Standardvorgehen für diese Rekursion an. Sei dazu `memo`$[j]$ ein Dictionary welches die Werte von $k(j)$ speichert.

Algorithm 3 `subseq`(j) ▷ *`memo` sei global gegeben*

```
if j = 1 then return 1                        ▷ Basisfall
if memo[j] = Null then                        ▷ Ergebnis liegt noch nicht vor
    k ← 1                                     ▷ Mindestwert von 1
    for ℓ ← 1 to j − 1 do
        if s_ℓ ≤ s_j then
            k ← max(k, 1 + subseq(ℓ))         ▷ Behalte den besseren Wert
    memo[j] = k                               ▷ Ergebnis speichern
return memo[j]
```


Das gewünschte Ergebnis kann dann schlussendlich mit folgender Formel bestimmt werden

$$\max_{1 \leq j \leq n} \texttt{subseq}(j).$$

Wenn man die Dauer der rekursiven Unteraufrufe vernachlässigt benötigt die Berechnung von `subseq(j)` höchstens $\mathcal{O}(n)$ Zeitschritte. Die *Gesamtanzahl* rekursiver Aufrufe von `subseq(j)` ist höchstens n, denn danach liegen alle Werte `subseq(j)` für $j = 1, \ldots, n$ in `memo`$[j]$ vor. Insgesamt also $\mathcal{O}(n^2)$ Zeitschritte.

(b) Wie in Teil (a) berechnen wir die Werte $k(j)$ welcher die Länge einer längsten Teilfolge $(s_{i_1}, \ldots, s_{i_k})$ von S ist, sodass $s_{i_1} \leq \cdots \leq s_{i_k}$ und $i_1 < \cdots < i_k = j$.

Symmetrisch zu Teil (a) berechnen wir die Werte $k'(j)$ welcher die Länge einer längsten Teilfolge $(s_{i_1}, \ldots, s_{i_{k'}})$ von S ist, sodass $s_{i_1} \geq \cdots \geq s_{i_{k'}}$ und $j = i_1 < \cdots < i_{k'}$.

Dies lässt sich einfach bewerkstelligen indem man die Folge S umdreht und den gleichen Algorithmus aus (a) benutzt.

Die Länge der geforderten Sequenz ist dann gegeben durch

$$\max_{j=1\ldots n} k(j) + k'(j).$$

Die Berechnung der Werte $k(j)$ und $k'(j)$ für alle $j = 1, \ldots, n$ dauert jeweils $\mathcal{O}(n^2)$ wie in (a). Die Berechnung des obigen Maximums dauert lediglich $\mathcal{O}(n)$. Insgesamt also $\mathcal{O}(n^2)$.

Aufgabe 8: Rabin-Karp Algorithmus *(9 Punkte)*

Sei $T = 334710367$ der Text und $P = 103$ das zu suchende Muster der Länge $m = 3$ jeweils gegeben zur Basis $b = 10$. Die Hashfunktion ist der Modulus mit $M = 11$. Sei

$$t_s := T\big[s \ldots (s+m-1)\big] \mod 11$$

der vom Rabin-Karp Algorithmus in Iteration s genutzte Hashwert eines Teilstrings von T.

Geben Sie die fehlenden Werte von t_s in der gegebenen Tabelle an. Setzen Sie eine Markierung ($\times$) wenn die Hashwerte von P und $T\big[s \ldots (s+m-1)\big]$ in Iteration s übereinstimmen. Setzen Sie außerdem eine Markierung wenn in Iteration s das Muster P in T erkannt wird.

Iteration s	0	1	2	3	4	5	6
Hash value t_s	4						
Hash match?	$\times$						
Pattern match?							

Musterlösung

Iteration s	0	1	2	3	4	5	6
Hash value t_s	4	6	9	6	4	3	4
Hash match?	$\times$				$\times$		$\times$
Pattern match?					$\times$		

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

Klausur Algorithmen und Datenstrukturen

27. August 2020, 10:00 -13:00

Name: ....

Matrikel Nr.: ....

Unterschrift: ....

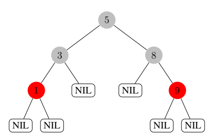
Erst öffnen, wenn die Klausuraufsicht die Erlaubnis erteilt!

- Schreiben Sie Ihren **Namen** und Ihre **Matrikelnummer** an die vorgesehenen Stellen.
- **Unterschreiben Sie** diese Seite um zu bestätigen, dass Sie die Fragen ohne unerlaubte Hilfsmittel beantwortet haben und die Klausuraufsicht über (gesundheitliche) Probleme informiert haben.
- Dies ist eine **Open Book** Klausur, weshalb alle gedruckten und handgeschriebenen Materialien erlaubt sind. **Elektronische Hilfsmittel** sind **nicht** erlaubt.
- Schreiben Sie **lesbar** und nur mit **dokumentenechten** Stiften. Benutzen Sie **keine rote** Farbe und **keinen Bleistift**!
- 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, Zeichnen Sie ...** zeigen an, dass Sie lediglich die geforderte Antwort oder Zeichnung und keine Begründung liefern müssen (falls nicht explizit verlangt).
- 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 als Blackbox** verwendet werden.
- **Lesen Sie jede Aufgabe sorgfältig** durch und stellen Sie sicher, dass Sie diese verstanden haben!
- Falls Sie eine Frage zur Aufgabenstellung haben, geben Sie der Klausuraufsicht ein **Handzeichen**.
- Schreiben Sie Ihren Namen auf **alle Blätter**!
- Füllen Sie die **Erklärung über Ihren gesundheitlichen Status** aus.

Aufgabe	1	2	3	4	5	6	7	8	**Total**
Maximum	18	19	17	10	17	12	15	12	120
Punkte									

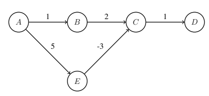
Aufgabe 1: Kurze Fragen *(18 Punkte)*

(a) Führen Sie auf folgendem Rot-Schwarz Baum die Operation `delete`(5) aus. Zeichnen Sie den resultierenden Baum. Sie können neben die Knoten die Farben schreiben. *(4 Punkte)*



(b) Betrachten Sie den folgenden *gerichteten, gewichteten* Graphen G. Führen Sie den Bellman-Ford Algorithmus auf G mit Startknoten A aus. Geben Sie die berechneten Distanzen aller Knoten *nach jeder* Iteration der äußeren Schleife an. *(4 Punkte)*

Hinweis: Die Iteration über die Kanten in der inneren Schleife ist nicht eindeutig. Wir fordern in dieser Aufgabe, dass Sie über die Kanten in alphabetischer Reihenfolge des Ausgangsknotens der Kante iterieren, d.h., die Kante (A, E) wird vor der Kante (B, C) iteriert.

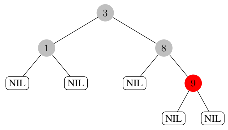


(c) Zeichnen Sie einen *ungerichteten, gewichteten* Graphen $G = (V, E, w)$ und markieren Sie einen Startknoten $s \in V$ so, dass sich jeder *"Shortest Path Tree"* mit Wurzel s in G von jedem *minimalen Spannbaum* von G unterscheidet. *(4 Punkte)*

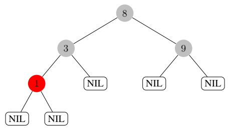
(d) Gegeben sei ein *ungerichteter, ungewichteter* (nicht notwendig zusammenhängender) Graph $G = (V, E)$. Wir möchten testen, ob dieser Graph *fast* ein Spannbaum ist. Darunter verstehen wir, dass G aus einem Spannbaum und höchstens c vielen zusätzlichen Kanten besteht, wobei $c \in O(1)$ eine gegebene Konstante ist. Beschreiben Sie einen Algorithmus der diesen Test in $\mathcal{O}(|V|)$ Zeit durchführt und begründen Sie die Laufzeit. *(6 Punkte)*

Musterlösung

(a) Wir suchen zunächst den Vorgänger v der Wurzel (alternativ Nachfolger). Dies ist der Knoten mit Schlüssel 3. Wir ersetzen den Schlüssel der Wurzel durch 3 und löschen v. Der Knoten mit Schlüssel 1 wird dabei schwarz gefärbt.



Wählt man den Nachfolger statt den Vorgänger, erhält man folgenden Baum:

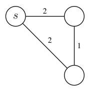


(b) Nach 1. Iteration: $\delta(A,A)=0$, $\delta(A,B)=1$, $\delta(A,C)=2$, $\delta(A,D)=4$, $\delta(A,E)=5$.

Nach 2. Iteration: $\delta(A,A)=0$, $\delta(A,B)=1$, $\delta(A,C)=2$, $\delta(A,D)=3$, $\delta(A,E)=5$.

Nach 3., 4. Iteration: $\delta(A,A)=0$, $\delta(A,B)=1$, $\delta(A,C)=2$, $\delta(A,D)=3$, $\delta(A,E)=5$.

(c)



(d) Wir führen eine leicht modifizierte Tiefensuche durch, bei der wir (als zusätzlichen Parameter) einen Zähler der besuchten Kanten mitführen. Jeder Aufruf der Rekursion erhält den

aktuellen Zählerstand und gibt die neue Anzahl besuchter Kanten zurück (aber maximal $|V|+c$). Jedem rekursiven Aufruf auf einen weißen Knoten wird der aktuelle Zählerstand plus eins mitgegeben (initialer Aufruf mit 0!). Nach jedem rekursiven Aufruf wird der Zählerstand auf den Rückgabewert aktualisiert. Der Zähler sorgt dafür, dass wir die Tiefensuche nach spätestens $|V|+c$ besuchten Kanten abbrechen (es werden keine neuen rekursiven Aufrufe mehr gestartet wenn der Zähler diesen Wert erreicht).

Nach dem Traversierungsalgorithmus testen wir noch ob alle Knoten besucht wurden (keine weißen Knoten mehr). Wir geben `True` zurück (G fast ein Spannbaum) falls die Anzahl besuchter Kanten *kleiner gleich* $|V|-1+c$ ist (mehr Kanten kann ein "Fast-Spannbaum" nicht haben) *und* alle Knoten besucht wurden. Sonst geben wir `False` zurück. Da die Graphtraversierung nach $|V|+c$ Iterationen bzw. Rekursionen abbricht, dauert dies höchstens $\mathcal{O}(|V|+c) = \mathcal{O}(|V|)$ viele Zeitschritte. Der nachgelagerte Test ob es noch weiße Knoten gibt dauert ebenfalls höchstens $\mathcal{O}(|V|)$ viele Zeitschritte.

Alternative Lösung mit Breitensuche: Wir führen eine leicht modifizierte Breitensuche durch. Wir führen dazu Markierungen wie bei der Tiefensuche ein: Weiß falls der Knoten noch nicht besucht wurde, schwarz falls er bereits aus der queue entfernt und abgearbeitet wurde, sonst grau (mit den entsprechenden Änderungen im Code). Wir führen außerdem einen Zähler für die Anzahl der besuchten Kanten ein (initial 0) welcher für jeden weißen Nachbarn des aktuellen Knotens inkrementiert wird. Mittels einer Überprüfung in jeder Iteration ob der Zähler den Wert $|V|+c$ schon erreicht hat werden äußere und innere Schleife in diesem Fall abgebrochen. Die Breitensuche gibt dabei die Gesamtanzahl besuchter Kanten zurück (aber maximal $|V|+c$).

Aufgabe 2: Sortieralgorithmen *(19 Punkte)*

Gegeben seien k *sortierte* Arrays $A_1, \ldots, A_k$ mit insgesamt n Elementen. Wir wollen diese Arrays zu einem einzelnen *sortierten* Array A der Länge n zusammenfassen.

(a) Eine Lösungsmöglichkeit ist folgender Algorithmus

Algorithm 1 `sequential_merge`$(A_1, \ldots, A_k)$
```
A = A_1
for i = 2 to k do
    A = merge(A, A_i)
return A
```

wobei *merge*() die Merge-Operation wie im Merge-Sort Algorithmus ist.

Angenommen k ist ein Teiler von n und alle Arrays haben die Länge n/k. Geben Sie die Laufzeit von `sequential_merge` als Funktion von n und k an. Begründen Sie Ihre Antwort. *(7 Punkte)*

(b) Ein Student schlägt stattdessen vor, alle Elemente auf beliebige Weise in ein Array der Länge n zu schreiben und dieses mit dem Merge-Sort Algorithmus aus der Vorlesung zu sortieren. Ist dieses Vorgehen schneller oder langsamer als `sequential_merge`? Begründen Sie Ihre Antwort. *(3 Punkte)*

Hinweis: Nehmen Sie wie in Teil (a) an, dass alle Arrays die Länge n/k haben.

(c) Wir möchten das gegebene Problem nun in Zeit $\mathcal{O}(n \log k)$ lösen, für beliebige Werte $k \leq n$, unter Verwendung von binären Heaps. Vervollständigen Sie dazu den folgenden Algorithmus `heap_merge` (schreiben Sie den Pseudo-Code, welcher an die Stelle "???" gehört). Begründen Sie die Laufzeit.

Algorithm 2 `heap_merge`$(A_1, \ldots, A_k)$
```
H = create_binary_heap()                  ▷ creates an empty binary heap
for i = 1 to k do
    key = A_i[0]
    H.insert((i, 0), key)
A = Array of length n                     ▷ allocate an array of length n
for j = 0 to n − 1 do
    ???
return A
```

Hinweis: H verwaltet Daten der Form (i, ℓ), wobei ℓ eine Position im Array A_i angibt. *(9 Punkte)*

Musterlösung

(a) Die Kosten von *merge*(A, A_i) betragen $\mathcal{O}(|A| + |A_i|)$. Initial hat A die Länge n/k und wächst in jedem Schleifendurchlauf um n/k. Die Gesamtkosten betragen also

$$\mathcal{O}\left(\sum_{i=2}^{k} i \cdot \frac{n}{k}\right) = \mathcal{O}\left(\frac{n}{k}\sum_{i=2}^{k} i\right) = \mathcal{O}\left(\frac{n}{k} \cdot k^2\right) = \mathcal{O}(n \cdot k)\,.$$

(b) Das Vorgehen des Studenten dauert $\mathcal{O}(n \log n)$. Für $k = o(\log n)$ ist dieses Vorgehen asymptotisch langsamer als `sequential_merge`, für $k = \omega(\log n)$ schneller und für $k = \Theta(\log n)$ gleich schnell.

(c) **Algorithm 3** `heap_merge`$(A_1, \ldots, A_k)$
```
H = create_binary_heap()                  ▷ creates an empty binary heap
for i = 1 to k do
    key = A_i[0]
    H.insert((i, 0), key)
A = Array of length n                     ▷ allocate an array of length n
for j = 0 to n − 1 do
    (x, y) = H.deleteMin()
    A[j] = A_x[y]
    if y ≤ |A_x| − 2 then
        key = A_x[y + 1]
        H.insert([x, y + 1], key)
Return A
```

In der ersten Schleife wird ein Heap mit k Elementen gebildet. Dies kostet $\mathcal{O}(k \log k)$. Danach werden n delete-min und $\leq n$ insert Operationen auf dem Heap ausgeführt. Der Heap hat dabei stets die Größe $\leq k$. Die Gesamtkosten betagen also $\mathcal{O}(n \log k)$.

Aufgabe 3: O-Notation *(17 Punkte)*

Geben Sie an, ob die folgenden Aussagen wahr oder falsch sind. Beweisen oder widerlegen Sie die Aussagen anhand der Mengendefinition der Landau-Notation (d.h. insbesondere ohne Verwendung von Grenzwerten).

(a) $n^2 - 3n \in \Omega(n^2)$ *(4 Punkte)*

(b) $(\log n)^2 \in \mathcal{O}(\log(n^3))$ *(4 Punkte)*

(c) $n^2 \in \mathcal{O}\Big(\sum_{i=1}^{n} i\Big)$ *(4 Punkte)*

(d) Falls $f(n) \in o(g(n))$ und $h(n)$ monoton steigt, so gilt $h(f(n)) \in \mathcal{O}(h(g(n)))$ *(5 Punkte)*

Musterlösung

(a) Wahr. Wähle $c = \frac{1}{2}$ und $n_0 = 6$. Für $n \geq n_0$ gilt dann $n^2 \geq 6n$ und daher

$$n^2 - 3n = \tfrac{1}{2} \cdot n^2 + \tfrac{1}{2}(n^2 - 6n) \geq \tfrac{1}{2} \cdot n^2.$$

(b) Falsch. Für $c > 0$ gilt

$$\begin{array}{rccc} & (\log n)^2 & \leq & c\log(n^3) \\ \Leftrightarrow & (\log n)^2 & \leq & 3c\log n \\ \Leftrightarrow & \log n & \leq & 3c \\ \Leftrightarrow & n & \leq & 2^{3c} \end{array}$$

Für gegebenes $c, n_0 > 0$ wähle also $n := \max\{2^{3c}+1, n_0\}$. Dann ist $n \geq n_0$, aber $(\log n)^2 > c\log(n^3)$.

(c) Wahr. In $\sum_{i=1}^{n} i$ sind mindestens $n/2$ Summanden $\geq n/2$, d.h. es gilt (für $n \geq 1$)

$$4\sum_{i=1}^{n} i \geq 4 \cdot \frac{n}{2} \cdot \frac{n}{2} = n^2\,.$$

Alternativ mit Gaußscher Summenformel.

(d) Wahr. Sei $c \leq 1$. Aus $f(n) \in o(g(n))$ folgt, dass es ein n_0 gibt, so dass für alle $n \geq n_0$ gilt $f(n) \leq c \cdot g(n) \leq g(n)$ und somit $h(f(n)) \leq h(g(n))$. Daher gilt $h(f(n)) \in \mathcal{O}(h(g(n)))$.

Aufgabe 4: Hashing mit offener Adressierung *(10 Punkte)*

Wir betrachten Hashtabellen mit offener Adressierung und zwei Methoden zur Auflösung von Kollisionen: *Doppel-Hashing* und *Cuckoo-Hashing*. Sei m die Größe der Hashtabelle. Wir definieren

$$\begin{aligned} h_1(x) &:= (5 \cdot x) \mod m, \\ h_2(x) &:= 1 + \big(2x \mod (m-1)\big), \\ h_3(x) &:= (3 \cdot x - 2) \mod m. \end{aligned}$$

(a) Sei $h_d(x,i) := (h_1(x) + i \cdot h_2(x)) \bmod m$. Fügen Sie die Schlüssel 13, 14, 2, 3, 11 der Reihe nach in eine Hashtabelle der Größe $m := 11$ ein. Benutzen Sie dafür h_d und *Doppelhashing* zur Kollisionsauflösung. *(5 Punkte)*

(b) Fügen Sie die Werte 3, 10, 7 der Reihe nach in eine Hashtabelle der Größe $m := 7$ ein. Benutzen Sie *Cuckoo-Hashing* mit den Funktionen h_1 und h_3 zur Kollisionsauflösung. Geben Sie den Zwischenstand der Tabelle nach dem Einfügen jeden Wertes an (d.h. es sind drei Tabellen anzugeben). *(5 Punkte)*

Hinweis: Die Lösungen sind in die Tabellen auf dem Lösungsblatt dieser Aufgabe einzutragen.

Musterlösung

Aufgabenteil (a):

Tabellenzustand nach Einfügen aller Werte:

3			11	14					2	13
0	1	2	3	4	5	6	7	8	9	10

Aufgabenteil (b):

Nach Einfügen von 3:

	3					
0	1	2	3	4	5	6

Nach Einfügen von 10:

3	10					
0	1	2	3	4	5	6

Nach Einfügen von 7:

10	3				7	
0	1	2	3	4	5	6

Aufgabe 5: Graphen *(17 Punkte)*

Gegeben ein *gerichteter, ungewichteter* Graph $G = (V, E)$ definiert man $G^2 = (V, E^2)$ durch

$(u,v) \in E^2$ genau dann, wenn $u \neq v$ und es in G einen (gerichteten) Pfad von u nach v der Länge höchstens 2 gibt.

(a) Beschreiben Sie einen Algorithmus mit Laufzeit $\mathcal{O}(|E| \cdot |V|)$, der aus der Adjazenzlisten-Repräsentation von G die Adjazenzlisten-Repräsentation von G^2 berechnet. D.h., v muss genau dann in der Adjazenzliste von u enthalten sein, wenn es einen Pfad von u nach v der Länge höchstens 2 gibt. Wir nehmen hier erstmal an, dass v in diesem Fall auch **mehrfach** in der Liste vorkommen darf. Begründen Sie die Laufzeit. *(6 Punkte)*

(b) Beschreiben Sie einen Algorithmus mit Laufzeit $\mathcal{O}(|E| \cdot |V|)$, der aus der Adjazenzlisten-Repräsentation von G die Adjazenzlisten-Repräsentation von G^2 **ohne Mehrfachvorkommen** eines Knotens in einer Adjazenzliste berechnet. Begründen Sie die Laufzeit.

Hinweis: Falls Sie eine Hashtabelle verwenden, dürfen Sie annehmen, dass Hinzufügen und Nachschauen von Werten $\mathcal{O}(1)$ Zeit benötigt. *(4 Punkte)*

(c) Beschreiben Sie, wie man aus der Adjazenzmatrix von G in Zeit $\mathcal{O}(|V|^3)$ die Adjazenzmatrix von G^2 berechnet. Erklären Sie die Laufzeit. *(7 Punkte)*

Musterlösung

(a) Man *ergänzt* die Adjazenzliste A von G wie folgt

Algorithm 4 `square-adj-list`$(A[0..n-1])$ ▷ *Assume nodes are numbered* $0 \ldots n-1$
```
initialize L_u as empty list for each u ∈ V
allocate array A′ of length n
for each node u ∈ V do
    for each v in the adjacency list A[u] do
        for each w in the adjacency list A[v] do
            Add w to L
    append list A[u] to L_u
    A′[u] = L_u
return A′
```

In den ersten beiden Schleifen hat man insgesamt $\mathcal{O}(\max\{|V|, |E|\})$ Durchläufe da *alle Iterationen* der zweiten Schleife der Anzahl der Kanten entspricht und man mindestens Anzahl Knoten viele Iterationen in der ersten Schleife hat. In der dritten Schleife hat man höchstens $\min\{|V|, |E|\}$ viele Iterationen, da ein Knoten nur maximal so viele Nachbarn haben kann. Wegen $\max\{x,y\} \cdot \min\{x,y\} = x \cdot y$ ist die Gesamtlaufzeit der Schleifen daher $\mathcal{O}(|E| \cdot |V|)$. (zusammenhängen von Listen geht in $\mathcal{O}(1)$).

(b) Man führt zuerst den Algorithmus aus (a) aus und löscht dann alle Mehrfachvorkommen.

Algorithm 5 `clean-up`$(A'[0..n-1])$ ▷ *Assume nodes are numbered* $0 \ldots n-1$
```
for each node u ∈ V do
    allocate a dictionary/hash table D
    add u to D
    for each v in the adjacency list A′[u] do
        if v ∈ D then
            delete v from A′[u]           ▷ deletion in O(1) by remembering predecessor
        else
            add v to D
return A′
```

Die Laufzeit des Löschalgorithmus beträgt $\mathcal{O}(|V| + |E| + \sum_{u \in V} |L_u|)$. Mit $\sum_{u \in V} |L_u| = \mathcal{O}(|V||E|)$ erhält man als Laufzeit $\mathcal{O}(|V||E|)$.

Man kann auch direkte Adressierung verwenden und statt D ein Array der Länge n nehmen (der Speicherplatz war ja in keinster Weise eingeschränkt). Man muss auch nicht zuerst den Algorithmus aus (a) anwenden und dann Mehrfachvorkommen löschen, sondern kann den Test ob v schon in der $A'[i]$ enthalten ist direkt in den Algorithmus aus (a) einbauen.

(c) *Algebraische Beschreibung:* Sei A die Adjazenzmatrix von G. Wir berechnen $B = A^2 + A$ und setzen alle Einträge ≥ 1 von B auf 1 und setzen die Hautpdiagonale von B auf 0.

Intuitivere Beschreibung: Man berechnet Eintrag (i,j) in B folgendermaßen: Die Einträge auf der Hauptdiagonalen setzt man auf 0. Für $i \neq j$ und $A[i,j] = 1$ (d.h. in G gibt es eine Kante von i nach j) setzt man $B[i,j] = 1$. Für $i \neq j$ und $A[i,j] = 0$ geht man durch alle Nachbarn von i und schaut, ob diese eine Kante nach j haben. In Pseudocode:

Algorithm 6 `square-adj-matrix`$(A[0..n-1][0..n-1])$
```
allocate an n × n matrix B
for i = 0 to n − 1 do
    for j = 0 to n − 1 do
        if i == j then
            B[i][j] = 0
        else if A[i][j] = 1 then
            B[i][j] = 1
        else
            for k = 0 to n − 1 do
                if A[i][k] == 1 and A[k][j] == 1 then
                    B[i][j] = 1
return B
```

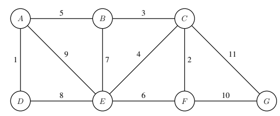
Für jeden der n^2 Einträge muss man $\mathcal{O}(n)$ Operationen ausführen. Die Laufzeit beträgt daher $\mathcal{O}(n^3)$.

Aufgabe 6: Mystische Funktion *(12 Punkte)*

Betrachten Sie den folgenden Algorithmus in abstraktem Pseudocode. Dieser erhält als Eingabe einen *gewichteten, ungerichteten, zusammenhängenden* Graphen $G = (V, E, w)$.

Algorithm 7 `myst-edge-set`(V, E, w)
```
for jedes e ∈ E nach absteigendem Gewicht w(e) do    ▷ beachte Iterationsreihenfolge!
    entferne e aus E
    if (V, E) ist nicht zusammenhängend then
        füge e zu E hinzu
return E
```

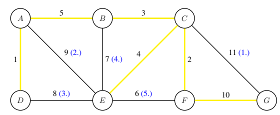
(a) Führen Sie den Algorithmus `myst-edge-set`(V, E, w) auf dem unten stehenden Graphen aus. Nummerieren Sie in dem Graphen die Reihenfolge, in der Kanten von dem Algorithmus gelöscht werden (als Zahl neben der jeweiligen gelöschten Kante). Markieren Sie außerdem alle Kanten, die zurück gegeben werden (umranden oder fett nachzeichnen). *(5 Punkte)*



(b) Welche Ausgabe gibt `myst-edge-set`(V, E, w) zurück? Beweisen Sie Ihre Behauptung. *(7 Punkte)*

Musterlösung

(a)



(b) Die Rückgabe entspricht einem MST. Für den Beweis beobachten wir zunächst folgende Eigenschaften des Algorithmus.

(1.) Der Algorithmus entfernt Kanten nur dann, wenn der Graph dann zusammenhängend bleibt. Damit ist der durch die zurückgegebenen Kanten induzierte Graph am Ende ebenfalls zusammenhängend.

(2.) Die zurückgegebene Kantenmenge hat keinen Zyklus. Denn andernfalls gäbe es einen Schnitt "durch" diesen Zyklus, welcher dann zwei Kanten in der zurückgegebenen Menge enthielte. Der Algorithmus hätte aber eine dieser beiden Kanten entfernt, da die Kantenmenge nach Entfernung noch zusammenhängend ist.

(3.) Sei e eine Kante in der Rückgabemenge. Es gibt einen Schnitt $(S, V \setminus S)$ auf dem e die einzige Kante in der Rückgabemenge ist, da diese zyklenfrei ist (trenne einen "Ast" im Baum ab). Da die Kanten im Schnitt $(S, V \setminus S)$ von G in absteigender Reihenfolge nach Gewicht entfernt wurden, ist e die (eine) leichteste Kante von $(S, V \setminus S)$.

Nach (3.) haben wir nur sichere Kanten in der Rückgabemenge. Die Rückgabemenge ist nach (2.) zyklenfrei und nach (1.) zusammenhängend, ist also ein *Spannbaum*. Eine sichere, zyklenfreie Kantemenge ist immer Teilmenge eines MST (nach Vorlesung). Damit ist der zurückgegebene Spannbaum insbesondere ein minimaler Spannbaum.

Aufgabe 7: Wörter trennen *(15 Punkte)*

Gegeben sei ein Dictionary D, welches Wörter der Länge höchstens $k \in \mathcal{O}(1)$ enthält. Sei T eine Zeichenkette bestehend aus n Zeichen. Wir möchten berechnen, ob sich T in zusammenhängende Zeichenketten zerlegen lässt, sodass jede Zeichenkette einem Wort in D entspricht.

Beispiel: Sei $D = \{$'Flugzeug', 'Algorithmen', 'Zug', 'super', 'sind'$\}$. Dann ist die Lösung des obigen Problems für die Eingaben $T_1 =$ 'Algorithmensinddoof' oder $T_2 =$ 'FlugZugzeug' jeweils False. Für $T_3 =$ 'sindAlgorithmensuper' ist die Antwort zu obigem Problem True.

Hinweise: Gehen Sie davon aus, dass die Konstante k als Teil der Eingabe gegeben ist und dass Sie (bspw. mittels Hashing) in $\mathcal{O}(1)$ Zeitschritten testen können, ob eine Zeichenkette der Länge höchstens k in D enthalten ist. Sie dürfen außerdem annehmen, dass die Zeichen von T in einem Array $T[0..n-1]$ vorliegen.

(a) Sei $t : \{0, \dots, n-1\} \to \{\texttt{True}, \texttt{False}\}$ eine Funktion, sodass $t(i) = \texttt{True}$ genau dann, wenn sich die Zeichenkette gegeben durch $T[0..i]$ in zusammenhängende Zeichenketten zerlegen lässt, sodass jede dieser Zeichenketten in D enthalten ist.

Bestimmen Sie zunächst einen rekursiven Zusammenhang für $t(i)$. Das heißt, geben Sie eine Vorschrift wie $t(i)$ mittels $t(j)$ mit $j < i$ berechnet werden kann. Begründen Sie kurz. *(8 Punkte)*

Hinweis: Sei T eine Zeichenkette, die sich zerlegen lässt in Zeichenketten $T = T_1 \dots T_\ell$ sodass $T_i \in D$. Betrachten Sie $T_1 \dots T_{\ell-1}$ und T_ℓ gesondert.

(b) Geben Sie einen Algorithmus an, der das obige Problem in $\mathcal{O}(n)$ vielen Zeitschritten mittels *dynamischer Programmierung* löst. Begründen Sie die Laufzeit. *(7 Punkte)*

Musterlösung

(a) Wenn sich eine Zeichenkette T zerlegen lässt in $T = T_1 \dots T_\ell$ mit $T_i \in D$, dann ist insbesondere $T_\ell \in D$ und hat weniger als k Zeichen. Wir müssen also nur testen ob T ein Suffix mit höchstens k Zeichen hat welches in D ist **und** ob sich das übrig bleibende Präfix ebenfalls in Wörter aus D zerlegen lässt. Die Frage ob $t(i) = \texttt{True}$ ist, lässt sich also zurückführen auf die Frage ob es ein Suffix $T[j..i] \in D$ von $T[0..i]$ gibt, welches höchstens aus höchstens k Zeichen besteht und außerdem $t(j-1) = \texttt{True}$ ist. Wir formalisieren diese Beobachtung wie folgt

$$t(i) = \bigvee_{j=\max(i-k+1,0)}^{i} \Big(t(j-1) = \texttt{True} \textbf{ and } T[j..i] \in D \Big)$$

Um den Fall abzudecken in dem $T[0..i]$ schon in D ist, setzen wir $t(j) := \texttt{True}$ für $j < 0$.

(b) Ausgehend von der obigen Rekursion, wenden wir die Strategie des dynamischen Programmierens an.

Algorithm 8 `is-decomposable(i)` ▷ *D, k, T, memo seien global gegeben*

if $j < 0$ **then return** True
$d =$ False
for $j = \max(i - k + 1, 0)$ **to** i **do** ▷ *Schleife inklusive i*
  if memo$[j-1] =$ Null **then** ▷ *Ergebnis für $j-1$ liegt noch nicht vor*
    memo$[j-1] =$ is-decomposable$(j-1)$
  $d = d$ **or** (memo$[j-1]$ **and** $T[j..i] \in D$)
return d

Jeder Aufruf von `is-decomposable(i)` dauert $\mathcal{O}(1)$ Zeitschritte (unter Vernachlässigung rekursiver Aufrufe), da wir eine Schleife nur $\mathcal{O}(k) = \mathcal{O}(1)$ Durchläufe hat und der Test $T[j..i] \in D$ in $\mathcal{O}(1)$ Zeit durchgeführt werden kann. Es gibt auch nur $\mathcal{O}(n)$ Aufrufe von `is-decomposable(i)`, denn spätestens dann liegt das Ergebnis von memo$[i]$ für jedes $i \in \{0, \dots, n-1\}$ vor.

Aufgabe 8: Textsuche *(12 Punkte)*

Gegeben sei das Muster $P = BACBAB$ und der Text $T = CBACABBACBABACBABA$.

(a) Geben Sie das Array S des Knuth-Morris-Pratt Algorithmus an. *(4 Punkte)*

(b) Finden Sie mittels des Knuth-Morris-Pratt Algorithmus alle Vorkommen von P in T. Die Schritte des Algorithmus sollen klar erkennbar sein. Sie können dazu die Tabelle auf dem Lösungsblatt verwenden. *(8 Punkte)*

Musterlösung

(a) $[-1, 0, 0, 0, 1, 2, 1]$

(b)

C	B	A	C	A	B	B	A	C	B	A	B	A	C	B	A	B	A
B	A	C	B	A	B												
	B	A	C	B	A	B											
				B	A	C	B	A	B								
					B	A	C	B	A	B							
						B	A	C	B	A	B						✓
											B	A	C	B	A	B	✓
																B	A

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

Klausur Informatik 2

Mittwoch, 8. März 2019, 13:00-16:00

Name: ....

Matrikel Nr.: ....

Unterschrift: ....

Erst öffnen wenn die Klausuraufsicht die Erlaubnis erteilt!

- Legen Sie Ihren **Studentenausweis** sichtbar vor sich.
- Schreiben Sie Ihren **Namen** und Ihre **Matrikelnummer** an die vorgesehenen Stellen.
- **Unterschreiben Sie** diese Seite um zu bestätigen, dass Sie die Fragen ohne unerlaubte Hilfsmittel beantwortet haben und die Klausuraufsicht über (gesundheitliche) Probleme informiert haben.
- Dies ist eine **Open Book** Klausur weshalb alle gedruckten und handgeschriebenen Materialien erlaubt sind. **Elektronische Hilfsmittel** sind **nicht** erlaubt.
- Schreiben Sie **lesbar** und nur mit **dokumentenechten** Stiften. Benutzen Sie **keine rote** Farbe und **keinen Bleistift**!
- 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 als Blackbox** verwendet werden.
- **Lesen Sie jede Aufgabe sorgfältig** durch und stellen Sie sicher dass Sie diese verstanden haben!
- Falls Sie eine Frage zur Aufgabenstellung haben, geben Sie der Klausuraufsicht ein **Handzeichen**.
- Schreiben Sie Ihren Namen auf **alle Blätter**!

Aufgabe	1	2	3	4	5	6	7	**Total**
Maximum	23	9	15	7	18	21	27	120
Punkte								

Aufgabe 1: Kurze Fragen *(23 Punkte)*

(a) Sortieren Sie die folgenden Funktionen **aufsteigend** nach ihrem asymptotischen Wachstum.

$$\begin{array}{llll} n^n & \log(n^n) & 2^{\log n} & (\log n)^n \\ n \log n & 3^n & \log\left((\sqrt{n})^n\right) & \big((2n)!\big)^2 \end{array}$$

Hinweise: $\log(\cdot)$ bezeichnet den Logarithmus zur Basis 2. Zwischen jeweils aufeinanderfolgenden Funktionen $f(n), g(n)$ in Ihrer Sortierung notieren Sie "$f(n) < g(n)$" falls $f(n) \in \mathcal{O}\big(g(n)\big)$ und $g(n) \notin \mathcal{O}\big(f(n)\big)$ gilt. Notieren Sie "$f(n) = g(n)$" wenn $f(n) \in \mathcal{O}\big(g(n)\big)$ und $g(n) \in \mathcal{O}\big(f(n)\big)$ gilt. *(4 Punkte)*

(b) Gegeben sei ein gerichteter Graph $G = (V, E)$, abgespeichert via **Adjazenzlisten**, und ein Knoten $v \in V$. Geben Sie einen Algorithmus an, mit welchem Sie in möglichst kurzer Zeit die Menge $U = \{u \in V : \exists \text{ Pfad von } u \text{ nach } v\}$ finden, d.h., alle Knoten u von welchen aus ein Pfad nach v existiert. *(4 Punkte)*

(c) Gegeben seien k **bereits sortierte** Arrays $A_1, \dots, A_k$ mit je m Sortierschlüsseln. Beschreiben Sie einen Algorithmus der ein **sortiertes** Array A der Größe km aus allen Werten der Arrays $A_1, \dots, A_k$ berechnet und dabei $\mathcal{O}(km \log k)$ Vergleiche benötigt. *(4 Punkte)*

(d) Sei A ein **bereits sortiertes** Array mit n paarweise verschiedenen Sortierschlüsseln. Es ist möglich **zwei** Sortierschlüssel in A zu vertauschen, um ein Array A' mit folgender Eigenschaft zu erhalten: Um das Array A' korrekt zu sortieren werden $\Omega(n)$ Durchläufe der äußeren Schleife des Sortieralgorithmus **Bubblesort** (wie in den Vorlesungsunterlagen) benötigt. Geben Sie die Vertauschung an und begründen Sie Ihre Antwort. *(5 Punkte)*

(e) Gegeben seien die Hashfunktionen $h_1(x) = x \mod m$ und $h_2(x) = 1 + (x \mod (m-1))$. Fügen Sie die Schlüssel 28, 59, 47, 13, 39 nacheinander in die unten stehende Hashtabelle der Größe $m = 11$ ein, indem Sie die Methode des **Doppel-Hashing** nutzen. *(4 Punkte)*

0	1	2	3	4	5	6	7	8	9	10

(f) Nennen Sie einen Vorteil des **Doppel-Hashing** gegenüber Hashing mit **quadratischer Sondierung**. *(2 Punkte)*

Musterlösung

(a) *Korrekturhinweis: Einen Punkt Abzug für jedes aufeinanderfolgende Paar mit falscher Reihenfolge bzw. falschem Zeichen < bzw. =.*

$$\begin{array}{ccccccccc} & 2^{\log n} & < & \log(n^n) & = & \log\left((\sqrt{n})^n\right) & = & n \log n \\ < & 3^n & < & (\log n)^n & < & n^n & < & \big((2n)!\big)^2 \end{array}$$

(b) Alle Knoten zu finden, von denen aus ein Pfad nach v existiert, ist fast das Gleiche wie alle Knoten zu finden, die von v aus erreicht werden können – nur umgekehrt. Betrachte den Graphen $G' = (V, E')$, in welchem jede Kante aus E umgedreht existiert, d.h., $E' = \{(y, x) : (x, y) \in E\}$. Wenn von u ein Pfad nach v in G existiert, dann existiert ein Pfad von v nach u in G'. D.h., wir führen BFS auf G' aus, startend mit v. DFS geht genauso. Alle Knoten, die von v aus in G' erreicht werden können, fügen wir zu U hinzu. *(2 Punkte)*

Korrekturhinweis: Die zwei Punkte gibt es für den Hinweis dass man DFS auf dem "umgedrehten" Graphen G' ausführt. Einfach nur DFS auf G auszuführen gibt 1 Punkt da immerhin auf ungerichteten Graphen korrekt (laut Aufgabenstellung ist der Graph aber gerichtet).

Das Einzige, was wir noch sicherstellen müssen, ist, dass wir G' schnell aus G kreieren können. Dazu gehen wir ähnlich vor wie in der vorhergehenden Aufgabe, indem wir erst eine Kopie des Knoten-Arrays erstellen und dann durch die Adjazenzlisten durchgehen. Gibt es in der Adjazenzliste von x einen Zeiger zu y, d.h., eine Kante (x, y) im Graphen, dann machen wir im zweiten Array einen Eintrag in der Adjazenzliste von y, mit einem Zeiger auf x. Jede Kante wird so einmal kopiert und dabei umgedreht. *(2 Punkte)*

Die Laufzeit von beiden Algorithmen ist in der Größenordnung des Speichplatzbedarfs von Adjazenzlisten, d.h., $O(n + m)$, und das ist optimal, da wir jede Kante und jeden Knoten mindestens einmal anschauen müssen.

Korrekturhinweis: Eine Laufzeitanalyse muss laut Aufgabenstellung nicht erbracht werden. Algorithmen mit schlechteren Laufzeiten die aber immerhin polynomiell in der Eingabegröße sind geben bis zu einem Punkt Abzug.

(c) Beobachtung: Wir haben eine Situation ähnlich zum Combine-Schritt bei Mergesort, bei der k bereits sortierte Teilarrays zusammengeführt werden müssen. Sei $k_1 := k$ und seien die Arrays $A_1^1 := A_1, \dots, A_{k_1}^1 := A_k$ die inital gegebenen, bereits sortierten Arrays.

Wir benutzen den in der Vorlesung beschriebenen Merge-Schritt iterativ als Black-box um sortierte Arrays $A_1^i, \dots, A_{k_i}^i$ jeweils paarweise in sortierte Arrays $A_1^{i+1}, \dots, A_{k_{i+1}}^{i+1}$ zusammenzuführen, solange bis $k_i = 1$ ist. Falls $k_i > 1$ und ungerade ist kombinieren wir nur die ersten $(k_i - 1)/2$ Paare.

Seien $A_1^{i+1}, \dots, A_{k_{i+1}}^{i+1}$ die Arrays nach dem $(i+1)$-ten Merge Schritt. Es gilt dass $k_{i+1} = \lceil k_i/2 \rceil$. Nach spätestens zwei Merge-Schritten gilt

$$k_{i+2} \le \lceil k_{i+1}/2 \rceil \le \lceil \lceil k_i/2 \rceil / 2 \rceil = \lceil k_i/4 \rceil \overset{k_i \ge 2}{\le} k_i/2,$$

d.h. nach spätestens zwei Merge-Schritten halbiert sich die restliche Anzahl der Arrays. Also ist das Verfahren nach spätestens $2 \log(k)$ Merge-Schritten abgeschlossen und es bleibt lediglich ein einziges sortiertes Array A übrig.

Korrekturhinweis: Man kann auch zunächst annehmen dass k eine Zweierpotenz ist und dann argumentieren, dass das Verfahren nur einen Mergeschritt mehr benötigt, wenn man k auf die nächste Zweierpotenz aufrundet und entsprechend viele Dummyelemente mit Wert

∞ einfügt. Es reicht sogar aus zu erwähnen dass höchstens ein konstanter Faktor mehr Schritte benötigt wird wenn k keine Zweierpotenz ist. Wer überhaupt nicht darauf eingeht, dass k_i ungerade sein könnte bekommt einen Punkt abgezogen.

Das Mergen zweier Arrays B, C dauert $\mathcal{O}(|B| + |C|)$. Damit benötigt jeder Merge Schritt insgesamt höchstens $\mathcal{O}\big(|A_1^i| + \ldots + |A_{k_i}^i|\big) = \mathcal{O}(km)$ Zeitschritte. Multipliziert mit der Anzahl der Merge-Schritte ergibt das $\mathcal{O}(km \log k)$

Korrekturhinweis: Eine Analyse muss laut Aufgabenstellung nicht gemacht werden. Wenn der Algorithmus länger dauert gibt es aber entsprechend Abzug. Maximal einen Punkt gibt es für Algorithmen mit Laufzit $\mathcal{O}(km \log(km))$, weil das trivialierweise dem Hintereinanderschreiben der Listen $A_1, \dots, A_k$ in eine Liste A und Anwendung von Mergesort auf A entspricht. Einen Punkt kann man für die Beschreibung eines "k-fach Merges" geben, bei der inital Pointer auf die ersten Arrayelemente gesetzt werden, das Maximum aus diesen bestimmt wird, und der entsprechende Pointer eine Stelle weiter gerückt wird. Das hat zwar Laufzeit $O(k^2 m)$ ist aber immerhin für $k \in o(\log m)$ besser als die triviale Lösung.

(d) Wir erhalten A' indem wir das größte Element am Ende von A und das kleinste Element am Anfang von A miteinander vertauschen. *(2 Punkte)*

Nach jedem Durchlauf der inneren Schleife von Bubblesort rückt das kleinste Element genau eine Stelle nach vorne. Damit benötigt die äußere Schleife mindestens $n-1 \in \Omega(n)$ Durchläufe bis das Array A' wieder sortiert ist. *(3 Punkte)*

(e)

		13	47	59	39	28				
0	1	2	3	4	5	6	7	8	9	10

(f) Die Wahrscheinlichkeit, dass zwei Elemente x und y die gleiche Abfolge an Positionen haben, ist beim Doppelhashing geringer, da hierfür $h_1(x) = h_1(y)$ und $h_2(x) = h_2(y)$ gelten muss.

Aufgabe 2: Textsuche *(9 Punkte)*

Gegeben sei der Suchtext $T = 15653212631$ der Länge $n = 11$ und das Muster $P = 21$ der Länge $m = 2$. Wir führen den Rabin-Karb Algorithmus mit Basis $b = 7$ und der Hashfunktion $h(x) := x \mod 8$ durch.

Hinweis: Die Wahl der Basis $b = 7$ bedeutet, dass wir das Muster P (sowie Auschnitte des Textes T) als Zahlen zur Basis 7 interpretieren. Beispielsweise entspricht 32 der Zahl $3 \cdot 7^1 + 2 \cdot 7^0 = 23$.

(a) Geben Sie für jeden Schritt $s \in \{0, \dots, 9\}$ den Hashwert $h\big(T[s, s+1]\big)$ an. *(5 Punkte)*

Hinweis: $T[s, s+1]$ ist das Teilwort bestehend aus dem s-ten und $(s+1)$-ten Zeichen von T.

(b) Für welche $s \in \{0, \dots, 9\}$ ist $h\big(T[s, s+1]\big) = h(P)$? *(4 Punkte)*

Musterlösung

(a) Wir erhalten die folgenden Ergebnisse:

s	0	1	2	3	4	5	6	7	8	9
$T[s, s+1]_{10}$	12	41	47	38	23	15	9	20	45	22
$h\big(T[s, s+1]\big)$	4	1	7	6	7	7	1	4	5	6

Korrekturhinweis: 0,5 Punkte Abzug für jeden Fehler in der letzten Zeile.

(b) $s \in \{2, 4, 5\}$.

Korrekturhinweis: 1,5 Punkte Abzug für jedes vergessene oder falsch eingefügte Element.

Aufgabe 3: Algorithmenanalyse *(15 Punkte)*

Der folgende Algorithmus erhält als Eingabe k Arrays $A_1, \dots, A_k$ der Größe m und ein Array B der Größe k. Die Arrays $A_i[0..m-1]$ enthalten ganze Zahlen in **sortierter** Reihenfolge. Das Array $B[0..k-1]$ enthält ganze Zahlen in **beliebiger Reihenfolge**.

Algorithm 1 `algorithm`$(A_1, \dots, A_k, B)$

$x \leftarrow 0$
for $i \leftarrow 1$ **to** k **do**
  $j \leftarrow 0$
  while $j < m$ **and** $A_i[j] < B[i-1]$ **do**
    $j \leftarrow j+1$
  if $A_i[j] \neq B[i-1]$ **then** $x \leftarrow x+1$
if $x \bmod 2 = 0$ **then return** `True`
else return `False`

(a) Geben Sie die Ausgabe von `algorithm`(A_1, A_2, A_3, B) für die folgende Eingabe an:

$$A_1 = [1,4,5,6], A_2 = [1,3,4,7], A_3 = [2,4,6,9], B = [4,2,7].$$

(3 Punkte)

(b) Fassen Sie zusammen welche Aussage `algorithm` im Bezug auf die Eingabe $A_1, \dots, A_k, B$ erzeugt. *(4 Punkte)*

(c) Geben Sie die **asymptotische** Laufzeit von `algorithm` im **worst case** in Abhängigkeit von k und m an. *(3 Punkte)*

(d) Beschreiben Sie einen Algorithmus, welcher die **gleiche Ausgabe** erzeugt wie `algorithm` und dessen Laufzeit $\mathcal{O}(k \log m)$ beträgt (oder beschreiben Sie eine entsprechende Modifikation von `algorithm`). *(5 Punkte)*

Musterlösung

(a) `True` *(3 Punkte)*

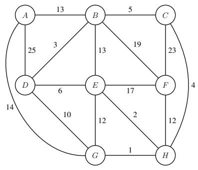
(b) `algorithm` zählt zunächst in wie vielen der Arrays A_i ein Element vorkommt das jeweils *nicht* mit $B[i-1]$ übereinstimmt. Dann liefert `algorithm` den `True` zurück falls diese Anzahl gerade ist, sonst `False`. *(4 Punkte)*

(c) $\Theta(km)$ *Korrekturhinweis: $\mathcal{O}(km)$ ist ebenfalls in Ordnung.* *(3 Punkte)*

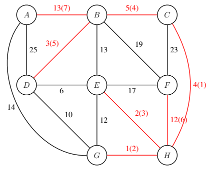
(d) Wir nutzen aus dass die Arrays $A_1, \dots, A_k$ sortiert sind und modifizieren `algorithm` dahingehend dass wir statt der linearen Suche in der inneren Schleife eine binäre Suche durchführen (*Korrekturhinweis: Das ist schon ausreichend für die volle Punktzahl*). Dass heißt, wir springen jeweils zur Mitte des übrig gebliebenen Suchbereichs (inital das gesamte Array A_i) vergleichen das dortige Element mit $B[i-1]$ und machen mit der linken bzw. rechten Hälfte weiter wenn das gefundene Element größer bzw. kleiner war. *(5 Punkte)*

Aufgabe 4: Minimaler Spannbaum *(7 Punkte)*

Führen Sie auf dem abgebildeten Graphen **Prim**'s Algorithmus zur Berechnung eines minimalen Spannbaums aus. Starten Sie mit Knoten C. Markieren Sie (falls farbig: nicht mit rot!) die Kanten, die am Ende im Baum sind und schreiben Sie neben die Kanten, in welcher Reihenfolge diese eingefügt werden.



Musterlösung



Aufgabe 5: Tankproblem *(18 Punkte)*

Ein Reisender möchte eine Strecke vom Startpunkt $a = 0$ zur Zielmarke $b \in \mathbb{N}$ mit seinem Auto zurücklegen. Mit vollem Tank kann das Auto eine **Strecke von $x \in \mathbb{N}$** zurücklegen. Zu Beginn der Reise ist der Tank voll. Auf der Strecke von a nach b liegen n **Tankstellen** (mit Nummern $1, \dots, n$) mit **aufsteigenden** Distanzen $t_1, \dots, t_n \in \mathbb{N}$ zu a die der Reisende kennt.

Der Reisende muss an einer Teilmenge $T \subseteq \{1, \dots, n\}$ von Tankstellen anhalten um zu tanken. Dabei darf die Distanz zwischen zwei nacheinander besuchten Punkten aus $T \cup \{a, b\}$ jeweils höchstens x sein, damit die Tankfüllung bis zum jeweils nächsten Ziel ausreicht.

(a) Geben Sie eine **Greedy-Strategie** um eine Auswahl T von Tankstellen zu treffen, welche $|T|$ minimiert. *(2 Punkte)*

(b) Beweisen Sie, dass Ihre Strategie $|T|$ minimiert. *(7 Punkte)*

Es herrscht eine Ölkrise und an den Tankstellen haben sich Warteschlangen gebildet. Der Reisende hat sich informiert und kennt die **Wartezeit w_i** an jeder Tankstelle $i \in \{1, \dots n\}$.

(c) Geben Sie einen effizienten Algorithmus nach dem **Prinzip des Dynamischen Programmierens** an, der eine Menge von Tankstellen $T \subseteq \{1, \dots, n\}$ ermittelt, welche die aufsummierte Wartezeit $W = \sum_{j \in T} w_j$ minimiert. Bestimmen Sie die Laufzeit Ihres Algorithmus. *(9 Punkte)*

Musterlösung

(a) **Als Text:** Sei $t_0 := 0$, $j := 0$ und $T = \emptyset$. (Schleife:) Solange $t_j + x < b$ (wir erreichen wir die Zielmarke b mit der aktuellen Tankfüllung noch nicht) fügen wir die weiteste Tankstelle t_k $(k > j)$ die vom aktuellen Standort t_j noch innerhalb unserer Reichweite x ist zu T hinzu setzen $j := k$ und führen die nächste Iteration der Schleife durch. Am Ende geben wir T zurück.

Korrekturhinweis: Es darf davon ausgegangen werden, dass die Abstände zwischen Tankstellen kleiner als x sind. Deshalb kann, aber muss nicht erwähnt werden: Falls weder eine Tankstelle noch das Ziel von der aktuellen Position aus erreichbar sind, kann man `False` o.ä. zurückgeben.

Algorithm 2 `Greedy-Refueling`$(x, t_1, \dots, t_n, b)$

$j \leftarrow 0$; $t_0 \leftarrow 0$; $T \leftarrow \emptyset$
while $t_j + x < b$ **do**
  if there exists $k > j$ with $t_j + x \geq t_k$ **then**
    $j \leftarrow \underset{k>j,\, t_j+x \geq t_k}{\arg\max}\ t_k$
    $T \leftarrow T \cup \{j\}$
  else return `False` ▷ *Korrekturhinweis: Fallabfrage nicht notwendig*
return T

(b) Sei $T^* = \{i_1, \dots, i_\ell\}$ eine Tankstrategie, sodass $|T^*|$ minimal ist und sei $T = \{j_1, \dots, j_m\}$ die Lösung unserer Greedy Strategie (jeweils geordnet nach Reihenfolge ihres Auftretens auf der Route). Sei die k-te Tankstop der erste, sodass $T^* \ni i_k \neq j_k \in T$, d.h. der erste Tankstop entlang der Route, der sich bei den Tankstrategien T^* und T unterscheidet.

Nach unserer Greedy Wahl von $j_k \in T$ ist $t_{j_k} > t_{i_k}$ und $t_{j_{k-1}} + x \geq t_{j_k}$ und $t_{j_{k-1}} + x < b$. Da die vorherige Tankstelle $j_{k-1} = i_{k-1}$ in T bzw. T^* übereinstimmen und da $t_{i_k} > t_{j_k}$, können wir in T^* die Tankstelle j_k mit der Tankstelle i_k ersetzen. Die neue Lösung $T^{(1)} := (T^* \setminus \{i_k\}) \cup \{j_k\}$ ist ebenfalls zulässig und minimal da $|T^*| = |T^{(1)}|$.

Falls $T = T^{(1)}$ ist $|T| = |T^*|$ und somit ebenfalls minimal. Sonst wiederholen wir den obigen Austausch mit $T^* = T^{(1)}$ um Lösungen $T^{(2)}, T^{(3)}, \dots$ zu erhalten, solange bis das Ergebnis mit T übereinstimmt. Da wir stets gleich große Lösungen erhalten ist schließlich $|T^*| = |T^{(1)}| = |T^{(2)}| = \dots = |T|$.

(c) Wir definieren W_i als minimale Wartezeit die ausgehend vom Standort t_i notwendig ist um (unter der Annahme eines *bereits vollen Tanks*) das Ziel b zu erreichen. Für $i = 0$ sei $t_0 := 0$ der Startpunkt. Sei $T_i \subseteq \{1, \dots, n\}$ die Menge der Tankstellen die der Reisende ab t_i ansteuern muss um die minimale Wartezeit W_i zu verwirklichen.

Wir definieren W_i rekursiv wie folgt: Im Basisfall ist $t_i + x \geq b$ und somit $W_i := 0$ und $T_i = \emptyset$ (wir können direkt zum Ziel fahren ohne weiter tanken zu müssen). Sonst setzen wir $j := \arg\min_{i<j\leq n,\, t_j+x\geq t_i} (W_j + w_j)$ und $W_i := W_j + w_j$ und $T_i := T_j \cup \{j\}$. Das gewünschte Ergebnis ist $W_0 = W$ und $T_0 = T$. Diese Rekursion können wir für das übliche Algorithmendesign nach dem Prinzip des dynamischen Programmierens benutzen. Nach Ausführung von `Dynamic-Refueling(0)` liegt das Ergebnis in `T`$[0]$ vor. *(7 Punkte)*

Algorithm 3 `Dynamic-Refueling`(i) ▷ globale dictionaries `W` und `T`

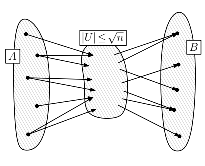
if $t_i + x \geq b$ **then**
  `W`$[i] \leftarrow 0$; `T`$[i] \leftarrow \emptyset$
  return 0
else if `W`$[i]$ not empty **then** ▷ *Zwischenergebnis bereits berechnet*
  return `W`$[i]$
else $j \leftarrow \underset{i<j\leq n,\, t_j+x\geq t_i}{\arg\min} \big($`Dynamic-Refueling`$(j) + w_j\big)$
  `W`$[i] \leftarrow$ `W`$[j] + w_j$; `T`$[i] \leftarrow$ `T`$[j] \cup \{j\}$
  return `W`$[i]$

Korrekturhinweis: Zwei Punkte Abzug für die Berechnung von W aber nicht T.

Die Laufzeit eines Rekursionsschrittes `Dynamic-Refueling`(i) ist $\mathcal{O}(n)$ für die Bestimmung des $\arg\min$ unter Missachtung der rekursiven Unteraufrufe. Wir berechnen jeden Rekursionsschritt `Dynamic-Refueling`(i), $i = 1, \dots, n$ höchstens ein mal bevor das Ergebnis jeweils im Dictionary vorliegt. Die Gesamtlaufzeit ist also $\mathcal{O}(n^2)$. *(2 Punkte)*

Aufgabe 6: Kürzeste Pfade *(21 Punkte)*

Gegeben sei eine Klasse $\mathcal{G}$ von **gerichteten, gewichteten** Graphen. Graphen $G = (V, E, w)$ der Klasse $\mathcal{G}$ haben **positive Kantengewichte** und deren Knoten V $(|V| = n)$ bestehen aus **drei disjunkten Teilmengen A, B und U**, so dass folgende Eigenschaften erfüllt sind. **Knoten in A haben nur ausgehende Kanten** zu Knoten in U und **Knoten in B haben nur eingehende Kanten** von Knoten in U. Die Knoten in U können durch beliebige Kanten verbunden sein, allerdings **enthält die Menge U höchstens $\sqrt{n}$ Knoten**. Eine Verbildlichung:



Optimieren Sie die folgenden, aus der Vorlesung bekannten Algorithmen zur Berechnung der Distanz zwischen **allen Paaren von Knoten**, **für Graphen der Klasse $\mathcal{G}$**. Argumentieren Sie jeweils, warum der optimierte Algorithmus **korrekt** ist und drücken Sie die asymptotische **Laufzeit** als Funktion von n aus.

(a) Bellman-Ford Algorithmus. *(7 Punkte)*

(b) Das Verfahren mittels Matrix-Multiplikationen in der Min-Plus-Algebra. *(7 Punkte)*

(c) Floyd-Warshall-Algorithmus. *(7 Punkte)*

Hinweise: Sie können davon ausgehen, dass die Aufteilung in die Mengen A, B und U bekannt ist. Es genügt die Modifikation der genannten Algorithmen ausreichend genau zu beschreiben, Sie müssen keinen Pseudocode angeben.

Musterlösung

Für die Optimierung der drei Wegsuchalgorithmen benutzen wir die folgende *Beobachtung: Da alle kürzesten Pfade in Graphen der Klasse $\mathcal{G}$ nur innere Knoten aus der Menge U haben können, haben kürzeste Pfade höchstens $|U|+1 = \sqrt{n}+1$ Kanten.*

(a) Wir ändern lediglich die äußere Schleife des Bellman-Ford Algorithmus (mit einem Startknoten s), sodass diese nur $\lfloor\sqrt{n}\rfloor+1$ Iterationen durchläuft (statt $n-1$). Da der Bellman-Ford Algorithmus nach i Durchläufen der äußeren Schleife alle kürzesten Pfade findet die höchstens $i+1$ Kanten haben, reichen aufgrund der obigen Beobachtung $\lfloor\sqrt{n}\rfloor+1$ Durchläufe aus um die Distanzen zu s korrekt zu berechnen. Die Laufzeit von Bellman-Ford verringert sich auf $\mathcal{O}(\sqrt{n}|E|) = \mathcal{O}(n^{5/2})$. Da wir Bellman-Ford insgesamt n mal ausführen müssen um die Distanzen zwischen allen Knotenpaaren zu ermitteln, erhöht sich die Laufzeit auf $\mathcal{O}(n^{3/2}|E|) = \mathcal{O}(n^{7/2})$. Der Speedup ist also $\sqrt{n}$.

(b) Wir benutzen wie zuvor die obige Beobachtung um das Verfahren zu optimieren. Der Iterationsparameter t der äußeren Schleife muss nur noch bis $\lceil\log(\sqrt{n})\rceil$ laufen (statt $\lceil\log n\rceil$) damit wir die korrekte Ausgabe erhalten. Der modifizierte Algorithmus funktioniert für Graphen der Klasse $\mathcal{G}$, da nach allen Durchläufen der Schleife die Distanzen von kürzesten Pfaden mit höchstens $2^{\lceil\log(\sqrt{n})\rceil}+1 \geq \sqrt{n}+1$ Kanten korrekt berechnet sind, was nach obiger Beobachtung für alle kürzesten Pfade der Fall ist. Die asymptotische Laufzeit ist $\mathcal{O}(n^3\log(\sqrt{n})) = \mathcal{O}(n^3\log n)$, ändert sich also nicht. Wir erhalten dennoch einen konstanten Speedup von $\frac{n^3\cdot\log n}{n^3\cdot\log(\sqrt{n})} = 2$.

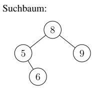
(c) Ähnlich wie bei Bellman-Ford können wir die Anzahl der Durchläufe bei Floyd-Warshall auf $\lfloor\sqrt{n}\rfloor$ absenken. Wir benutzen dazu, dass die Menge U bekannt ist (Hinweis). Seien ohne Einschränkung $U = \{1, \dots, \ell\}$ mit $\ell \leq \sqrt{n}$ die Nummern der Knoten von U. Nach dem k-ten Durchlauf der Hauptschleife des Algorithmus, sind die Distanzen zwischen allen kürzesten Pfaden die lediglich die Knoten mit Nummern $1, \dots, k$ als innere Knoten verwenden, korrekt berechnet. Da aufgrund obiger Beobachtung lediglich die Knoten $1, \dots, \ell$ mit $\ell \leq \sqrt{n}$ innere Knoten von kürzesten Pfaden sein können, reicht es die Hauptschleife bis $\lfloor\sqrt{n}\rfloor$ laufen zu lassen um die korrekten Distanzen zwischen allen Knotenpaaren zu erhalten. Die asymptotische Laufzeit ist dann $\mathcal{O}(n^{5/2})$ und der Speedup ist $\sqrt{n}$.

Korrekturhinweis: Alle korrekten Modifikationen die einen Speedup geben, bzw. für gewisse Eingaben einen Speedup geben könnten, werden anteilig bewertet, selbst wenn dies unter dem Speedup der Musterlösung bleibt. Es können folgende Unterscheidung gemacht werden (Liste nicht erschöpfend):

- *Ist der Speedup "additiv variabel" (gemeint ist $T_{original} - T_{mod} = x$ für $x \in \Omega(n)$ für **manche** Eingaben G) gibt das je nach Speedup höchstens zwei Punkte mit korrektem Algorithmus und korrekter Analyse (ohne letzteres 1).*
- *Ist der Speedup "multiplikativ variabel" (gemeint ist $T_{original}/T_{mod} = x$ für $x \in \omega(1)$ für **manche** Eingaben G) gibt das je nach Speedup x bis zu drei Punkte mit korrektem Algorithmus und korrekter Analyse (ohne letzteres 1,5).*
- *Ist der Speedup "multiplikativ" (gemeint ist $T_{original}/T_{mod} = x$ für $x \in \Omega(1)$ für **alle** Eingaben G) gibt das je nach Speedup x die volle Punktzahl mit korrektem Algorithmus und korrekter Analyse (ohne letzteres die Hälfte).*

Aufgabe 7: Binäre Suchbäume *(27 Punkte)*

(a) Gegeben sei der folgende binäre Suchbaum:



Geben Sie **alle Folgen** von `insert`(key) Operationen an, welche diesen Baum erzeugen. Entspricht einer dieser Folgen der **Pre-Order**-Traversierung? Entspricht einer dieser Folgen der **Post-Order**-Traversierung? Wenn ja, welche? *(5 Punkte)*

(b) Zeigen Sie folgende Eigenschaft binärer Suchbäume: Wenn ein Knoten zwei Kinder hat, dann hat sein Vorgänger kein rechtes Kind und sein Nachfolger kein linkes Kind. *(6 Punkte)*

(c) Seit T ein binärer Suchbaum, bei dem die Knoten statt pointer zum **parent**-Knoten jeweils einen pointer zu ihrem **Vorgänger** und **Nachfolger** haben, d.h. für alle Knoten u gibt es die Attribute u.*pred* und u.*succ*.

Beschreiben Sie einen (möglichst effizienten) Algorithmus, der von einem Knoten u den parent-Knoten zurückgibt. Begründen Sie die Korrektheit Ihres Algorithmus und analysieren Sie die Laufzeit (in Abhängigkeit von der Tiefe des Baums). *(8 Punkte)*

Hinweis: Überlegen Sie sich zunächst wie man vorgehen würde, wenn man wüsste, ob u das linke oder das rechte Kind seines parents ist (oder ob u gar keinen parent hat).

(d) Ein binärer Suchbaum heiße **perfekt balanciert**, wenn für jeden Knoten u sich die Anzahl der Knoten im linken Teilbaum von u von der Anzahl der Knoten im rechten Teilbaum von u um höchstens 1 unterscheidet. Beschreiben Sie einen (möglichst effizienten) Algorithmus, der testet, ob ein binärer Suchbaum perfekt balanciert ist. Analysieren Sie die Laufzeit Ihres Algorithmus. *(8 Punkte)*

Musterlösung

(a) (i) 8 5 6 9
(ii) 8 5 9 6
(iii) 8 9 5 6
(i) entspricht der pre-order Traversierung. Keine entspricht der post-order Traversierung.

(b) Wenn ein Knoten x ein linkes Kind hat, dann ist sein Vorgänger der Knoten mit dem größten Schlüssel im linken Teilbaum von x. Dieser hat kein rechtes Kind, da dies auch im linken Teilbaum von x wäre und einen noch größeren Schlüssel hätte.

Wenn ein Knoten x ein rechtes Kind hat, dann ist sein Nachfolger der Knoten mit dem kleinsten Schlüssel im rechten Teilbaum von x. Dieser hat kein linkes Kind, da dies auch im rechten Teilbaum von x wäre und einen noch kleineren Schlüssel hätte.

(c) Sei $\min(x)$ der Knoten mit kleinstem und $\max(x)$ der Knoten mit größtem Schlüssel des Teilbaums mit Wurzel x. Diese kann man (ohne einen parent-pointer) in $\mathcal{O}$(Tiefe) finden. Falls $x = x$.*parent.left*, so ist x.*parent* $= \max(x)$.*succ*. Falls $x = x$.*parent.right*, so ist x.*parent* $= \min(x)$.*pred*. Der Algorithmus lautet also

Algorithm 4 `findparent` (x)

```
if x = max(x).succ.left then
    x.parent = max(x).succ
else if x = min(x).pred.right then
    x.parent = min(x).pred
else
    x.parent =none
```

(d) Der folgende Algorithmus bekommt als Eingabe einen Schlüssel x und gibt ein Tupel (y, z) aus, wobei y entweder true oder false ist, je nachdem ob der Teilbaum mit Wurzel x perfekt balanciert ist, und z die Anzahl der Elemente im Teilbaum mit Wurzel x ist (wird für die Rekursion benötigt).

Algorithm 5 A(x)

```
if x == None then
    return (true,0)
else
    if A(x.left)[0] == true & A(x.right)[0] == true & |A(x.left)[1]-A(x.right)[1]|≤ 1 then
        return (true,A(x.left)[1]+A(x.right)[1]+1)
    else
        return (false,0)
```

Der Algorithmus wird für jeden Knoten einmal aufgerufen und ein Durchgang (ohne Rekursion) dauert $\mathcal{O}(1)$. Die Laufzeit beträgt also $\mathcal{O}(n)$.

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

Klausur Informatik 2

Montag den 3. September 2018, 13:00-16:00

Name: ....

Matrikel Nr.: ....

Unterschrift: ....

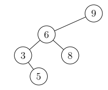
Erst öffnen wenn die Klausuraufsicht die Erlaubnis erteilt!

- Legen Sie Ihren **Studentenausweis** sichtbar vor sich.
- Schreiben Sie Ihren **Namen** und Ihre **Matrikelnummer** an die vorgesehenen Stellen.
- **Unterschreiben Sie** diese Seite um zu bestätigen, dass Sie die Fragen ohne unerlaubte Hilfsmittel beantwortet haben und die Klausuraufsicht über (gesundheitliche) Probleme informiert haben.
- Dies ist eine **Open Book** Klausur weshalb alle gedruckten und handgeschriebenen Materialien erlaubt sind. **Elektronische Hilfsmittel** sind **nicht** erlaubt.
- Schreiben Sie **lesbar** und nur mit **dokumentenechten** Stiften. Benutzen Sie **keine rote** Farbe und **keinen Bleistift**!
- 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 als Blackbox** verwendet werden.
- **Lesen Sie jede Aufgabe sorgfältig** durch und stellen Sie sicher dass Sie diese verstanden haben!
- Falls Sie eine Frage zur Aufgabenstellung haben, geben Sie der Klausuraufsicht ein **Handzeichen**.
- Schreiben Sie Ihren Namen auf **alle Blätter**!

Aufgabe	1	2	3	4	5	6	7	8	**Total**
Maximum	13	9	9	11	12	12	13	11	90
Punkte									

Aufgabe 1: Kurze Fragen *(13 Punkte)*

(a) Gegeben sei der folgende binäre Suchbaum mit Wurzel 9:



Geben Sie **alle Folgen** von `insert`(key) Operationen an, welche diesen Baum erzeugen. *(2 Punkte)*

(b) Geben Sie jeweils die Besuchsreihenfolge der Knoten für die **In-Order, Pre-Order, Post-Order und Level-Order** Traversierung für den obigen Baum an. *(2 Punkte)*

(c) Nennen Sie **einen Vorteil** von **einfach** verketteten Listen gegenüber **doppelt** verketteten Listen. Geben Sie für beide Datenstrukturen die Laufzeit der **Operation Löschen** an, falls ein Pointer auf das zu löschende Element gegeben ist. *(2 Punkte)*

(d) Arrangieren Sie die Werte des folgenden Arrays $A = [8, 7, 7, 6, 5, 4, 3, 3, 3, 2, 1]$ um, so dass A einen **gültigen Min-Heap** repräsentiert. Sie können davon ausgehen, dass das Array mit Index 1 beginnt. *(2 Punkte)*

(e) Angenommen wir wissen, dass ein gegebener Algorithmus A für eine Eingabe der Größe n eine Laufzeit von $\Omega(n \log n)$ hat. Wir wissen außerdem, dass ein Algorithmus B für die selbe Eingabe eine Laufzeit von $\Theta(n^2)$ hat. Können wir folgern, dass A eine bessere **asymptotische** Laufzeit hat als B? **Begründen Sie** Ihre Antwort kurz. *(2 Punkte)*

(f) Sei $\Sigma = \{a, b, c, d, e\}$ ein Alphabet. Die Wahrscheinlichkeit (relative Häufigkeit) p_x für ein Zeichen $x \in \Sigma$, sei wie folgt:

$$p_a = \tfrac{3}{10},\ p_b = \tfrac{3}{10},\ p_c = \tfrac{2}{10},\ p_d = \tfrac{1}{10},\ p_e = \tfrac{1}{10}.$$

Konstruieren Sie eine **Huffman-Kodierung** von Σ bzgl. dieser Häufigkeitsverteilung mit dem Greedy-Verfahren aus der Vorlesung. *(2 Punkte)*

(g) Geben Sie die **erwartete (durchschnittliche) Codewortlänge** Ihrer zuvor berechneten Huffman Kodierung an. *(1 Punkt)*

Musterlösung

(a) $9, 6, 3, 8, 5$
$9, 6, 3, 5, 8$

$9, 6, 8, 3, 5$

Ein Punkt für die erste Folge, ein halber für jede weitere.

(b) In-Order: $3, 5, 6, 8, 9$ *($\frac{1}{2}$ Punkte)*
Pre-Order: $9, 6, 3, 5, 8$ *($\frac{1}{2}$ Punkte)*
Post-Order: $5, 3, 8, 6, 9$ *($\frac{1}{2}$ Punkte)*
Level-Order: $9, 6, 3, 8, 5$ *($\frac{1}{2}$ Punkte)*

(c) Eine einfach verkettete Liste benötigt weniger Speicherplatz da weniger Pointer gespeichert werden müssen. *(1 Punkt)*

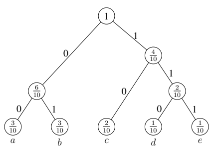
Die Laufzeit von Löschen (bei gegebenem Pointer auf das zu löschende Element) ist bei einfach verkettete Listen $\Theta(n)$ (da der Vorgänger ermittelt werden muss). Bei doppelt verketteten Listen hingegen $\mathcal{O}(1)$. *(1 Punkt)*

(d) Nein. *(1 Punkt)*

Der Grund ist dass für die Laufzeit von A nur eine untere Schranke für die asymptotische Laufzeit gegeben ist. A könnte mit basierend auf unserem Wissen trotzdem beliebig langsam sein. *(1 Punkt)*

(e) $A = (1, 2, 3, 3, 3, 4, 5, 6, 7, 7, 8)$. *Für jeden Knoten der die Min-Heap Eigenschaft verletzt gibt es einen Punkt Abzug.*

(f) Mit dem Verfahren aus der Vorlesung erhalten wir den folgenden Baum:



Wir lesen die folgende Kodierung ab:

$$a : 00,\ b : 01,\ c : 10,\ d : 110,\ e : 111.$$

(g) Die mittlere Codeworlänge ist $2 \cdot \frac{3}{10} + 2 \cdot \frac{3}{10} + 2 \cdot \frac{2}{10} + 3 \cdot \frac{1}{10} + 3 \cdot \frac{1}{10} = \frac{22}{10} = 2{,}2$. *(Rechnung nicht notwendig).*

Aufgabe 2: Landau Notation *(9 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 *(jeweils 2 Punkte)*.

(a) $10n^{1/3} \in \mathcal{O}(\sqrt{n})$

(b) $n! \in \Omega(n^n)$ *Dabei ist* $n! := n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$.

(c) $2^n + 2^n \in \Theta(4^n)$

Musterlösung

(a) Die Aussage ist wahr. *(1 Punkt)*

Wir zeigen dass es ein $c > 0, n_0 \in \mathbb{N}$ gibt sodass für alle $n \geq n_0$

$$\begin{aligned} & 10n^{1/3} \leq c \cdot \sqrt{n} \\ \iff \quad & 10 \leq c \cdot n^{1/6} \end{aligned}$$

Das ist beispielsweise für $c = 1$ für alle $n \geq 10^6 =: n_0$ der Fall. *(2 Punkte)*

Alternativ darf man zeigen, dass $\frac{10n^{1/3}}{\sqrt{n}}$ gegen einen endlichen Wert konvergiert:

$$\lim_{n\to\infty} \frac{10n^{1/3}}{\sqrt{n}} = 10 \cdot \lim_{n\to\infty} \frac{1}{n^{1/6}} = 10 \cdot 0 = 0.$$

(b) Die Aussage ist falsch. *(1 Punkt)*

Angenommen die Aussage wäre richtig, dann gibt es $c > 0, n_0 \in \mathbb{N}$ sodass für alle $n \geq n_0$

$$\begin{aligned} & n! \geq c \cdot n^n \\ \iff \quad & n \cdot (n-1) \cdots 3 \cdot 2 \geq c \cdot n^n \\ \implies \quad & n^{n-1} \geq c \cdot n^n \\ \iff \quad & \tfrac{1}{c} \geq n \end{aligned}$$

was für $n > \frac{1}{c}$ widersprüchlich ist.

Alternativ zeigen wir dass $n! \in o(n^n)$ was keine Schnittemenge mit $\Omega(n^n)$ hat.

$$0 \leq \lim_{n\to\infty} \frac{n!}{n^n} \leq \lim_{n\to\infty} \frac{n^{n-1}}{n^n} = \lim_{n\to\infty} \frac{1}{n} = 0.$$

Damit ist auch $\lim\limits_{n\to\infty} \frac{n!}{n^n} = 0$ (Sandwich Theorem).

(c) Die Aussage ist falsch. *(1 Punkt)*

Angenommen es gäbe ein $c > 0, n_0 \in \mathbb{N}$ sodass für alle $n \geq n_0$, dann ist

$$\begin{aligned} & 2^n + 2^n \geq c \cdot 4^n \\ \iff \quad & \tfrac{1}{c} \geq \tfrac{4^n}{2^n+2^n} \\ \iff \quad & \tfrac{1}{c} \geq 2^{n-1} \end{aligned}$$

ein Widerspruch für alle $n \geq 1 + \log \frac{1}{c}$. Daher $2^n + 2^n \notin \Omega(4^n)$ und somit $2^n + 2^n \notin \Theta(4^n)$. *(2 Punkte)*

Alternativ zeigen wir dass $2^n + 2^n \in o(4^n)$ was keine Schnittemenge mit $\Theta(4^n)$ hat.

$$\lim_{n\to\infty} \frac{2^n + 2^n}{4^n} = \lim_{n\to\infty} \frac{1}{2^{n-1}} = 0.$$

Aufgabe 3: Sortieralgorithmen *(9 Punkte)*

Gegeben sei der folgende Sortieralgorithmus `BucketSort` als Pseudocode. Der Algorithmus erhält als Argument eine **unsortierte, einfach verkettete Liste** L mit n Listenelementen, denen jeweils ein **positiver, ganzzahliger** Sortierschlüssel zugeordnet ist, sowie den größten Schlüssel m in L.

Algorithm 1 `BucketSort`(List L, max. key m)

```
Create new lists B_0, ..., B_m
while L is not empty do
    e ← L.pop()              ▷ L.pop() removes and returns the first element of L
    k ← e.key
    B_k.append(e)            ▷ Attach e to the tail of B_k
return B_0 ∘ ... ∘ B_m       ▷ The "∘" operator concatenates two lists in constant time
```

(a) Begründen Sie kurz die **Korrektheit** von `BucketSort` und geben Sie die **Laufzeit** in Abhängigkeit von m und n an. *(2 Punkte)*

Wir wollen nun die asymptotische Laufzeit von `BucketSort` der asymptotischen Laufzeit des Sortierverfahrens `QuickSort` gegenüberstellen. Gehen Sie davon aus, dass `QuickSort` die zu sortierende Schlüsselfolge in einem Array erhält und wir stets **das vorderste Element** des aktuellen (Teil-) Arrays als **Pivotelement** wählen.

(b) Geben Sie die Laufzeit dieser Variante von `QuickSort` für den Fall an, dass das Eingabearray ***n* mal den Schlüssel 1** enthält. **Begründen Sie Ihre Antwort.** *(3 Punkte)*

(c) Geben Sie die Laufzeit von `BucketSort` für den Fall an, dass die Eingabeliste ***n* mal den Schlüssel 1** enthält. *(1 Punkt)*

(d) Geben Sie für den **Worst-Case** und den **Best-Case** von `QuickSort` obere Schranken m_w, m_b für den größten Schlüssel m an, sodass `BucketSort` die Eingabefolge in dem entsprechenden Fall **asymptotisch** genauso schnell oder schneller sortiert als `QuickSort`. Drücken Sie m_w, m_b mit Hilfe der $\mathcal{O}$-Notation aus. *(3 Punkte)*

Musterlösung

(a) *Korrektheit*: Nach dem Durchlaufen der Schleife enthält jede Liste B_i (Bucket) alle Elemente aus L mit Schlüssel i und jedes Element von L befindet sich in einem Bucket. Da wir die B_i in aufsteigender Reihenfolge in einer Liste zusammenfassen ist die Liste danach sortiert. *(1 Punkt)*

Laufzeit: $\Theta(n+m)$ (ebenfalls ok ist $\mathcal{O}(n+m)$). *(1 Punkt)*

Das liegt daran dass die Buckets wahlfrei adressiert werden können (d.h. in $\mathcal{O}(1)$). Sollte jemand auf die Laufzeit $\Theta(nm)$ kommen, gibt das nur Punkte wenn auch erklärt wird warum das so sein soll. Wenn man beispielsweise argumentiert dass das entsprechende Bucket erst gesucht werden muss, weil die Buckets beispielsweise in einer Liste gespeichert sind (oder einer ähnlich langsamen Datenstruktur). Auch wenn das nicht optimal ist kann man in Kombination mit einer Erklärung den ganzen Punkt geben.

(b) $\Theta(n^2)$. Ok wäre auch $\mathcal{O}(n^2)$ zu schreiben (auch wenn man hier eigentlich die untere asymptotische Schranke betonen möchte). *(1 Punkt)*

Begründung: Quicksort trennt immer nur das Pivot vom Eingabearray ab und partitioniert dann alle restlichen Elemente auf eine Seite des Arrays (je nach Implementierung entweder **alle** nach rechts oder **alle** nach links). *(1 Punkt)*

Quicksort wird also einerseits rekursiv auf ein Array aufgerufen dass nur ein Element weniger hat, während die Rekursion auf das andere (leere) Teilarray sofort terminiert. Die Laufzeit ist also $\sum_{i=0}^{n} ci \in \Omega(n^2)$ (für eine geeignete Konstante $c > 0$). *(1 Punkt)*

(c) $\Theta(n)$. Ok wäre auch $\mathcal{O}(n)$. *(1 Punkt)*

(d) $m_w \in \mathcal{O}(n^2)$. *(1,5 Punkte)*
$m_b \in \mathcal{O}(n \log n)$. *(1,5 Punkte)*

Aufgabe 4: Hashing *(11 Punkte)*

Gegeben seien zwei leere Hashtabellen H_1 der Größe m_1 und H_2 der Größe m_2. Gegeben seien zudem die zugehörigen Hashfunktionen

$$h_1(x) := (x + 5) \mod m_1, \quad h_2(x, i) := (x + 2i) \mod m_2$$

wobei x der einzufügende Schlüssel ist und i die Anzahl der Kollisionen bei dem aktuellen Einfügeprozess. Mittels h_1 wollen wir Elemente in H_1 nach dem Prinzip **Hashing mit Chaining** einfügen. Mittels h_2 wollen wir dasselbe mit H_2 nach dem Prinzip **Hashing mit linearem Sondieren** tun.

(a) Geben Sie für ein allgemeines n eine Folge von n **paarweise verschiedenen** Schlüsseln $x_1, \ldots, x_n$ und einen Suchschlüssel x_j für $1 \leq j \leq n$ an, sodass nach Einfügen von $x_1, \ldots, x_n$ in dieser Reihenfolge das Finden von x_j mindestens $\Omega(n)$ Zeit benötigt. Tun Sie dies **sowohl für H_1 als auch für H_2** unter der Annahme, dass diese initial leer sind und $m_2 > 2n$. *(4 Punkte)*

(b) Sei $m_1 := 7$ und $m_2 := 11$. Fügen Sie die Schlüssel $5, 12, 1, 8, 19$ in der gegebenen Reihenfolge jeweils in **beide** Datenstrukturen H_1, H_2 ein und geben Sie deren Zustand nach dem Einfügen aller Schlüssel in den dafür vorgesehenen Tabellen an. *(3 Punkte)*

(c) Sei H eine weitere Hashtabelle der Größe 7, welche für die Kollisionsauflösung das Prinzip des **Cuckoo-Hashing** benutzt. Seien

$$g_1(x) := (x + 5) \mod 7, \quad g_2(x) := 2x \mod 7$$

die zugehörigen Hashfunktionen. Fügen Sie der Reihe nach die Schlüssel $1, 2, 4, 8, 9$ ein und geben Sie den Zustand von H **nach** Einfügen der 8 und der 9 in den vorgesehenen Tabellen an. *(4 Punkte)*

Musterlösung

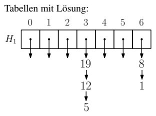
(a) Für H_1: Schlüsselfolge bspw. $x_i := i \cdot m_1$ *(1 Punkt)*

mit Suchschlüssel x_1. *(1 Punkt)*

Für H_2: Schlüsselfolge bspw. $x_i := i \cdot m_2$ *(1 Punkt)*

mit Suchschlüssel x_n. *(1 Punkt)*

(b) Tabellen mit Lösung:



H_2		12		1		5			8		19
	0	1	2	3	4	5	6	7	8	9	10

1,5 Punkte pro korrekter Tabelle. Daumenregel: Ein Punkt Abzug pro falschem Wert. Es ist beim Hashing mit Chaining auch in Ordnung die Reihenfolge innerhalb einer Liste anders herum zu machen, denn man könnte das so implementieren dass immer am Ende der Liste eingefügt wird (mit einem zusätzlichen Pointer pro Liste). Meine Vermutung ist, dass bei H_1 viele das $+5$ in der Hashfunktion übersehen. Wenn sonst alles richtig ist (also alles nur 5 Zellen nach links verschoben ist) kann man immerhin noch einen halben Punkt geben.

(c) Tabellen mit Lösung:

Zustand vor Einfügen der 8 (unbewertet):

H	2		4				1
	0	1	2	3	4	5	6

Zustand nach Einfügen der 8:

H	2	4	1				8
	0	1	2	3	4	5	6

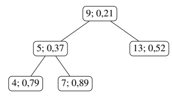
Zustand nach Einfügen der 9:

H	9	4	1		2		8
	0	1	2	3	4	5	6

Zwei Punkte pro korrekter Tabelle. Daumenregel: Ein Punkt Abzug pro falschem Wert. Wenn es dann nicht zu einfach wird (keine Verdrängung), kann man bei der zweiten Tabelle Folgefehler berücksichtigen.

Aufgabe 5: Treap *(12 Punkte)*

Betrachten Sie den folgenden **Treap**, dessen Knoten Paare von Schlüsseln $(key1; key2)$ enthalten.

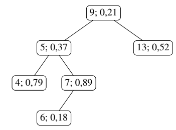


(a) **Fügen Sie das Paar (6; 0,18) in den Treap ein** und führen Sie die notwendigen Operationen durch, um die Treap Eigenschaften wiederherzustellen. Geben Sie nach **jeder** Rotation den entsprechenden Baum an und dokumentieren Sie kurz Ihre Schritte. *(6 Punkte)*

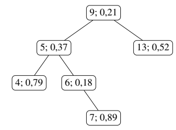
(b) Führen Sie die notwendigen Operationen durch, um **den Knoten mit dem Schlüssel 9** aus dem **ursprünglichen** Treap zu **löschen** (siehe oben). Geben Sie nach **jeder** Rotation den entsprechenden Baum an und dokumentieren Sie kurz Ihre Schritte. *(6 Punkte)*

Musterlösung

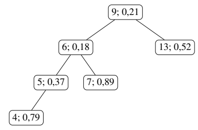
(a) Füge (6; 0,18) zunächst entsprechend der binären Suchbaumeigenschaft ein:



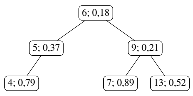
`rotate-right`(7,6): *(2 Punkte)*



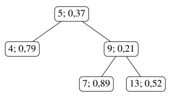
`rotate-left`(5,6): *(2 Punkte)*



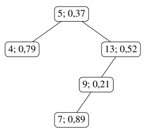
`rotate-right`(9,6): *(2 Punkte)*



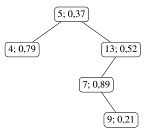
(b) `rotate-right`(9,5): *(2 Punkte)*



`rotate-left`(9,13): *(2 Punkte)*



Da die 9 jetzt nur noch ein Kind hat können wir sie direkt löschen und die 7 direkt an die 13 hängen. *Alternativ* kann man die 9 mittels `rotate-right`(9,7) auch noch weiter bis zu einem Blatt nach unten rotieren und erst dort löschen: *(2 Punkte)*



Aufgabe 6: Algorithmenanalyse *(12 Punkte)*

Betrachten Sie folgenden Algorithmus, der als Eingabe ein Array $A[0, \ldots, n-1]$ der Länge n erhält:

Algorithm 2 `algorithm`(A)

```
for i ← 2 to n − 1 do
    for j ← 1 to i − 1 do
        for k ← 0 to j − 1 do
            if |A[k] − A[j]| = |A[k] − A[i]| then
                return True
return False
```

(a) Geben Sie die **Ausgabe** von `algorithm` für die Eingaben $A_1 = [3, 2, 6, 0]$, $A_2 = [4, 1, 5, 6]$ und $A_3 = [-3, 0, 3, 6]$ an. *(3 Punkte)*

(b) Geben Sie die asymptotische Laufzeit von `algorithm` in Abhängigkeit von n an. *(2 Punkte)*

(c) Beschreiben Sie einen Algorithmus mit Laufzeit $\mathcal{O}(n \log n)$, der für ein Array $A[0, \ldots, n–1]$ überprüft, ob es $i, j \in \{0, \ldots, n-1\}$ mit $i \neq j$ gibt, sodass $A[i] = A[j]$. *(2 Punkte)*

(d) Beschreiben Sie einen Algorithmus, der für jedes Array die gleiche Ausgabe erzeugt wie `algorithm`, aber eine bessere asymptotische Laufzeit hat. Ihr Algorithmus darf einen höheren Speicherbedarf haben als `algorithm`. Erklären Sie die Laufzeit Ihres Algorithmus. *(5 Punkte)*

Hinweis: Sie dürfen den Algorithmus aus (c) *als Blackbox verwenden, auch wenn Sie* (c) *nicht gelöst haben.*

Musterlösung

(a) `algorithm`(A_1) =`True`, `algorithm`(A_2) =`algorithm`(A_3) =`False`.

(b) $\Theta(n^3)$. $\mathcal{O}(n^3)$ ist ebenfalls in Ordnung.

(c) Man sortiert das Array ($\mathcal{O}(n \log n)$) und überprüft es in einem Durchlauf auf gleiche Werte, welche dann, falls vorhanden, nebeneinander stehen ($\mathcal{O}(n)$).

(d) Für jedes $k \in \{0, \ldots, n-3\}$ erstellt man eine Liste mit Einträgen $|A[k] - A[j]|$ für $k < j < n-1$ ($\mathcal{O}(n^2)$). *(2 Punkte)*

Dann überprüft man jede Liste auf gleiche Werte. Dies benötigt nach (c) $\mathcal{O}(n \log n)$ für eine einzelne Liste, also insgesamt $\mathcal{O}(n^2 \log n)$. *(2 Punkte)*

Zusammen benötigt man also $\mathcal{O}(n^2 + n^2 \log n) = \mathcal{O}(n^2 \log n)$. *(1 Punkt)*

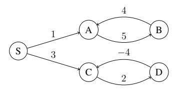
Aufgabe 7: Graphenalgorithmen *(13 Punkte)*

Sei $G = (V, E, w)$ ein ungerichteter, gewichteter Graph. Angenommmen es gebe **für jeden Schnitt** $(S, V \setminus S)$ eine **eindeutige leichte Schnittkante**, d.h. eine Schnittkante, deren Gewicht echt kleiner als das aller anderen Schnittkanten ist.

(a) Zeigen Sie, dass G einen minimalen Spannbaum hat und das dieser **eindeutig** ist. *(6 Punkte)*

(b) Folgt umgekehrt aus der Existenz eines eindeutigen minimalen Spannbaums auch die Existenz einer eindeutigen leichten Schnittkante für jeden Schnitt? **Beweisen Sie** Ihre Antwort. *(3 Punkte)*

Gegeben sei der folgende gewichtete, gerichtete Graph:



(c) Führen Sie auf diesem Graph den **Bellman-Ford** Algorithmus mit Startknoten S aus und tragen Sie **nach jeder** Iteration der äußeren Schleife die bisher berechneten Distanzen von S **in der vorgegeben Tabelle** ein (siehe Lösungsblatt). Iterieren Sie in der inneren Schleife die Kanten **aufsteigend nach Größe der Gewichte**. *(4 Punkte)*

Musterlösung

(a) *Existenz eines MST:* Zeige, dass G zusammenhängend ist. Sei dazu $u \in V$ und S die Zusammenhangskomponente, welche u enthält. Falls $S \neq V$, gibt es nach Voraussetzung eine Kante $\{v, w\} \in E$ mit $v \in S$ und $w \in V \setminus S$. Da w von u aus erreichbar ist, muss $w \in S$ gelten, Widerspruch. *(2 Punkte)*

Eindeutigkeit des MST: Angenommen es gebe zwei verschiedene MST T und T'. Dann gibt es eine Kante $\{u, v\}$ in T, welche nicht in T' ist. Durch Löschen von $\{u, v\}$ aus T zerfällt T in zwei Zusammenhangskomponenten, deren Knotenmengen wir mit T_u und T_v bezeichnen (mit $u \in T_u$ und $v \in T_v$) *(1 Punkt).*

Sei $\{x, y\}$ die eindeutige leichteste Schnittkante zwischen T_u und T_v. Falls $\{x, y\} \neq \{u, v\}$, so kann man in T $\{u, v\}$ durch $\{x, y\}$ ersetzen und erhält einen Spannbaum mit kleinerem Gewicht als T, Widerspruch. *(1 Punkt)*

Nehme also an, dass $\{u, v\}$ die eindeutige leichteste Kante zwischen T_u und T_v ist. Sei p ein Pfad in T' von u nach v und e eine Kante im Pfad, welche Schnittkante von (T_u, T_v) ist. Fügt

man $\{u, v\}$ zu T' hinzu, erhält man einen Kreis bestehend aus p und $\{u, v\}$. Durch Löschen einer beliebigen Kante aus dem Kreis erhält man wieder einen Spannbaum. Fügt man also $\{u, v\}$ zu T' hinzu und löscht e, erhält man wegen $w(\{u, v\}) < w(e)$ einen Spannbaum mit kleinerem Gewicht, Widerspruch. *(2 Punkte)*

(b) Sei $V = \{x, y, z\}$, $E = \{\{x, y\}, \{x, z\}\}$ und $w(\{x, y\}) = w(\{x, z\}) = 1$. Dann hat $G = (V, E)$ einen eindeutigen MST (G selbst), aber der Schnitt $(\{x\}, \{y, z\})$ hat keine eindeutige leichte Schnittkante. *(3 Punkte)*

(c) Es gibt höchstens einen Punkt pro korrekter Zeile (Folgefehler können ggf. berücksichtigt werden). Innerhalb einer Zeile gibt es einen halben Punkt Abzug pro falschem Wert. Die Reihenfolge der Relaxierung der Kante muss eingehalten werden (siehe Aufgabe)! *(4 Punkte)*

	$\delta(S,S)$	$\delta(S,A)$	$\delta(S,B)$	$\delta(S,C)$	$\delta(S,D)$
Initial.	0	∞	∞	∞	∞
$i = 1$	0	1	6	3	∞
$i = 2$	0	1	6	3	5
$i = 3$	0	1	6	1	3
$i = 4$	0	1	6	-1	1

Aufgabe 8: Dynamische Programmierung *(11 Punkte)*

Betrachten Sie die folgenden Funktionen auf natürlichen Zahlen:

$$f_1(n) = n - 1$$

$$f_2(n) = \begin{cases} \frac{n}{2} & \text{falls } 2 \mid n \\ n & \text{sonst} \end{cases}$$

$$f_3(n) = \begin{cases} \frac{n}{3} & \text{falls } 3 \mid n \\ n & \text{sonst} \end{cases}$$

$m \mid n$ ("m teilt n") bedeutet es gibt ein $k \in \mathbb{N}$ so dass $k \cdot m = n$.

Man betrachtet folgendes Problem: Für ein gegebenes $n \geq 1$, finde die **minimale Anzahl an Anwendungen** von f_1, f_2, f_3, die man benötigt, um 1 zu erhalten. Formal: Finde das minimale k, so dass es $i_1, \ldots, i_k \in \{1, 2, 3\}$ gibt mit $f_{i_1}(f_{i_2}(\ldots(f_{i_k}(n))\ldots)) = 1$.

Ein Student schlägt dazu folgenden Algorithmus vor:

Algorithm 3 `steps_to_one`(n)

```
s ← 0
while n > 1 do
    if 3 | n then
        n ← n/3
    else if 2 | n then
        n ← n/2
    else
        n ← n − 1
    s ← s + 1
return s
```

(a) Welches Algorithmendesign-Prinzip benutzt der obige Algorithmus? *(1 Punkt)*

(b) Löst `steps_to_one` für jedes $n \geq 1$ das gegebene Problem? Beweisen Sie Ihre Antwort. *(3 Punkte)*

(c) Geben Sie einen Algorithmus **in Pseudocode** an, der das Prinzip der dynamischen Programmierung anwendet, um das obige Problem zu lösen. Analysieren Sie die (asymptotische) Laufzeit ihres Algorithmus. Diese sollte maximal polynomiell in n sein. *(7 Punkte)*

Musterlösung

(a) Beim gegebenen Algorithmus handelt es sich um einen Greedy-Algorithmus.

(b) Für $n = 10$ führt der Algorithmus folgende Schritte aus:

$$10 \xrightarrow{f_2} 5 \xrightarrow{f_1} 4 \xrightarrow{f_2} 2 \xrightarrow{f_2} 1$$

Optimal hingegen ist

$$10 \xrightarrow{f_1} 9 \xrightarrow{f_3} 3 \xrightarrow{f_3} 1$$

(c) Algorithmus: *(5 Punkte)*

memo $= \{\}$

Algorithm 4 `steps_to_one`(n)

```
if n in memo then
    return memo[n]
if n == 1 then
    s = 0
else
    x = steps_to_one(n − 1)
    if n | 2 then
        y = steps_to_one(n/2)
    else
        y = ∞
    if n | 3 then
        z = steps_to_one(n/3)
    else
        z = ∞
    s = 1 + min{x, y, z}
memo[n] = s
return s
```

Laufzeit: *(2 Punkte)*

Es gibt n Aufrufe, bei denen nicht der Wert aus *memo* genommen wird. Ein Aufruf benötigt $\mathcal{O}(1)$ (Rekursion ignoriert). Die Laufzeit beträgt also $\mathcal{O}(n)$.

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* $\forall n \in \mathbb{N} : \log_3(n) \leq n$.

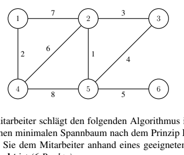
(b) $\sqrt[3]{n} \in \Theta(\sqrt{n})$

(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 = ∅ oder |V| = 1 then return ∅                  ▷ Basisfall

Teile V in gleichgroße Mengen V1,V2 auf
Seien G1 bzw. G2 die durch V1 bzw. V2 induzierten Teilgraphen.

M1 ← DivConMST(G1)                   ▷ Rekursive Ermittlung der min. Spannbäume
M2 ← DivConMST(G2)

Ermittle leichteste Kante e = {v1, v2} ∈ E mit v1 ∈ V1, v2 ∈ V2.
return M1 ∪ M2 ∪ {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)

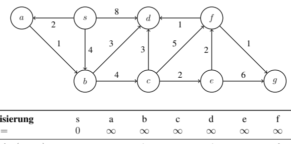
(a) Gegeben sei ein **binärer Baum** T dessen Knoten **paarweise unterschiedliche** Schlüssel enthalten. Geben Sie einen **rekursiven** Algorithmus an, welcher prüft ob T ein binärer **Such**baum ist *(10 Punkte)*.

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.

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

Informatik II: Algorithmen & Datenstrukturen

Montag, 29. August, 2014, 14:00 – 17:00

Name: ....

Matrikelnummer: ....

Unterschrift: ....

Blättern Sie nicht um bevor Sie dazu aufgefordert werden!

- Schreiben Sie **auf alle Blätter** (inklusive Deckblatt und etwaiger zusätzlicher Blätter) Ihren **Vornamen, Nachnamen** und Ihre **Matrikelnummer**.
- **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. Schreiben Sie nicht in **rot** oder **grün** und benutzen Sie keinen Bleistift!
- Alle schriftlichen Hilfsmittel (Bücher, Vorlesungsunterlagen, handschriftliche Notizen, etc.) sind erlaubt. **Elektronische Hilfsmittel sind nicht erlaubt.**
- Die Klausur besteht aus 8 Aufgaben und 120 Punkten. Zum Bestehen reichen 50 Punkte.
- Benutzen Sie für jede Aufgabe eine eigene Seite.
- Es wird nur eine Lösung pro Aufgabe gewertet. Vergewissern Sie sich, dass Sie zusätzliche Lösungen selbst entwerten. Falls mehrere Lösungen zu einer Aufgabe existieren, so wird die schlechtere Lösung gewertet.
- Die folgenden Regeln gelten überall, außer sie werden explizit außer Kraft gesetzt. Bei Laufzeitfragen ist wie üblich nur die asymptotische Laufzeit notwendig. Wenn Sie einen Algorithmus angeben sollen, so können Sie Pseudocode angeben, eine Beschreibung der Funktionsweise Ihres Algorithmus ist allerdings ausreichend. Algorithmen sind immer effizient zu konstruieren, d. h., mindestens polynomiell und i. d. R. schneller als eine naive Lösungsmethode. 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.**
- **Erklären Sie Ihre Lösungen, außer es wird explizit darauf hingewiesen, dass dies nicht nötig ist! Nur das Endresultat aufzuschreiben ist nicht ausreichend.**

Frage	1	2	3	4	5	6	7	8	**Total**
Punkte									
Maximum	29	14	16	17	9	10	10	15	120

1 Kurze Fragen (29 Punkte)

a) (4 Punkte) Welche Eigenschaften müssen die Elemente eines Arrays $H[1,\ldots,n]$ besitzen, damit sie bei der Array-Implementierung aus der Vorlesung einen gültigen Heap (Prioritätswarteschlange) mit n Elementen bilden?

b) (3 Punkte) Ist es möglich, vergleichsbasiert, einen binären Suchbaum so zu implementieren, dass man mit $o(n \log n)$ Vergleichen n Schlüssel einfügen kann? Begründen Sie ihre Antwort.

c) (5 Punkte) Es sei ein DAG (gerichteter azyklischer Graph) gegeben. Nehmen Sie an, der Graph hat einen Knoten s, so dass jeder andere Knoten des Graphen über einen gerichteten Pfad von s erreichbar ist. Ergibt die Besuchsreihenfolge der Knoten einer bei s gestarteten BFS Traversierung eine topologische Sortierung des Graphen? Erkären Sie weshalb oder geben Sie ein Gegenbeispiel an.

d) (5 Punkte) Gegeben sei die folgende Häufigkeitsverteilung:

$$A : 10\%, B : 15\%, C : 8\%, D : 23\%, E : 11\%, F : 33\%.$$

Geben sie einen optimalen präfixfreien binären Code an.

e) (4 Punkte) Bestimmen Sie die Editierdistanz der Wörter *SONNE* und *MOND*. Geben Sie auch eine minimale Sequenz von Editieroperationen an, um von *SONNE* zu *MOND* zu kommen.

f) (3 Punkte) Sie haben eine Anwendung bei der zunächst n Elemente in eine Hashtable eingefügt werden und danach sehr viele Anfragen ($\gg n$) ausgeführt werden. Welches der in der Vorlesung besprochenen Hashing-Verfahren eignet sich am besten? Begründen Sie Ihre Wahl.

g) (5 Punkte) Zeichnen Sie einen gültigen Rot-Schwarz-Baum, welcher Tiefe 5 hat und welcher möglichst wenige Knoten enthält (NIL-Knoten sind hier nicht mitgezählt). Die Tiefe eines Rot-Schwarz-Baumes ist der größte Abstand eines Nicht-NIL-Knoten von der Wurzel.

Hinweis: *Es gibt einen solchen Baum mit 14 Nicht-NIL-Knoten. Sie müssen nur einen gültigen Baum zeichnen. Eine Sequenz von Operationen, welche den Baum erzeugt, müssen Sie* ***nicht*** *angeben.*

2 Landau Notation (14 Punkte)

Begründen Sie alle Antworten zu dieser Aufgabe ausführlich.

a) (6 Punkte) Zeigen Sie mit der Definition von $\mathcal{O}(\cdot)$, dass $\sqrt[3]{3n+10}$ in $\mathcal{O}(\sqrt[3]{n})$ enthalten ist.

b) (4 Punkte) Zeigen Sie, dass n^n in $\Omega(n!)$ liegt.

c) (4 Punkte) Gilt $\log_2(n^3) \in \Theta\big(\ln(3n^7)\big)$? Begründen Sie ihre Antwort.

3 Dynamische Programmierung (16 Punkte)

Wir suchen einen Algorithmus, der die minimale Münzenanzahl bestimmt, die man benötigt um eine vorgegeben Betrag B zu bezahlen (Man muss genau den Betrag B bezahlen, d.h. es gibt kein Rückgeld).

Genauer: Gegeben sei ein Array M mit k verschiedenen ganzzahligen Münzwerten $m_0,\ldots,m_{k-1} \in \mathbb{N}$, $k > 0$, $m_i \geq 1$ sowie ein ganzzahliger Betrag $B \in \mathbb{N}$. Wir wollen die Anzahl der Münzen $A := \sum_{i=0}^{k-1} a_i$ minimieren, wobei $a_0,\ldots,a_{k-1} \in \mathbb{N}_0$ ganze Zahlen sind, die $\sum_{i=0}^{k-1} a_i m_i = B$ erfüllen.

Ein fleißiger Student hat dazu den folgenden Algorithmus entworfen:

```
int minimum(int[] M,  int B):
    sort(M)          # sorts M in decreasing order
    int i = 0
    int c = 0
    while B > 0:
        if B < M[i]:
            i += 1
            if i >= len(M):
                return −1       # no solution found
        else:
            B = B − M[i]
            c += 1

    return c
```

a) (3 Punkte) Welches Algorithmendesign-Prinzip benutzt der obige Algorithmus (Algorithmendesign-Prinzipien sind z. B. Dynamische Programmierung oder Divide and Conquer)?

b) (5 Punkte) Geben Sie Münzwerte $m_0,\ldots,m_{k-1}$, sowie einen Betrag B vor, so dass zwar eine Lösung existiert, der obige Algorithmus jedoch keine optimale Lösung findet.

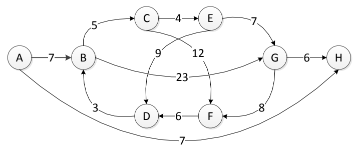
c) (8 Punkte) Beschreiben Sie einen auf dynamischer Programmierung basierenden Algorithmus, der die minimale Münzanzahl berechnet, die nötig ist um den Betrag B zu bezahlen.

Was ist die asymptotische Laufzeit Ihres Algorithmus?

Hinweis: *Sie können bei Teilaufgabe c) davon ausgehen, dass es immer möglich ist den Betrag B mit den Münzen zu bezahlen.*

4 Kürzeste Wege (17 Punkte)

a) (8 Punkte) Führen Sie den Dijkstra Algorithmus auf dem gegebenen gerichteten Graph aus um kürzeste Wege von Knoten A zu allen anderen Knoten zu finden. Schreiben Sie für jeden der Knoten $\{B,\ldots,H\}$ die Länge des Pfades von A zu diesem Knoten aus und geben Sie an, in welcher Reihenfolge der Dijkstra Algorithmus die Knoten bearbeitet.



Bei Teilaufgaben b) und c) ist ein ungerichteter Graph $G = (V, E)$ mit Kantengewichtsfunktion $w : E \to \mathbb{N}$ gegeben. Das Ziel dieser Aufgabe ist es für einen Startknoten $s \in V$, kürzeste Pfade zu allen anderen Knoten zu finden.

b) (3 Punkte) Geben Sie einen möglichst effizienten Algorithmus an um das Problem zu lösen, wenn alle Kantengewichte identisch 1 sind, d.h., $w(e) = 1$ für alle $e \in E$.

Was ist die Laufzeit Ihres Algorithmus?

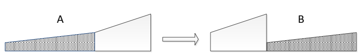
c) (6 Punkte) Geben Sie einen möglichst effizienten Algorithmus an um das Problem zu lösen, wenn alle Kantengewichte 1 oder 2 sind, d.h., $w(e) \in \{1, 2\}$ für alle $e \in E$.

Was ist die Laufzeit Ihres Algorithmus?

Hinweis: *Schnellere Algorithmen liefern mehr Punkte. Für (zu) langsame Algorithmen erhalten Sie nur wenige Punkte.*

5 Minimum Finden (9 Punke)

Sie sollen ein Array A der Länge n, welches aufsteigend sortiert mit n verschiedenen ganzen Zahlen gefüllt ist, erhalten. Leider geht bei der Übergabe etwas schief, so dass das Array nicht mehr korrekt sortiert ist. Statt des Arrays A erhalten Sie ein Array B, welches man wie folgt aus A erhält: Das Array A wird an einer Ihnen unbekannten Stelle r, $0 \leq r \leq n-1$ geteilt und dann vertauscht zusammen gesetzt, d.h. für $0 \leq j < r$ gilt $B[j] := A[r+j]$ und für $r \leq j < n$ gilt $B[j] := A[j-r]$.



Geben Sie einen Algorithmus an, welcher das kleinste Element im Array B findet und Laufzeit $\mathcal{O}(\log n)$ hat. Ihr Algorithmus bekommt lediglich das Array B als Eingabe und kennt das Array A nicht!

6 Mystische Suche (10 Punkte)

Betrachten Sie folgenden Pseudocode der Funktion *mystery(array)*, welche als Eingabe ein Array mit positiven ganzen Zahlen erhält.

```
int mystery(int[] array):
    result = []
    for i in range(0, len(array)):
        current = array[i]
        for j in range(i + 1, len(array)):
            if current == array[j]:
                result.append(current)
                break          # aborts inner for loop
    if len(result) == 0:
        return −1
    else:
        return result[0]
```

a) (5 Punkte) Welches Problem löst die Funktion *mystery*? Welche Laufzeit hat diese Funktion im *Worst Case*?

b) (5 Punkte) Wie kann die Funktion verbessert werden um das gleiche Problem in Linearzeit zu lösen?

7 Minimale Spannbäume (10 Punkte)

Bei den folgenden zwei Teilaufgaben müssen Sie jeweils einen zusammenhängenden, ungerichteten und gewichteten Graphen konstruieren, welcher aus einem Knoten v und noch mindestens 2 weiteren Knoten besteht. Die Kantengewichte des Graphen sollen alle verschieden sein.

a) (5 Punkte) Konstruieren Sie einen Graphen, so dass der minimale Spannbaum (MST) und der Shortest Path Tree von v **verschieden** sind (den Shortest Path Tree betrachten wir hierbei als ungerichteten Baum).

b) (5 Punkte) Konstruieren Sie einen Graphen, so dass der minimale Spannbaum (MST) und der Shortest Path Tree von v **gleich** sind (den Shortest Path Tree betrachten wir hierbei als ungerichteten Baum).

8 String Matching (15 Punkte)

Gegeben sei der folgende Pseudocode, welcher das String-Matching-Problem lösen soll.

```
int pattern_finder(String text, String pattern):
    int i = 0
    while i < len(text) − len(pattern) + 1:
        int counter = 0
        for j in range(0, len(pattern)):
            if text[i+j] == pattern[j]:
                counter +=1
            else:
                break
        if counter == 0:
            i += 1
        elif counter != len(pattern):
            i += counter
        else:
            return i
    return −1
```

a) (8 Punkte) Wieso ist der obige String-Matching Algorithmus nicht korrekt? Geben Sie ein Gegenbeispiel an. Welche Laufzeit hat der Algorithmus?

b) (7 Punkte) Der Algorithmus schlägt nicht für jedes Beispiel fehl. Für welche Eingaben funktioniert der Algorithmus trotzdem? Geben Sie eine (möglichst große) Klasse von Eingaben an.

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

Informatik II: Algorithmen & Datenstrukturen

Mittwoch, 4. März, 2015, 9:00 – 12:00

Name: ....

Matrikelnummer: ....

Unterschrift: ....

Blättern Sie nicht um bevor Sie dazu aufgefordert werden!

- Schreiben Sie **auf alle Blätter** (inklusive Deckblatt und etwaiger zusätzlicher Blätter) Ihren **Vornamen, Nachnamen** und Ihre **Matrikelnummer**.
- **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. Schreiben Sie nicht in **rot** oder **grün** und benutzen Sie keinen Bleistift!
- Alle schriftlichen Hilfsmittel (Bücher, Vorlesungsunterlagen, handschriftliche Notizen, etc.) sind erlaubt. **Elektronische Hilfsmittel sind nicht erlaubt. Das inkludiert Mobiltelefone.**
- Die Klausur besteht aus 7 Aufgaben und 120 Punkten. Zum Bestehen reichen 50 Punkte.
- Bleiben Sie nicht unnötig lange an einer Aufgabe hängen.
- Benutzen Sie für jede Aufgabe eine eigene Seite.
- Markieren Sie Schmierpapier als solches. Dieses können Sie dann auch abgeben, Notizen darauf können im Zweifelsfall zu Ihren Gunsten verwendet werden, nicht jedoch zu Ihrem Ungunsten.
- Es wird nur eine Lösung pro Aufgabe gewertet. Vergewissern Sie sich, dass Sie zusätzliche Lösungen selbst entwerten. Falls mehrere Lösungen zu einer Aufgabe existieren, so wird die schlechtere Lösung gewertet.
- Die folgenden Regeln gelten überall, außer sie werden explizit außer Kraft gesetzt. Bei Laufzeitfragen ist wie üblich nur die asymptotische Laufzeit notwendig. Wenn Sie einen Algorithmus angeben sollen, so *können* Sie Pseudocode angeben — eine Beschreibung der Funktionsweise Ihres Algorithmus ist allerdings ausreichend. Algorithmen sind immer effizient zu konstruieren, d.h., mindestens polynomiell und i.d.r. schneller als eine naive Lösungsmethode. Algorithmen aus der Vorlesung können grundsätzlich als Blackbox verwendet werden.
- **Erklären Sie Ihre Lösungen, außer es wird explizit darauf hingewiesen, dass dies nicht nötig ist! Nur das Endresultat aufzuschreiben ist nicht ausreichend.**

Frage	1	2	3	4	5	6	7	**Total**
Punkte								
Maximum	20	12	10	12	14	18	34	120

Aufgabe 1: Kurze Fragen (20 Punkte)

Beantworten Sie die folgenden Fragen kurz.

(a) **(4 Punkte)** Sie haben $m = 2^k$ ($k \in \mathbb{N}$) aufsteigend sortierte Arrays $A_1, \dots, A_m$ mit jeweils n Elementen gegeben. Geben Sie einen effizienten Algorithmus an, um all diese Arrays zu einem einzelnen **sortierten** Array A der Größe mn zusammenzufassen. Was ist die Laufzeit Ihres Algorithmus in Abhängigkeit von m und n?

(b) **(6 Punkte)** Gegeben ist ein Array mit n **positiven** Integers. Geben Sie einen Divide & Conquer Algorithmus in **Pseudocode** an, um das kleinste gerade Element im Arrays zu finden, ohne das Array zu sortieren. Wenn es kein solches Element gibt, soll der Algorithmus -1 zurückgeben. Geben Sie auch die Rekursionsgleichung an; diese müssen Sie nicht lösen.

(c) **(3 Punkte)** Beweisen Sie oder geben Sie ein Gegenbeispiel an zu folgendem Satz: "Editierdistanz erfüllt die Dreiecksungleichung." D.h., für alle Strings x, y und z gilt $d(x,y) + d(y,z) \geq d(x,z)$, wenn d die Editierdistanz ist.

(d) **(3 Punkte)** Ein Graph $G = (V, E)$ mit n Knoten und $m = n^{4/3}$ Kanten ist aus Speicherplatzgründen als Adjazenzliste abgespeichert. Für ein beliebiges Knotenpaar (u, v) soll mit dem Befehl EXISTS($\{u, v\}$) in möglichst kurzer Zeit getestet werden können, ob $\{u, v\}$ in E enthalten ist. Wie erweitern Sie die Datenstruktur, um das möglichst effizient zu realisieren und den Speicherplatz dabei gering zu halten? Geben Sie den asymptotischen Platzbedarf der Datenstruktur und die asymptotische Laufzeit von EXISTS jeweils in Abhängigkeit von n an.

(e) **(4 Punkte)** Für ein großes Programmierprojekt müssen Sie wiederholt eine Vielzahl von Dateien kompilieren. Dabei treten Abhängigkeiten bezüglich der Kompilierreihenfolge auf, wie z.B. dass Datei A vor Datei B kompiliert werden muss (abgekürzt als $A \to B$). Die Abhängigkeitsrelationen sind als **gerichteter** und **kreisfreier** Graph $G = (V, E)$ gegeben (mit $n = |V|$ und $m = |E|$). Geben Sie einen Algorithmus an, welcher effizient das Problem löst, in welcher Reihenfolge die Dateien zu kompilieren sind. Was ist die asymptotische Laufzeit Ihres Algorithmus als Funktion von n und m?

Aufgabe 2: Landau Notation (12 Punkte)

Entscheiden Sie für jede der folgenden Aussagen, ob diese wahr oder falsch ist. Geben Sie im letzteren Fall eine Funktion f und/oder eine Funktion g an, welche die Ungültigkeit der Aussage zeigen. Eine weitere Begründung ist nicht notwendig.

1) $\forall f(n) \in \Omega(\log n),\ g(n) \in \mathcal{O}(n): \quad g(n) \in \mathcal{O}(f(n))$

2) $\forall f(n) \in \Omega(\log n),\ g(n) \in \mathcal{O}(n): \quad f(n) \in \mathcal{O}(g(n))$

3) $\forall f(n) \in \Omega(\log n),\ g(n) \in \mathcal{O}(n): \quad f(n) \in \Omega(\log(g(n)))$

4) $\forall f(n) \in \Omega(\log n),\ g(n) \in \mathcal{O}(n): \quad f(n) \in \Theta(\log(g(n)))$

5) Falls $f(n) \in \mathcal{O}(g^2(n))$, dann gilt $f(n) \in \Omega(g(n))$.

6) Falls $f(n) \in \Theta(2^n)$, dann gilt $f(n) \in \Theta(3^n)$.

7) Falls $f(n) = \mathcal{O}(n^3)$, dann gilt $\log f(n) \in \mathcal{O}(\log n)$.

Aufgabe 3: Mengen vereinigen (10 Punkte)

Gegeben sei ein Array A von Tupeln (a, b), wobei das Tupel (a, b) das Interval $[a, b] \subseteq \mathbb{R}$ repräsentiert . Ihre Aufgabe ist es, eine Prozedur SIMPLIFY zu schreiben, welches so ein Array nimmt und ein neues Array dieser Form produziert, welches die Vereinigung **aller** Intervalle in A darstellt und dabei eine **minimale** Anzahl an Tupeln verwendet. Angewandt auf sich zwei überschneidende Intervalle $[a_1, b_1]$ und $[a_2, b_2]$ wird das Tupel $(\min\{a_1, a_2\}, \max\{b_1, b_2\})$ zurückgegeben, gibt es keine Überschneidung, so ist keine Simplifizierung möglich und beide Tupel werden unverändert zurückgegeben.

Beispiel: SIMPLIFY angewandt auf $A = \langle(3,7),(1,4),(10,12),(6,8)\rangle$ gibt entweder das Array $\langle(10,12),(1,8)\rangle$ oder das Array $\langle(1,8),(10,12)\rangle$ zurück, da $[1,8] = [1,4] \cup [3,7] \cup [6,8]$; eine weitere Simplifizierung ist allerdings nicht möglich.

Geben Sie einen Algorithmus an, der dieses Problem möglichst effizient löst. Welche Laufzeit hat Ihr Algorithmus bei einem Array aus n Elementen?

Aufgabe 4: Algorithmus raten (12 Punkte)

Gegeben seien zwei Integer Arrays a und b der Länge n und m, und folgender Code:

```
// add(w) fuegt w am Ende der verketteten Liste an.

List<Integer> myst(int[] a, int[] b) {
  List<Integer> c = new LinkedList<Integer>();
  riddle(a, b, c);
  riddle(b, a, c);
  return c;
}

void riddle(int[] x, int[] y, List<Integer> z) {
  for (int i = 0; i < x.length; i++) {
    boolean take = true;
    for (int j = 0; j < y.length; j++)
      if (x[i] == y[j]) take = false;
    if (take == true) z.add(x[i]);
  }
}
```

(a) **(6 Punkte)** Erklären Sie was die Funktion MYST berechnet. Welche Komplexität hat diese Funktion in Abhängigkeit von n und m?

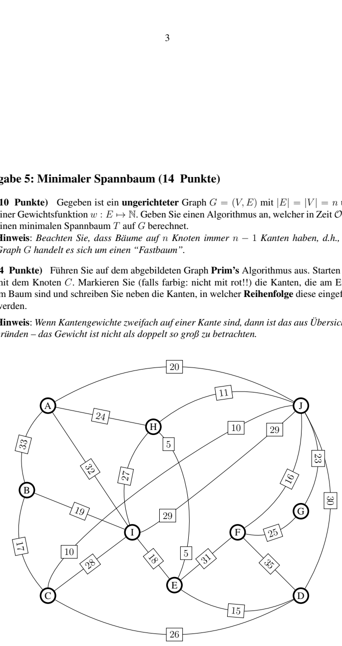
(b) **(6 Punkte)** Können Sie den gegebenen Algorithmus so modifizieren, dass die Funktion das gleiche Ergebnis berechnet, allerdings asymptotisch möglichst effizient ist (in n und m)? Beschreiben Sie Ihre Lösung – die Verwendung von Pseudocode oder Java ist nicht notwendig. Welche Zeitkomplexität hat Ihr Algorithmus?

Aufgabe 5: Minimaler Spannbaum (14 Punkte)

(a) **(10 Punkte)** Gegeben ist ein **ungerichteter** Graph $G = (V, E)$ mit $|E| = |V| = n$ und einer Gewichtsfunktion $w : E \mapsto \mathbb{N}$. Geben Sie einen Algorithmus an, welcher in Zeit $\mathcal{O}(n)$ einen minimalen Spannbaum T auf G berechnet.
Hinweis: *Beachten Sie, dass Bäume auf n Knoten immer $n - 1$ Kanten haben, d.h., bei Graph G handelt es sich um einen "Fastbaum".*

(b) **(4 Punkte)** Führen Sie auf dem abgebildeten Graph **Prim's** Algorithmus aus. Starten Sie mit dem Knoten C. Markieren Sie (falls farbig: nicht mit rot!!) die Kanten, die am Ende im Baum sind und schreiben Sie neben die Kanten, in welcher **Reihenfolge** diese eingefügt werden.

Hinweis: *Wenn Kantengewichte zweifach auf einer Kante sind, dann ist das aus Übersichtsgründen – das Gewicht ist nicht als doppelt so groß zu betrachten.*



Aufgabe 6: Ab ins Kino (18 Punkte)

Gegeben sei ein Straßennetz der großen Stadt Kinopolis, dargestellt als **gerichteter** Graph $G = (V, E)$ (mit $n = |V|$ und $m = |E|$) sowie einer Gewichtsfunktion $w : E \mapsto \mathbb{N}$, welche Reisezeiten mit dem **Auto** widerspiegelt. $K \subset V$ sind die Positionen der Kinos von Kinopolis.

(a) **(6 Punkte)** Zwei Freunde (wohnhaft an den Knoten v_1 und v_2) möchten zusammen ein Kino besuchen. Es gibt $k = |K|$ Kinos, und sie wollen dasjenige Kino in K wählen, welches für beide günstig zu erreichen ist, d.h., die **Summe** der Reisezeiten von beiden soll möglichst kurz sein. Beschreiben Sie, wie Sie dieses Problem lösen würden. Geben Sie die Laufzeit Ihres Algorithmus in Abhängigkeit von n, m und k an.

(b) **(6 Punkte)** Nehmen Sie nun an, dass $k = |K| = \log n$. Mit der Ankunft von "Star Wars Zero" in den Kinos beschließen sogar $l = \sqrt{n}$ Freunde (wohnhaft bei $v_1, v_2, \dots, v_l$), gemeinsam ein Kino zu besuchen und wieder soll das Kino gefunden werden, bei dem die Summe der Reisezeiten minimiert wird. Sie merken, dass der direkte Transfer Ihres vorherigen Algorithmus eine recht lange Laufzeit ergibt. Wie müssen Sie Ihren Algorithmus ändern, so dass Ihr Algorithmus schneller läuft? Was ist die Laufzeit?

Einige Kinos sind schwer mit dem Auto zu erreichen und es empfiehlt sich, das Auto ein Stück weit entfernt zu parken und den Rest zu Fuß zurückzulegen (es kann an jedem Knoten in V geparkt werden). Die Gewichtsfunktion $w' : E \mapsto \mathbb{N}$ spiegelt die Laufzeiten **für Fußgänger** wider.

(c) **(6 Punkte)** Eine einzelne Person will ins Kino. Wie berechnen Sie für diese Person den Parkplatz $p \in V$ und das Kino $x \in K$, so dass die Reisezeit minimiert wird? Was ist die Laufzeit Ihres Algorithmus?

Aufgabe 7: Binäre Suchbäume und Prioritätswarteschlangen (34 Punkte)

(a) **(6 Punkte)** Gegeben sei ein einfacher binärer Suchbaum T (die simple Variante aus der Vorlesung, d.h., ohne Rotationen) und zwei Elemente x und y, so dass $x \notin T$ und $y \in T$. Wenn auf diesem Baum `insert(x)` und direkt danach `remove(x)` ausgeführt werden - ist der resultierende Baum immer identisch zu T? Was, wenn `remove(y)` ausgeführt wird, und direkt danach `insert(y)`?
Argumentieren Sie kurz Ihre Antwort bzw. konstruieren Sie ein Beispiel, falls Ihre Antwort "nein" ist.

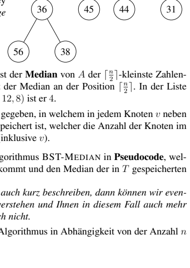
(b) **(6 Punkte)** $pre_1 = \{15, 9, 8, 5, 14, 11, 22, 25, 23, 28\}$ und $pre_2 = \{5, 3, 2, 4, 7, 8, 6\}$ seien zwei Schlüsselreihenfolgen, die angeblich als Ergebnis einer **Preorder** Traversierung eines binären Suchbaums erzeugt wurden. Zeichnen Sie die zugehörigen Suchbäume T_1 und T_2 sofern dies möglich ist; falls nicht, erklären Sie, warum nicht.

(c) **(6 Punkte)** Führen Sie **nacheinander** die nachfolgenden Operationen auf der abgebildeten (Min-Heap) Prioritätswarteschlange aus und zeichnen Sie den resultierenden Baum nach **jeder Gruppe** von Operationen.

1) *insert(42), insert(19), insert(12)*

2) *decreaseKey(v,15), deleteMin()*

Hinweis: decreaseKey(x, newkey) *setzt den Wert des Knotens x auf* newkey *falls* newkey *kleiner ist als der derzeitige Schlüssel von x.*



(d) In einer Auflistung A von n Zahlenwerten ist der **Median** von A der $\lceil \frac{n}{2} \rceil$-kleinste Zahlenwert, d.h., wenn man A sortiert, dann steht der Median an der Position $\lceil \frac{n}{2} \rceil$. In der Liste $(4, 1, 37, 2, 0)$ ist der Median 2, in $(1, 0, 9, 4, 12, 8)$ ist er 4.

Nehmen Sie an, ein binärer Suchbaum T ist gegeben, in welchem in jedem Knoten v neben dem Schlüssel **key** auch ein Wert **size** abgespeichert ist, welcher die Anzahl der Knoten im in v gewurzelten Unterbaum T_v wiedergibt (inklusive v).

- **(12 Punkte)** Schreiben Sie einen Algorithmus BST-MEDIAN in **Pseudocode**, welcher als Eingabe die Wurzel von T bekommt und den Median der in T gespeicherten Werte zurückgibt.
Hinweis: *Wenn Sie Ihren Algorithmus auch kurz beschreiben, dann können wir eventuelle Fehler in Ihrem Code leichter verstehen und Ihnen in diesem Fall auch mehr Punkte geben; notwendig ist dies jedoch nicht.*
- **(4 Punkte)** Welche Laufzeit hat Ihr Algorithmus in Abhängigkeit von der Anzahl n der Knoten und der Höhe h von T?

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

Informatik II: Algorithmen & Datenstrukturen

Donnerstag, 28. August, 2014, 14:00 – 17:00

Name: ....

Matrikelnummer: ....

Unterschrift: ....

Blättern Sie nicht um bevor Sie dazu aufgefordert werden!

- Schreiben Sie **auf alle Blätter** (inklusive Deckblatt und etwaiger zusätzlicher Blätter) Ihren **Vornamen, Nachnamen** und Ihre **Matrikelnummer**.
- **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. Schreiben Sie nicht in **rot** oder **grün** und benutzen Sie keinen Bleistift!
- Alle schriftlichen Hilfsmittel (Bücher, Vorlesungsunterlagen, handschriftliche Notizen, etc.) sind erlaubt. **Elektronische Hilfsmittel sind nicht erlaubt.**
- Die Klausur besteht aus 7 Aufgaben und 120 Punkten. Zum Bestehen reichen 50 Punkte.
- Benutzen Sie für jede Aufgabe eine eigene Seite.
- Es wird nur eine Lösung pro Aufgabe gewertet. Vergewissern Sie sich, dass Sie zusätzliche Lösungen selbst entwerten. Falls mehrere Lösungen zu einer Aufgabe existieren, so wird die schlechtere Lösung gewertet.
- Die folgenden Regeln gelten überall, außer sie werden explizit außer Kraft gesetzt. Bei Laufzeitfragen ist wie üblich nur die asymptotische Laufzeit notwendig. Wenn Sie einen Algorithmus angeben sollen, so können Sie Pseudocode angeben, eine Beschreibung der Funktionsweise Ihres Algorithmus ist allerdings ausreichend. Algorithmen sind immer effizient zu konstruieren, d.h., mindestens polynomiell und i.d.r. schneller als eine naive Lösungsmethode. Algorithmen aus der Vorlesung können grundsätzlich als Blackbox verwendet werden.
- **Erklären Sie Ihre Lösungen, außer es wird explizit darauf hingewiesen, dass dies nicht nötig ist! Nur das Endresultat aufzuschreiben ist nicht ausreichend.**

Frage	1	2	3	4	5	6	7	**Total**
Punkte								
Maximum	26	8	13	22	17	16	18	120

Aufgabe 1: Kurze Fragen (26 Punkte)

Beantworten Sie die folgenden Fragen kurz.

(a) **(2 Punkte)** Was ist die Best-Case Laufzeit von Selection Sort (in $\mathcal{O}$-Notation, ohne Begründung).

(b) **(2 Punkte)** Welches der in der Vorlesung besprochenen Sortierverfahren verhält sich am Besten, wenn die Eingabe ein bereits sortiertes Array ist? (ohne Begründung)

(c) **(3 Punkte)** Eine Anwendung verwendet eine Hashtabelle so, dass zur einmaligen Initialisierung alle (Schlüssel,Wert)-Paare eingefügt werden und danach nur noch (sehr viele) *find*-Operationen ausgeführt werden. Welche Hashtabellen-Implementierung aus der Vorlesung eignet sich besonders für diese Anwendung und wieso?

(d) **(3 Punkte)** Bestimmen Sie die Editierdistanz von `Tier` und `Tor` und geben Sie die notwendigen Operationen an. Begründen Sie auch, wieso es nicht besser geht?

(e) **(4 Punkte)** Geben Sie einen effizienten Algorithmus an, um einen Spannbaum mit maximalem Gesamtgewicht zu berechnen (kurze Beschreibung genügt).

(f) **(4 Punkte)** Ihnen ist von einem großen Netzwerk eine Liste der Knoten gegeben, zusammen mit den Knotengraden. Ihre Aufgabe ist es, die Knoten des Netzwerks nach ihren Knotengraden zu sortieren. Welches Sortierverfahren aus der Vorlesung eignet sich am besten dazu (mit Begründung)?

(g) **(4 Punkte)** Beschreiben Sie, wie eine FIFO-Warteschlange *Queue* mit Hilfe einer Prioritätswarteschlange (Heap) implementiert werden kann.

(h) **(4 Punkte)** Gegeben sei ein Baum T mit n Elementen, welcher für die gleichen Schlüssel sowohl ein binärer Suchbaum, als auch ein Min-Heap ist (alle Schlüssel seien verschieden). Welche Höhe kann T haben?

Hinweis: Überlegen Sie sich zuerst an einem kleinen Beispiel, was passiert.

Aufgabe 2: Landau-Notation (8 Punkte)

Beweisen oder widerlegen Sie die folgenden Behauptungen durch Benutzen der exakten Definitionen von $\mathcal{O}(\cdot)$ und $\Omega(\cdot)$.

(a) **(4 Punkte)** $\sqrt{n+7} \in \mathcal{O}(\sqrt{n})$.

(b) **(4 Punkte)** $\log(n^2) \in \Omega\big((\log n)^2\big)$.

Aufgabe 3: Mystische Multiplikationen (13 Punkte)

Gegeben sei ein Integer Array A der Länge n und folgender Code:

```
boolean myst(int[] A) {
   int n = A.length;
   for (int i = 0; i < n-1; i++) {
      for (int j = i+1; j < n; j++) {
         for (int k = 0; k < n; k++) {
            if (A[i] * A[j] == A[k]) {
               return true;
            }
         }
      }
   }
   return false;
}
```

(a) **(4 Punkte)** Was berechnet die Funktion *myst* (bzw., in welchen Fällen gibt *myst* true zurück)?

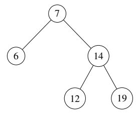
(b) **(3 Punkte)** Welche (asymptotische) Laufzeit hat *myst(int[] A)* im Worst Case und im Best Case (in Abhängigkeit von n, ohne Begründung)?

(c) **(6 Punkte)** Geben Sie, z.B. unter Zuhilfenahme einer geeigneten Datenstruktur, einen Algorithmus mit gleicher Ausgabe wie *myst* an, dessen *asymptotische* Laufzeit im Worst Case jedoch strikt besser als diejenige von *myst* ist. Was ist die Laufzeit Ihres Algorithmus?

Aufgabe 4: Binäre Suchbäume (22 Punkte)

(a) **(6 Punkte)** Führen Sie nacheinander die nachfolgenden 8 Einfüge- und Lösch-Operationen auf den unten angegebenen binären Suchbaum aus und zeichnen Sie den resultierenden Baum nach **jeder Gruppe** von Operationen.

1) *insert(23)*, *delete(19)*
2) *insert(13)*, *insert(10)*, *insert(4)*, *insert(18)*
3) *delete(6)*
4) *delete(14)*



(b) **(6 Punkte)** Können die folgenden Folgen in einer *Suchabfrage* in einem binären Suchbaum auftreten? Zeichnen Sie einen Beispielbaum auf, wenn die Antwort "ja" lautet, andernfalls begründen Sie Ihre Antwort.

- Folge 1: $44, 12, 15, 37, 19, 29, 28$
- Folge 2: $53, 64, 83, 75, 69, 60, 66$

(c) **(10 Punkte)** Man nennt einen binären Suchbaum T einen *AVL-Baum*, wenn für *jeden* Knoten u gilt, dass die Höhen der beiden in $u.\texttt{left}$ und $u.\texttt{right}$ gewurzelten Teilbäume sich um maximal 1 unterscheiden. Beschreiben Sie eine Methode *isAVL* als Pseudocode, welche einen binären Suchbaum T auf diese Eigenschaft testet. Was ist die Laufzeit Ihres Algorithmus, und warum?

Hinweis: Schreiben Sie zuerst eine rekursive Methode avlHeight(v)*, welche, angewandt auf einen Knoten v, die Höhe des in v gewurzelten Teilbaums wiedergibt* falls *der Teilbaum selbst ein AVL-Baum ist, und -1 sonst.*

Aufgabe 5: Divide & Conquer, Sortieren (17 Punkte)

(a) **(5 Punkte)** Gegeben sei ein Integer-Array A der Länge n. Finden Sie einen Divide & Conquer Algorithmus, der das Maximum der Elemente in A findet, ohne das Array zu sortieren. Geben Sie die Rekursionsgleichung für die Laufzeit an – diese muss *nicht* gelöst werden.

(b) **(12 Punkte)** Gegeben sei ein aufsteigend sortiertes Integer-Array B mit n Elementen. Nun werden k beliebige Elemente ($k \ll n$) individuell verringert (man weiß nicht, welche k Elemente verändert wurden), so dass das resultierende Array im Allgemeinen nicht mehr sortiert ist.

Geben Sie einen Algorithmus an, der das Array in Zeit $\mathcal{O}(n + k \log k)$ erneut in einen sortierten Zustand bringt.

Hinweis: Finden Sie zuerst alle Elemente, welche dadurch, dass sie heruntergesetzt wurden, die Sortierreihenfolge zerstören (wenn man diese Elemente entfernt, sind die restlichen Elemente richtig sortiert). Beachten Sie, dass das höchstens *k Elemente sind, aber nicht exakt k Elemente sein müssen.*

Aufgabe 6: Graphen & Graphtraversierung (16 Punkte)

Gegeben sei ein *gerichteter* Graph $G = (V, E)$. Herausgefunden werden soll, ob G einen Kreis enthält. Ein Kreis ist eine Knotenfolge $u_1, u_2, \ldots, u_k$, so dass $(u_i, u_{i+1}) \in E$ für alle $1 \leq i < k$, sowie $(u_k, u_1) \in E$. Es gelte $k \geq 2$, d.h., es kann Kreise der Länge 2 geben.

(a) **(4 Punkte)** Erläutern Sie einen Algorithmus, welcher möglichst effizient herausfindet, ob ein gegebener gerichteter Graph G einen Kreis enthält.

(b) **(2 Punkte)** Welche Datenstruktur würden Sie verwenden, um den Graph zu repräsentieren und warum: Adjazenzlisten oder Ajazenzmatrix?

(c) **(10 Punkte)** Geben Sie Pseudocode für Ihren Algorithmus an.

Aufgabe 7: Kürzeste Wege zur Arbeit (über den Berg) (18 Punkte)

Geben Sie jeweils einen Algorithmus an, um die folgenden Probleme zu lösen (dabei dürfen in der Vorlesung behandelte Algorithmen als Blackbox verwendet werden):

Gegeben sei ein Straßennetz als ein ungerichteter, gewichteter Graph $G = (V, E, w)$, $n = |V|$, $m = |E|$.

(a) **(2 Punkte)** Sie wohnen in Punkt $a \in V$ und arbeiten in Punkt $b \in V$. Wie berechnen Sie den kürzesten Weg zur Arbeit?

Geben Sie außerdem die asymptotische Laufzeit Ihres Algorithmus (ohne Begründung) an.

(b) **(4 Punkte)** Sie möchten umziehen in eine neue Wohnung $a' \in W$, wobei $W \subseteq V$ eine Menge von potentiellen Wohnungen ist. Wie finden Sie a', wenn diese von allen Wohnungen in W den kürzesten Weg zur Arbeit $b \in V$ haben soll?

Geben Sie außerdem die asymptotische Laufzeit Ihres Algorithmus an (ohne Begründung).

(c) **(12 Punkte)** Sie haben sich wegen der schönen Lage für eine andere Wohnung $c \in V$ mit Fahrradnähe zu $b \in V$ entschieden. Jeder Knoten v im Graphen hat eine Höhe $h_v \in \mathbb{R}$ (der Einfachheit halber seien alle Knotenhöhen voneinander verschieden) und je nachdem, in welcher Richtung man über eine Kante $e = \{u, v\}$ geht, ist diese eine Bergaufkante oder eine Bergabkante. Aus unbekanntem Grund wollen Sie von Ihrer Wohnung c aus zunächst ausschließlich bergauf fahren – bis zu einem Punkt $x \in V$ – und von dort aus ausschließlich bergab bis zum Punkt b.

Beschreiben Sie einen (effizienten) Algorithmus, welcher herausfindet, ob so ein Weg existiert und welcher bei Existenz den kürzesten aller solchen Wege, inklusive des höchsten Punkts x, zurückgibt. Welche Laufzeit hat Ihr Algorithmus (ohne Begründung)?

Hinweis: Verfahren aus der Vorlesung können als Blackbox verwendet werden. Beschreiben Sie Ihren Algorithmus nur, zu hohe Genauigkeit (z.B. via Pseudocode) kostet Sie zu viel Zeit.