Name:	Klausur-ID:
Vorname:	
Matrikelnummer:	

Lösungsvorschlag

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. D. Hofheinz, Jun.-Prof. H. Meyerhenke 16.02.2016

Nachklausur Algorithmen I

Aufgabe 1.	Kleinaufgaben	16 Punkte
Aufgabe 2.	Algorithmensimulation	10 Punkte
Aufgabe 3.	Kürzeste Wege	12 Punkte
Aufgabe 4.	Multiple Choice	7 Punkte
Aufgabe 5.	Pseudocodeanalyse	7 Punkte
Aufgabe 6.	Algorithmenentwurf	8 Punkte

Bitte beachten Sie:

- Merken Sie sich Ihre **Klausur-ID**.
- **Schreiben** Sie auf **alle Blätter** der Klausur und Zusatzblätter Ihre **Klausur-ID**.
- Die Klausur enthält 23 Blätter.
- Die durch Übungsblätter gewonnenen Bonuspunkte werden erst nach Erreichen der Bestehensgrenze hinzugezählt. Die Anzahl der Bonuspunkte entscheidet nicht über das Bestehen der Klausur.
- Die Bearbeitungszeit beträgt 120 Minuten.

Aufgabe		1	2	3	4	5	6	Summe
max. Punkte		16	10	12	7	7	8	60
Punkte								

Bonuspunkte:	Summe:	**Note:**

Lösungsvorschlag

Aufgabe 1. Kleinaufgaben [16 Punkte]

a. In der Vorlesung wurden binäre Heaps behandelt. Im Folgenden betrachten wir 3-äre Heaps. Hier hat jeder Knoten maximal drei anstatt maximal zwei Kinder.
Wie binäre Heaps werden auch 3-äre Heaps implizit repräsentiert, indem die Einträge ebenenweise in ein Array geschrieben werden. Sei ein solches Array $h[1..n]$ gegeben. Dann gibt die Funktion parent(i) den Elternknoten des Knotens i zurück.
Daraus ergibt sich folgende Invariante: $\forall i > 1 : h[parent(i)] \leq h[i]$.

1. Für folgenden 3-ären Heap ist die Invariante verletzt. Markieren Sie für alle Positionen i, ob die Invariante erfüllt (mit einem ✓) oder verletzt (mit einem **X**) ist.

Index	1	2	3	4	5	6	7	8	9	10	11	12
Eintrag	2	11	30	17	47	15	29	28	34	21	20	22

(1 Punkt)

Index	1	2	3	4	5	6	7	8	9	10	11	12
Eintrag	2	11	30	17	47	15	29	28	34	21	20	22
	✓	✓	✓	✓	✓	✓	✓	X	✓	X	✓	✓

2. Geben Sie die Funktion parent(i) im Pseudocode an! Der Elternknoten der Wurzel soll dabei die Wurzel selbst sein. (3 = 1+2 Punkte)

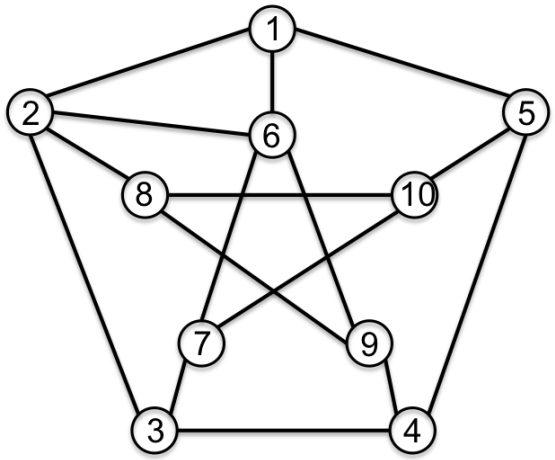
```
function parent(i)
if (i = 1)
    then return 1
else
    return ⌊(i+1)/3⌋
```

[4 Punkte]

b. Eine Knotenfärbung $C : V \mapsto \mathbb{N} \setminus \{0, \infty\}$ bildet jeden Knoten eines Graphen $G = (V, E)$ auf eine Farbe (repräsentiert durch eine natürliche Zahl) ab. Eine solche Knotenfärbung ist *zulässig*, wenn es keine Kante $e \in E$ gibt, deren Endknoten beide die gleiche Farbe haben.
Der unten stehende Algorithmus findet eine zulässige Knotenfärbung und versucht dabei, mit möglichst wenig Farben auszukommen.

Initialisiere $C(v) \leftarrow \infty$ für alle $v \in V$
for vertex $v \leftarrow 1$ **to** n do
    $C(v) \leftarrow$ kleinste Farbe, die noch nicht in der Nachbarschaft von v verwendet wurde
return C

1. Wie färbt der obige Algorithmus den folgenden Graphen? Schreiben Sie Ihre Lösung in die vorbereitete Tabelle!



Knoten	1	2	3	4	5	6	7	8	9	10
Farbe										

(2 Punkte)

Knoten	1	2	3	4	5	6	7	8	9	10
Farbe	1	2	1	2	3	3	2	1	4	4

2. Wie viele Farben braucht der Algorithmus höchstens bei einem Graphen mit n Knoten, m Kanten und höchstem Knotengrad d?
Kreuzen Sie **genau eine** Lösung an!

2	
d	
$d+1$	
n	
$\lceil \frac{1}{2} \cdot \mathrm{m} \rceil$	

(2 Punkte)

$d+1$

[4 Punkte]

c. Der Rang einer Zahl x in einer Liste von paarweise verschiedenen Zahlen sei die Position von x, falls die Liste aufsteigend sortiert wäre. Die erste Position in der Liste sei hierbei 1. In der Liste 3, 7, 1, 8, 4, 5 hätte die Zahl 5 also den Rang 4. Gegeben sei folgender Algorithmus *BogoRank*, um den Rang einer Zahl zu bestimmen.

```
function BogoRank(list A, number x)        // erstes Element in A ist A[1]
n ← |A|
sortiere Liste A mit InsertionSort
if n = 1 then
    if A[1] < x then return 1
             else return 0
//Teile Eingabeliste in zwei Hälften:
left ← A[1, ⌊n/2⌋]        // benötigt Θ(n) Schritte
right ← A[⌊n/2⌋ + 1, n]   // benötigt Θ(n) Schritte
return BogoRank(left, x) + BogoRank(right, x)
```

1. Stellen Sie eine Rekurrenzgleichung für die asymptotische Worst-Case-Laufzeit von *BogoRank* auf.

   (2 Punkte)
   $T(n) = \mathcal{O}(n^2) + 2 \cdot T(n/2)$

2. Angenommen, der Sortierschritt in Zeile 2 entfällt. Geben sie die Laufzeit des veränderten Algorithmus mit dem Master-Theorem an. Ist diese Laufzeit optimal für das Berechnen des Ranges? Begründen Sie kurz.

   (3 Punkte)
   Die Rekurrenzgleichung ist $T(n) = \Theta(n) + 2 \cdot T(n/2)$. Das folgende Schema war in der Vorlesung:

   $$T(n) = \begin{cases} a & \text{falls n = 1} \\ cn + dT(n/b) & \text{sonst} \end{cases}$$

   Da $d = b$, gilt $T(n) = \Theta(n \log n)$. Dies ist nicht optimal, da der Rang in linearer Zeit ermittelt werden kann: Man muss nur einmal über die Liste iterieren und zählen, wie viele Zahlen kleiner sind als x.

[5 Punkte]

d. Beweisen oder widerlegen Sie die folgenden Aussagen. Geben Sie bei jeder Aussage explizit an, ob sie wahr oder falsch ist.

1. Die Rekurrenz $T(n) = \begin{cases} 5 & \text{falls n = 1} \\ 3n + 3 \cdot T(n/2) & \text{sonst} \end{cases}$
   liegt in $\mathcal{O}(n \log n)$.

   Die Behauptung ist falsch. Mit dem Master-Theorem folgt, dass $T(n)$ in $\Theta(n^{\log_2 3})$ liegt. Die Funktion $n^{\log_2 3}$ wächst schneller als $n \log n$.

2. $1024^{\log_2 n} \in \mathcal{O}(n^{1024})$

   Die Behauptung stimmt. $1024^{\log_2 n} = 2^{10 \cdot \log_2 n} = n^{10} \in \mathcal{O}(n^{1024})$.

3. $n! \in \mathcal{O}(n^n)$

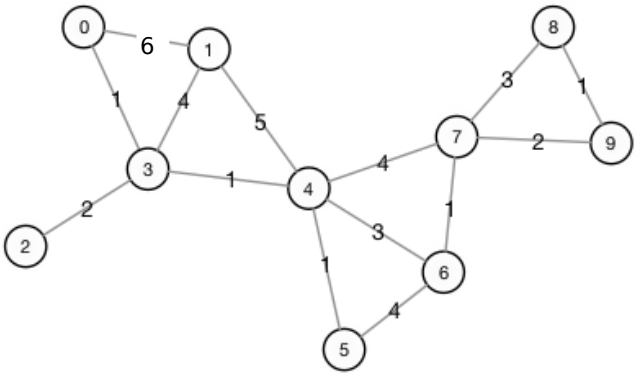
   Die Behauptung stimmt, da $n! \leq n^n$.
   Dies gilt, da $n! = \prod_{i=1}^{n} i \leq \prod_{i=1}^{n} n = n^n$.

[3 Punkte]

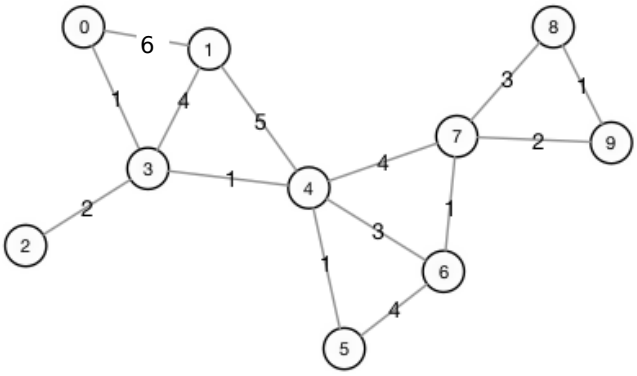
Lösungsvorschlag

Aufgabe 2. Algorithmensimulation [10 Punkte]

a. In dieser Aufgabe geht es darum, die Ausführung des aus der Vorlesung bekannten Jarnik-Prim-Algorithmus für minimale Spannbäume zu simulieren.
Gegeben ist der folgende Eingabegraph:



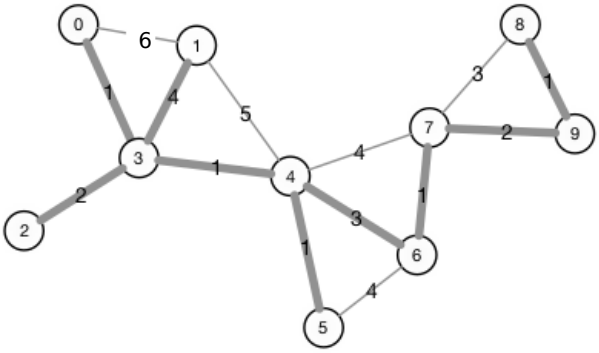
Beginnen Sie bei der Ausführung mit dem Knoten 0 und berechnen Sie den minimalen Spannbaum. Geben Sie die Liste der Kanten in der Reihenfolge an, in der sie der Algorithmus einfügt. Notieren Sie eine Kante in der Form (u, v), wobei u der bereits im Baum enthaltene Knoten ist (falls schon einer der Knoten im Baum enthalten ist).
(Es folgt eine Kopie des Graphen für Notizen – diese wird in keinem Fall bewertet.)



Tragen Sie Ihre Lösung in das folgende Schema ein:

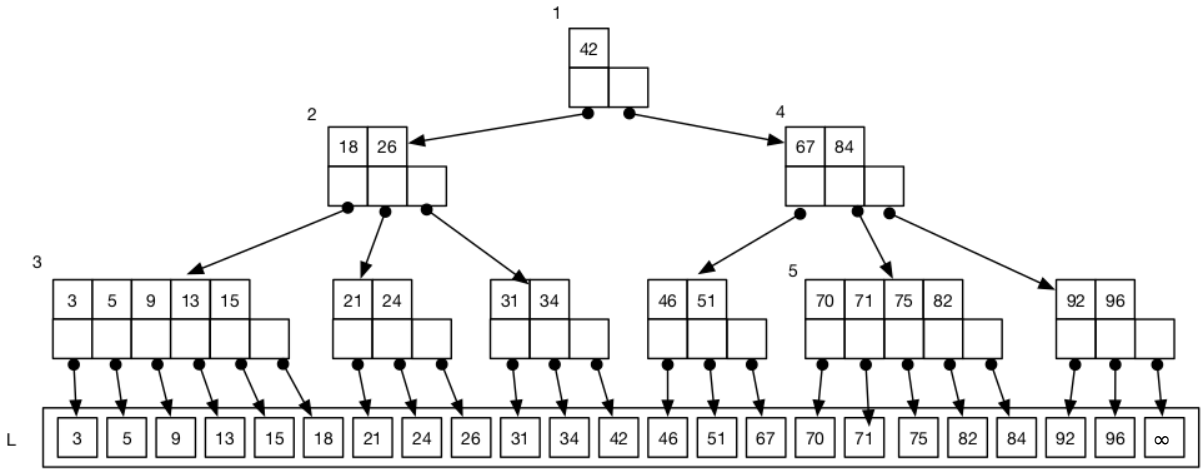
[(___, ___), (___, ___),(___, ___),(___, ___),(___, ___),(___, ___),(___, ___),(___, ___),(___, ___)]

Liste der Kanten: [(0,3), (3,4), (4,5), (3,2), (4,6), (6,7), (7,9), (9,8), (3,1)]



[4 Punkte]

b. Gegeben sei folgender (2,6)-Baum:



Dieser Baum ermöglicht die Suche in einer sortierten Folge, die zusätzlich als doppelt verkettete Liste *L* gespeichert ist.
Führen Sie nacheinander folgende Operationen auf dem Baum aus:

1. insert(29)
2. insert(16)

Tragen Sie dazu jeweils in die unten angegebene Schablone ein, wie sich der Inhalt der Knoten 1, 2 und 3 verändert. (Beachten Sie: auch andere Knoten können sich verändern, dies muss aber nicht eingetragen werden.) Sollte es nötig sein, einen der Knoten zu spalten, tragen Sie den Inhalt des linken (bzw. des ersten) Knotens nach der Spaltung ein.

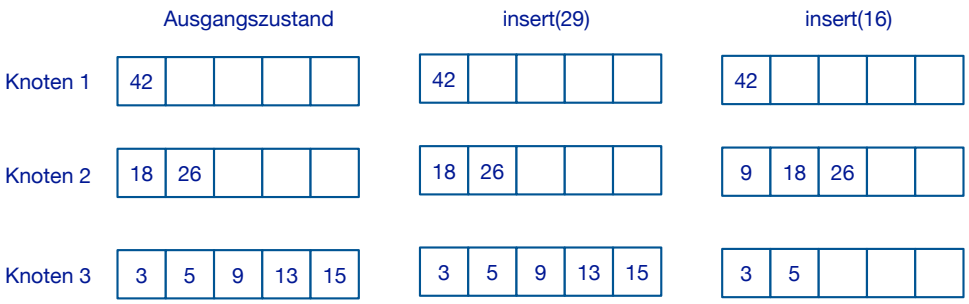
Lösung bitte hier eintragen:

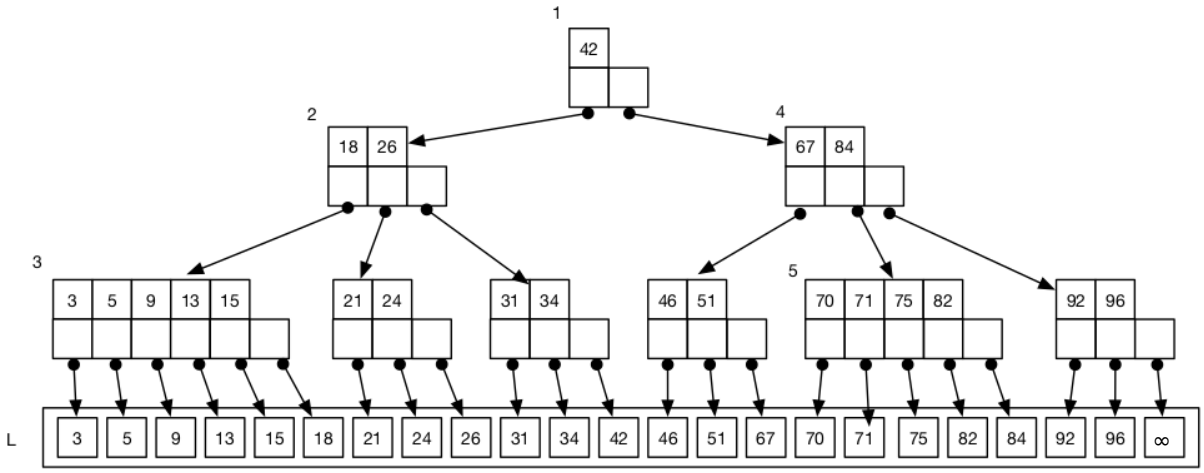
	Ausgangszustand	insert(29)	insert(16)
Knoten 1	42		
Knoten 2	18 26		
Knoten 3	3 5 9 13 15		

Kopie für Notizen: (wird in keinem Fall bewertet)

	Ausgangszustand	insert(29)	insert(16)
Knoten 1	42		
Knoten 2	18 26		
Knoten 3	3 5 9 13 15		

Lösung:


Ausgangszustand
insert(29)
insert(16)
Knoten 1
42
42
42
Knoten 2
18
26
18
26
9
18
26
Knoten 3
3
5
9
13
15
3
5
9
13
15
3
5

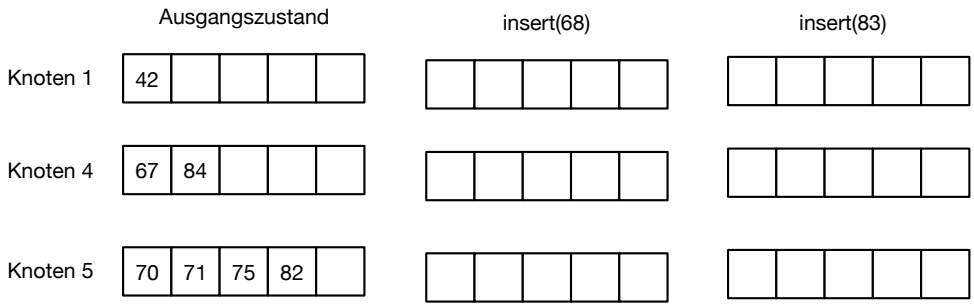


Beginnen Sie wieder beim Ausgangszustand des Baumes (oben dargestellt) und führen Sie darauf nun nacheinander folgende Operationen durch:

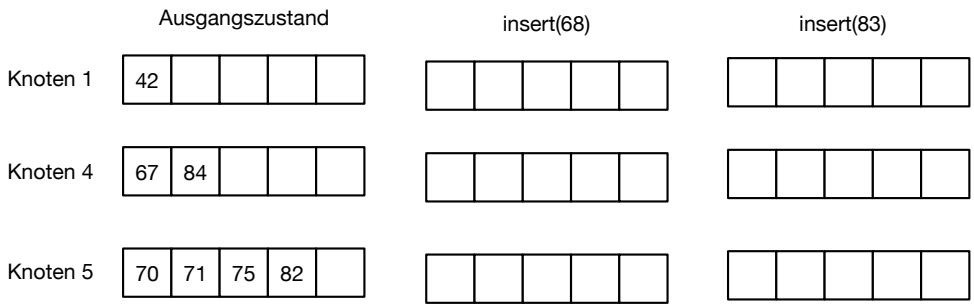
1. insert(68)
2. insert(83)

Tragen Sie in die Schablone ein, wie sich der Inhalt der Knoten 1, 4 und 5 verändert. (Beachten Sie: auch andere Knoten können sich verändern, dies muss aber nicht eingetragen werden.) Sollte es nötig sein, einen der Knoten zu spalten, tragen Sie den Inhalt des linken (bzw. des ersten) Knotens nach der Spaltung ein.

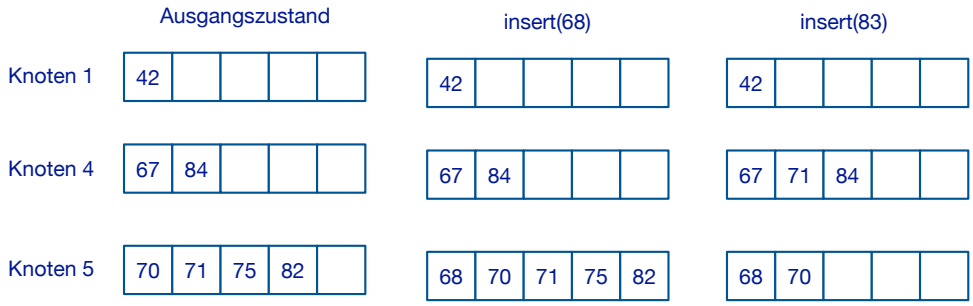
Lösung bitte hier eintragen:



Kopie für Notizen: (wird in keinem Fall bewertet)



Lösung:


Ausgangszustand
insert(68)
insert(83)
Knoten 1
42
42
42
Knoten 4
67
84
67
84
67
71
84
Knoten 5
70
71
75
82
68
70
71
75
82
68
70

[6 Punkte]

Lösungsvorschlag

Aufgabe 3. Kürzeste Wege [12 Punkte]

In der Vorlesung haben wir das Problem der kürzesten Wege von einem Quellknoten aus kennengelernt (SSSP = single source shortest paths).
Betrachten wir nun APSP (all-pairs shortest paths) für einen gewichteten und gerichteten Graphen $G = (V, E, \omega)$ mit n Knoten: Für jedes geordnete Knoten**paar** $(u, v) \in V \times V$ berechne den kürzesten Weg von u nach v!
Der unten angegebene Algorithmus FWD löst das Problem zum Teil. Er berechnet nur die paarweisen *Distanzen*, aber nicht die zugehörigen Wege. Für jedes Paar (u, v) wird die Distanz von u nach v in der Matrix D an der Position $D_{u,v}$ gespeichert.

Hinweis: Gehen Sie in dieser Aufgabe immer davon aus, dass G nichtnegative Kantengewichte hat, stark zusammenhängend ist und keine Schleifen (d. h. Kanten der Form (v, v)) hat!

FWD-Algorithmus für gewichteten Graphen $G = (V, E, \omega)$

```
// init D
for u ← 1 to n do
    for v ← 1 to n do
        if (u,v) ∈ E then
            D_{u,v} ← ω(u,v)
        else
            D_{u,v} ← ∞
// compute distances
for k ← 1 to n do
    for u ← 1 to n do
        for v ← 1 to n do
            D_{u,v} ← min(D_{u,v}, D_{u,k} + D_{k,v})
return D
```

a. Welche Laufzeit hat der Algorithmus FWD? Begründen Sie Ihre Antwort kurz. [1 Punkt]
$O(n^3)$, weil drei im Hauptteil ineinander verschachtelte Schleifen jeweils $O(n)$ mal aufgerufen werden und die Hauptoperation in der innersten Schleife konstante Laufzeit hat. Die Initialisierung ist quadratisch und daher asymptotisch schneller.

b. Betrachten Sie nun den Algorithmus APDD, der das gleiche Problem wie FWD löst (PQ steht in der Beschreibung für Prioritätswarteschlange bzw. Priority Queue):

APDD für gewichteten Graphen $G = (V, E, \omega)$
// init D, wie bei FWD-Algorithmus
for $u \leftarrow 1$ **to** n do
    for $v \leftarrow 1$ **to** n do
        if $(u, v) \in E$ then
            $D_{u,v} \leftarrow \omega(u, v)$
        else
            $D_{u,v} \leftarrow \infty$
// Beginn eigentlicher Algorithmus
foreach $v \in V$ do
    Rufe Dijkstras Algorithmus (PQ als binärer Heap implementiert) mit v als Startknoten auf;
    Aktualisiere $D_{v,\cdot}$, also alle Distanzen von v ausgehend, anhand des Dijkstra-Resultats
return D

Kreuzen Sie in der unten stehenden Tabelle an, für welche jeweilige Eingabe welcher Algorithmus **asymptotisch schneller** ist!

Hinweis: Falsche Kreuze führen zu Abzügen, die sich allerdings nur innerhalb dieser Aufgabe auswirken!

Eingabetypus	**FWD schneller**	**APDD schneller**	**beide gleich schnell**
1) Dünner Graph ($m = O(n)$)			
2) Dichter Graph ($m = \Omega(n^2)$)			
3) Nahezu-dünner Graph ($m = \Theta(n \log n)$)			

Geben Sie hier unten zur **Begründung** Ihrer Antworten die notwendige(n) Laufzeitrechnung(en) an! [3 Punkte]

Eingabetypus	**FWD schneller**	**APDD schneller**	**beide gleich schnell**
1) Dünner Graph ($m = O(n)$)		X	
2) Dichter Graph ($m = \Omega(n^2)$)	X		
3) Nahezu-dünner Graph ($m = \Theta(n \log n)$)		X	

- 1x Dijkstra wie angegeben: $O((n+m)\log n)$
- APDD: $O(n(n+m)\log n)$; 1) $O(n^2 \log n)$, 2) $O(n^3 \log n)$, 3) $O(n^2 \log^2 n)$
- FWD: $O(n^3)$

c. Welche Zeile in der obigen Tabelle müsste anders ausgefüllt werden, wenn die Prioritätswarteschlange des Dijkstra-Algorithmus in APDD mit einem Fibonacci-Heap implementiert wäre? Begründen Sie!

- 1x Dijkstra wie angegeben: $O(m + n \log n)$
- APDD: $O(nm + n^2 \log n)$; 1) $O(n^2 \log n)$, 2) $O(n^3)$, 3) $O(n^2 \log n)$
- FWD: $O(n^3)$

Also: In Zeile 2 (dichter Graph) hätten wir nun ein Unentschieden und demnach ein Kreuz bei „beide gleich schnell“. [1 Punkt]

d. Erweitern Sie den FWD-Algorithmus nun derart, dass er nicht nur die Distanzen, sondern auch kürzeste Wege (pro Knotenpaar reicht **ein** kürzester Weg!) berechnet und zurückgibt. Repräsentieren Sie dabei einen kürzesten Weg, indem Sie in einer Vorgängermatrix P im Eintrag $P_{u,v}$ den Vorgängerknoten von v auf dem kürzesten Weg von u nach v speichern! Vervollständigen Sie dazu den Pseudocode unten!

FWDP-Algorithmus für gewichteten Graphen $G = (V, E, \omega)$

for $u \leftarrow 1$ **to** n do

    for $v \leftarrow 1$ **to** n do

        if $(u, v) \in E$ then

            $D_{u,v} \leftarrow \omega(u, v)$

---

        else

            $D_{u,v} \leftarrow \infty$

---

for $k \leftarrow 1$ **to** n do

    for $u \leftarrow 1$ **to** n do

        for $v \leftarrow 1$ **to** n do

---

return pair of D and P

[3 Punkte]

Lösung

FWDP-Algorithmus für gewichteten Graphen $G = (V, E, \omega)$

for $u \leftarrow 1$ **to** n do
    for $v \leftarrow 1$ **to** n do
        if $(u, v) \in E$ then
            $D_{u,v} \leftarrow \omega(u, v)$
            $P_{u,v} \leftarrow u$ (0,5 Punkte)
        else
            $D_{u,v} \leftarrow \infty$
            $P_{u,v} \leftarrow nil$ (0,5 Punkte)
for $k \leftarrow 1$ **to** n do
    for $u \leftarrow 1$ **to** n do
        for $v \leftarrow 1$ **to** n do
            if $(D_{u,k} + D_{k,v} < D_{u,v})$ then (0,5 Punkte)
                $D_{u,v} \leftarrow D_{u,k} + D_{k,v}$ (0,5 Punkte)
                $P_{u,v} \leftarrow P_{k,v}$ (1 Punkt)
return pair of D and P

Lösungsende

e. Geben Sie nun einen Algorithmus `getPath`(P, u, v) **in Pseudocode** an, der mit Hilfe der Vorgängermatrix P die Knoten des kürzesten Weges von u nach v in richtiger Reihenfolge ausgibt! Verwenden Sie Datenstrukturen und deren Basisoperationen aus der Vorlesung! Zur Ausgabe eines Knotens v können Sie `output`(v) verwenden!

Lösung

```
procedure getPath(P, u, v)
    S ← empty stack
    current ← v
    while current ≠ u do
        S.push(current)
        current ← P_{u,current}
    S.push(u)
    while not S.empty() do
        current ← S.pop()
        output(current)
```

Alternativ auch okay:

- Statt mit Stack mit Liste arbeiten und dann nach Listen-Operation reverse die Ausgabe machen.
- Den Rekursionsstack der Laufzeitumgebung verwenden.

Lösungsende

[4 Punkte]

Lösungsvorschlag

Aufgabe 4. Multiple Choice [7 Punkte]

Markieren Sie für jede der folgenden Aussagen, ob sie zutreffend ist. Korrekt markierte Aussagen geben entweder $0,5$ Punkte oder 1 Punkt (wie angegeben), inkorrekt markierte Aussagen geben $0,5$ bzw. 1 Punkt Abzug. Aussagen, die nicht markiert sind, ergeben weder Punkte noch Abzug. Eine negative Gesamtpunktzahl in dieser Aufgabe wird als 0 Punkte gewertet.

Aussage	**trifft zu**	**trifft nicht zu**
Folgen und Felder (Arrays) sind zwei konkrete Implementierungen der abstrakten Datenstruktur "Liste". *(0,5 Punkte)*		
Jeder Algorithmus mit erwarteter Laufzeit $O(f(n))$ hat eine Worst-Case-Laufzeit von $O(f(n)^2)$. *(0,5 Punkte)*		
Ein binärer Heap kann genutzt werden, um eine Prioritätswarteschlange zu implementieren. *(0,5 Punkte)*		
Der Jarnik-Prim-Algorithmus findet einen minimalen Spannbaum in einem Graphen mit ganzzahligen Kantengewichten in Zeit $O(m + n\log(n))$, wenn er mit Fibonacci-Heaps implementiert ist. (m =Anzahl Kanten, n =Anzahl Knoten.) *(0,5 Punkte)*		
Der Dijkstra-Algorithmus ist ein Beispiel für einen Greedy-Algorithmus. *(0,5 Punkte)*		
Mit der Karatsuba-Ofman-Multiplikation können zwei Langzahlen mit jeweils n Ziffern mit $O(n^{\log_3(2)})$ Ziffernoperationen multipliziert werden. *(0,5 Punkte)*		
Nehmen Sie an, eine Hashtabelle wird mit einer vollkommen zufälligen Hashfunktion (d.h. einer Hashfunktion, die für jede Eingabe eine unabhängig gleichverteilte Ausgabe gibt) implementiert. Dann reicht eine Hashtabelle mit n Einträgen aus, um n Elemente mit Wahrscheinlichkeit von mindestens 50% kollisionsfrei abzulegen. *(1 Punkt)*		
Tiefensuche kann benutzt werden, um einen ungerichteten und zusammenhängenden Graph auf Kreisfreiheit zu testen. *(1 Punkt)*		
Mit Branch-and-Bound kann das ganzzahlige Rucksackproblem in Worst-Case-Zeit $O(n)$ gelöst werden, wobei n die Anzahl der gegebenen Gegenstände ist. *(1 Punkt)*		
Das Garbage-In-Garbage-Out-Prinzip erlaubt das Finden eines minimalen Schnitts in einem gerichteten Graphen mittels Rekursion. *(1 Punkt)*		

Lösung

n, n, j, j, j, n, n, j, n, n

Lösungsende

Lösungsvorschlag

Aufgabe 5. Pseudocodeanalyse [7 Punkte]

Betrachten Sie den folgenden Pseudocode und beantworten Sie dann die nachfolgenden Teilaufgaben bezüglich seiner Funktion.

```
Procedure whoopsie(list: doubly linked list of non-negative integers):
    if list.length=1
        return list.first
    for el1 in list do      // iteriere über Liste, vom ersten bis zum letzten Element
        for el2 in list do      // iteriere über Liste, vom ersten bis zum letzten Element
            if el1 < el2 and el1 + el2 ≥ 0
                list.removeElement(el1)
                return whoopsie(list)
    return list.first
```

a. Welche Ausgabe liefert der Aufruf whoopsie([4,53,1,7,3,34,78,42])? [2 Punkte]

Lösung

78

Lösungsende

b. Geben Sie die asymptotische Worst-Case-Laufzeit von whoopsie an, in Abhängigkeit der Länge n der eingegebenen Liste. Geben Sie eine Folge von Eingaben $(\text{list}_n)_{n\in\mathbb{N}}$ an, so dass whoopsie für diese Eingaben asymptotisch die Worst-Case-Laufzeit benötigt. [3 Punkte]

Lösung

Worst-Case-Laufzeit $\Theta(n^2)$.
Eingaben mit Worst-Case-Laufzeit (z.B.): Maximum am Anfang. (Spezialfall davon: alle Elemente in Liste gleich, also z.B. $l_n := \overbrace{(c,\ldots,c)}^{n \text{ mal}}$ für ein $c \in \mathbb{N}$.)

Lösungsende

c. Was berechnet whoopsie? Kreuzen Sie **genau eine Antwort** an.

- [ ] Das Minimum der Elemente aus list
- [ ] Das Maximum der Elemente aus list
- [ ] Den Durchschnittswert der Elemente aus list [2 Punkte]
- [ ] Den Median von list
- [ ] Das erste Primzahl in list

Lösung

Das Maximum der Elemente aus list.

Lösungsende

Lösungsvorschlag

Aufgabe 6. Algorithmenentwurf [8 Punkte]

Sie sollen die Auftragsverwaltung des aufstrebenden Versandunternehmens „SendMeStuff“ entwerfen. Sie möchten hierzu ein System implementieren, welches Aufträge der Form $A = (\mathit{Produkt}, \mathit{Menge})$ verwaltet. Das System soll folgende Operationen unterstützen:

- `auftrag`(A) fügt einen Auftrag A zum System hinzu.
- `abarbeiten` liefert den *ältesten* im System befindlichen Auftrag A zurück und entfernt ihn aus dem System.
- `menge`$(\mathit{Produkt})$ liefert die Gesamtmenge von *Produkt* aus allen Aufträgen im System zurück. (D.h. `menge`$(\mathit{Produkt})$ liefert die Summe aller Menge_i für derzeit im System befindliche Aufträge $A_i = (\mathit{Produkt}_i, \mathit{Menge}_i)$ mit $\mathit{Produkt}_i = \mathit{Produkt}$ zurück.)

(Zweck der Operation `menge` ist es, Nachfragespitzen zu erfassen, um dann geeignet produzieren bzw. Ware bereitstellen zu können.)

a. Skizzieren Sie eine (möglicherweise aus mehreren elementaren Datenstrukturen zusammengesetzte) Datenstruktur, die die obigen Operationen unterstützt. Der Zeitbedarf pro Operation soll konstant sein, wobei

- keine a-priori-Schranke für die Anzahl der Aufträge bekannt ist,
- die Anzahl N_P der verschiedenen Produkte konstant und bekannt ist, und Produkte als Ganzzahlen $\mathit{Produkt} \in \{0, \ldots, N_P - 1\}$ kodiert sind.

Beschreiben Sie, welche in der Vorlesung behandelten Datenstruktur(en) Sie benutzen. Beschreiben Sie in Pseudocode jede der Operationen `auftrag`(A), `abarbeiten`, `menge`$(\mathit{Produkt})$.
[5 Punkte]

Mehrere Lösungen möglich. Beispiel: verwende Queue (d.h. FIFO) Q für Verwaltung der Aufträge, und Array M (indiziert über Produkte) für Verwaltung der jeweiligen Gesamtmenge von Produkten aus Aufträgen. Genauer: anfangs sei Q leer und $M = (M[0], \ldots, M[N_P - 1])$ ein Array der Größe N_P. Betrachte folgenden Pseudocode:

function `auftrag`(A) // füge Auftrag zu System hinzu
parse $A = (Produkt, Menge)$
Q.enqueue(A)
$M[Produkt] = M[Produkt] + Menge$

function `abarbeiten` // liefert ältesten Auftrag zurück und entfernt ihn
if Q empty then return $\perp$
A=Q.dequeue
parse $A = (Produkt, Menge)$
$M[Produkt] = M[Produkt] - Menge$
return A

function `menge`(***Produkt***) // liefert Gesamtmenge von *Produkt* aus allen Aufträgen
return $M[Produkt]$

Die Laufzeit von allen Operationen ist konstant, weil enqueue und dequeue, sowie Array-Abfragen nur konstante Laufzeit haben.

b. Nehmen Sie nun an, dass die Vereinfachung aus Aufgabenteil **a** wegfällt. Genauer:

- Die Anzahl N_P der Produkte sei immer noch konstant und bekannt. Allerdings seien die Produkte nun durch Zahlen nicht a priori beschränkter Größe $Produkt \in \mathbb{N}$ kodiert. (Nehmen Sie an, dass diese Zahlen jeweils in eine Speicherzelle passen.)

Skizzieren Sie, wie Ihre Datenstruktur verändert werden muss, um in diesem allgemeineren Szenario mit *erwartet* konstantem Zeitbedarf pro Operation zu funktionieren. Beschreiben Sie die Operationen `auftrag`(A), `abarbeiten`, `menge`$(Produkt)$ wieder in Pseudocode. [3 Punkte]

Wieder mehrere Lösungen möglich. Beispiel: verwende wie in Aufgabenteil **a** eine Queue Q für die Verwaltung der Aufträge, aber nun eine Hashtabelle T der Größe $O(N_P)$ für die Verwaltung der Produktmengen. Genauer: anfangs seien Q und T leer. Betrachte folgenden Pseudocode:

function `auftrag`(A) // füge Auftrag zu System hinzu
parse $A = (Produkt, Menge)$
Q.enqueue(A)
$T[Produkt] = T[Produkt] + Menge$

function `abarbeiten` // liefert ältesten Auftrag zurück und entfernt ihn
if Q empty then return $\perp$
A=Q.dequeue
parse $A = (Produkt, Menge)$
$T[Produkt] = T[Produkt] - Menge$
return A

function `menge`$(Produkt)$ // liefert Gesamtmenge von $Produkt$ aus allen Aufträgen
return $T[Produkt]$

Die Laufzeit von allen Operationen ist erwartet konstant, weil enqueue und dequeue konstante Laufzeit, und sowie Hashtabellen-Abfragen und -Updates erwartet konstante Laufzeit haben.