Karlsruher Institut für Technologie

Algorithmen 1 – Sommersemester 2018
Zuverlässige Softwaresysteme im Kontext der Automobilindustrie
`http://verialg.iti.kit.edu`

Prof. Dr. Carsten Sinz – Markus Iser

Klausur **04.09.2018**

Name, Vorname:

Matrikelnummer:

Allgemeine Hinweise

- Als Hilfsmittel ist nur *eine* DIN-A4-Seite 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.

Tabelle 1: Punkteverteilung

	Teil A							Teil B						Teil C		
16 Aufgaben	1	2	3	4	5	6	7	1	2	3	4	5	6	1	2	3
60 Punkte	20							20						20		
	5	2	2	3	3	1	4	4	3	3	3	3	4	7	6	7

A Gemischte Kleinaufgaben (20 Punkte)

A.1 Union-Find Datenstruktur (5 Punkte)

A.1.1 Oberflächlich betrachtet (1 Punkt)

Was verwaltet die Union-Find Datenstruktur? Nennen Sie einen Algorithmus, in dem diese Datenstruktur eingesetzt wird.

A.1.2 Naive Implementierung (2 Punkte)

Erklären Sie die beiden Operationen `union` und `find` und die Grundidee hinter deren naiven Implementierung. Wie sind die worst-case Laufzeiten der beiden Operationen (im $\mathcal{O}$-Kalkül)?

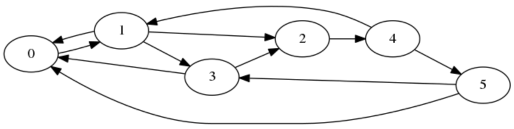
A.1.3 Optimierte Implementierung (2 Punkte)

Was ist Union-Find mit *einfacher Pfadkompression* und welches Problem löst man mit der Kompression?

Geben Sie die amortisierte Laufzeit der elementaren Operationen von Union-Find mit einfacher Pfadkompression im $\mathcal{O}$-Kalkül an.

A.2 Adjazenzfelddarstellung (2 Punkte)

Geben Sie den folgenden Graphen als Adjazenzfeld an.[1]



A.3 O-Kalkül (2 Punkte)

Zeigen oder widerlegen Sie, dass $n^{n+3} \in \mathcal{O}(n^n)$.

A.4 Master-Theorem (3 Punkte)

Lösen Sie folgende Rekurrenzen im Θ-Kalkül mit $n = 7^k$ und $k \in \mathbb{N} \setminus \{0\}$.

$$X(n) = X\left(\frac{n}{7}\right) + 2018 \cdot n \qquad X(1) = 42 \tag{1}$$

$$Y(n) = 8 \cdot Y\left(\frac{n}{7}\right) + \frac{n}{2018} \qquad Y(1) = 6 \tag{2}$$

$$Z(n) = 7 \cdot Z\left(\frac{n}{7}\right) + X(n) \qquad Z(1) = 1 \tag{3}$$

[1] Das vorgegebene Raster ist als Hilfestellung gedacht, es geht *nicht* darum, unbedingt in jedes Feld etwas reinzuschreiben.

A.5 Sortieren (3 Punkte)

A.5.1 Ordnen (1 Punkt)

Ordnen Sie die folgenden Algorithmen *absteigend*[2] nach ihrer worst-case Laufzeitkomplexität: Quicksort, Bucketsort, Mergesort.

A.5.2 Untere Schranke (1 Punkte)

Wie lautet die untere Schranke für die worst-case Laufzeitkomplexität bei vergleichsbasiertem Sortieren? Gilt diese untere Schranke auch für Sortieralgorithmen für ganze Zahlen? Begründen Sie Ihre Antwort.

A.5.3 Stabiles Sortieren (1 Punkt)

Was macht ein *stabiles* Sortierverfahren aus? Nennen Sie ein Beispiel für ein stabiles Sortierverfahren.

A.6 Dijkstras Algorithmus (1 Punkt)

In welchem Fall ist Dijkstras Algorithmus auf Graphen nicht anwendbar und warum?

[2]Das heißt, die höchste Laufzeitkomplexität kommt in Ihrer Ordnung zuerst.

A.7 Hashing (4 Punkte)

A.7.1 Perfekte Hashfunktion (1 Punkt)

Unter welcher Bedingung erhält eine Hashfunktion das Attribut *perfekt*?

A.7.2 Kollisionen (2 Punkte)

Nennen und erklären Sie kurz zwei Möglichkeiten, um mit Kollisionen beim Hashing umzugehen.

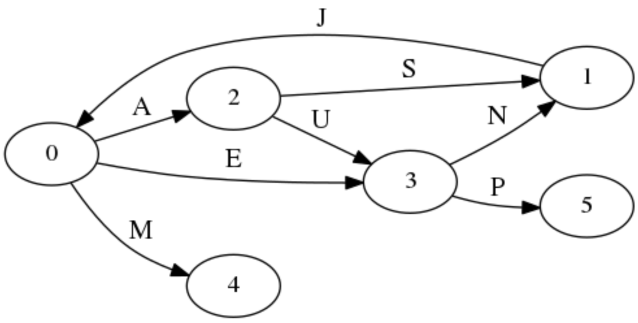
A.7.3 Universelle Hashfunktionen (1 Punkt)

Welche Eigenschaften muss eine Familie X von Hashfunktionen haben, damit X *universell* ist?

B Algorithmen Ausführen (20 Punkte)

B.1 Breitensuche (4 Punkte)

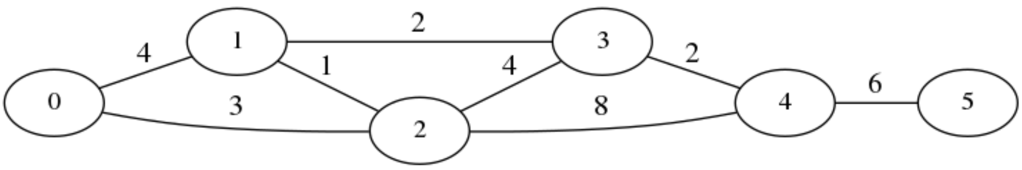
Führen Sie auf folgendem Graphen beginnend mit Knoten 0 eine Breitensuche aus. Halten Sie sich, wenn Sie die Wahl haben, in jeder Ebene an die von den Knotennamen induzierte Reihenfolge. Geben sie die Kanten in derjenigen Reihenfolge an, in der Sie sie bei der Breitensuche besucht haben, und benennen Sie, ob es sich um eine Tree-, Cross- oder Backward-Kante handelt.[3] Warum gibt es bei der Breitensuche keine Forward-Kanten?



0			5		
1			6		
2			7		
3			8		
4			9		

B.2 Minimium Spanning Tree (3 Punkte)

Konstruieren Sie mit dem Algorithmus von Jarník und Prim für den unten dargestellten Graphen einen minimalen Spannbaum. Starten Sie bei Knoten 0. Zeichnen Sie den so berechneten minimalen Spannbaum noch einmal deutlich erkennbar darunter.



[3] Das vorgegebene Raster ist als Hilfestellung gedacht, es geht *nicht* darum, unbedingt in jedes Feld etwas reinzuschreiben.

B.3 Dynamische Programmierung (3 Punkte)

Beim *Longest Common Subsequence Problem (LCS)* geht es darum, für zwei Zeichenketten S und T deren längste gemeinsame Teilfolge L zu berechnen. D.h. es ist die maximale Zeichenkette L gesucht, die sowohl in S als auch in T auftritt. L muss in S und T nicht zusammenhängend sein, aber die Reihenfolge der Zeichen muss beibehalten werden.

Beispiel: Für $S = \texttt{abazdc}$ und $T = \texttt{bacbad}$ ist $L = \texttt{abad}$ die gesuchte Lösung.

Das LCS-Problem lässt sich mittels dynamischer Programmierung lösen. Dabei werden Teilprobleme für Präfixe $S[1..i]$, $i \leq |S|$ und $T[1..j]$, $j \leq |T|$ von S und T betrachtet. Wenn man die *Länge des LCS* der Präfixe $S[1..i]$ und $T[1..j]$ mit $\mathrm{LCS}(i, j)$ bezeichnet, so ergibt sich $\mathrm{LCS}(i, j)$ aus kleineren Teilproblemen wie folgt:

$$\mathrm{LCS}(i,j) = \begin{cases} \max(\mathrm{LCS}(i-1,j), \mathrm{LCS}(i,j-1)) & \text{falls } S[i] \neq T[j], \\ 1 + \mathrm{LCS}(i-1,j-1) & \text{falls } S[i] = T[j]. \end{cases}$$

Dabei bezeichnet $S[i]$ das i-te Zeichen in Zeichenkette S. Wir nehmen an, dass $\mathrm{LCS}[0,0] = \mathrm{LCS}[i,0] = \mathrm{LCS}[0,j] = 0$ ist.

B.3.1 Ausführen (2 Punkte)

Berechnen Sie die Werte $\mathrm{LCS}(i, j)$ für $1 \leq i, j \leq 6$ für $S = \texttt{abazdc}$ und $T = \texttt{bacbad}$ und tragen Sie diese in die folgende Tabelle ein:

T

S

	b	a	c	b	a	d
a						
b						
a						
z						
d						
c						

B.3.2 Bestimmung der längsten gemeinsamen Teilsequenz (1 Punkt)

Beschreiben Sie, wie sich anhand der Werte LCS(i, j) die längste gemeinsame Teilsequenz, d.h. die eigentliche Lösung des LCS-Problems, berechnen lässt.

B.4 Hashing mit Linear Probing (3 Punkte)

Gegeben sei die Hashfunktion $h(v) = v \bmod 13$. Sei T eine Hashtabelle, die unter Verwendung von Funktion $h(v)$ und mit *linear probing* ganzzahlige Werte in einem Feld der Länge 13 speichert. Führen Sie die folgende Menge von Befehlen auf T aus, und geben Sie den Zustand der Hashtabelle (des Feldes) nach jedem Schritt an.

	0	1	2	3	4	5	6	7	8	9	10	11	12	
`T.insert 24`														
`T.insert 13`														
`T.insert 19`														
`T.insert 26`														
`T.insert 1`														
`T.insert 2`														
`T.remove 1`														
`T.insert 12`														
`T.remove 13`														
`T.remove 24`														

B.5 Gnome Sort (3 Punkte)

Algorithmus 1 : Gnome Sort

Input : Array a

Output : Array a (sorted in ascending order)

```
pos ← 0
while pos < length(a) do
    if pos = 0 or a[pos] ≥ a[pos −1] then
        pos ← pos +1
    else
        swap a[pos] and a[pos −1]
        pos ← pos −1
```

B.5.1 Ausführen (1,5 Punkte)

Führen Sie Algorithmus 1 für $a = [7, 3, 5, 9, 8]$ aus und geben Sie a nach jeder swap-Operation an.

B.5.2 Laufzeitverhalten (1,5 Punkte)

Geben Sie die worst-case Laufzeit von Algorithmus 1 im Θ-Kalkül an und begründen Sie ihre Antwort.

B.6 Binäre Heaps (4 Punkte)

Gegeben Sei ein Feld mit den Zahlen $7, 8, 13, 20, 6, 19, 35, 12$.

Erstellen Sie mit den oben genannten Zahlen zunächst einen binären Heap.[4] Führen Sie dann zweimal deleteMin aus. Dann fügen Sie erst 17 und dann 4 ein.

Tragen Sie nach jedem Schritt den Zustand des Feldes in die Tabelle ein.

	7	8	13	20	6	19	35	12
`makeheap`								
`deleteMin`								
`deleteMin`								
`insert 17`								
`insert 4`								

[4] Zur Klarstellung: Es geht in dieser Aufgabe um binäre min-Heaps, die in einem Array gespeichert werden.

C Algorithmenentwurf (20 Punkte)

C.1 Konvexe Hülle (7 Punkte)

Zur Ermittlung der konvexen Hülle von n Punkten im $\mathbb{R}^2$ betrachten wir einen Algorithmus (in der Literatur als `Graham-Scan` Algorithmus bekannt), der gegeben eine Punktmenge P eine *minimale* Liste von Punkten $K \subseteq P$ berechnet, welche die *konvexe Hülle* von P aufspannt. Zur Vereinfachung gehen wir davon aus, dass für alle Punkte $(x, y) \in P$ gilt, dass $x \geq 0$ und $y \geq 0$.[5]

Initialisierung

Die Laufzeit des `Graham-Scan` Algorithmus wird durch das Sortieren bestimmt. In einem Initialisierungsschritt werden die Punkte $p \in P$ aufsteigend nach dem Winkel $\angle\vec{p}\vec{x}$ zwischen Vektor $\vec{p}$ und Einheitsvektor $\vec{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ sortiert.[6]

Build Convex Hull

Anschließend wird im Schritt `buildConvexHull` die Liste $K \subseteq P$ von Punkten erzeugt, welche die konvexe Hülle von P aufspannen. Implementieren Sie nur den Schritt `buildConvexHull` (in Pseudocode), der ein nach den oben genannten Kriterien sortiertes Array P von n Punkten engegennimmt und in $\mathcal{O}(n)$ Rechenschritten die konvexe Hülle K erzeugt.

Wählen Sie geschickt eine geeignete Datenstruktur für K. Verwenden Sie Funktion d aus Gleichung 1, um die relative Position eines Punktes zu einer durch zwei Punkte aufgespannten Geraden zu bestimmen.

Zeigen Sie, dass Ihre Implementierung die Laufzeit $\mathcal{O}(n)$ nicht überschreitet.

$$d(A,B,C) = \begin{vmatrix} 1 & x_A & y_A \\ 1 & x_B & y_B \\ 1 & x_C & y_C \end{vmatrix} = (x_B - x_A)(y_C - y_A) - (x_C - x_A)(y_B - y_A)$$

$$\begin{aligned} d(A,B,C) < 0 &\iff C \text{ liegt rechts von } \overrightarrow{AB} \\ d(A,B,C) > 0 &\iff C \text{ liegt links von } \overrightarrow{AB} \\ d(A,B,C) = 0 &\iff C \text{ liegt auf } \overrightarrow{AB} \end{aligned}$$

Gleichung 1: Relative Position eines Punktes C bezüglich eines Vektors $\vec{AB}$

[5] Punkte $\begin{pmatrix} x \\ y \end{pmatrix}$ werden in P als Tupel (x, y) dargestellt. Tun Sie das auch in Ihrem Pseudocode.

[6] Bei gleichem Winkel werden die Punkte p aufsteigend nach der Länge $|\vec{p}|$ sortiert.

C.2 Stabiles Partitionieren (6 Punkte)

Beim Partitionieren eines Feldes A von Elementen nach einem gegebenen Prädikat P werden alle Elemente $e \in A$, welche die Eigenschaft $P(e)$ haben, an den Anfang der Liste verschoben. Das Ergebnisfeld A' ist also eine Permutation von A, für welches es einen maximalen Index k gibt, sodass folgendes gilt:

$$P(A'[i]) = \mathsf{true}, \text{ falls } i \leq k$$
$$P(A'[i]) = \mathsf{false}, \text{ falls } i > k.$$

Bei *stabilem* Partitionieren gilt zusätzlich, dass die relative Ordnung der Elemente innerhalb der Partitionen in A' gleich bleibt. Also für alle Elemente $e_1, e_2 \in A$ mit $P(e_1) = P(e_2)$ gilt, dass

$$\mathsf{index}(A', e_1) < \mathsf{index}(A', e_2) \quad \text{g.d.w.} \quad \mathsf{index}(A, e_1) < \mathsf{index}(A, e_2).$$ [7]

Implementieren Sie einen *in-place* Algorithmus (in Pseudocode), der in einem gegebenen Array A mit einem ebenfalls gegebenen Prädikat P eine stabile Partitionierung herstellt. Ihr Algorithmus darf die worst-case Laufzeitschranke von $\mathcal{O}(n^2)$ nicht überschreiten. Zeigen Sie, dass Ihr Algorithmus die Laufzeitbedingung erfüllt. Zeigen Sie die Korrektheit Ihrer Implementierung mithilfe von Invarianten.

[7] $\mathsf{index}(A, e)$ steht hier für die Position des Elements e in Array A.

C.3 Directed Acyclic Graphs, DAG (7 Punkte)

Gegeben Sei ein gerichteter, *nicht notwendigerweise zusammenhängender* Graph $G = (V, E)$ mit n Knoten und m Kanten in Adjazenzfelddarstellung. Schreiben Sie einen Algorithmus isDAG (in Pseudocode), der genau dann true zurückgibt, wenn G ein DAG ist, und false zurückgibt, wenn G einen Zyklus enthält. Die Laufzeit darf $\mathcal{O}(n + m)$ nicht überschreiten, dies ist zu zeigen.