Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Ulf Leser
E. Angriman, A. Ermshaus, F. Nelles, M. Sänger

Berlin, den 17.05.2021

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 3

Abgabe: Montag, den 31.05.2021, bis 11:10 Uhr über Moodle. Die Übungsblätter sind in Gruppen von zwei (in Ausnahmefällen drei) Personen zu bearbeiten. Die Lösungen sind auf nach Aufgaben getrennten **PDFs** über Moodle abzugeben. Alle Teilaufgaben einer Aufgabe sind in einem PDF hochzuladen. Pro Gruppe genügt es, wenn eine Person die Lösung der Gruppe abgibt. Zur Bewertung wird der zuletzt hochgeladene Stand herangezogen. Vermerken Sie auf allen Abgaben Ihre **Namen**, Ihre **CMS-Benutzernamen** und Ihre **Abgabegruppe (z.B. AG123)** aus Moodle. Benennen Sie die hochgeladenen PDF-Dateien nach dem Schema *A<Aufgabe>-<Person1>-<Person2>.pdf*, bspw. A03-Musterfrau-Beispiel.pdf für Aufgabe 3 von Lisa Musterfrau und Mark Beispiel. Die Auflistung der Namen kann in beliebiger Reihenfolge erfolgen.
Beachten Sie die Informationen im Moodle-Kurs (https://hu.berlin/algodat21).

Konventionen:

- Für ein Array A ist $|A|$ die Länge von A, also die Anzahl der Elemente in A. Die Indizierung aller Arrays beginnt bei 1 (und endet also bei $|A|$). Bitte beginnen Sie die Indizierung der Arrays in Ihren Lösungen auch bei 1.
- Mit der Aufforderung "Analysieren Sie die Laufzeit" ist gemeint, dass Sie eine möglichst gute obere Schranke der (Worst Case) Zeitkomplexität angeben und diese begründen sollen.
- Die Menge der natürlichen Zahlen $\mathbb{N}$ enthält die Zahl 0.

Aufgabe 1 (Schreibtischtests) — 4+4+4=12 Punkte

In den folgenden Unteraufgaben sollen Sie jeweils einen *Schreibtischtest* durchführen. Das heißt, Sie führen auf Papier einen gegebenen Algorithmus für gegebene Eingaben aus. In der Unteraufgabe ist jeweils angegeben, welche Zwischenergebnisse Sie als Lösung einreichen sollen. Notieren Sie ein Array entweder als Liste in eckigen Klammern (also in der Form $[a_1, \ldots, a_n]$) oder wie in den VL-Folien als Tabelle der Form $\boxed{a_1 \mid \ldots \mid a_n}$.

a) Führen Sie einen Schreibtischtest für den Algorithmus **QuickSort** aus der VL für das Eingabe-Array $A = [5, 20, 18, 7, 10, 3, 16, 12]$ durch, wobei Sie als Sortierreihenfolge die natürliche Ordnung auf natürlichen Zahlen annehmen. Als Pivot-Element wählen Sie stets das am weitesten rechts stehende Element des aktuellen Teil-Arrays. Geben Sie den aktuellen Wert von A nach jeder Swap-Operation an. Unterstreichen Sie jeweils das in diesem Aufruf betrachtete Teil-Array.

b) Führen Sie einen Schreibtischtest für den Algorithmus **MergeSort** aus der VL für das Eingabe-Array $A = [\mathrm{f}, \mathrm{h}, \mathrm{m}, \mathrm{k}, \mathrm{g}, \mathrm{x}, \mathrm{e}, \mathrm{t}, \mathrm{j}]$ durch, wobei Sie als Sortierreihenfolge die alphabetische Ordnung auf den Buchstaben annehmen. Geben Sie die Zwischenergebnisse nach jedem Aufruf von MergeSort bzw. Merge in Form eines Graphen wie auf den VL-Folien an (siehe z.B. Folie 8).

c) Führen Sie einen Schreibtischtest für den Algorithmus **BucketSort** aus der VL für das Alphabet $\Sigma = \{0, 1, 2, 3\}$, $m = 3$, und das Eingabe-Array

$$A = [012, 123, 022, 030, 111, 112, 003, 102, 020]$$

durch. Wir nehmen weiterhin an, dass das Alphabet Σ linear geordnet ist, und dass die Sortierreihenfolge auf den Zeichenketten die *lexikographische* Ordnung ist. Geben Sie die Zwischenergebnisse wie auf den Vorlesungsfolien an (siehe Folie 23).

Aufgabe 2 (Sortierung spezieller Arrays) 3+3+3+3+3=15 Punkte

Ein *allgemeines Sortierverfahren* ist ein Sortierverfahren, welches die zu sortierenden Elemente nur vergleichen kann und sonst keinerlei Eigenschaften der Elemente ausnutzt. Beweisen oder widerlegen Sie für jede der folgenden Teilaufgaben a)-e) getrennt: Es gibt ein allgemeines Sortierverfahren, welches ein Array A von n beliebigen (in konstanter Zeit vergleichbaren) Elementen im Worst Case in Laufzeit $\mathcal{O}(n)$ sortiert, falls...

a) ...50% aller Elemente im Array gleich sind.

b) ...die erste Hälfte des Array bereits in aufsteigender und die zweite Hälfte des Array bereits in absteigender Reihenfolge sortiert sind.

c) ...die Länge von A durch 10 teilbar ist, und jeder Bereich $[i * 10 + 1,\ (i + 1) * 10]$ für $0 \leq i \leq n/10 - 1$ in sich aufsteigend sortiert ist.

d) ...für jedes $1 \leq i \leq n - 10$ gilt: $A[i] \leq A[i + 10]$.

e) ...in A nur höchstens $m \in \mathbb{N}_{>0}$ viele unterschiedliche Elemente vorkommen. Hierbei sei m eine Konstante und $m < |a|$.

Hinweis: Für den Fall, dass es ein solches allgemeines Sortierverfahren gibt, können Sie als Beweis die Idee eines konkreten Verfahrens beschreiben. Sie dürfen beliebig viel zusätzlichen Speicherplatz verwenden. Sie dürfen stetes auf die in der VL bewiesene untere Schranke für allgemeine Sortierverfahren verweisen.

Aufgabe 3 (Untere Schranke für `merge`) 8 Punkte

Gegeben seien zwei aufsteigend sortierte Arrays X und Y mit jeweils n Schlüsseln. Beweisen Sie: Lässt man als einzige Operation Vergleiche von je zwei Schlüsseln zu, so benötigt jeder deterministische Algorithmus im Worst Case mindestens $2n - 1$ Vergleiche, um beide Arrays zu einem aufsteigend sortierten Gesamtarray Z der Länge $2n$ zu verschmelzen.

Hinweis: Beweisen Sie per Widerspruch: Überlegen Sie sich zunächst eine Instanz, in der beim `Merge` möglichst viele Vergleiche anfallen. Zeigen Sie für diese Instanz dann, dass jeder der Vergleiche nötig ist, da sich der Algorithmus sonst für diese Instanz und eine leicht modifizierte Instanz exakt gleich verhält und der Algorithmus somit nicht korrekt sein kann.

Aufgabe 4 (Implementierung von Sortieralgorithmen) 6+6+3=15 Punkte

In dieser Aufgabe sollen Sie die Sortieralgorithmen Quick Sort und RadixESort in Java implementieren. Die Sortieralgorithmen werden auf Java `int`-Arrays angewendet. Sie können davon ausgehen, dass alle Zahlen der übergebenene Arrays nichtnegativ sind und jedes Array immer aus mindestens einem Element ($|A| \geq 1$) besteht.

Ergänzen Sie den fehlenden Code in der Vorlage `Sorting.java`, welche Sie im Moodle-Kurs vorfinden. Bei Ihrer Implementierung dürfen Sie keine zusätzlichen komplexen Datenstrukturen (bspw. Arrays oder Listen) verwenden. Sie können jedoch weitere Variablen von primitiven Datentypen sowie eigene Hilfsmethoden hinzufügen.

1. Implementieren Sie die Methode `sort()` in der privaten Klasse `Quicksort`: Die Methode nimmt als Eingabe ein `int`-Array A und zwei Ganzzahlen f und r entgegen und sortiert das übergebene Array rekursiv innerhalb des Intervalls $[\texttt{f}, \ldots, \texttt{r})$ mit Hilfe des Quick Sort Algorthmus. Beachten Sie, dass die rechte Grenze r nicht Teil des zu sortierenden Intervalls ist, d.h. $A[r]$ soll beim Aufruf `sort(a, f, r)` nicht sortiert werden. Verwenden Sie dass Element ganz rechts im Intervall als Pivotelement. Die Implementierung des Algorithmus soll dabei in-place erfolgen, d.h. Sie dürfen nur konstant viel zusätzlichen Speicher verwenden.

2. Implementieren Sie die Methode `sort()` in der privaten Klasse `RadixEsort`: Die Methode nimmt als Eingabe ein `int`-Array A und drei Ganzzahlen f r, i entgegen und sortiert die Elemente im übergebenen Array innerhalb des Intervalls $[\texttt{f}, \ldots, \texttt{r})$ nach dem i-ten Bit rekursiv mittels des RadixESort Algorithmus.

   Sei $e[i]$ das i-te Bit des Elements e. Der Aufruf `sort(A, f, r, i)` ordnet die Elemente von A innerhalb des Intervalls $[\texttt{f}, \ldots, \texttt{r})$ so um, dass alle Elemente e mit $e[i] = 0$ auf die linke und alle Elemente e mit $e[i] = 1$ auf die rechte Seite verschoben werden. Anschließend werden die beiden Teile des Arrays rekursiv, anhand des $(i-1)$-ten Bits, sortiert. Beachten Sie, dass (wie zuvor) die rechte Grenze r nicht Teil des zu sortierenden Intervalls ist.

   Hinweis: Eine `int`-Variable in Java hat eine Größe von 32 Bits. Die Bitpositionen werden von rechts und bei 0 startend indiziert. Das höchstwertige Bit ist für das Vorzeichen reserviert, d.h. Ihre Implementierung sollte mit den Bits am Index 30 beginnen. Um die Belegung des i-ten Bits einer `int`-Variable e zu ermitteln, kann folgende Bedingung formuliert werden:

```
if (((e >> i) & 1) == 0) {
    // Das i-te Bit von 'e' ist 0
} else {
    // Das i-te Bit von 'e' ist 1
}
```

3. Beschreiben Sie informell, wie Sie Ihre Implementierung des RadixESort Algorithmus ändern müssten, um auch negative Zahlen berücksichtigen zu können.

Stellen Sie sicher, dass alle Testfälle in der `main`-Methode der Datei Sorting.java erfolgreich durchlaufen. Achten Sie außerdem auf Randbedingungen und Spezialfälle, die von den Testfällen vielleicht nicht vollständig abgedeckt werden.

Hinweis: Ihr Java-Programm muss auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, sodass die einzelnen Schritte nachvollziehbar sind.