Matthias Krause
Alexander Moch

Mannheim, den 12.10.2020

CS 307 Algorithmen und Datenstrukturen, Herbstsemester 2020

Lösungsskizze zum Übungsblatt 3

AUFGABE 3.1: Der folgende Algorithmus (in Pseudocode) findet das Minimum der Einträge in einem (unsortierten) Array $A[1 \cdot \cdot n]$ $(n > 1)$.

algorithm *MINIMUM* (A, n)
1 min := $A[1]$;
2 **for** $i := 2$ **to** n
3   **if** (min $> A[i]$)
4   min := $A[i]$;
5 **return** min;

Für die average-case Analysen in dieser Aufgabe soll angenommen werden, dass jede Permutation von n verschiedenen Zahlen $b_1, \ldots, b_n$ mit der gleichen Wahrscheinlichkeit (nämlich $\frac{1}{n!}$) als Belegung von $A[1], \ldots, A[n]$ vorkommen kann.
Wie oft wird der Test in Zeile 3 in worst case, best case und average case ausgeführt? Kann ein anderer Algorithmus zur Bestimmung des Minimums in A mit weniger Vergleichen in worst, best oder average case auskommen?
$n - 1$ *Vergleiche in allen Fällen. Dies kann nicht unterboten werden, da jedes Element außer das Minimum mindestens einen Vergleich „verlieren“ muss.*

AUFGABE 3.2: In vielen Anwendungen taucht das folgende modifizierte Sortierproblem auf: Es sind n Objekte (hier vereinfachend Integers) $a_1, \ldots, a_n$ in einem Array $A[1 \cdot \cdot n]$ gegeben. Es soll auf der Basis einer geeigneten Datenstuktur eine Routine entwickelt werden, die wiederholt aufgerufen werden kann und bei dem i-ten Aufruf, $i = 1, \ldots, n$, die i-tgrößte Zahl (genauer $a_{\pi(i)}$ für eine Permutation π mit $a_{\pi(1)} \geq \ldots \geq a_{\pi(n)}$) liefert. Die Zahl der Aufrufe k ist a priori nicht bekannt. Es soll allerdings von der Tatsache, dass k wesentlich kleiner als n sein kann, Nutzen gezogen werden; es ist also nicht ratsam, die Zahlen vorzusortieren und dann einfach eine nach der anderen auszugeben.

a) Wie würden Sie dieses Problem mit Hilfe eines Heaps lösen? (Hier und in der Teilaufgabe (b) reicht eine grobe Beschreibung der wesentlichen Schritte und deren worst-case Laufzeit; es soll kein Pseudocode angegeben werden).

Zuerst wird ein (Max-)Heap aufgebaut (Zeit $O(n)$). Danach wird bei jedem Aufruf die Operation ExtractMax ausgeführt (Zeit pro Aufruf: $O(\log n)$).

b) Wie würden Sie dieses Problem mit Hilfe eines Partitionierungsansatzes, wie er bei Quicksort verwendet wird, lösen?

(Es wird zuerst das rechte Teilarray bearbeitet.) Nach einem Aufruf von PARTITION(A, p, r) wird $(p, q - 1)$ auf den Stack gelegt und das Teilarray mit Indizes $(q + 1, r)$ weiter bearbeitet. Falls man mit dem aktuellen Teilarray fertig ist, so wird ein Paar (p', r') dem Stack entnommen und mit dem entsprechenden Teilarray weiter gemacht. Es bietet sich an, das Array stückweise zu sortieren: Falls i sortierte

Elemente vorliegen sollen, kann man aufhören, wenn $p' > i$ gilt. Falls später mehr Elemente benötigt werden, kann man mit dem Stack in seinem letzten Zustand weiter machen.
Im worst case kann schon der erste Aufruf (Bestimmung des Maximums) Zeit $\Theta(n^2)$ benötigen (und zwar, wenn PARTITION immer p zurückliefert). Allerdings taucht diese Situation bei einer „guten“ Implementierung von Quicksort kaum auf; empirisch hat sich diese Methode oft sogar als wesentlich schneller erwiesen als die Methode unter (a).

AUFGABE 3.3$^{\mathbf{K}}$: In dieser Aufgabe betrachten wir die Ausführung des in der Vorlesung beschriebenen Algorithmus HEAPSORT auf das Feld $A = [5, 2, 4, 3, 13, 8, 9, 11, 7, 6]$.

a) Geben Sie den Inhalt von A an, nachdem der Algorithmus zum ersten Mal einen (Max-) Heap daraus erstellt hat.

$[13, 11, 9, 7, 6, 8, 4, 3, 5, 2]$

b) Geben Sie jeweils den (vollständigen) Inhalt von A an, jedes Mal wenn der Algorithmus eine weitere HEAPIFY ausgeführt hat.

$[11, 7, 9, 5, 6, 8, 4, 3, 2, 13]$
$[9, 7, 8, 5, 6, 2, 4, 3, 11, 13]$
$[8, 7, 4, 5, 6, 2, 3, 9, 11, 13]$
$[7, 6, 4, 5, 3, 2, 8, 9, 11, 13]$
$[6, 5, 4, 2, 3, 7, 8, 9, 11, 13]$
$[5, 3, 4, 2, 6, 7, 8, 9, 11, 13]$
$[4, 3, 2, 5, 6, 7, 8, 9, 11, 13]$
$[3, 2, 4, 5, 6, 7, 8, 9, 11, 13]$
$[2, 3, 4, 5, 6, 7, 8, 9, 11, 13]$

AUFGABE 3.4$^{\mathbf{K}}$: Geben Sie für die unten aufgeführten Sortieralgorithmen die asymptotische Laufzeit in Θ-Form (also z.B. $\Theta(n^2)$) zum Sortieren eines n-elementigen Feldes mit dem angegebenen Inhalt an.

a) Insertionsort auf $[1, 2, 3, \ldots, n - \lfloor\sqrt{n}\rfloor, n, n-1, n-2, \ldots, n - \lfloor\sqrt{n}\rfloor + 1]$

$\Theta(n)$

b) Mergesort auf $[2, 1, 4, 3, \ldots, n, n-1]$

$\Theta(n \log n)$

c) Quicksort auf $[1, 2, 3, \ldots, \lfloor\frac{n}{2}\rfloor, n, n-1, n-2, \ldots, \lfloor\frac{n}{2}\rfloor + 1]$

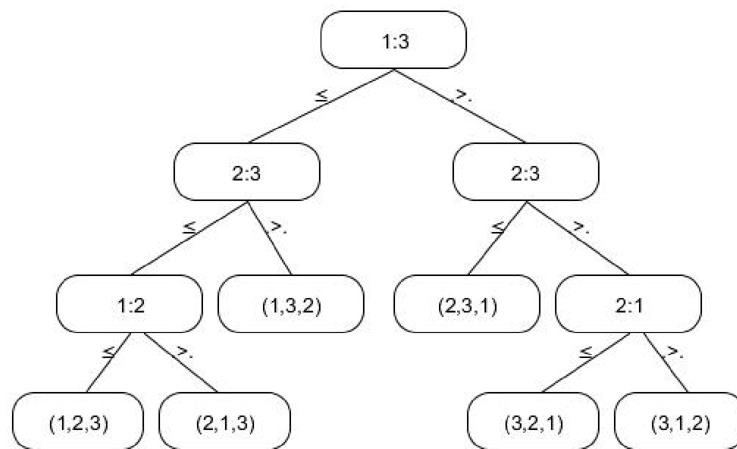
$\Theta(n^2)$

d) Heapsort auf $[n, n-1, n-2, \ldots, 3, 2, 1]$

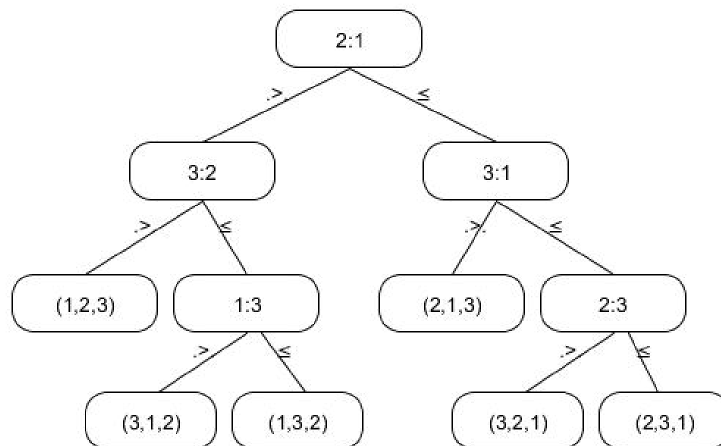
$\Theta(n \log n)$

AUFGABE 3.5:
Zeichnen Sie den Entscheidungsbaum $T(n)$ für

a) Quicksort und $n = 3$,



b) Heapsort und $n = 3$.



AUFGABE 3.6$^{\mathbf{K}}$:

a) Gegeben ist das folgende Feld $A[1 \cdot \cdot 10]$, in dem Schlüssel aus der Menge $\{0, \ldots, 999\}$ gespeichert sind. Benutzen Sie den in der Vorlesung vorgestellten Algorithmus *Radix Sort*, um das Feld in aufsteigender Reihenfolge zu sortieren. Tragen Sie dabei die Ergebnisse der Zwischenschritte in nachfolgendes Schema ein!

129	Schritt 1 ⇒	450	Schritt 2 ⇒	111	Schritt 3 ⇒	58
944		111		318		111
318		322		322		129
322		922		922		318
450		944		129		322
111		745		944		397
397		397		745		450
745		318		450		745
58		58		58		922
922		129		397		944

b) Wir betrachten nun das allgemeine Schema für *Radix Sort*. Es soll ein Feld mit n Elementen sortiert werden. Die Schlüssel seien dabei d-stellige Zahlen mit Ziffern aus $\{0, \ldots, k-1\}$. Im Teil (a) gilt also beispielsweise $n = 10$, $d = 3$, $k = 10$.
In der Vorlesung wurde dargestellt, dass *Radix Sort* für jeden Zwischenschritt ein geeignetes Sortierverfahren verwenden kann. Geben Sie an, welche der nachfolgend aufgeführten (aus der Vorlesung bekannten) Algorithmen (ohne Modifizierung) für eine Implementierung von *Radix Sort* in Frage kommen. Geben Sie für die geeigneten Algorithmen die resultierende (worst case) Laufzeit von *Radix Sort* (in Abhängigkeit von n, d und k) in der O-Notation an.

Jedes stabile Sortierverfahren ist geeignet.

	Geeignet? (ja/nein)	Falls geeignet: Laufzeit Radix Sort
Insertion Sort	ja	$O(d \cdot n^2)$
Merge Sort	ja	$O(d \cdot n \cdot \log(n))$
Quick Sort	nein	-
Heap Sort	nein	-
Counting Sort	ja	$O(d \cdot (n + k))$