Prof. Dr. Ir. Joost-Pieter Katoen Christian Dehnert, Jonathan Heinen, Thomas Ströder, Sabrina von Styp

Aufgabe 1 ($\mathcal{O}$-Notation): (3 + 3 + 4 = 10 Punkte)

Beweisen oder widerlegen Sie die folgenden Aussagen:

a) $\log_2 n \in \Omega(\frac{1}{n})$

b) $2^n \in \Theta(3^n)$

c) $\sum_{i=0}^{n} i! \in \Theta(n!)$

Lösung:

Beweisen oder widerlegen Sie die folgenden Aussagen:

a) Es gilt:

$$g \in \Omega(f) \text{ gdw. } lim\ inf_{n\to\infty} \frac{g(n)}{f(n)} > 0$$

$$\lim_{n\to\infty} log_2 n \cdot n = \infty > 0$$

Da es einen Limes gibt, gibt es auch einen Limes inferior, der gleich diesem Limes ist.

q.e.d

b) Offensichtlich gilt $2^n < 3^n$ für alle $n \in \mathbb{N}$ und somit $2^n \in \mathcal{O}(3^n)$. Somit bleibt zu zeigen, dass $2^n \in \Omega(3^n)$. Es gilt:

$$2^n \in \Omega(3^n) \text{ gdw. } \exists c > 0, n_0 \text{ mit } \forall n \geq n_0 : c \cdot 3^n \leq 2^n$$

Wir betrachten nun für welches $c > 0$ die Aussage $c \cdot 3^n \leq 2^n$ gilt:

$$\begin{array}{lll} & c \cdot 3^n & \leq 2^n \quad | \ 3 = 2 \cdot 3/2 \\ \Leftrightarrow & c \cdot (3/2 \cdot 2)^n & \leq 2^n \\ \Leftrightarrow & c \cdot (3/2)^n \cdot 2^n & \leq 2^n \quad | \ /2^n \\ \Leftrightarrow & c \cdot (3/2)^n & \leq 1 \end{array}$$

Dies gilt offensichtlich für kein $c > 0$. Somit gilt $2^n \notin \Omega(3^n)$ und auch $2^n \notin \Theta(3^n)$

c) Ziehen wir den letzten Summanden aus der Summe heraus erhalten wir:

$$\sum_{i=0}^{n-1} i! + n!.$$

Dies können wir leicht wie folgt nach oben und unten abschätzen:

$$n! \quad \leq \quad \sum_{i=0}^{n-1} i! + n! \quad \leq \quad \sum_{i=0}^{n-1} (n-1)! + n! = \underbrace{(n-1)! \cdot \ldots \cdot (n-1)!}_{= n \cdot (n-1)! \ = \ n!} + n! \ = \ 2 \cdot n!$$

Da beide Abschätzungen $n!$ und $2 \cdot n!$ in $\Theta(n!)$ liegen gilt dies auch für $\sum_{i=0}^{n} i! \in \Theta(n!)$.

q.e.d

Name: **Matrikelnummer:**

Aufgabe 2 (Sortieren): (3 + 2 + 5 = 10 Punkte)

a) Sortieren Sie das folgende Array mittels des Mergesort-Algorithmus.

7	13	3	2	9	5	4	8

Geben Sie nach jeder Merge-Operation den jeweils verschmolzenen Arraybereich an.

b) Gegeben sei ein Array E, das Datenobjekte enthält, welche je zwei Schlüssel K1 und K2 besitzen. Anna sortiert das Array E mittels eines Sortieralgorithmus S zuerst nach den Schlüsseln K1. Anschließend sortiert sie diejenigen Arraybereiche, bei denen die Werte des Schlüssels K1 gleich sind, mit demselben Algorithmus S nach den Werten des zweiten Schlüssels K2. Boris sortiert das Array E mittels des Algorithmus S zuerst nach den Schlüsseln K2. Anschließend sortiert er das gesamte resultierende Array mit dem Algorithmus S nach den Schlüsseln K1.

Unter welchen Bedingungen erhalten Anna und Boris für ein beliebiges Eingabearray E immer das gleiche Ergebnis? Begründen Sie Ihre Antwort!

c) Betrachten Sie den folgenden Sortieralgorithmus `gnome`.

```
void gnome(int E[]) {
    int i = 0;
    while (i < E.length) {
        if (i == 0 || E[i-1] <= E[i]) {
            i++;
        } else {
            swap(E,i,i-1);
            i--;
        }
    }
}

void swap(int E[], int i1, int i2) {
    int store = E[i1];
    E[i1] = E[i2];
    E[i2] = store;
}
```

Bestimmen Sie die Worst-Case Laufzeit (Θ) dieses Algorithmus in Abhängigkeit der Arraylänge n und geben Sie an, ob dieser Algorithmus ein stabiler Sortieralgorithmus ist. Begründen Sie Ihre Antwort!

Lösung:

a)

2	3	4	5	7	8	9	13

2	3	7	13		4	5	8	9

7	13		2	3		5	9		4	8

b) Anna und Boris erhalten immer das gleiche Ergebnis, wenn der Sortieralgorithmus S stabil ist. Denn beide sortieren das gesamte Array zum letzten Mal nach den Schlüsseln K1. Während Anna die Einträge mit gleichen K1 Schlüsseln noch nach den K2 Schlüsseln sortiert, hat Boris dies zuvor getan. Bei einem stabilen Sortieralgorithmus bleibt diese Sortierung bei gleichen K1 Schlüsseln erhalten.

c) Die Worst-Case Laufzeit des `gnome` Algorithmus ist $\Theta(n^2)$. Die `swap` Operation hat offensichtlich konstante Laufzeit. Damit entspricht die asymptotische Laufzeit des Algorithmus genau der asymptotischen Anzahl an Schleifendurchläufen. Da die Schleife endet, sobald `i` den Wert n erreicht hat, tritt der Worst-Case ein, wenn `i` so oft wie möglich dekrementiert statt inkrementiert wird. Demnach muss so oft wie möglich die Situation eintreten, dass zwei hintereinander liegende Arrayeinträge in der falschen Reihenfolge sind. Dies ist genau dann der Fall, wenn das Array rückwärts sortiert ist. Dann muss der Algorithmus für jede Position j im Array j-mal die Schleife durchlaufen, um diese Position zu erreichen, und weitere j-mal, um den dortigen Wert zum Anfang des Arrays zurückzutauschen. Damit ergibt sich eine Laufzeit von

$$\begin{aligned}\Theta(\Sigma_{j=1}^{n} 2 \cdot j) &= \Theta(2 \cdot \Sigma_{j=1}^{n} j) \\ &= \Theta(2 \cdot \frac{n \cdot (n-1)}{2}) \\ &= \Theta(n^2).\end{aligned}$$

Der `gnome` Algorithmus ist stabil, denn es werden nur nebeneinander liegende Elemente vertauscht, die in falscher Reihenfolge sind. Gleiche Elemente können nicht vertauscht werden und bleiben damit in ihrer ursprünglichen Reihenfolge.

Name: **Matrikelnummer:**

Aufgabe 3 (Datenstruktur): (2 + 4 + 4 = 10 Punkte)

Zum effizienten Speichern einer Matrix M mit n Zeilen, m Spalten und z Nicht-Nulleinträgen verwaltet das sogenannte CRS-Format drei Arrays wie folgt:

- Das Array $values$ speichert ausschließlich die Nicht-Nulleinträge von M in fortlaufender Reihenfolge, wobei diese Werte zeilenweise von links nach rechts abgelesen werden.
- Das Array $columns$ speichert zu jedem Nicht-Nulleintrag in $values$ die zugehörige Spalte, die dieser Wert in der Matrix einnimmt.
- Das Array $rows$ speichert für jede Zeile i von M den Index des ersten Elements der Zeile im Array $values$. Um anzuzeigen, wo die Werte der letzten Zeile in $values$ enden, enthält es am Schluss noch ein zusätzliches Sentinel-Element mit dem Wert $length(values)$.

Betrachten wir nun beispielsweise die Darstellung der Matrix

$$\begin{pmatrix} 10 & 0 & 2 & 0 & 0 \\ 0 & 5 & 0 & 13 & 0 \\ 0 & 0 & 18 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{pmatrix}$$

Die entsprechenden Arrays des CRS-Formats sind gegeben durch

- $values = \begin{bmatrix} 10 & 2 & 5 & 13 & 18 & 4 & 1 & 3 \end{bmatrix}$
- $columns = \begin{bmatrix} 1 & 3 & 2 & 4 & 3 & 1 & 4 & 5 \end{bmatrix}$
- $rows = \begin{bmatrix} 0 & 2 & 4 & 5 & 7 & 8 \end{bmatrix}$

Sei M nun eine beliebige Matrix mit n Zeilen, m Spalten und z Nicht-Nulleinträgen.

a) Geben Sie die exakte Anzahl D der Dateneintrage, die das CRS-Format für M speichern muss, in Abhängigkeit der Parameter n, m und z an. Als Dateneinträge werden hierbei alle Inhalte der Arrays $values$, $columns$ und $rows$ aufgefasst.

b) Das folgende Codefragment setzt einen Wert a in der i-ten Zeile und j-ten Spalte in eine CRS-Matrix ein ($1 \leq i \leq n$ und $1 \leq j \leq m$). Es wird vereinfachend davon ausgegangen, dass hinter das Ende eines Arrays sicher geschrieben werden kann und sich dadurch die Länge des Arrays entsprechend vergrößert.

```
public static void insert(CRSMatrix M, int a, int i, int j) {
    // Bestimme den Anfangsindex der Zeilen i und i+1
    int rowIndex = M.rows[i - 1];
    int nextRowIndex = M.rows[i];

    // Finde Position in Zeile i an der eingefuegt werden muss
    int pos = rowIndex;
    while (pos < nextRowIndex && j > M.columns[pos]) {
        pos++;
    }

    // Eintrag existiert schon
    if (pos < nextRowIndex && j == M.columns[pos]) {
        M.values[pos] = a; // ueberschreibe nur den Wert
    } else { // Eintrag existierte nicht
        // verschiebe entsprechende Arrayelemente
        for (int k = M.values.length - 1; k >= pos; k--) {
            M.values[k + 1] = M.values[k];
            M.columns[k + 1] = M.columns[k];
        }

        // setze Wert und Spalte
        M.values[pos] = a;
        M.columns[pos] = j;

        // die auf i folgenden Zeilen beginnen nun einen Index spaeter
        for (int k = i; k < M.rows.length; k++) {
            M.rows[k]++;
        }
    }
}
```

Geben Sie die Laufzeitkomplexität von $insert(M, a, i, j)$ in Abhängigkeit der Zeilen n, Spalten m und Nicht-Nulleinträge z für den Best-Case als auch den Worst-Case an. Begründen Sie ihre Antwort.

c) Es soll nun ein Algorithmus entworfen werden, der für eine natürliche Zahl k das erste Vorkommen dieses Wertes bezüglich der Reihenfolge in $values$ in M bestimmt und dann ausgibt, in welcher Zeile und in welcher Spalte von M dieser Wert steht. Kommt k nicht in M vor, so soll nichts ausgegeben werden.

Beschreiben Sie **in Stichpunkten** ein möglichst effizientes Verfahren, welches die Aufgabe erfüllt, und geben Sie die resultierende Worst-Case-Laufzeitkomplexität ihres Verfahrens an. Orientieren Sie sich hierbei an bekannten Algorithmen aus der Vorlesung.

Lösung:

a) Die Arrays $values$ und $columns$ haben beide die Länge z. Das Array $rows$ besitzt Länge $n+1$. Insgesamt ergeben sich $D = 2z + n + 1$ Elemente, die gespeichert werden müssen.

b) Im besten Fall existiert die spezifizierte Position (i, j) bereits (if-Branch) und das Element in Spalte j ist das erste in Zeile i (da dann die While-Schleife direkt terminiert). In diesem Fall sind alle durchlaufenen Teile in konstanter Zeit durchführbar, die Komplexität liegt also in $\mathcal{O}(1)$.

Im schlechtesten Fall existiert die spezifizierte Position (i, j) noch nicht (else-Branch) und das Element muss nach dem letzten Nicht-Nullelement aus Zeile i in $values$ eingefügt werden. Zuerst muss in $\mathcal{O}(\min(m, z))$ Schritten die richtige Position im Array $values$ gesucht werden. Dann müssen $\mathcal{O}(z)$ Arrayeinträge in $values$ und $columns$ um eine Position weiter kopiert werden und anschließend $\mathcal{O}(n)$ Indizes in $rows$ um jeweils eins erhöht werden. Insgesamt liegt die Laufzeit also in $\mathcal{O}(\min(m, z) + z + n) = \mathcal{O}(z + n)$.

c) Ein mögliches Verfahren zur Lösung der Aufgabe könnte wie folgt aussehen:

a) Führe eine lineare Suche nach k in $values$ in $\mathcal{O}(z)$ durch.

b) Wurde k nicht gefunden, so stoppe.

c) Wurde k an Index l in $values$ gefunden, kann dessen Spalte direkt aus $columns[l]$ abgelesen werden.

d) Führe eine Art binäre Suche nach l in $rows$ durch: gilt $rows[i] \leq l < rows[i + 1]$ für einen Index i, so ist die entsprechende Zeile $(i + 1)$ gefunden. Dies ist in $\mathcal{O}(\log n)$ Schritten möglich.

Damit liegt Gesamtlaufzeit in $\mathcal{O}(z + \log n)$.

Name: **Matrikelnummer:**

Aufgabe 4 (Rekursionsgleichungen): (4 + 2 + 4 = 10 Punkte)

a) Geben Sie für das folgende Programm eine Rekursionsgleichung für die asymptotische Laufzeit des Aufrufes `berechne(n)` an.

```
int berechne(int n){
    if(n <= 0)
        return 3;

    int value = 4;
    for(int i = 0; i < n*n; i++){
        value += pow(i);
    }

    value += 3 * berechne(n/2) + 4 * berechne(n/4) + 5
    return value;
}

int pow(n){
    int value = n;
    for(int i = 0; i < n; i++)
        value = value *n;
    return value;
}
```

b) Bestimmen Sie für die folgende Rekursionsgleichung die Komplexitätsklasse Θ mit Hilfe des Mastertheorems. Begründen Sie Ihre Antwort.

$$T(n) = 4T(n/2) + n \cdot \log_2 n + 4n + 3$$

c) Skizzieren Sie den Rekursionsbaum zu der folgenden Rekursionsgleichung. Lesen Sie die Komplexitätsklasse Θ der Gleichung ab. Die abgelesene Komplexitätsklasse **muss nicht bewiesen**, **Summen nicht aufgelöst** werden. Eine Angabe der Laufzeit der Art $\Theta(\sum \ldots)$ ist zulässig.

$$T(n) = 3 \cdot T(n-2) + 2n^2, \quad T(0) = T(1) = 1$$

Lösung:

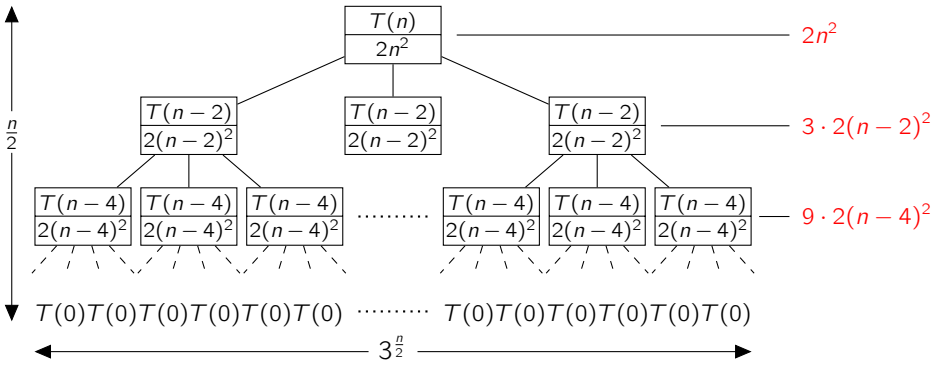
a) `pow(n)` hat lineare Laufzeit im Parameter n, da die `for`-Schleife n-fach durchlaufen wird. Jeder Aufruf von `berechne(n)` führt zu zwei rekursiven Aufrufen, einer mit $n/2$ und einer mit $n/4$. Darüber hinaus wird die `for`-Schleife n^2-fach durchlaufen. Der Aufruf von `pow(i)` erzeugt jeweils kosten linear in i. Somit ergibt sich die folgende Laufzeit für den Aufruf:

$$T(n) = T(n/2) + T(n/4) + \underbrace{\sum_{i=0}^{n^2} i}_{\in\Theta(n^4)} \quad T(0) = 1$$

b) Da $E = \log_2 4 = 2$ und $f(n) = n \cdot \log_2 n + 4n + 3 \in \mathcal{O}(n^{E-1/2}) = \mathcal{O}(n^{1.5})$ handelt es sich um den ersten Fall des Mastertheorems und es gilt:

$$T(n) \in \Theta(n^E) = \Theta(n^2)$$

c) Es ergibt sich der folgende Rekursionsbaum:



Aus dem Rekursionsbaum lässt sich die Laufzeit wie folgt ablesen:

$$T(n) \in \Theta\left(\sum_{i=0}^{n/2}(3^i \cdot 2 \cdot (n-2i)^2) + 3^{\frac{n}{2}}\right)$$

Name: **Matrikelnummer:**

Aufgabe 5 (Bäume): (2 + 1 + 3 + 1 + 3 = 10 Punkte)

Ein höhenbalancierter Baum ist ein Binärbaum, bei dem sich für jeden Knoten die Höhe seiner Teilbäume höchstens um 1 unterscheiden darf.

a) Zeichnen Sie einen höhenbalancierten Baum der Höhe 3 mit **minimaler Anzahl** an Knoten. Geben Sie für jeden Knoten die Höhe des Teilbaumes an, der an diesem Knoten beginnt.
Hinweis: Der zu zeichnende Baum hat 7 Knoten.

b) 1. Stellen Sie eine Rekursionsgleichung $E(h)$ auf, mit der sich die **minimale Anzahl** von Knoten in einem höhenbalancierten Baum der Höhe h bestimmen lässt. *Geben Sie auch die nötigen Basisfälle an.*

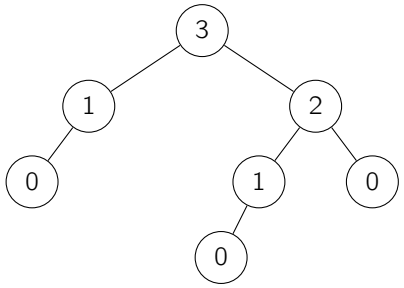
2. Beweisen Sie, dass $\frac{1}{2} \cdot (\frac{3}{2})^h + \frac{1}{2}$ eine *untere Schranke* für die Anzahl der Knoten ist.

c) Geben Sie die minimale Höhe $h(n)$ eines höhenbalancierten Baumes mit n Knoten an. Begründen Sie Ihre Antwort kurz. Ein formaler Beweis ist nicht nötig.

d) Zeigen Sie, dass für die Höhe $h(n)$ eines höhenbalancierten Baumes mit n Knoten gilt, dass $h(n) \in \mathcal{O}(\log_2 n)$. Zum Lösen dieser Aufgabe dürfen Sie die untere Schranke $\frac{1}{2} \cdot (\frac{3}{2})^h + \frac{1}{2}$ aus Aufgabe **5 b)** nutzen.

Lösung:

a) Die minimale Anzahl von Knoten in einem höhenbalancierten Baum wird erreicht, wenn für jeden Knoten der Höhenunterschied zwischen den Teilbäumen maximal, also 1 ist:



b) 1. Wie oben bereits erwähnt, wird die minimale Anzahl von Knoten genau dann erreicht, wenn für jeden Knoten der Höhenunterschied zwischen den Teilbäumen maximal, also 1 ist. Die Anzahl der Knoten setzt sich dann zusammen aus dem Wurzelknoten und der Anzahl der Knoten im linken und rechten Teilbaum. Um auf die Höhe h zu kommen muss mindestens einer der beiden Teilbäume die Höhe $h-1$ besitzen. Um die Differenz von 1 zu erhalten, muss der andere Teilbaum dann die Höhe $h-2$ besitzen.

$$E(h) = E(h-1) + E(h-2) + 1 \quad E(0) = 1, E(1) = 2$$

2. Um zu zeigen, dass $e(h) = \frac{1}{2} \cdot (\frac{3}{2})^h + \frac{1}{2}$ in der Tat eine untere Schranke für die Anzahl der Blätter ist, nutzen wir die Substitutionsmethode:

$$\begin{aligned}
E(h) &= E(h-1) + E(h-2) + 1 \\
&\geq e(h-1) + e(h-2) + 1 \\
&= 1/2 \cdot (3/2)^{h-1} + 1/2 \ + \ 1/2 \cdot (3/2)^{h-2} + 1/2 + 1 \\
&= 1/2 \cdot (2/3) \cdot (3/2)^h + 1/2 \ + \ 1/2 \cdot (4/9) \cdot (3/2)^h + 1/2 + 1 \\
&= 1/2 \cdot (2/3 + 4/9) \cdot (3/2)^h + 2 \\
&= 1/2 \cdot (6/9 + 4/9) \cdot (3/2)^h + 2 \\
&= 1/2 \cdot (10/9) \cdot (3/2)^h + 2 \\
&\geq 1/2 \cdot (3/2)^h + 1/2 \\
&= e(h)
\end{aligned}$$

Nun müssen wir noch die Basisfälle beweisen:

$$e(0) = 1/2 \cdot 1 + 1/2 = 1 \leq 1 = E(0)$$

und

$$e(1) = 1/2 \cdot 3/2 + 1/2 = 3/4 + 1/2 = 5/4 \leq 2 = E(1)$$

Somit gilt $E(h) > e(h)$ für alle $h \in \mathbb{N}$.

c) Die minimale Höhe eines höhenbalancierten Baumes mit n Elementen wird erreicht, wenn der Baum nahezu vollständig ist, d.h. wenn sich die Länge aller Pfade von der Wurzel zu jedem Blatt maximal um eins unterscheiden. Dies entspricht den Bäumen, die wir als Heaps genutzt haben, die eine Höhe von $\lfloor \log_2 n \rfloor$ besitzen. Somit gilt $h(n) \geq \lfloor \log_2 n \rfloor$.

d) Wir stellen die oben gegebene Formel $e(h) = \frac{1}{2} \cdot (\frac{3}{2})^h + \frac{1}{2}$ nach h um und erhalten:

$$\begin{array}{lll} & n & \geq \frac{1}{2} \cdot (\frac{3}{2})^h + \frac{1}{2} \\ \Leftrightarrow & n - \frac{1}{2} & \geq \frac{1}{2} \cdot (\frac{3}{2})^h \\ \Leftrightarrow & 2n - 1 & \geq (\frac{3}{2})^h \\ \Leftrightarrow & \log_{\frac{3}{2}}(2n-1) & \geq h \end{array}$$

Somit gilt für die Höhe $h(n) \leq \log_{\frac{3}{2}}(2n-1) \in \mathcal{O}(\log_2 n)$.