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

Klausur

Dienstag 24. August 2021, 14:00 - 16:30 Uhr, Gebäude 101 und 051 der Technischen Fakultät

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 150 Minuten Zeit. Wenn Sie vier Aufgaben bearbeiten, haben Sie im Durchschnitt 30 Minuten pro Aufgabe Zeit und dann noch 30 Minuten Puffer.

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 und am nächsten Tag korrigieren und die Noten dann gleich im HISinOne eintragen. Zum Ablauf und Termin der Klausureinsicht wird es nach der Korrektur noch eine Ankündigung auf dem Forum geben.

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, wie z.B. ein fehlendes Semikolon, 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 (Sortieren und O-Notation, 25 Punkte)

1.1 (10 Punkte) Betrachten Sie die folgende Funktion, die ein gegebenes Feld absteigend sortiert. Geben Sie für jede der Zeilen 2 - 5 an, wie oft die Zeile für ein Feld der Größe n ausgeführt wird. Wenn die Anzahl der Ausführungen von der Eingabe abhängt, geben sie an, wie viele Ausführungen es mindestens gibt und wie viele höchstens (mit Begründung). Eine Zeile mit einer *for* Anweisung wird dabei so oft ausgeführt, wie die Menge groß ist, über die iteriert wird.

```
1.  def max_sort(A):
2.    for i in range(len(A)):
3.      for j in range(i + 1, len(A)):
4.        if A[i] < A[j]:
5.          A[i], A[j] = A[j], A[i]
```

Zeile 2 wird immer genau n mal ausgeführt.
Zeile 3 wird immer genau $(n-1) + (n-2) + \ldots + 1 = n \cdot (n-1)/2$ mal ausgeführt.
Zeile 4 wird genauso oft ausgeführt wie Zeile 3.
Zeile 5 wird mindestens 0 mal ausgeführt (wenn das Feld bereits absteigend sortiert ist) und höchstens so oft wie Zeile 3 bzw. Zeile 4 (wenn das Feld genau falsch herum sortiert ist).

1.2 (5 Punkte) Jeder vergleichsbasierte Algorithmus lässt sich leicht so erweitern, dass er zu Beginn vergleichsbasiert in Zeit $O(n)$ überprüft, ob die Folge schon sortiert ist, und falls ja, nichts weiter tut. Warum widerspricht das nicht der unteren Schranke von $\Omega(n \cdot \log n)$ für vergleichsbasiertes Sortieren?

1. Der so abgeänderte Algorithmus hat Laufzeit $O(n)$ für *manche* Eingaben, insbesondere solche, die bereits sortiert sind.
2. Die untere Schranke besagt, dass die Laufzeit nicht für *alle* Eingaben $O(n)$ sein kann.

1.3 (10 Punkte) Seien f_1, f_2, g_1, g_2 Funktionen $\mathbb{N} \to \mathbb{R}_0^+$. Beweisen Sie, dass wenn $f_1 = O(g_1)$ und $f_2 = O(g_2)$, dann auch $f_1 + f_2 = O(g_1 + g_2)$. Argumentieren Sie dabei über die Definition von $O(...)$ und nicht über Grenzwerte.

1. $f_1 = O(g_1)$ heißt: es gibt $C_1 > 0$ und n_1, so dass für all $n \geq n_1$, $f_1(n) \leq C_1 \cdot g_1(n)$.
2. $f_2 = O(g_2)$ heißt: es gibt $C_2 > 0$ und n_2, so dass für all $n \geq n_2$, $f_2(n) \leq C_2 \cdot g_2(n)$.
3. Wähle $n_0 = \max(n_1, n_2)$ und $C = \max(C_1, C_2)$.
4. Dann gilt für alle $n \geq n_0$, dass $f_1(n) + f_2(n) \leq C_1 \cdot g_1(n) + C_2 \cdot g_2(n) \leq C \cdot (g_1(n) + g_2(n))$.
5. Damit ist $f_1 + f_2 = O(g_1 + g_2)$.

Aufgabe 2 (Hashing und dynamische Felder, 25 Punkte)

2.1 (5 Punkte) Zeichnen Sie den Zustand der Hashtabelle der Größe 5 nach Einfügen der Elemente $11, 4, 21, 29, 6$ (in dieser Reihenfolge) mit der Hashfunktion $h(x) = 7 \cdot x \bmod 5$. Benutzen Sie Hashing mit offener Adressierung.

1. Die Hashwerte sind $h(11) = 2$, $h(4) = 3$, $h(21) = 2$, $h(29) = 3$, $h(6) = 2$.
2. Offene Adressierung: wenn Platz i in der Tabelle belegt ist, probieren wir $i + 1 \bmod 5$, usw.
3. Hashtabelle nach den fünf Einfügungen: $[\,29, 6, 11, 4, 21\,]$.

2.2 (10 Punkte) Seien H_1 und H_2 zwei c-universelle Klassen für Hashtabellen der Größe m, mit $H_1 \cap H_2 = \emptyset$. Beweisen Sie, dass dann auch $H_1 \cup H_2$ eine c-universelle Klasse für Hashtabellen der Größe m ist.

1. H_i universell heißt: für alle $x, y \in U$ mit $x \neq y$, ist $|\{h \in H_i : h(x) = h(y)\}| \leq c \cdot |H_i|/m$.
2. Betrachten wir jetzt beliebige $x, y \in U$ mit $x \neq y$ und $|\{h \in H_1 \cup H_2 : h(x) = h(y)\}|$.
3. Wegen $H_1 \cap H_2 = \emptyset$, ist das $= |\{h \in H_1 : h(x) = h(y)\}| + |\{h \in H_2 : h(x) = h(y)\}|$.
4. Nach 1 ist das $\leq c \cdot |H_1|/m + c \cdot |H_2|/m = c \cdot (|H_1| + |H_2|)/m$.
5. Wegen $H_1 \cap H_2 = \emptyset$ ist $|H_1| + |H_2| = |H_1 \cup H_2|$.

2.3 (5 Punkte) Seien bei einem dynamischen Feld s_i und c_i die Anzahl Elemente und die Kapazität nach der i-ten Operation. Zu Beginn ist $s_0 = 0$ und $c_0 = 1$. Wenn vor einer *push* Operation $s_{i-1} = c_{i-1}$, wird das Feld vergrößert, so dass nach der Operation $c_i = 2 \cdot s_i$. Wenn nach einer *pop* Operation $s_i < c_i/2$, wird das Feld verkleinert, so dass $c_i = s_i$. Geben Sie die Werte von s_i und c_i nach jeder der folgenden Operationen an: *push*, *push*, *push*, *pop*, *pop*, *push*. Die Argumente von *push* wurden weggelassen, weil sie für die Aufgabe unerheblich sind.

1. $s_1 = 1$, $c_1 = 1$ (*push* ohne Vergrößerung).
2. $s_2 = 2$, $c_2 = 4$ (*push* mit Vergrößerung).
3. $s_3 = 3$, $c_3 = 4$ (*push* ohne Vergrößerung).
4. $s_4 = 2$, $c_4 = 4$ (*pop* ohne Verkleinerung).
5. $s_5 = 1$, $c_5 = 1$ (*pop* mit Verkleinerun).
6. $s_6 = 2$, $c_6 = 4$ (*push* mit Vergrößerung).

2.4 (5 Punkte) Sind bei der Vergrößerungs- und Verkleinerungsstrategie von Aufgabe 2.3 die Gesamtkosten für eine Folge von n beliebigen Operationen auf einem zu Beginn leeren Feld $O(n)$? Geben Sie eine kurze aber stichhaltige Begründung für Ihre Antwort.

1. Sei nach einer Folge von n *push* Operationen $c_n = s_n = n$.
2. Dann ist nach einer weiteren *push* Operation $c_{n+1} = 2 \cdot s_{n+1}$.
3. Dann ist nach einer weiteren *pop* Operation $c_{n+2} = s_{n+2} = s_n = c_n = n$, wie bei 1.
4. Jeder dieser beiden Operationen hat Kosten $\Theta(n)$, wegen des Umkopierens.
5. Nach $n/2$ mal diese beiden Operationen, haben wir insgesamt Kosten $\Theta(n^2)$ für $2n$ Operationen.

Aufgabe 3 (Blockoperationen, Suchbäume und Heaps, 25 Punkte)

3.1 (10 Punkte) Sei σ eine Permutation der Zahlen $0, \ldots, 11$ und nehmen wir an, ein Programm greift nacheinander auf die Elemente $A[\sigma(0)]$, $A[\sigma(1)]$, ..., $A[\sigma(11)]$ eines Feldes A der Größe 12 zu. Bestimmen Sie ein σ, so dass das Programm genau *neun* Blockoperationen benötigt, mit Blockgröße $B = 4$. Mit Begründung! Nehmen Sie dabei an, dass das Feld genau an einer Blockgrenze beginnt und dass $M = B$ (d.h., in den schnellen Speicher passt genau ein Block).

1. Zugriff auf $A[0]$, $A[4]$, $A[8]$ $\rightarrow$ das sind drei Blockoperationen.
2. Zugriff auf $A[1]$, $A[5]$, $A[9]$ $\rightarrow$ drei weitere Blockoperationen.
3. Zugriff auf $A[2]$ und $A[3]$, $A[6]$ und $A[7]$, $A[10]$ und $A[11]$ $\rightarrow$ drei weitere Blockoperationen.
4. Das sind insgesamt neun Blockoperationen, $\sigma = 0, 4, 8, 1, 5, 9, 2, 3, 6, 7, 10, 11$.

3.2 (8 Punkte) In einem $(2, 4)$-Baum wurde ein Element eingefügt und es kam dabei zu einer Folge von $m \geq 1$ Aufspaltungen. Sei k die Anzahl der Knoten vom Grad 3 vor der letzten Aufspaltung und k' die Anzahl der Knoten vom Grad 3 nach der letzten Aufspaltung. Welche Möglichkeiten gibt es für den Wert von $k' - k$ und warum?

1. Die letzte Aufspaltung ersetzt einen Knoten vom Grad 5 durch Knoten vom Grad 2 und 3.
2. Dadurch kommt also mit Sicherheit ein Knoten vom Grad 3 dazu.
3. Der Elternknoten des letzten aufgespaltenen Knotens hat 3 oder 4 Kinder (2 kann nicht sein, weil durch die Auspaltung ein Kind dazu gekommen ist; 5 kann nicht sein, weil sonst nochmal aufgespalten werden müsste).
4. Wenn er 3 Kinder hat, hatte er vorher 2 Kinder und es gibt einen Knoten vom Grad 3 mehr.
5. Wenn er 4 Kinder hat, hatte er vorher 3 Kinder und es gibt einen Knoten vom Grad 3 weniger.
6. Insgesamt ist also $k' - k$ entweder 0 oder 2.
7. Speziallfall: wenn der Baum vor der Operation genau ein Blatt mit vier Kindern hat, wird genau einmal aufgespalten, die Wurzel hat zwei Kinder und damit $k' - k = 1$. Dieser Spezialfall musste nicht berücksichtigt werden, um die volle Punktezahl zu bekommen.

3.3 (7 Punkte) Schreiben Sie eine Funktion *check_heap_property(heap_array)*, die für einen in dem Feld *heap_array* abgespeicherten binären Heap *True* zurückgibt wenn die Heapeigenschaft erfüllt ist und *False* wenn nicht. Sie können dabei annehmen, dass das Feld für einen Heap mit n Elementen die Größe $n + 1$ hat. An Position 0 in dem Feld steht der Wert *None*.

```
1.  def check_heap_property(heap_array):
2.    for i in range(2, len(heap_array)):
3.      child_value = heap_array[i]
4.      parent_value = heap_array[i//2]
5.      if parent_value > child_value:
6.        return False
7.    return True
```

Aufgabe 4 (Graphen und Editierdistanz, 25 Punkte)

4.1 (10 Punkte) Schreiben Sie eine Funktion *explore(adjacency_lists, s)*, die auf einem durch seine Adjazenzlisten gegebenen ungerichteten Graphen alle Knoten berechnet, die von einem Startknoten s aus erreichbar sind. Das Ergebnis sollte als Menge zurückgegeben werden (Sie können dafür ein Python *dictionary* oder *set* verwenden). Sie können annehmen, dass in *adjacency_lists* alle Kanten in beide Richtungen abgespeichert sind, ohne Kantenkosten (die für diese Aufgabe irrelevant sind).

```
 1.  def explore(adjacency_lists, s):
 2.    frontier = [s]
 3.    visited_nodes = set()
 4.    while len(frontier) > 0:
 5.      u = frontier.pop()
 6.      if u not in visited_nodes:
 7.        visited_nodes.add(u)
 8.        for v in adjacency_lists[u]:
 9.          frontier.append(v)
10.    return visited_nodes
```

4.2 (10 Punkte) Geben Sie einen gerichteten Graphen an mit folgenden Eigenschaften. Der Graph hat mindestens ein negatives Kantengewicht, aber keinen Zyklus. Es gibt einen Knoten s, so dass Dijkstras Algorithmus, angefangen bei s und ausgeführt bis alle Knoten gelöst sind, zu einem inkorrekten Ergebnis führt. Das heißt, am Ende entspricht mindestens ein *dist* Wert an einem Knoten nicht der Distanz des kürzesten Weges von s aus zu diesem Knoten. Mit Begründung!

1. Betrachte einen Graph mit drei Knoten s, u, t und den folgenden drei Kanten.
2. Es gibt eine Kante von s nach t mit Kosten 1 und eine Kante von s nach u mit Kosten 2.
3. Es gibt außerdem eine Kante von u nach t mit Kosten -2.
4. Der kürzeste Weg von s nach t geht also über Knoten u und hat Kosten 0.
5. Dijkstras Algorithmus löst erst den Knoten s mit korrektem *dist* Wert 0.
6. In der PQ stehen dann t mit *dist* Wert 1 und u mit *dist* Wert 2.
7. Dijkstras Algorithmus löst also als nächstes t, aber der *dist* Wert ist nicht korrekt.

4.3 (5 Punkte) Seien x und y die Zeichenketten $ALGO$ und $ANGIE$. Bestimmen Sie $\mathrm{ED}(x, y)$, indem Sie die zugehörige 5×6 Tabelle ausfüllen.

	ε	A	N	G	I	E
ε	0	1	2	3	4	5
A	1	0	1	2	3	4
L	2	1	1	2	3	4
G	3	2	2	1	2	3
O	4	3	3	2	2	$\boxed{3}$

Aufgabe 5 (Vermischtes, 25 Punkte)

5.1 (10 Punkte) Schreiben Sie eine Funktion *second_largest(A)*, die die zweitgrößte Zahl in dem gegebenen Feld A findet. Die Laufzeit sollte $O(n)$ sein, wobei n die Anzahl der Elemente in dem Feld ist. Sie können annehmen, dass $n \geq 2$ und dass die Zahlen in dem Feld alle verschieden sind.

```
1.  def second_largest(A):
2.    max_1 = max(A[0], A[1])
3.    max_2 = min(A[0], A[1])
4.    for i in range(2, len(A)):
5.      if A[i] > max_1:
6.        max_1, max_2 = A[i], max_1
7.      elif A[i] > max_2:
8.        max_2 = A[i]
9.    return max_2
```

5.2 (5 Punkte) Sei G ein ungerichteter Graph mit Knotenmenge V und Kantenmenge E. Beweisen Sie: wenn G ein Baum ist, dann ist $|E| = |V| - 1$.

1. Wenn G ein Baum ist, können wir einen Knoten als Wurzel betrachten.
2. Jede Kante verbindet dann einen Elternknoten mit einem Kindknoten.
3. Wir weisen jede Kante ihrem Kindknoten zu.
4. Dann ist jede Kante genau einen Knoten zugewiesen.
5. Und jedem Knoten, außer der Wurzel, ist genau eine Kante zugewiesen.
6. Damit ist $|E|$ genau die Anzahl Knoten außer der Wurzel, also $|V| - 1$.

5.3 (10 Punkte) Betrachten Sie den Aufruf *ed_recursive("aa", "bb")* der folgenden rekursiven Funktion. Genau welche Aufrufe von *ed_recursive* gibt es dabei insgesamt und in welcher Reihenfolge? Stellen Sie dabei jeden Aufruf als Paar der beiden Argumente dar, ohne die Anführungszeichen. Zum Beispiel ist der erste Aufruf (aa, bb) und der zweite Aufruf (aa, b).

```
1.  def ed_recursive(x, y):
2.    n, m = len(x), len(y)
3.    if n == 0: return m
4.    if m == 0: return n
5.    ed1 = ed_recursive(x, y[0:m-1]) + 1
6.    ed2 = ed_recursive(x[0:n-1], y) + 1
7.    ed3 = ed_recursive(x[0:n-1], y[0:m-1]) + (0 if x[n-1] == y[m-1] else 1)
8.    return min(ed1, ed2, ed3)
```

Es gibt insgesamt die folgenden 19 Aufrufe, in dieser Reihenfolge: (aa, bb), (aa, b), (aa, ε), (a, b), (a, ε), (ε, b), $(\varepsilon, \varepsilon)$, (a, ε), (a, bb), (a, b), (a, ε), (ε, b), $(\varepsilon, \varepsilon)$, (ε, bb), (ε, b), (a, b), (a, ε), (ε, b), $(\varepsilon, \varepsilon)$.