Lösungen von Übungsblatt 8

Algorithmen (WS 2018, Ulrike von Luxburg)

Lösungen zu Aufgabe 1

a) Für Heapsort betrachten wir das Array $A = [3_0, 3_1, 3_2]$. Der Index im Subskript gibt die initiale Position des jeweiligen Schlüssels an. Es ergeben sich folgende Zwischenschritte

(i) $[3_2, 3_1, 3_0]$

(ii) $[3_1, 3_2, 3_0]$.

Das letzte Array stellt das Endergebnis dar. Somit ist Heapsort nicht stabil. Für Selection Sort betrachten wir das Array $B = [3_0, 3_1, 2]$. In der in-place Variante wird 3_0 mit 2 getauscht und wir erhalten:

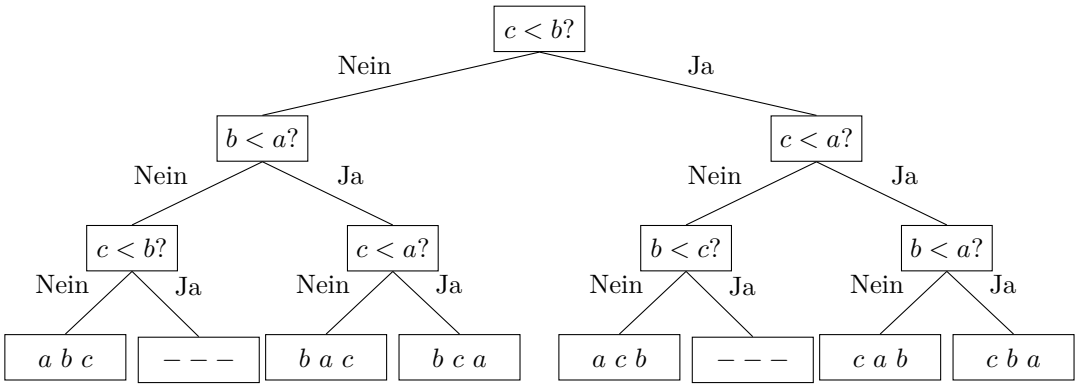
(i) $[2, 3_1, 3_0]$

Somit ist auch Selection Sort instabil.

b) Um Sortierverfahren zu stabilisieren, bekommt jeder Schlüssel ein Feld zugewiesen, das seine initiale Position speichert. Wandle dann die Vergleichsoperation so ab, dass im Falle gleicher Elementwerte als Zweitkriterium aufsteigend nach der Indexposition sortiert wird. Dadurch stehen letztlich alle Elemente zu denselben Werten in ihrer originalen Reihenfolge.

Lösungen zu Aufgabe 2

a)



b) Wir wollen den Algorithmus abbrechen, sobald in einem Durchgang der äußeren Schleife in der inneren Schleife keine Vertauschung stattgefunden hat, denn dann ist das Array offensichtlich bereits sortiert. Die best-case Laufzeit $\mathcal{O}(n)$ wird dann auf der geordneten Eingabe [1 2 ... n] erzielt.

c)

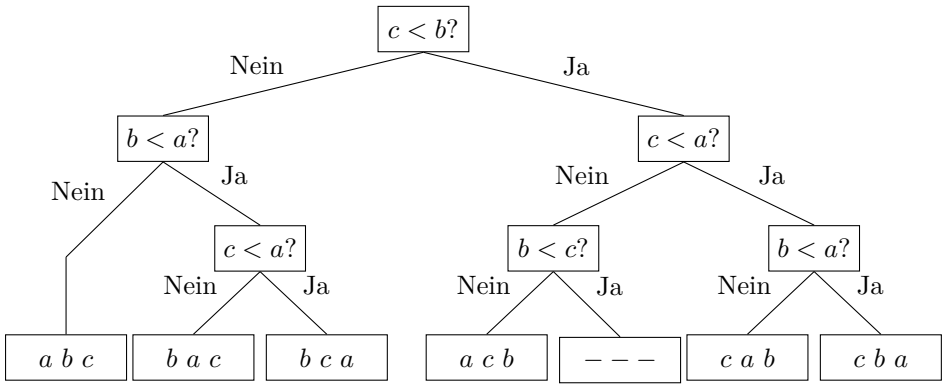
```
1:  function BubblesortAdapt(A)
2:      for i = 1 to A.length − 1 do
3:          swapped=FALSE;
4:          for j = A.length downto i + 1 do
5:              if A[j] < A[j − 1] then
6:                  exchange A[j] with A[j − 1]
7:                  swapped=TRUE;
8:              end if
9:          end for
10:         if swapped==FALSE then
11:             return A
12:         end if
```

13: **end for**
14: **return** A
15: **end function**

d) Sei n die Länge des Eingabe-Arrays. Im worst-case einer absteigend sortierten Folge mit paarweise verschiedenen Einträgen wird zusätzlich zu den Kosten des BUBBLESORT-Algorithmus aus der Vorlesung $n-1$ mal eine Zuweisung FALSE gemacht (Zeile 3), $\sum_{i=1}^{n-1}(n-i) = n(n-1)/2$ mal eine Zuweisung TRUE gemacht (Zeile 7) sowie $n-1$ mal eine if-Abfrage durchgeführt (Zeile 10). Sowohl die Zuweisungen als auch die if-Abfragen zählen als je eine Elementaroperation (und benötigen je konstante Laufzeit $\Theta(1)$), der Mehraufwand beträgt also $2(n-1)+n(n-1)/2$ Elementaroperationen (bzw. $\Theta(n^2)$ Laufzeit).

Dieser Mehraufwand lohnt sich aber: wenn die Eingabe bereits sortiert oder auch nur vorsortiert ist, wird die Laufzeit deutlich reduziert. Man kann auch folgendes Resultat zeigen: Unter Annahme einer Gleichverteilung auf den $n!$ Permutationen von $[1 \ \ldots \ n]$ beträgt die erwartete Anzahl an Vergleichen $\frac{1}{2}(n^2 - n\ln n - (\gamma + \ln 2 - 1)\cdot n) + \mathcal{O}(\sqrt{n})$ im Gegensatz zu einer (deterministischen) Anzahl von $n(n-1)/2$ Vergleichen in der Version der Vorlesung ($\gamma \approx 0.58$ bezeichnet hierin die Euler-Mascheroni-Konstante).

e)



Im Vergleich zum Baum aus (a) hat dieser Baum einen inneren Knoten weniger (der einem neuerlichen Vergleich $c < b$? im Falle einer sortierten Eingabe entspricht). Im Baum aus (a) haben alle Blätter gleiche Höhe ($3 \cdot (3-1)/2 = 3$), in diesem Baum nicht mehr (das spiegelt die Reduktion der Laufzeit bei gewissen Eingaben wider).

Lösungen zu Aufgabe 3

a) Das Finden des Medians einer Eingabesequenz der Länge ℓ benötigt nach Aufgabenstellung $\mathcal{O}(\ell)$ Zeitaufwand. Dieses Pivotelement halbiert die Eingabesequenz in zwei Teile der Länge $\ell/2$, auf welchen dann rekursiv fortgefahren wird. Im worst-case taucht das Pivotelement zudem nur einmal in der Eingabesequenz auf. Die Laufzeit auf Eingabegröße n ist somit $T(n) \leq 2T(\lceil n/2 \rceil) + \mathcal{O}(n)$, wobei in $\mathcal{O}(n)$ sowohl das Finden des Pivotelement, das Zerlegen in die Teilsequenzen, als auch das Zusammensetzen der Teillösungen zur Gesamtlösung enthalten ist. Mit $T(1) = \Theta(1)$ und dem Master-Theorem folgt, dass $T(n) = \mathcal{O}(n \log n)$ die worst-case Laufzeit dieser Quicksort-Variante ist. Asymptotisch im worst-case ist diese Variante also besser als der randomisierte Quicksort oder andere Quicksort-Varianten mit worst-case Laufzeit $\mathcal{O}(n^2)$. Zudem ist $\mathcal{O}(n \log n)$ scharf, da auch diese Quicksort-Variante vergleichsbasiert ist, und somit $\Omega(n \log n)$ nicht unterbieten kann.

Die Laufzeit-Konstanten zum Finden des Medians in jedem Zwischenschritt sind schlichtweg zu groß, so dass in der Praxis auf den meisten Eingaben der randomisierte Quicksort derart schneller ist, dass die worst-case Zusicherung des Median-Pivots irrelevant ist. Wenn man wirklich $\mathcal{O}(n \log n)$ zusichern will, kann man ggf. besser Merge-Sort wählen.

b) Nein, da das arithmetische Mittel nicht wie der Median unabhängig von einer starken Unbalancierung der Eingabedaten ist. Betrachte beispielsweise die Eingabe $(n, n^2, n^3, \ldots, n^n)$ der Länge

n. Das arithmetische Mittel all dieser Zahlen ist $\overline{a}_n = \frac{1}{n}\sum_{i=1}^{n} n^i = \sum_{i=0}^{n-1} n^i$ und ist offensichtlich größer als n^{n-1}. Damit wird bestenfalls das vorletzte Element des Arrays als Pivot gewählt. Dies teilt die Eingabe in

$$\underbrace{(n, n^2, n^3, \ldots, n^n - 2)}_{a}\underbrace{(n^{n-1})}_{b}\underbrace{(n^n)}_{c},$$

also insbesondere eine Sequenz a der Länge mindestens $n-2$.

Nun wird rekursiv auf Sequenz a der Länge k fortgefahren, also auf der Eingabesequenz $(n, n^2, n^3, \ldots, n^k)$. Dessen arithmetisches Mittel ist ganz analog zu oben $\overline{a}_k = \frac{1}{n}\sum_{i=1}^{k} n^i = \sum_{i=0}^{k-1} n^i$ und offensichtlich größer als n^{k-1}. Wieder teilt sich die Eingabe bestenfalls in eine Sequenz der Länge $k-2$, die Pivotsequenz b der Länge 1 und die Sequenz c der Länge 1.

Insgesamt wird also mindestens $n/2$ Mal rekursiv abgestiegen. Da die Ermittlung des arithmetischen Mittels einer Eingabesequenz der Länge k mindestens Zeit $c \cdot k$ für eine Konstante $c \geq 1$ benötigt, fällt somit mindestens Aufwand $cn + c(n-2) + c(n-4) + \ldots + 2c = c\sum_{i=1}^{n/2} 2i \in \Omega(n^2)$ an. Damit erhalten wir die Gesamtlaufzeit $\Omega(n^2)$.

Lösungen zu Aufgabe 4

a) $[6, 21, 14, 19, 15, 18, 20]$ als Heap ist $[21, 19, 20, 6, 15, 14, 18]$

1. Tauschen: $[18, 19, 20, 6, 15, 14, 21]$
2. MaxHeapify: $[20, 19, 18, 6, 15, 14], 21$
3. Tausch: $[14, 19, 18, 6, 15, 20], 21$
4. MaxHeapify: $[19, 15, 18, 6, 14], 20, 21$
5. Tauschen: $[14, 15, 18, 6, 19], 20, 21$
6. MaxHeapify: $[18, 15, 14, 6], 19, 20, 21$
7. Tauschen: $[6, 15, 14, 18], 19, 20, 21$
8. MaxHeapify: $[15, 6, 14], 18, 19, 20, 21$
9. Tauschen: $[14, 6, 15], 18, 19, 20, 21$
10. MaxHeapify: $[14, 6], 15, 18, 19, 20, 21$
11. Tauschen: $[6, 14], 15, 18, 19, 20, 21$
12. MaxHeapify: $[6], 14, 15, 18, 19, 20, 21$
13. Ergebnis: $6, 14, 15, 18, 19, 20, 21$

b) 1. Aufgabe: Hier gibt es mehrere Möglichkeiten, eine richtige Lösung ist:

$$\begin{pmatrix} 3 & 4 & 5 & 9 \\ 5 & 8 & 14 & \infty \\ 9 & 12 & \infty & \infty \\ 16 & \infty & \infty & \infty \end{pmatrix}$$

2. Aufgabe

```
function EXTRACT-MIN(M)
    x = M(1,1)
    M(1,1) = ∞
    YOUNGIFY(M,1,1)
    return x
end function
function YOUNGIFY(M, i, j)
    smallest_i = i
    smallest_j = j
    if i + 1 ≤ m and M(i,j) > M(i+1,j) then
        smallest_i = i + 1
```

```
    end if
    if j + 1 ≤ n and M(smallest_i, j) > M(i, j + 1) then
        smallest_i = i
        smallest_j = j + 1
    end if
    if smallest_i ≠ i or smallest_j ≠ j then
        SWAP(M(i, j), M(smallest_i, smallest_j))
        YOUNGIFY(M, smallest_i, smallest_j)
    end if
end function
```

Der gegebene Algorithmus ist korrekt, da man in jedem Rekursionsschritt eine kleinere Submatrix M' der ursprünglichen Matrix M betrachtet. Diese Submatrix hat jedes mal wieder die Eigenschaften der Ursprungsmatrix. Dies stellt sicher, dass das nächst-kleinste Element entweder in $M'(1,2)$ (sofern wir nicht am rechten Rand sind) oder in $M'(2,1)$ (sofern wir nicht am unteren Rand sind) zu finden ist. Dieses muss an die Stelle $M'(1,1)$ der Submatrix wandern. Abgebrochen wird, wenn das Ende der Matrix $M'(m,n)$ erreicht ist oder an den Stellen $M'(1,2)$ und $M'(2,1)$ jeweils ∞ steht. (Erneut unter der Berücksichtigung eventuell am Rand der Matrix zu sein.) Im schlimmsten Fall wandert unser ∞ von der Position $M(1,1)$ ganz bis zum Ende der Matrix $M(1,1)$ ganz bis zum Ende der Matrix $M'(m,n)$ und benötigt demnach $O(m+n)$.