Name:	Matrikelnummer:

Prof. für Algorithmen
und Datenstrukturen
Prof. Dr. Hannah Bast
Patrick Brosi

Algorithmen und Datenstrukturen
SS 2021

http://ad-wiki.informatik.uni-freiburg.de/teaching

UNI FREIBURG

Nachklausur

Freitag 25. Februar 2022, 14:00 - 16:00 Uhr, Gebäude 101 der TF, HS 026 + HS 036

Es gibt *fünf* Aufgaben. Für jede davon gibt es maximal 25 Punkte. Wir zählen nur die *vier besten* Aufgaben. Sie können also maximal 100 Punkte erreichen. Zum Bestehen reichen 50 Punkte.

Sie haben insgesamt 120 Minuten Zeit. Wenn Sie vier Aufgaben bearbeiten, haben Sie im Durchschnitt 30 Minuten pro Aufgabe Zeit.

An Lehrmaterialien dürfen Sie maximal **ein** nach Ihrer Wahl beschriftetes DIN-A4 Blatt Papier verwenden. Sie dürfen keinerlei elektronische Geräte verwenden, insbesondere keine Geräte, mit denen Sie mit Dritten kommunizieren oder sich mit dem Internet verbinden können.

Wir werden die Klausur am selben Tag korrigieren und die Noten dann gleich im HISinOne eintragen. Die Klausureinsicht wird voraussichtlich am Dienstag, den 01.03.2022 um 14:00 Uhr stattfinden. Genauere Informationen dazu werden noch auf dem Forum bekannt gegeben.

Bemerkung zu den Programmieraufgaben: Wann immer das Schreiben von Code gefragt ist, können Sie frei zwischen Python, Java und C++ wählen. Für kleinere, rein syntaktische Fehler gibt es keinen Punktabzug. Alle Programmieraufgaben dieser Klausur lassen sich mit wenig Code lösen, bei mehr als 12 Zeilen riskieren Sie Punktabzug.

Was Sie wie abgeben sollen: Schreiben Sie bitte oben in die blaue Box Ihren Namen und Ihre Matrikelnummer. Schreiben Sie Ihre Lösungen bitte auf die Klausur: benutzen Sie dabei für jede Aufgabe zuerst die Vorderseite und dann die Rückseite des entsprechenden Blattes. Wenn Sie zusätzliches Papier benötigen, schreiben Sie ebenfalls auf jedes dieser Blätter Ihren Namen und Ihre Matrikelnummer.

Tipps: Lesen Sie die Aufgaben sorgfältig durch und vermeiden Sie Flüchtigkeitsfehler. Verbringen Sie nicht zu viel Zeit mit einer einzelnen Teilaufgabe. Wenn Sie nicht weiterkommen, machen Sie erstmal mit einer anderen (Teil)aufgabe weiter.

Dies ist eine Version der Klausur mit Lösungsskizzen. Verteilen Sie sie nicht weiter, sie ist ausschließlich für den persönlichen Gebrauch bestimmt. Benutzen Sie sie nicht, wenn Sie alte Klausuren zur Prüfungsvorbereitung durchrechnen, es ist die schlechteste Art zu lernen. Benutzen Sie sie nur, um Ihre Lösungen nachzuprüfen, nachdem Sie selber ernsthaft versucht haben, die Aufgaben zu lösen.

Aufgabe 1 (O-Notation und Prioritätswarteschlangen, 25 Punkte)

1.1 (7 Punkte) Sei $f(n) = \log(n^3)$ und $g(n) = (\log n)^2$, jeweils für alle $n \in \mathbb{N}$. Beweisen oder widerlegen Sie: $f = O(g)$. Beweisen oder widerlegen Sie: $g = O(f)$. Sie können dabei die Definition der O-Notation benutzen oder über Grenzwerte argumentieren.

1. Durch einfache Umformung ergibt sich $f(n) = 3 \cdot \log n$.
2. Daher $f(n)/g(n) = 3 \cdot \log n/(\log n)^2 = 3/\log n$.
3. Da der Zähler konstant ist und $\lim_{n\to\infty} \log n = \infty$, gilt $\lim_{n\to\infty} f(n)/g(n) = 0$.
4. Deshalb gilt $f = O(g)$, aber nicht $g = O(f)$.

1.2 (8 Punkte) Schreiben Sie eine Funktion *pq_sort*, die ein gegebenes Feld mittels einer Prioritätswarteschlange aufsteigend sortiert. Das gegebene Feld soll geändert und kein neues Feld zurückgegeben werden. Sie können dabei annehmen, dass es eine Klasse *PriorityQueue* gibt, die die üblichen Funktionen einer Prioritätswarteschlange bereitstellt (*insert*, *get_min*, *delete_min*).

```
1.  def pq_sort(array):
2.      pq = PriorityQuery()
3.      for x in array:
4.          pq.insert(x)
5.      for i in range(len(array)):
6.          A[i] = pq.get_min()
7.          pq.delete_min()
```

1.3 (10 Punkte) Sei A ein Feld der Größe $n + 1$, in dem die n Elemente eines binären Heaps abgespeichert sind (in $A[0]$ steht ein beliebiger Wert). Beweisen oder widerlegen Sie jede der folgenden beiden Aussagen. Wenn die Heapeigenschaft erfüllt ist, sind $A[1], \ldots, A[n]$ aufsteigend sortiert. Wenn $A[1], \ldots, A[n]$ aufsteigend sortiert sind, ist die Heapeigenschaft erfüllt.

1. Aussage 1 gilt nicht. Ein Gegenbeispiel ist $A[1] = 1, A[2] = 3, A[3] = 2$. Dies ist ein Heap mit 1 an der Wurzel und Kindern 3 und 2. Die Heapeigenschaft ist erfüllt, da $A[1] < A[2]$ und $A[1] < A[3]$. Aber die Elemente sind offensichtlich nicht aufsteigend sortiert, da $A[2] > A[3]$.

2. Aussage 2 gilt. Sei i der Index von einem inneren Knoten und j der Index eines der beiden Kinder, sofern existent. Dann ist entweder $j = 2i$ oder $j = 2i + 1$, aber auf jeden Fall $i > j$. Da die Elemente aufsteigend sortiert sind, ist dann $A[i] < A[j]$.

Aufgabe 2 (Sortieren und Potenzialfunktionsmethode, 25 Punkte)

2.1 (8 Punkte) Betrachten Sie die folgende Funktion *sort(array)*. Geben Sie für jede der Zeilen 2-7 an, wie oft diese für ein Feld *array* der Länge n mindestens und höchstens ausgeführt wird. Eine Zeile mit einer *for* Anweisung wird dabei so oft ausgeführt wie die Menge groß ist, über die iteriert wird. Geben Sie zusätzlich die Laufzeit des Algorithmus als $\Theta(...)$ in möglichst einfacher Form an.

```
1.  def sort(array):
2.    for i in range(len(array)):
3.      k = i
4.      for j in range(i + 1, len(array)):
5.        if array[j] < array[k]:
6.          k = j
7.      array[i], array[k] = array[k], array[i]
```

1. Zeile 2, 3 und 7: immer genau n mal.
2. Zeile 4 und 5: immer genau $n \cdot (n-1)/2$ mal.
3. Zeile 6: mindestens 0 mal und höchstens $n \cdot (n-1)/2$ mal.
4. Die Gesamtlaufzeit ist $\Theta(n^2)$.

2.2 (7 Punkte) Schreiben Sie eine Funktion *merge(A, B)*, die zwei aufsteigend sortierte Felder A und B in Laufzeit $O(n)$ zu einem aufsteigend sortierten Feld zusammenfügt und dieses zurückgibt.

```
1.  def merge(A, B):
2.    i = j = 0
3.    C = []
4.    while i < len(A) or j < len(B):
5.      if i == len(B) or A[i] < B[j]:
6.        C.append(A[i])
7.        i += 1
8.      else:
9.        C.append(B[j])
10.       j += 1
11.   return result
```

2.3 (10 Punkte) Eine Studentin kauft ein Smartphone für 100 EUR, das einen Monat hält. In jedem Folgemonat schaut sie, ob ihr Smartphone noch funktioniert und wenn nicht, kauft sie ein neues, das doppelt so lange hält und doppelt so teuer ist wie das vorherige. Beweisen Sie, dass ihre Kosten in n Monaten $O(n)$ sind. Wählen Sie dazu eine geeignete Potenzialfunktion Φ und Konstanten A und B, sodass für die Kosten T_i im i-ten Monat gilt: $T_i \leq A + B \cdot (\Phi_i - \Phi_{i-1})$.

1. Definere Φ_i als die Anzahl der Monate nach Monat i bis zum nächsten Kauf und $\Phi_0 = 0$.
2. Wähle $A = B = 100$ EUR.
3. Fall 1: Kein Kauf in Monat i. Dann ist $\Phi_i = \Phi_{i-1} - 1$ und $T_i = 0 = A + B \cdot (\Phi_i - \Phi_{i-1})$.
4. Fall 2: Sie kauft in Monat i ein Smartphone für $k \cdot 100$ Euro, das k Monate hält. Dann ist $\Phi_i = \Phi_{i-1} + k - 1$ und $T_i = 100 \cdot k = 100 + 100 \cdot (k-1) = A + B \cdot (\Phi_i - \Phi_{i-1})$.
5. Gemäß Mastertheorem ist dann $\sum_i T_i = O(n + \Phi_n)$. Das ist $O(n)$, da $\Phi_n \leq n$.

Aufgabe 3 (Blockoperationen und Editierdistanz, 25 Punkte)

3.1 (5 Punkte) Seien x und y die Zeichenketten *FREI* und *FERIEN*. Bestimmen Sie $\mathrm{ED}(x, y)$, indem Sie die zugehörige 5×7 Tabelle ausfüllen.

	ε	F	E	R	I	E	N
ε	0	1	2	3	4	5	6
F	1	0	1	2	3	4	5
R	2	1	1	1	2	3	4
E	3	2	1	2	2	2	3
I	4	3	2	2	2	3	$\boxed{3}$

3.2 (5 Punkte) Die Präfix-Editier-Distanz sei definiert als $\mathrm{PED}(x, y) = \min_{y'} \mathrm{ED}(x, y')$ wobei y' ein Präfix von y ist. Bestimmen Sie $\mathrm{PED}(x, y)$ für x und y aus Aufgabe 3.1 anhand der bereits gezeichneten Tabelle. Geben Sie alle y' an, für die diese minimale Editier-Distanz erreicht wird.

1. Die letzte Zeile der Tabelle enthält die Editier-Distanzen zwischen *FREI* und allen Präfixen von *FERIEN*. Das Minimum dieser Zeile ist also die Präfix-Editier-Distanz: $\mathrm{PED}(x, y) = 2$.
2. Die Präfixe von y, für die das Minimum erreicht wird, sind: *FE*, *FER*, *FERI*.

3.3 (7 Punkte) Beweisen oder widerlegen Sie: $\mathrm{ED}(x, y) \leq \max\{|x|, |y|\}$

1. Es reicht, die Aussage für $|x| \leq |y|$ zu beweisen. Für den Fall $|x| > |y|$ können wir dann x und y vertauschen und die Aussage folgt aus $\mathrm{ED}(x, y) = \mathrm{ED}(y, x)$.
2. Für $|x| \leq |y|$ können wir x durch folgende Operationen in y überführen: Ersetzen von $x[i]$ durch $y[i]$, für $i = 1, \ldots, |x|$. Einfügen von $y[i]$ am Ende, für $i = |x| + 1, \ldots, |y|$.
3. Das sind $|x| + (|y| - |x|) = |y| = \max\{|x|, |y|\}$ Operationen.

3.4 (8 Punkte) Sei σ eine Permutation der Zahlen $0, \ldots, 11$ und nehmen Sie an, ein Programm greift nacheinander auf die Elemente $A[\sigma(0)]$, $A[\sigma(1)]$, ..., $A[\sigma(11)]$ eines Feldes A der Größe $n = 12$ zu. Sei die Blockgröße $B = 4$ und $M = B$ (d.h. in den schnellen Speicher passt genau ein Block). Das Feld beginnt genau an einer Blockgrenze. Wie viele Permutationen gibt es, für die das Programm die kleinstmögliche Anzahl an Blockoperationen benötigt? Mit Begründung!

1. Das Programm benötigt mindestens $\lceil n/B \rceil = 3$ Blockoperationen, nämlich genau dann, wenn für jeden Block, der gelesen wird, auf alle Elemente darin zugegriffen wird, bevor der nächste Block gelesen wird.
2. Es gibt $3! = 6$ mögliche Reihenfolgen, in denen die 3 Blöcke gelesen werden können.
3. Es gibt $4! = 24$ mögliche Reihenfolgen, in denen auf die 4 Elemente innerhalb eines Blocks zugegriffen werden kann.
4. Da die Zugriffsreihenfolge innerhalb eines Blocks unabhängig von den anderen Blöcken ist, gibt es $6 \cdot 24^3 = 82.944$ Permutationen, für die das Programm 3 Blockoperationen benötigt.

Aufgabe 4 (Graphen und Suchbäume, 25 Punkte)

4.1 (5 Punkte) Zeichnen Sie den durch die Adjazenzmatrix A beschriebenen gerichteten und gewichteten Graph G. Der Eintrag 8 bedeutet zum Beispiel, dass es eine Kante von Knoten 4 zu Knoten 2 gibt, mit Kosten 8. Ein $\times$ bedeutet "keine Kante". Führen Sie Dijkstras Algorithmus auf G genau so lange aus, bis die Kosten des kürzesten Pfades von Knoten 4 nach Knoten 2 sicher gefunden sind. Stellen Sie jede Iteration als Zeile in einer Tabelle dar, deren Spalten die fünf dist-Werte der Knoten sind. Geben Sie in einer sechsten Spalte den in der nächsten Iteration gelösten Knoten an. Unterstreichen Sie die Kosten des gefundenen Pfades.

$$A = \begin{pmatrix} \times & 2 & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 3 & 2 & \times & \times & 0 \\ \times & 8 & \times & \times & 4 \\ 4 & \times & 1 & \times & \times \end{pmatrix}$$

	dist[1]	dist[2]	dist[3]	dist[4]	dist[5]	next
1.	inf	inf	inf	0	inf	4
2.	inf	8	inf	0	4	5
3.	8	8	5	0	4	3
4.	8	_7_	5	0	4	2

4.2 (5 Punkte) Ändern Sie *ein* Kantengewicht von G so, dass die Kosten des kürzesten Pfades von Knoten 4 nach Knoten 2 gleich bleiben, aber Dijkstras Algorithmus *alle* Knoten lösen muss, um ihn zu finden.

Ändere Gewicht von Kante $(3, 1)$ auf 1.

4.3 (5 Punkte) Beweisen oder widerlegen Sie: in einem Graph mit n Knoten und nichtnegativen Kantengewichten ändert sich der dist-Wert eines Knotens bei Dijkstras Algorithmus höchstens $n - 1$ mal.

1. Jeder Knoten wird höchtens einmal gelöst, jede Kante (u, v) also höchstens einmal relaxiert.
2. Für einen Knoten v kann sich der dist-Wert nur ändern, wenn eine Kante (u, v) relaxiert wird, mit $u \neq v$ (eine Kante von u zu sich selber kann den dist-Wert nicht verbessern).
3. Für einen Knoten v gibt es maximal $n - 1$ Kanten (u, v) mit $u \neq v$.

4.4 (10 Punkte) Implementieren Sie die Methode *insert(root, newNode)* die einen Knoten *newNode* vom Typ *TreeNode* in einen durch die Wurzel *root* gegebenen binären Suchbaum einfügt oder einen entsprechenden vorhandenen Knoten aktualisiert. *TreeNode* stellt die Felder *key*, *value*, *leftChild* und *rightChild* bereit. Fehlende Kinder werden durch den Wert *None* repräsentiert.

```
1.  def insert(root, newNode):
2.    if root.key == newNode.key: root.value = newNode.value
3.    elif root.key < newNode.key:
4.      if root.rightChild == None: root.rightChild = newNode
5.      else insert(root.rightChild, newNode)
6.    else
7.      if root.leftChild == None: root.leftChild = newNode
8.      else insert(root.leftChild, newNode)
```

Aufgabe 5 (Hashtabellen und universelles Hashing, 25 Punkte)

5.1 (8 Punkte) Betrachten Sie eine Hashtabelle der Größe 5 mit der Hashfunktion $h(x) = 4 \cdot x \bmod 5$. Geben Sie eine Schlüsselmenge S_1 mit $|S_1| = 10$ Werten an, so dass es mit h zu einer maximalen Zahl von Kollisionen kommt. Geben Sie außerdem eine Schlüsselmenge S_2 mit $|S_2| = 10$ Werten an, so dass es mit h zu einer minimalen Zahl von Kollisionen kommt. Mit Begründung!

1. Für $S_1 = \{0, 5, 10, 15, 20, 25, 30, 35, 40, 45\}$ werden alle Schlüssel aus S auf denselben Platz in der Hashtabelle abgebildet. Mehr Kollisionen sind nicht möglich.
2. Für $S_2 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ werden auf jeden Platz in der Hashtabelle genau 2 Schlüssel abgebildet. Besser kann man die Schlüssel nicht verteilen.

5.2 (7 Punkte) Implementieren Sie eine Funktion *erase(T, h, key)*, die aus der Hashtabelle T mit Hashfunktion h das Element mit dem Schlüssel *key* möglichst effizient löscht. Wenn *key* nicht in der Hashtabelle enthalten ist, soll nichts passieren. Nehmen Sie dabei an, dass die Hashtabelle als Feld von Feldern implementiert ist, mit Elementen eines Typs mit den zwei Komponenten *key* und *value*. Das Löschen eines beliebigen Elements aus einem Feld sollen Sie selbst implementieren. Sie können davon ausgehen, dass es eine Funktion *pop()* gibt, die das letzte Element aus einem Feld löscht.

```
1.  def erase(T, h, key):
2.    bucket = T[h(key)]
3.    for i in range(len(bucket)):
4.      if (bucket[i].key == key):
5.        bucket[i] = bucket[len(bucket) - 1]
6.        bucket.pop()
7.        break
8.    return
```

5.3 (10 Punkte) Sei H eine 1-universelle Klasse von Hashfunktionen für ein Universum $U \subset \mathbb{N}$ und eine Größe m einer Hashtabelle. Sei $h_0 : U \to \{0, \ldots, m-1\}$ die Funktion, die alle Elemente aus U auf 0 abbildet. Beweisen Sie, dass dann $H' = H \cup \{h_0\}$ eine 2-universelle Klasse von Hashfunktionen für U und m ist. Hinweis: Sie dürfen ohne Beweis verwenden, dass jede 1-universelle Klasse mindestens m Hashfunktionen enthält.

1. Falls $h_0 \in H$, ist $H' = H$ und H' ist wie H 1-universell und damit auch 2-universell.
2. Ansonsten seien $x, y \in U$ beliebig mit $x \neq y$.
3. Wir müssen zeigen, dass $|\{h \in H' : h(x) = h(y)\}| \leq 2 \cdot |H'|/m$.
3. Es ist $|\{h \in H' : h(x) = h(y)\}| = |\{h \in H : h(x) = h(y)\}| + 1$, weil mit Sicherheit $h_0(x) = h_0(y)$.
4. Da H universell ist, gilt $|\{h \in H : h(x) = h(y)\}| + 1 \leq |H|/m + 1$.
5. Da wir annehmen dürfen, dass $|H| \leq m$, gilt $|H|/m + 1 \leq |H|/m + |H|/m = 2 \cdot |H|/m$.