Nachname:	Vorname:	Matrikelnummer:

↑ Bitte **sehr deutlich** schreiben und BLOCKBUCHSTABEN (groß geschriebene Druckbuchstaben) verwenden ↑

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

Algorithmen und Datenstrukturen
SS 2023

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

UNI FREIBURG

Nachklausur

Dienstag 12. März 2024, 14:00 - 16:30 Uhr, HS 26 und HS 36

Es gibt *fünf* Aufgaben. Für jede davon gibt es maximal 20 Punkte. Sie können also maximal 100 Punkte erreichen. Zum Bestehen reichen 50 Punkte. Sie haben insgesamt zweieinhalb Stunden Zeit. Damit haben Sie im Durchschnitt 30 Minuten Zeit pro Aufgabe. Jede Aufgabe sollte in 20 Minuten machbar sein, die restlichen 10 Minuten sind als Puffer gedacht.

An Lehrmaterialien dürfen Sie maximal **ein** DIN-A4 Blatt Papier verwenden, das beidseitig mit von Ihnen selber zusammengestellten Inhalten beschriftet oder bedruckt sein kann. Sie dürfen keinerlei elektronische Geräte verwenden, insbesondere keine Geräte, mit denen Sie mit Dritten kommunizieren oder sich mit dem Internet verbinden können.

Bemerkung zu den Programmieraufgaben: Wenn Code gefragt ist, können Sie frei zwischen Python, Java und C++ wählen. Wir empfehlen Python. Für kleinere, rein syntaktische Fehler gibt es keinen Punktabzug. Alle Programmieraufgaben dieser Klausur lassen sich mit wenig Code lösen. Pro Aufgabe ist eine Obergrenze für die Anzahl der Zeilen vorgegeben.

Bemerkung zu den Multiple-Choice-Aufgaben: Bei einigen der Aufgaben wird lediglich nach dem Wahrheitswert von Aussagen gefragt. Antworten Sie dann zu jeder Aussage mit den Worten *wahr* oder *falsch* oder nichts. Ohne Antwort gibt es für die betreffende Aussage 0 Punkte. Für eine richtige Antwort gibt es 2 Punkte, für eine falsche Antwort gibt es zwei Punkte Abzug. Insgesamt können Sie für eine solche Aufgabe nicht weniger als 0 Punkte bekommen. Achten Sie darauf, dass in Ihrer Abgabe klar ist, zu welcher Aussage eine Antwort gehört.

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 des Blattes, auf dem die Aufgabe steht und dann die Rückseite des *vorherigen* Blattes. Wenn Sie zusätzliches Papier abgeben wollen (das sollte die Ausnahme sein), schreiben Sie auf jedes zusätzliche Blatt Ihren Namen und Ihre Matrikelnummer.

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, 20 Punkte)

1.1 (5 Punkte) Schreiben Sie eine Funktion *zero_one_sort(items: list[int])* $\rightarrow$ *list[int]*, die eine Liste *items*, die nur Nullen und Einsen enthält, aufsteigend sortiert. Die Funktion soll für Eingabegröße n in Zeit $O(n)$ laufen und maximal 10 Zeilen lang sein.

```
def zero_one_sort(items: list[int]) -> list[int]:
    num_ones = 0
    for item in items:
        num_ones += item
    return [0] * (len(items) - num_ones) + [1] * num_ones
```

1.2 (7 Punkte) Schreiben Sie eine Funktion *merge_descending(a: list[int], b: list[int])* $\rightarrow$ *list[int]*, die zwei absteigend sortierte Listen erhält und eine absteigend sortierte Liste mit allen Elementen aus beiden Listen zurückgibt. Die Funktion soll für Gesamtlänge n der beiden Listen in Zeit $O(n)$ laufen und maximal 15 Zeilen lang sein.

```
def merge_descending(a: list[int], b: list[int]) -> list[int]:
    i = j = 0
    res = []
    while i < len(a) and j < len(b):
        if a[i] >= b[j]:
           res.append(a[i])
           i += 1
        else:
           res.append(b[j])
           j += 1
    res.extend(a[i:])
    res.extend(b[j:])
    return res
```

1.3 (8 Punkte) Geben Sie für jede der folgenden Aussagen an, ob sie *wahr* oder *falsch* ist. Eine richtige Antwort gibt 2 Punkte, eine falsche Antwort gibt 2 Punkte Abzug. Wenn Sie die Antwort nicht wissen, schreiben Sie *keine Aussage* oder nichts. Sie können nicht weniger als 0 Punkte für diese Aufgabe bekommen.

1. Ein vergleichsbasierter Sortieralgorithmus braucht für Eingabegröße n immer $\Theta(n \log n)$ Zeit.

Falsch: Die Laufzeit von *MinSort* ist zum Beispiel $\Theta(n^2)$.

2. Man kann in Zeit $O(n)$ feststellen, ob ein Feld der Größe n bereits sortiert ist.

Wahr: Iterieren und abbrechen, sobald ein Element kleiner als das vorherige Element ist.

3. Die Laufzeit von *MinSort* für Eingabegröße n ist $\mathrm{O}(n)$, wenn die Eingabe bereits sortiert ist.

Falsch: Die Laufzeit von *MinSort* ist für jede Eingabe $\Theta(n^2)$, siehe Beweis aus der Vorlesung.

4. *MergeSort* für Eingabegröße n ruft $O(n)$ mal die Subroutine auf, die zwei Felder mergt.

Wahr: $\frac{n}{2^1} + ... + \frac{n}{2^{\log_2 n}} \in O(n)$

Aufgabe 2 (I/E-Sequenzen und O-Notation, 20 Punkte)

2.1 (6 Punkte) Die folgende Funktion sortiert die gegebene Liste aufsteigend. Geben Sie die `I`/`E`-Sequenz für die Eingabe $[5, 1, 3]$ an und schreiben Sie dabei über jedes `I` bzw. `E` die zugehörige Zeilennummer des Programms, sowie den Wert der Schleifenvariablen i. Die *for*- und *while*-Schleifen sollen Sie dabei nicht umschreiben, sondern wie folgt berücksichtigen: Wenn eine Iteration einer Schleife ausgeführt wird, entspricht dies einem `E`, wenn die Schleife beendet wird einem `I`. Schreiben Sie über jedes `I` bzw. `E` die zugehörige Zeilennummer und den aktuellen Wert der Variablen i.

```
1. def insertion_sort(l: list[int]):
2.     for i in range(len(l)):
3.         j = i
4.         while j > 0 and l[j - 1] > l[j]:
5             l[j-1], l[j] = l[j], l[j-1]
6.            j = j - 1
```

0	0	1	1	1	2	2	2	3
2	4	2	4	4	2	4	4	2
E	I	E	E	I	E	E	I	I

2.2 (6 Punkte) Geben Sie für jede der folgenden Aussagen an, ob sie *wahr* oder *falsch* ist. Eine richtige Antwort gibt 2 Punkte, eine falsche Antwort gibt 2 Punkte Abzug. Wenn Sie die Antwort nicht wissen, schreiben Sie *keine Aussage* oder nichts. Sie können insgesamt nicht weniger als 0 Punkte für diese Aufgabe bekommen.

1. $n^3 + 6 \cdot n^2 + n \cdot \log n \in \mathrm{o}(0.1 \cdot n^3 \cdot \log n)$

Wahr: $\lim_{n\to\infty}(n^3 + 6 \cdot n^2 + n \cdot \log n)/(0.1 \cdot n^3 \cdot \log n) = 0$

2. $n! \in \mathrm{O}(n^n)$

Wahr: $\lim_{n\to\infty} n!/n^n = 0$

3. Wenn $f_1 \in \Theta(f_2)$ und $f_2 \in \mathrm{O}(f_3)$, dann gilt $f_1 \in \Theta(f_3)$

Falsch: nicht erfüllt falls $f_2 \in \mathrm{o}(f_3)$

2.3 (8 Punkte) Seien f_1, f_2, g_1, g_2 streng positive Funktionen (also $f_1(n) > 0$ für alle n, ebenso für f_1, g_1 und g_2) mit $f_1 \in \mathrm{O}(g_1)$ und $f_2 \in \mathrm{O}(g_2)$. Beweisen Sie, dass $f_1 \cdot f_2 \in \mathrm{O}(g_1 \cdot g_2)$. Argumentieren Sie über die Definition von O und **nicht** über Grenzwerte.

1. $f_1 \in O(g_1)$ heißt: es gibt $n_1 > 0$ und $C_1 > 0$, sodass $f_1(n) \leq C_1 \cdot g_1(n)$ für alle $n > n_1$
2. $f_2 \in O(g_2)$ heißt: es gibt $n_2 > 0$ und $C_2 > 0$, sodass $f_2(n) \leq C_2 \cdot g_2(n)$ für alle $n > n_2$
3. Sei nun $n_0 = \max(n_1, n_2)$.
4. Dann gilt für alle $n > n_0$: $f_1(n) \cdot f_2(n) \leq C_1 \cdot g_1(n) \cdot C_2 \cdot g_2(n)$ (da alle Terme > 0)
5. Also mit $C := C_1 * C_2$ auch $f_1(n) \cdot f_2(n) \leq C \cdot g_1(n) \cdot g_2(n)$ und damit $f_1 \cdot f_2 \in \mathrm{O}(g_1 \cdot g_2)$

Aufgabe 3 (Assoziative und dynamische Felder, 20 Punkte)

3.1 (5 Punkte) Sei $h(x) = 2 \cdot \text{asciisum}(x) \bmod 3$ eine Hashfunktion für Schlüssel vom Typ *string*, wobei asciisum(x) die Summe der ASCII-Codes der einzelnen Zeichen von x ist. Die ASCII-Codes von $A..Z$ sind 65..90. Berechnen Sie die Hashwerte für die Schlüssel BLA, BLI und BLU und fügen Sie diese Schlüssel in dieser Reihenfolge in eine entsprechende Hashtabelle ein, mittels offener Adressierung. Zeichnen Sie den Zustand der Hashtabelle nach dem letzten Einfügen.

1. $\text{asciisum}(BLA) \bmod 3 = (66 + 76 + 65) \bmod 3 = (0 + 1 + 2) \bmod 3 = 0$
2. $\text{asciisum}(BLI) \bmod 3 = (66 + 76 + 73) \bmod 3 = (0 + 1 + 1) \bmod 3 = 2$
3. $\text{asciisum}(BLU) \bmod 3 = (66 + 76 + 85) \bmod 3 = (0 + 1 + 1) \bmod 3 = 2$
4. $h(BLA) = 0$, $h(BLI) = 1$, $h(BLU) = 1$
5. Zustand der Hashtabelle nach dem letzten Einfügen: BLA, BLI, BLU

3.2 (10 Punkte) Eine Bank benutzt TANs, die aus n Großbuchstaben (jeweils von $A..Z$) bestehen, zum Beispiel für $n = 6$, $AHGTEF$. Nehmen Sie an, es gibt eine Methode *check(tan: list[str]) → bool*, die eine als Feld von Zeichen gegebene TAN in $O(1)$ Zeit prüft, und genau dann *True* zurück gibt, wenn sie korrekt ist. Schreiben Sie eine Funktion *crack(n: int) -> list[str]*, die alle n-stelligen TANs durchprobiert und die korrekte zurück gibt. Sie können annehmen, dass es genau eine korrekte TAN gibt. Die Funktion sollte nicht mehr als 15 Zeilen haben und in Zeit $O(26^n)$ laufen. Sie können die Methode *chr* verwenden, die aus einem ASCII-Code das entsprechende Zeichen macht, und *ord* für die umgekehrte Richtung; beide Funktionen laufen in Zeit $O(1)$.

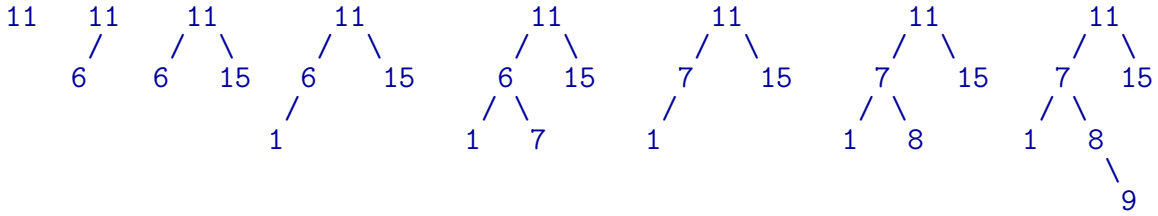
```
def crack(n: int) -> str:
    tan = ["A"] * (n + 1)
    while tan[n] == "A":
        if check(tan):
            return tan[:n]
        for i in range(n):
            tan[i] = chr(ord(tan[i]) + 1)
            if ord(tan[i]) > ord("Z"):
                tan[i] = "A"
            else:
               break
```

3.3 (5 Punkte) Beweisen Sie die folgende Variante des Mastertheorems. Sei Φ eine nichtnegative Potenzialfunktion, so dass für die Kosten der $i$ten Operation gilt: $T_i \leq A + B \cdot (\Phi_i - \Phi_{i-1})$, für irgendwelche Konstanten $A, B > 0$. Dann ist $\sum_{i=1}^{n} T_i = O(n + \Phi_n)$.

1. $\sum_{i=1}^{n} T_i \leq A \cdot n + B \cdot \sum_{i=1}^{n} (\Phi_i - \Phi_{i-1})$
2. $\sum_{i=1}^{n} (\Phi_i - \Phi_{i-1}) = \Phi_n - \Phi_0 \leq \Phi_n$, weil $\Phi_0 \geq 0$
3. Damit ist $\sum_{i=1}^{n} T_i \leq A \cdot n + B \cdot \Phi_n = O(n + \Phi_n)$

Aufgabe 4 (Binäre Suchbäume und Blockoperationen, 20 Punkte)

4.1 (4 Punkte) Zeichnen Sie einen zu Beginn leeren binären Suchbaum nach jeder der folgenden Operationen: *insert(11)*, *insert(6)*, *insert(15)*, *insert(1)*, *insert(7)*, *delete(6)*, *insert(8)*, *insert(9)*. Angegeben sind hierbei nur die Schlüssel. Die Werte dürfen für diese Aufgabe ignoriert werden.



4.2 (8 Punkte) Schreiben Sie eine rekursive Funktion *lookup(key: int, node: Node)* $\rightarrow$ *Node|None*, die in einem binären Suchbaum mit dem Wurzelknoten *node* den Knoten mit dem Schlüssel *key* zurückgibt, oder *None* falls kein solcher existiert. Sie können annehmen, dass jeder Schlüssel höchstens einmal im Baum vorkommt. Sie können außerdem annehmen, dass ein *Node* die folgenden Attribute hat: *key* (der Schlüssel, ein Integer), *value* (der Wert, ein beliebiges Objekt), *left* (linkes Kind, ein *Node* oder *None*) und *right* (rechtes Kind, ein *Node* oder *None*). Die Funktion soll maximal 10 Zeilen lang sein.

```
def lookup(key, node):
    if key == node.key:
        return node
    if key < node.key and node.left:
        return lookup(key, node.left)
    elif key > node.key and node.right:
        return lookup(key, node.right)
```

4.3 (8 Punkte) Sei A ein Feld mit n Elementen, das an einer Blockgrenze anfängt. Über die n Elemente des Feldes wird in zufälliger Reihenfolge iteriert. Sei M die Größe des schnellen Speichers und B die Blockgröße. *Blockoperationen* wird im folgenden mit *BO* abgekürzt. Geben Sie für jede der folgenden Aussagen an, ob sie *wahr* oder *falsch* ist. Eine richtige Antwort gibt 2 Punkte, eine falsche Antwort gibt 2 Punkte Abzug. Wenn Sie die Antwort nicht wissen, schreiben Sie *keine Aussage* oder nichts. Sie können nicht weniger als 0 Punkte für die Aufgabe bekommen.

1. Wenn $n = 13$ und $B = M = 4$, dann werden im besten Fall nur 4 BO benötigt. Wahr: Wenn z.B. zufälligerweise der Reihe nach über die Elemente des Felds iteriert wird.

2. Wenn $n = 20$ und $B = M = 4$, dann werden im schlechtesten Fall 20 BO benötigt. Wahr. Im schlechtesten Fall muss für jedes Element ein neuer Block geladen werden.

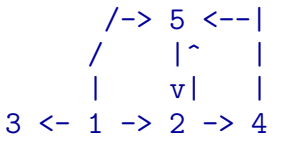
3. Wenn $n = 43$, $B = 4$, $M = 44$, dann werden im schlechtesten Fall 43 BO benötigt. Falsch. In diesem Fall passen alle Blöcke gleichzeitig in den schnellen Speicher.

4. Für alle $M \geq B$ werden im besten Fall nur $\lceil \frac{n}{B} \rceil$ BO benötigt. Wahr. Wenn immer erst über alle Elemente eines Blocks iteriert wird, dann muss jeder Block genau einmal geladen werden.

Aufgabe 5 (Heaps und Graphen, 20 Punkte)

5.1 (5 Punkte) Betrachten Sie die Adjazenzmatrix A eines gerichteten und gewichteten Graphen G. Der Eintrag 6 beschreibt zum Beispiel eine Kante von Knoten 1 zu Knoten 5 mit Kosten 6. Ein $\times$ bedeutet "keine Kante". Malen Sie den Graphen G. Führen Sie Dijkstras Algorithmus auf G genau so lange aus, bis die Kosten des kürzesten Pfades von Knoten 1 nach Knoten 4 sicher gefunden sind. Stellen Sie jede Iteration als Zeile in einer Tabelle mit sechs Spalten dar: in den ersten fünf Spalten stehen die *dist*-Werte der Knoten **vor** der Iteration, in der sechsten Spalte steht der in der Iteration gelöste Knoten.

$$A = \begin{bmatrix} \times & 3 & 0 & \times & 6 \\ \times & \times & \times & 4 & 5 \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & 1 \\ \times & 2 & \times & \times & \times \end{bmatrix}$$



	dist[1]	dist[2]	dist[3]	dist[4]	dist[5]	gelöst
1.	0	inf	inf	inf	inf	1
2.	0	3	0	inf	6	3
3.	0	3	0	inf	6	2
4.	0	3	0	7	6	5
5.	0	3	0	7*	6	4

5.2 (10 Punkte) Schreiben Sie eine Methode *reach(v: int, G: list[list[int]])* $\rightarrow$ *int*, die die Anzahl der von Knoten v aus erreichbaren Knoten in einem gerichteten Graphen G mittels Breitensuche berechnet. Die Adjazenzliste für einen Knoten v steht in *G[v]*. Ihre Methode soll in $O(|V| + |E|)$ laufen (V = Menge der Knoten, E = Menge der Kanten) und nicht mehr als 15 Zeilen haben.

```
def reach(v, G):
  cur = [v], next = [], vis = {v}
  while len(cur) != 0:
    for v in cur:
      for w in G[v]:
        if w not in vis:
          next.append(w)
          vis.add(w)
    cur, next = next, cur
    next = []
  return len(res) - 1
```

5.3 (5 Punkte) Erfüllt der durch das Feld $A = [\times, 1, 2, 20, 19, 4]$ beschriebene binäre Heap die Heap-Eigenschaft? Mit Begründung. Sei nun $B = [\times, 14, 6, 12, 13, 7]$. Führen Sie auf dem durch B beschriebenen binären Heap die Methode *repair_heap_downwards* auf dem ersten in B gespeicherten Knoten aus. Malen Sie jeden Zwischenschritt als Baum und geben Sie das Endergebnis zusätzlich als Feld an.