Karlsruher Institut für Technologie

Algorithmen 1 – Sommersemester 2019
Institut für Theoretische Informatik
http://verialg.iti.kit.edu
Prof. Dr. Carsten Sinz – Markus Iser

Name, Vorname:

Matrikelnummer:

Allgemeine Hinweise

- Als Hilfsmittel ist nur *ein* DIN-A4-Blatt mit Ihren *handschriftlichen* Notizen zugelassen.
- Schreiben Sie auf *alle* Blätter Ihren Namen und Ihre Matrikelnummer.
- Die durch die Übungsblätter gewonnenen Bonuspunkte werden erst nach Erreichen der zum Bestehen der Klausur nötigen Punktzahl hinzugezählt. Die Anzahl der Bonuspunkte entscheidet nicht über das Bestehen.
- **Die Klausur nur mit Erlaubnis umdrehen!**

Übersicht Punkteverteilung

Die Klausur besteht aus drei Teilen, in denen Sie jeweils 20 Punkte erreichen können.

Aufgaben	Teil A					Teil B						Teil C		
	1	2	3	4	5	1	2	3	4	5	6	1	2	3
60 Punkte	20					20						20		
	5	8	4	1	2	4	2	4	3	4	3	6	6	8

A Gemischte Kleinaufgaben (20 Punkte)

A.1 Begriffe (5 Punkte)

A.1.1 Binäre Min-Heaps (1 Punkt)

Wie lautet die Heap-Eigenschaft für einen Min-Heap?

A.1.2 Hashfunktionen (1 Punkt)

Sei $\mathcal{H}$ eine nicht-leere Menge von Funktionen $h : U \to \{1, \ldots, m\}$. Was muss gelten, damit $\mathcal{H}$ eine universelle Familie von Hash-Funktionen ist?

A.1.3 Graphen (1 Punkt)

Welche beiden Eigenschaften machen einen beliebigen Graphen zu einem DAG?

A.1.4 Minimale Spannbäume (2 Punkte)

Definieren Sie die Kreis- und die Schnitt-Eigenschaft, welche zur Berechnung von minimalen Spannbäumen verwendet werden.

A.2 Asymptotische Laufzeit (8 Punkte)

A.2.1 O-Kalkül (2 Punkte)

Beweisen oder widerlegen Sie: $(\log_5 n)^n \in \mathcal{O}\big(n^{\log_3 n}\big)$.

A.2.2 Rekurrenzgleichung (2 Punkte)

Gegeben sei die folgende Rekurrenzgleichung:

$$T(n) = \begin{cases} 1 & \text{für } n = 1 \\ 4 \cdot T(\frac{n}{2}) + n^2 & \text{für } n > 1 \end{cases}$$

Zeigen Sie für $n = 2^k$ mit $k \in \mathbb{N}_0$ mithilfe vollständiger Induktion, dass

$$T(n) = n^2 \cdot \big(\log_2(n) + 1\big)$$

A.2.3 Mastertheorem (4 Punkte)

Gegeben seien zwei rekursive Algorithmen A und B, wobei B in jedem Rekursionsschritt nicht nur sich selbst sondern auch A aufruft. Die Laufzeiten von A und B seien durch die Rekurrenzen T_A und T_B beschrieben mit $n = 5^k$ und $k \in \mathbb{N}$.

Geben Sie alle ganzzahligen Lösungen für $a, b > 0$ an, für die laut Mastertheorem aus der Vorlesung die Laufzeit von B in $\Theta(n \log n)$ liegt, und begründen Sie Ihre Wahl von a und b kurz.

$$T_A(n) = \begin{cases} 1024 & \text{für } n = 1 \\ 5n + a \cdot T_A(\frac{n}{5}) & \text{sonst} \end{cases}$$

$$T_B(n) = \begin{cases} 256 & \text{für } n = 1 \\ T_A(n) + b \cdot T_B(\frac{n}{5}) & \text{sonst} \end{cases}$$

A.3 Union-Find (4 Punkte)

Welche beiden Beschleunigungstechniken wurden in der Vorlesung für die Union-Find-Datenstruktur eingeführt? Nennen und beschreiben Sie diese kurz.

A.4 Inversionen (1 Punkt)

Bestimmen Sie die Anzahl der Inversionen in der folgenden Liste,[1] und nennen Sie ein Beispiel für einen Verwendungszweck dieses Maßes.

$$(87, 2, 3, 45, 8, 23, 43, 65)$$

A.5 Wächter (2 Punkte)

Erklären Sie, welchen Zweck Sentinel-Elemente bei Suchalgorithmen erfüllen. Haben Sentinels einen Einfluss auf die *asymptotische* Laufzeit eines Algorithmus? Begründen Sie kurz.

[1] Eine Inversion ist ein Paar (i, j) mit $i < j$ und $a[i] > a[j]$

B Algorithmen Ausführen (20 Punkte)

B.1 Dynamische Programmierung (4 Punkte)

Die Editierdistanz (auch Levenshtein-Distanz genannt) zwischen zwei Zeichenketten ist die minimale Anzahl von Einfüge-, Lösch- und Ersetz-Operationen von Zeichen, die benötigt wird, um eine Zeichenkette in eine andere umzuwandeln. Die Editierdistanz zwischen zwei Worten u und v kann mittels dynamischer Programmierung berechnet werden. Wir verwenden eine Matrix D für Teillösungen, wobei $D_{i,j}$ die Levenshtein-Distanz der Präfixe $u_{0...i}$ und $v_{0...j}$ angibt. Die erste Zeile ($i = 0$) und Spalte ($j = 0$) von D sollen wie folgt initialisiert werden:[2]

$$\begin{aligned} D_{i,0} &= i, \text{für } 0 \leq i \leq |u| \\ D_{0,j} &= j, \text{für } 1 \leq j \leq |v| \end{aligned} \qquad \text{(Initialisierungsschritte)}$$

Die Rekurrenzschritte, die beim Erstellen der Matrix verwendet werden, sind für $1 \leq i \leq |u|, 1 \leq j \leq |v|$ wie folgt gegeben:

$$D_{i,j} = \min \begin{cases} 0 + D_{i-1,j-1} & \text{falls } u_i = v_j \\ 1 + D_{i-1,j-1} & \text{Ersetzung} \\ 1 + D_{i,j-1} & \text{Einfügung} \\ 1 + D_{i-1,j} & \text{Löschung} \end{cases} \qquad \text{(Rekurrenzschritte)}$$

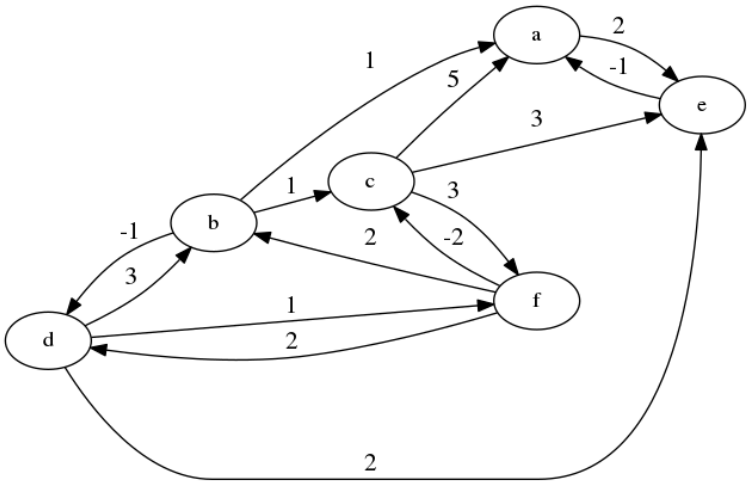
Berechnen Sie die Editierdistanz der beiden Worte “Hausboot” und “ausbooten”, indem Sie rechts die bereits initialisierte Matrix D ausfüllen. Markieren Sie anschließend den Pfad in der Matrix, an dem Sie eine minimale Folge von Editierschritten ablesen können.

	ϵ	a	u	s	b	o	o	t	e	n
ϵ	0	1	2	3	4	5	6	7	8	9
H	1									
a	2									
u	3									
s	4									
b	5									
o	6									
o	7									
t	8									

[2] Hierbei bezeichne $|w|$ die Länge der Zeichenkette w.

B.2 Kürzeste-Wege-Algorithmen (2 Punkte)

Berechnen Sie in dem gegebenen Graphen alle kürzesten Wege, die bei Knoten c beginnen, und tragen Sie die kürzesten Distanzen zu c in die dafür vorgesehene Tabelle ein. Benutzen Sie dabei einen der beiden in der Vorlesung vorgestellten Algorithmen.



Knoten	Distanz von c
a	
b	
d	
e	
f	

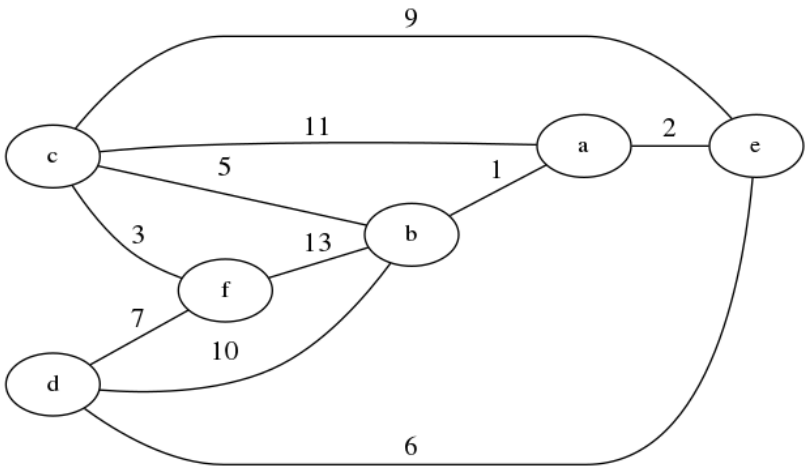
B.3 Sortieren (4 Punkte)

Sortieren Sie die Folge $(17, 9, 3, 76, 15, 23, 107, 90)$ aufsteigend mit Mergesort. Geben Sie dabei die einzelnen merge-Schritte in ihrer *Ausführungsreihenfolge* an, und notieren Sie dabei für jedes $\mathsf{merge}(e_1, e_2) \to a$ in der Tabelle unten sowohl die Eingabetupel e_1 und e_2, als auch das Ausgabetupel a.

	e_1	e_2	a
1.			
2.			
3.			
4.			
5.			
6.			
7.			

B.4 Minimale Spannbäume (3 Punkte)

Berechnen Sie einen minimalen Spannbaum (MST) für den unten gegebenen Graphen mit dem Algorithmus von Jarnik und Prim. Verwenden Sie Knoten c als Startknoten. Weisen Sie dabei in der Tabelle rechts jeder Baumkante eine Nummer zu, die angibt beim wievielten Schnitt sie dem MST hinzugefügt wurde.



$\{a, e\}$	
$\{a, c\}$	
$\{a, b\}$	
$\{b, c\}$	
$\{b, d\}$	
$\{b, f\}$	
$\{c, e\}$	
$\{c, f\}$	
$\{d, e\}$	
$\{d, f\}$	

B.5 Min-Heaps (4 Punkte)

Gegeben ist ein Array A mit Werten wie in der ersten Zeile der unten stehenden Tabelle. Verwenden Sie *buildHeap* aus der Vorlesung, um einen binären Heap mit den Zahlen nachfolgender Tabelle aufzubauen. Führen Sie anschließend einmal *deleteMin* und danach *insert 4* aus. Notieren Sie den Zustand des Heaps nach jeder Ausführung.

A	10	24	13	39	40	45	36	17	25	89
buildHeap()										
deleteMin()										
insert(4)										

B.6 Hashing mit Linear Probing (3 Punkte)

Folgende Hashtabelle sei mit der Hashfunktion $f(x) = x \bmod 13$ erstellt worden. Führen Sie die unten angegebenen Operationen aus und geben Sie jeweils den Zustand der Hashtabelle nach Ausführen der Operation an.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Wert	13	1	15	0	3	30	19	14	20	9	7		
delete(3)													
delete(0)													
insert(6)													

C Algorithmenentwurf (20 Punkte)

C.1 Anagramme (6 Punkte)

Gegeben seien ein Wort W und ein String S über einem fixen Alphabet Σ der konstanten Größe k. Finden Sie alle Indizes i, an denen der Teilstring $S[i : i+|W|]$ ein Anagramm von W ist.[3] Als Anagramm wird eine Buchstabenfolge bezeichnet, die aus einer anderen Buchstabenfolge allein durch Umstellung (Permutation) der Buchstaben gebildet ist. Sie dürfen annehmen, dass jedes Zeichen z des Alphabets mit einer ganzen Zahl $0 \leq z < |\Sigma|$ identifiziert werden kann.

Beispiel: Gegeben seien das Wort W=“ab” und der String S=“xbyaba”, dann lautet die Anagramm-Indexfolge A von W in S wie folgt:

$$A = \{4, 5\} \ .$$

Geben Sie einen Algorithmus an, der das Anagramm-Index-Problem mit einer Worst-Case-Laufzeit von $\mathcal{O}(|S|)$ löst und begründen Sie die Laufzeit. Für einen korrekten Algorithmus mit schlechterer Lauftzeit aber vorhandener Laufzeitbegründung erhalten Sie noch maximal 3 Punkte.

[3] Anmerkungen zur Notation: Sei S ein String, dann bezeichnet S[i] den Buchstaben an Stelle i und S[i:j] denjenigen zusammenhängenden Teilstring in S der mit S[i] beginnt und mit S[j-1] endet. Das erste Zeichen im String S ist S[1].

C.2 Die fehlende Zahl (6 Punkte)

Gegeben sei ein Array $A[1..n]$ von n beliebigen ganzen Zahlen. Gesucht ist die kleinste Zahl $z > 0$, die nicht in A vorkommt. Zahlen können auch mehrfach in A vorkommen, d.h. es kann Duplikate geben. Der hier zu entwickelnde Algorithmus soll das Problem *in $\mathcal{O}(n)$ und mit nur konstantem zusätzlichem Speicherplatz* lösen. Das Array A darf verändert werden. Zeigen Sie für eine Lösung, dass Ihr Algorithmus diese Bedingungen einhält.

C.2.1 Eingeschränkter Fall (4 Punkte)

Lösen Sie die Aufgabe unter der Annahme, dass in A nur ganze Zahlen z zwischen 1 und n vorkommen (d.h. $1 \leq z \leq n$).

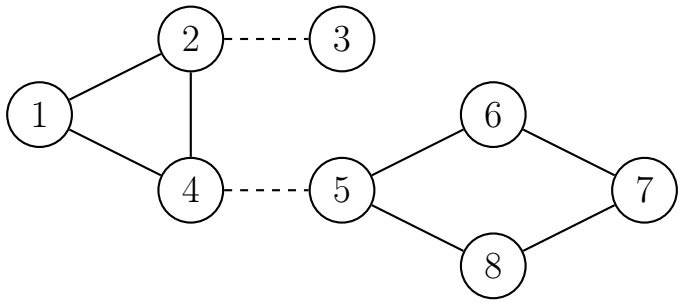
C.2.2 Allgemeiner Fall (2 Punkte)

Lösen Sie die Aufgabe für den allgemeinen Fall, dass also in A beliebige ganze Zahlen auftreten dürfen.

C.3 Brücken in Graphen (8 Punkte)

Sei $G = (V, E)$ ein einfacher, zusammenhängender Graph, d.h., G sei ungerichtet und enthalte keine Mehrfachkanten oder Schleifen (Kanten, die einen Knoten mit sich selbst verbinden). Eine *Brücke* $b \in E$ ist eine Kante, deren Löschen die Anzahl der Zusammenhangskomponenten in G erhöht.

Beispiel: Im unten angegebenen Graph sind die gestrichelten Kanten Brücken.



a) **(1 Punkt)** Zeigen Sie: Eine Kante $e \in E$ ist eine Brücke genau dann, wenn die Kante e nicht auf einem einfachen Zyklus von G liegt. (Ein Weg $(v_0, \ldots, v_k)$ ist ein einfacher Zyklus, wenn $k > 2$, $v_0 = v_k$ und $v_1, \ldots, v_k$ paarweise verschieden sind.)

Brücken können über eine Modifikation der Tiefensuche (DFS) für ungerichtete Graphen berechnet werden. Sei T_G der DFS-Baum zu einer Tiefensuche von G und $d[v]$ die zugehörige DFS-Nummerierung der Knoten $v \in V$. Wir definieren $low[v]$ *als den kleinsten Wert* $d[w]$*, ermittelt über alle Rückwärtskanten* (u, w) *der Tiefensuche, für die* u *im DFS-Baum von* v *aus erreichbar ist und für die* $d[w] < d[v]$ *gilt.* Falls keine solche Rückwärtskante existiert, sei $low[v] = d[v]$. $low[v]$ kann wie folgt berechnet werden:

$$low[v] = \min \begin{cases} d[v] & \\ d[w] & \text{für alle } w, \text{ für die } (v, w) \text{ eine Rückwärtskante ist} \\ low[w] & \text{für alle } w, \text{ für die } (v, w) \text{ eine Baumkante ist} \end{cases}$$

b) **(1 Punkt)** Geben Sie für den oben angeführten Beispielgraph die Werte $low[v]$ für alle $v \in V$ an. Die Nummern in den Knoten sollen dabei als Werte der DFS-Nummerierung interpretiert werden. Sie können Ihre Lösung direkt im Graph oben eintragen.

c) **(2 Punkte)** Begründen Sie, warum *nur Kanten des DFS-Baums* Brücken in G sein können. Geben Sie außerdem eine Bedingung (unter Verwendung der Arrays d und low) dafür an, dass eine Kante (u, v) des DFS-Baums Brücke von G ist.

d) **(4 Punkte)** Geben Sie einen Algorithmus an, der für einen einfachen, zusammenhängenden Graph G sämtliche Brücken berechnet. Verwenden Sie dazu das unten angegebene Tiefensuchschema und ergänzen Sie Definitionen für die Funktionen `init`, `root`(s), `traverseNonTreeEdge`(u, v, w), `traverseTreeEdge`(v, w) und `backtrack`(u, v). Sie können hierbei annehmen, dass eine ungerichtete Kante des Graphen G durch zwei gerichtete Kanten dargestellt ist.

Tiefensuchschema:

```
unmark all nodes
init
foreach s ∈ V do
   if s is not marked then
      mark s
      root(s)
      DFS(s, s)
```

```
Procedure DFS(u, v: NodeId)
   foreach (v, w) ∈ E do
      if w is marked then
         traverseNonTreeEdge(u, v, w)
      else
         traverseTreeEdge(v, w)
         mark w
         DFS(v, w)
   backtrack(u, v)
```

init:

root(s):

traverseNonTreeEdge(u, v, w):

traverseTreeEdge(v, w):

backtrack(u, v):