Alexander Neuwirth (439218)
Leonhard Segger (440145)
Jonathan Sigrist (441760)

Aufgabe 15.

(a) (i) $f(n) = n$ und $g(n) = n^2$.

(ii) $f(n) \notin \Omega(g(n))$ bedeutet, dass $f(n)$ langsamer wächst als $g(n)$. $f(n) \notin \mathrm{O}(g(n))$ bedeutet, dass $f(n)$ schneller wächst als $g(n)$. Beide Bedingungen sind nicht gleichzeitig erfüllbar.

(iii) $f(n)$ kann nicht gleichzeitig genau so schnell und langsamer als $g(n)$ wachsen.

(iv) $f(n)$ kann nicht gleichzeitig genau so schnell und schneller als $g(n)$ wachsen.

(v) $f(n) = n^2$ und $g(n) = n$.

(vi) $f(n) = n^2$ und $g(n) = n$.

(b) (i) Nach dem ersten Mastertheorem ergibt sich für $T(n) = 4T(n/2) + n$, dass $a = 4$, $b = 2$ und $f(n) = n$.
Also folgt $n^{log_b a} = n^{log_2 4} = n^2$ und es muss $f(n) \in O(n^{log_2 4 - \varepsilon})$ gelten, also $0 < \varepsilon \leq 1$.
Es gilt also $T(n) \in \Theta(n^2)$.

(ii) Für $T(n) = 4T(n/2) + n^2$ ergibt sich, dass $a = 4$, $b = 2$ und $f(n) = n^2$.
Es folgt $n^{log_b a} = n^{log_2 4} = n^2$ und da $f(n) \in \Theta(n^2)$ gilt, folgt nach dem zweiten Mastertheorem $T(n) \in \Theta(n^2 log n)$.

(iii) Für $T(n) = 4T(n/2) + n^3$, gilt $a = 4$, $b = 2$ und $f(n) = n^3$.
Somit folgt $n^{log_a b} = n^2$. Da $f(n) \in \Omega(n^{2+\varepsilon})$ für $0 < \varepsilon \leq 1$ und $a \cdot f(n/b) \leq c \cdot f(n)$, also $4/8 \cdot n^3 \leq cn^3$ für $0.5 \leq c < 1$ erfüllt ist, folgt nach dem dritten Mastertheorem $T(n) \in \Theta(f(n))$.

(iv) $\log_2(n) + 2(\log_2(n^{0.5}) + 2(\log_2(n^{0.25}) + 2T(n^{0.125})))$
$n^{0.5^k} = 2 \quad (\sqrt{2} = 1)$
$k = \log_2(\log_2(n))$
$\Rightarrow \log_2(n) \cdot (\log_2(\log_2(n)) + 1)$

(v) $2 \cdot (2 \cdot (2 \cdot T(n/8) + n/4\ \log_2(n/4)) + n/2\ \log_2(n/2)) + n \log(n)$
$= \sum_{i=1}^{\log_2(n)} n \log_2(n/(2^i)) + n \log_2(n)$
$= n \log_2(n) \cdot (1 + \log_2(n))/2$

Aufgabe 16.

Die Informatiker verwenden das Prinzip der binären Suche. Dabei werden die ersten n Schubladen verwendet. Ein Informatiker soll dabei immer die niedrigste Schublade finden und einen Stein reinlegen. Falls er die n-te Schublade befüllt, so ist er der letzte Kandidat.
Das heißt, das der k-te Informatiker zuerst die mittlere $(n/2)$ Schublade öffnet und dann je nachdem ob bereits ein Stein in dieser liegt den Bereich auf $[1, n/2 - 1]$ oder $[n/2 + 1, 0]$ eingrenzt. Er ist fertig, wenn nur noch eine einzige Schublade übrig bleibt, also $[k, k]$ und legt dort einen Stein rein. Die Methode entspricht also der binären Suche und stimmt so auch mit dessen Komplexität und Korrektheit überein, erfüllen also die Aufgabenstellung.

Aufgabe 17.

Alexander Neuwirth (439218)
Leonhard Segger (440145)
Jonathan Sigrist (441760)

(a) Die Methode gibt zurück, ob das Array ein gleiches Element wie `x` enthält, also `k.compareTo(x)` auf irgendeinem Element k aus dem Array `true` zurückgibt.

(b) Der worst-case ist erreicht, wenn das gesammte Array kein gleiches Element wie `x` enthält. Dann wird rekursiv das Array per divide-and-conquer-Verfahren aufgeteilt und jedes Element wird genau einmal überprüft. Also ist die Laufzeitkomplexität $O(n)$.

(c) Das betrachtete Interval kann nicht direkt betrachtet werden. Falls das Interval die größe 2 erreicht, wird das erste Element mit `x` verglichen. Also wird `array[r]` nie aufgerufen und somit

- niemals `array[array.length]` aufgerufen, was das Programm zum Absturz bringen würde.
- kein Element mehrfach ausgewertet, da bei der Unterteilung die Mitte einmal als obere und einmal als untere Grenze auftritt. Somit wird also effektiv immer nur $[l, r-1]$, bzw. $[l, m-1]$ und $[m, r-1]$ ausgewertet.

Dieses verhalten zieht sich durch alle Stufen des divide-and-conquer-Verfahrens.
Falls das Array ein Element wie x entält, so ist es entweder in der ersten oder der zweiten Hälfte. Sobald die Intervalgröße 2 erreicht, wird der Inhalt verglichen.

Aufgabe 18.

(a) Paarweise verschieden heißt, dass keine zwei Zahlen gleich sind. Die geforderte 5×7 Matrix sähe zum Beispiel wie folgt aus:

$$\begin{pmatrix} 01 & 08 & 15 & 22 & 29 \\ 02 & 09 & 16 & 23 & 30 \\ 03 & 10 & 17 & 24 & 31 \\ 04 & 11 & 18 & 25 & 32 \\ 05 & 12 & 19 & 26 & 33 \\ 06 & 13 & 20 & 27 & 34 \\ 07 & 14 & 21 & 28 & 35 \end{pmatrix}$$

(b) Die oben beschriebene Matrix kann offensichtlich mit nur einem Vergleich $x \leq mn$ in $O(1)$ feststellen, ob x in dieser ist.
Eine beliebige Matrix kann man durch das divide-and-conquer-Verfahren erst Spalten und dann Zeilen ausfündig machen, welche die Zahl beinhalten könnten. Da der erste Eintrag einer Zeile oder Spalte jeweils der kleinste dieser ist, brauchen bei `x < array[i][j]` keine Zellen `array[k][l]` mit $k \geq i$ und $l \geq j$ getestet zu werden. Analog kann man kleinste Schranken für Spalten und Zeilen mit dem letzten Element dieser finden, sodass letzlich eine $g \times h$ Untermatrix entsteht. Bei großen Matrizen kann dieser Schritt den folgenden Schritt erheblich beschleunigen, muss aber nicht verwendet werden. Die Laufzeit zum ersten sortieren beträgt $O(2 \log m + 2 \log n)$
In der Untermatrix sucht man jetzt mit einem binären Suchalgorithmus in jeder Zeile oder Spalte, ob die gesuchte Zahl x vorkommt. Je nachdem ob g oder h kleiner ist, wählt man die Richtung aus. Die Laufzeit beträgt also $O(g \log h)$ bzw. $O(h \log g)$. Dabei kommt man nicht

Alexander Neuwirth (439218)
Leonhard Segger (440145)
Jonathan Sigrist (441760)

an die gewünschte Komplexität $\Theta(m+n)$.
Alternativ kann man auch jedes Element einzeln untersuchen und die Komplexität ist dann $\Theta(mn)$.

(d) Falls x nicht im Array enthalten ist, erreicht der Zähler irgendwann die letzte Spalte und terminiert danach. Falls in einer Spalte ein Element gleich x sein sollte, so wird diese Spalte durch die Iterierung um eins immer getestet. Da auch dieser Test mit eins iteriert, wird garantiert dieses Element verglichen.
Eine Spalte kann übersprungen werden, wenn $x <$`a[0][j]` bzw. $x >$`a[m][j]`,
da `a[0][j]`$< \cdots <$`a[m][j]`.