Lösungsskizzen zur Prüfung

(1)

(a) $T(n) = O(1) + 2T(\frac{n}{2}) = O(1) + 2O(1) + 4T(\frac{n}{4}) =$
$= O(1) + 2O(1) + \ldots + O(\frac{n}{2^k}) =$
$O(1) \cdot \underbrace{\sum_{i=0}^{k(=ld\ n)} 2^i}_{2^{k+1}-1=2n-1} = O(n)$

(b) $T(n) = O(n) + T(\frac{n}{2}) = O(n) + O(\frac{n}{2}) + T(\frac{n}{4}) = \ldots =$
$= O(n) + O(\frac{n}{2}) + O(\frac{n}{4}) + O(\frac{n}{8}) + \ldots = \underbrace{\sum_{i=0}^{ld\ n} \frac{1}{2^i}}_{\leq 2} \cdot O(n) = O(n)$

(c) $T(n) = O(n\ log\ n) + 2T(\frac{n}{2}) =$
$= O(n\ log\ n) + 2O(\frac{n}{2}\ log\frac{n}{2}) + 4T(\frac{n}{4}) = \ldots =$
$= O(n\ log\ n) + 2O(\frac{n}{2}\ log\frac{n}{2}) + 4O(\frac{n}{4}\ log\ \frac{n}{4}) + \ldots \leq$
$\leq \underbrace{O(n\ log\ n) + O(n\ log\ n) + O(n\ log\ n) + \ldots}_{O(log\ n) mal} = O(n\ log^2\ n)$

(d) $T(n) = O(n^2) + 2T(\frac{n}{2}) = O(n^2) + 2O(\frac{n^2}{4}) + 4T(\frac{n}{4}) = \ldots =$
$= O(n^2) + 2O(\frac{n^2}{4}) + 4O(\frac{n^2}{16}) + 8O(\frac{n^2}{64}) + \ldots =$
$= O(n^2) + O(\frac{n^2}{2}) + O(\frac{n^2}{4}) + O(\frac{n^2}{8}) + \ldots =$
$= O(n^2) \cdot \underbrace{\sum_{i=0}^{ld\ n} \frac{1}{2^i}}_{\leq 2} = O(n^2)$

(e) $T(n) = O(\sqrt{n}) + T(\sqrt{n}) = O(n^{\frac{1}{2}}) + O(n^{\frac{1}{4}}) + T(n^{\frac{1}{4}}) \leq$
$\leq O(n^{\frac{1}{2}}) + O(\frac{n^{\frac{1}{2}}}{2}) + O(\frac{n^{\frac{1}{2}}}{4}) + \cdots + O(1) \leq$
$\leq O(\sqrt{n}) \cdot \underbrace{\sum_{i=0}^{\infty} \frac{1}{2^i}}_{\leq 2} = O(\sqrt{n})$

(2) Prinzipiell gibt es einen Zusammenhang zwischen Rechenzeit und Speicherverbrauch eines Algorithmus: Um eine Speicherzelle zu beschreiben braucht ein Algorithmus eine bestimmte Zeit $O(1)$. Daraus folgt auch, daß ein Algorithmus es z.B. in $O(n)$ Zeit nicht schaffen kann $\Omega(n^2)$ Speicher zu benutzen. Dementsprechend kann bei den folgenden Punkten argumentiert werden.

(a) Richtig. Es ist möglich, daß ein Algorithmus in dieser Zeit $O(n\ log\ n)$ Speicher bearbeitet. $O(n\ log\ n)$ heißt ja auch, daß es weniger Speicher sein kann, also z.B. $\Theta(1)$.

(b) Richtig. Es wird $\Theta(n^2)$ Speicher benötigt, $\Omega(n\ log\ n)$ stellt jedoch nur eine untere Schranke für die Zeit dar. Das heißt, daß der Algorithmus durchaus $\Theta(n^2)$ oder noch mehr Zeit brauchen kann.

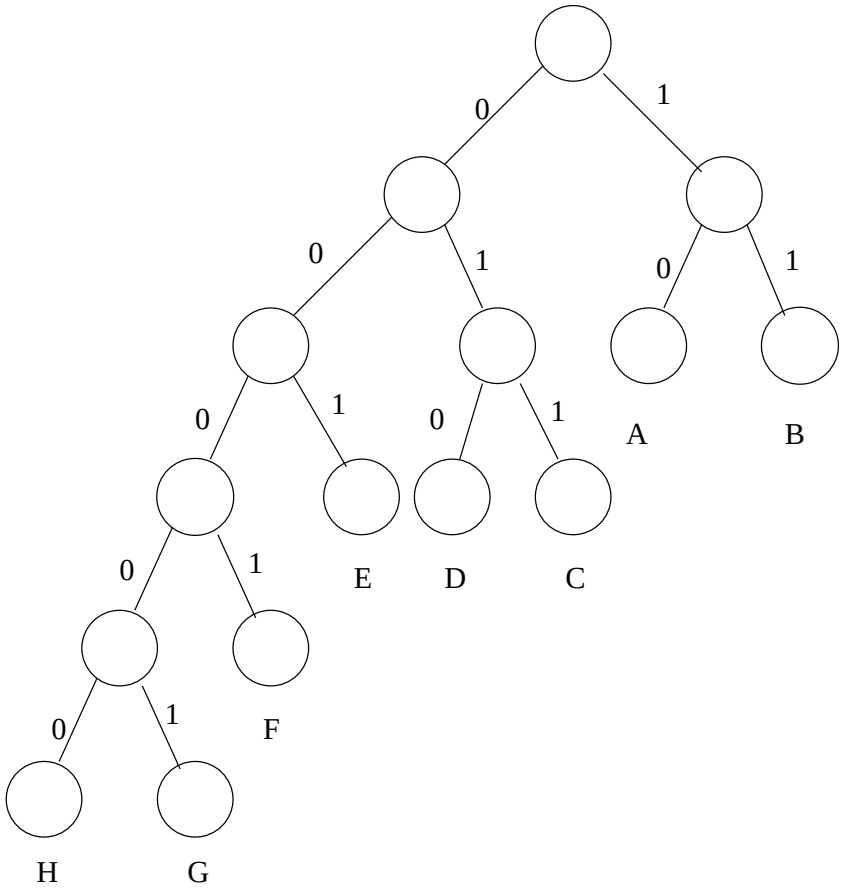
(c) Richtig. Da $O(n^3)$ nur eine obere Grenze darstellt erfüllt z.B. jeder in der Vorlesung vorgestellt Algorithmus dieses Kriterium.

(d) Falsch. Durch den Speicherbedarf allein, kann keine Aussage über die maximale Zeit gegeben werden. Ein Algorithmus den eine Zeit von $\Omega(42^n)$ braucht, um eine essentielle Frage zu beantworten, könnte genauso gut nur $O(1)$ Speicher verwenden, gleichzeitig kann ein Algorithmus mit einer Laufzeit von $O(n^3)$ eben diesen Speicher von $\Omega(n^3)$ verarbeiten.

(e) Falsch. Wie in den einleitenden Worten gesagt, kann der Algorithmus pro Zeiteinheit nur eine Speicherstelle verwenden, d.h. wenn er maximal $O(n\ log\ n)$ Zeit braucht (obere Schranke!), dann kann er nicht in dieser Zeit mindesten $\Omega(n^2)$ (untere Schranke!) Speicher verwenden.

(3) Der nachstehende Codebaum, zeigt den Zusammenhang zwischen der Codierung und den relativen Häufigkeiten.

Aufgrund der Vorschrift für den optimalen Code, müssen immer die Elemente mit den geringsten Auftrittswahrscheinlichkeiten zusammengefasst werden. Eine mögliche relative Häufigkeit ist z.B. einfach $p_i = \frac{1}{2^t}$ zu wählen, wobei t die Tiefe des Elementes bzw. die Länge des Codes des Zeichens ist. Damit ist auch $\sum_i p_i = 1$ garantiert. Es ergibt sich also: $p(A) = p(B) = \frac{1}{4}$, $p(C) = p(D) = p(E) = \frac{1}{8}$, $p(F) = \frac{1}{16}$, $p(G) = p(H) = \frac{1}{32}$.



Bemerkung: Es gibt natürlich unendlich viele verschiedene, korrekte Häufigkeiten. Und auch die angegebenen Häufigkeiten würden andere optimale Codes zulassen.

(4) Alle n Zeilen der Matrix werden mit einem beliebigen Sortierverfahren – dessen worst-case Verhalten $O(n\ log\ n)$ ist – aufsteigend sortiert. Dies benötigt $n \cdot O(n\ log\ n) = O(n^2\ log\ n)$ Zeit. Nun wird für jede Zahl der ersten Zeile in allen anderen Zeilen mittels Binärsuche überprüft, ob diese dort vorhanden ist. Dabei wird die Suche nach einer bestimmten Zahl abgebrochen, sobald in einer Zeile diese nicht gefunden werden konnte und mit der nächsten fortgesetzt. Konnte eine Zahl in allen Zeilen gefunden werden, gibt das Programm eine positive Meldung aus und wird beendet.

Dieser Algorithmus arbeitet offensichtlich korrekt: jede Zahl der ersten Zeile wird in jeder anderen Zeile gesucht. Ohne Sortierung würde dieses Verfahren $O(n^3)$ Zeit benötigen. Durch die Sortierung kann statt linearer Suche jedoch die Binärsuche verwendet werden, wodurch sich für den Überprüfungsprozess die Zeit wie folgt errechnet: (Anzahl der Elemente in der ersten Zeile) * (Anzahl der Zeilen in denen gesucht wird) * (Zeit für die Suche) $= n \cdot (n-1) \cdot log\ n = O(n^2\ log\ n)$.

Die Gesamtlaufzeit ist somit die Zeit für die Sortierung plus die Zeit für die Überprüfung: $O(n^2\ log\ n) + O(n^2\ log\ n) = O(n^2\ log\ n)$.