Lösungsskizzen zur Prüfung

(1)

(a) Siehe Skriptum Seite 8.

(b) D.h. gesucht ist eine Funktion die asymptotisch zwischen $O(n)$ und $O(n\ log\ n)$ liegt. Funktionen die diesen Anspruch erfüllen gibt es viele, z.B. $O(n\ log\ log\ n)$, $O(n\sqrt{log\ n})$, ...

(c) Die beiden sind identische Funktionsklassen!
Beweis: $n^{ln\ 5} = e^{ln\ n^{ln\ 5}} = e^{(ln\ 5)\cdot(ln\ n)} = e^{(ln\ n)\cdot(ln\ 5)} = e^{ln\ 5^{ln\ n}} = 5^{ln\ n}$

(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.

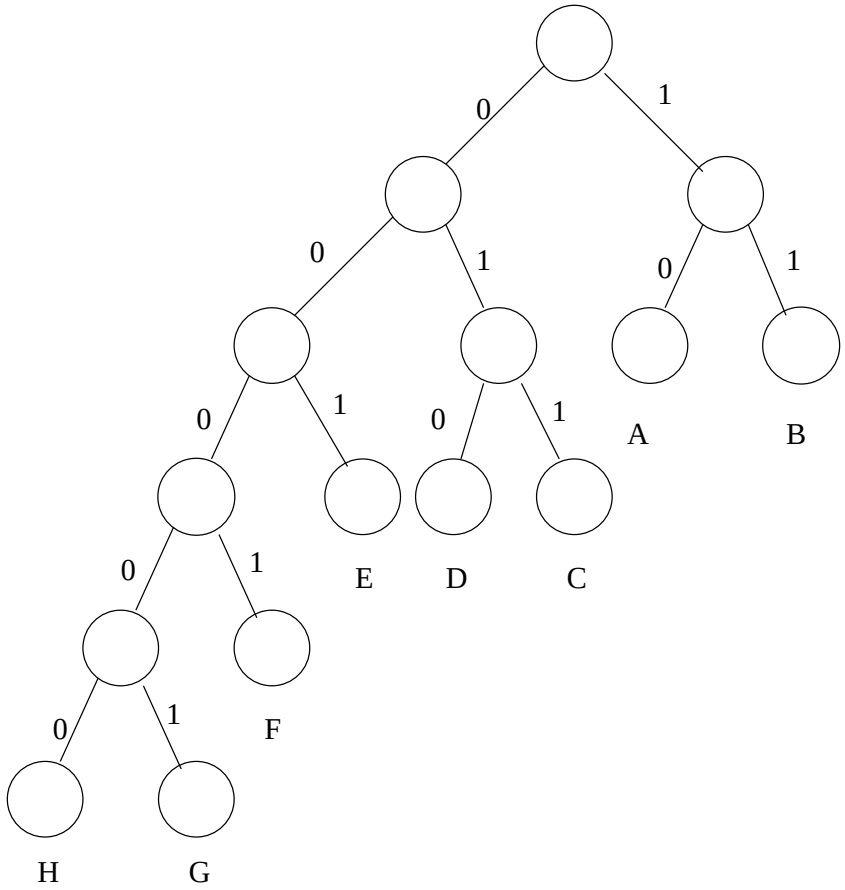
(c) Richtig. Da $O(n^3)$ nur eine obere Grenze darstellt erfüllt z.B. jeder in der Vorlesung vorgestellt Algorithmus dieses Kriterium.

(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) Für das Problem x in der Matrix A zu finden, gibt es mehrere Möglichkeiten:

* Die 'billigste' in $O(n^2)$: Zwei verschachtelte FOR-Schleifen. Benötigt werden zumindest zwei Zählvariablen, aber maximal konstanter Speicher: $S(1)$.
* Die 'einfache' in $O(n\ log\ n)$: Man nehme eine FOR-Schleife über alle Spalten (Zeilen) und wende pro Zeile (Spalte) einen Suchalgorithmus z.b. die Binärsuche an. Zeitverhalten: $T(n) = n \cdot O(log\ n) = O(n\ log\ n)$; min. eine Zählvariable, max. konstanter Speicher: $S(1)$
* Die 'lineare' Lösung entsteht aus folgender Beobachtung: Ist $A[i,j] > x$, so sind durch die Sortierung der Zeilen und Spalten auch alle Elemente unterhalb und rechts von $A[i,j]$ grösser als x:

$$A[i,j] > x \Rightarrow A[k,l] > x,\ i < k \leq n,\ j < l \leq n.$$

Analoges folgt aus $A[i,j] < x$. Der Algorithmus sieht wie folgt aus:

(a) (1) Starte mit dem Element $A[1,n]$.
(b) (2) $j \leftarrow j-1$ bis $A[i,j] \leq x$ (alle Spalten rechts von $A[i,j]$ sind dann grösser als x und müssen nicht mehr berücksichtigt werden). Ist $A[i,j] = x$ sind wir fertig.
(c) (3) $i \leftarrow i+1$ bis $A[i,j] \geq x$ (alle Spalten über $A[i,j]$ sind dann kleiner als x und müssen nicht mehr berücksichtigt werden.) Ist $A[i,j] = x$ sind wir fertig.
(d) (4) Wiederhole die Schritte (2) und (3) bis entweder ein $A[i,j] = x$ gefunden wurde oder $i \leq 0$ oder $j \leq 0$ ist (x wurde dann nicht gefunden).

Nachdem immer nur j um 1 dekrementiert, bzw. i um eins inkremtentiert wird und dies solang bis eine der beiden Variablen 0 ist, kann im schlimmsten Fall $2 \cdot n$ mal verglichen bzw. gerechnet werden: Laufzeit $O(n)$, Speicher: 2 Variablen $S(1)$.