Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Marius Kloft
Bux, Grußien, Schäfer, Tschorsch, Völlinger

Berlin, den 30.05.2016

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 4

Abgabe: Montag den 13.06.2016 bis 10:55 Uhr vor der Vorlesung im Hörsaal oder bis 10:45 Uhr im Briefkasten (RUD 25, Raum 3.321). Die Übungsblätter sind in Gruppen von zwei (in Ausnahmefällen auch drei) Personen zu bearbeiten. **Die Lösungen sind auf nach Aufgaben getrennten Blättern abzugeben. Heften Sie bitte die zu einer Aufgabe gehörenden Blätter vor der Abgabe zusammen.** Vermerken Sie auf allen Abgaben Ihre Namen, Ihre Matrikelnummern, den Namen Ihrer Goya-Gruppe und an welchem Übungstermin (Wochentag, Uhrzeit, Dozent) Sie die korrigierten Abgaben abholen möchten. Beachten Sie auch die aktuellen Hinweise auf der Übungswebsite unter https://hu.berlin/algodat16.

Konventionen:

- Für ein Array A ist $|A|$ die Länge von A, also die Anzahl der Elemente in A. Die Indizierung aller Arrays auf diesem Blatt 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 Zeitkomplexität angeben sollen und diese begründen sollen.
- $\mathbb{N}_{>0}$ bezeichnet die Menge der positiven natürlichen Zahlen.

Aufgabe 1 (Schreibtischtests) — 5 + 4 + 4 = 13 Punkte

In den folgenden Teilaufgaben sollen Sie jeweils einen *Schreibtischtest* für ein Sortierverfahren durchführen. Das heißt, Sie führen auf Papier einen gegebenen Algorithmus für gegebene Eingaben aus. In der jeweiligen Teilaufgabe steht, 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 (Folie 24) für das Eingabe-Array $A = [2, 10, 6, 7, 13, 4, 1, 12, 5, 9]$ durch, wobei Sie als Ordnung die natürliche Ordnung auf den natürlichen Zahlen annehmen. Als Pivot-Element wählen Sie das am weitesten rechts stehende Element des aktuellen Teil-Arrays. Geben Sie den aktuellen Wert von A nach jeder Swap-Operation (Folie 25, Zeilen 14, 17) an. Unterstreichen Sie jeweils das in diesem Aufruf betrachtete Teil-Array.

b) Führen Sie einen Schreibtischtest für den Algorithmus **Mergesort** aus der VL (Folie 7) für das Eingabe-Array $A = [\mathrm{x}, \mathrm{a}, \mathrm{b}, \mathrm{o}, \mathrm{k}, \mathrm{j}, \mathrm{c}, \mathrm{r}, \mathrm{g}]$ durch, wobei Sie als Ordnung die alphabetische Ordnung auf den Buchstaben annehmen. Geben Sie die Zwischenergebnisse in Form eines Graphen wie auf VL-Folie 7 an.

c) In dieser Teilaufgabe betrachten wir den Algorithmus **Bucketsort**. Dabei stammen die Einträge des Arrays immer aus einer Menge von Zeichenketten gleicher Länge über einem endlichen Alphabet Σ. Wir nehmen also an, dass die Einträge eines Arrays alles Elemente von Σ^m für eine bestimmte Zahl $m \in \mathbb{N}_{>0}$ sind, wobei Σ^m wie üblich die Menge aller Zeichenketten über Σ der Länge m ist. Wir nehmen weiterhin an, dass das Alphabet Σ linear geordnet ist, und dass die Ordnung auf den Zeichenketten die *lexikographische*

Ordnung ist. Es gilt also: $a_1 \dots a_m < b_1 \dots b_m$ g.d.w. ein i mit $1 \leq i \leq m$ existiert, so dass $a_i < b_i$ und $a_j = b_j$ für alle $j < i$.

Betrachten Sie den folgenden Pseudocode für den in der VL informal eingeführten Algorithmus **Bucketsort**(A, m):

Algorithmus Bucketsort(A, m)

Input: Array A mit Einträgen aus Σ^m
Output: Sortiertes Array A

```
1:  B := Array leerer Queues mit |B| = |Σ|;
2:  for i := m downto 1 do      # die m-te Stelle ist die Stelle ganz rechts
3:    for j := 1 to |A| do
4:      a := A[j];
5:      k := findBucket(a, i);
6:      B[k].enqueue(a);
7:    end for
8:    j := 1;
9:    for k := 1 to |B| do
10:     while not B[k].empty() do
11:       A[j] := B[k].dequeue();
12:       j := j + 1;
13:     end while
14:   end for
15: end for
16: return A;
```

Die Funktion **findBucket** gibt für ein $a = a_1 \dots a_m \in \Sigma^m$ und eine Stelle $1 \leq i \leq m$ den Index des für a_i vorgesehenen Buckets zurück: Ist a_i der k-te Buchstabe in Σ, gibt **findBucket**(a, i) den Index k zurück.

Ein Bucket wird dabei als (FIFO-)Queue umgesetzt. Die Operationen einer Queue sind wie üblich **enqueue**(a) zum Einfügen eines Elements a, **dequeue**() zum Auslesen und Entfernen eines Elements und **isEmpty**() zur Überprüfung, ob die Queue leer ist.

Führen Sie einen Schreibtischtest für $\Sigma = \{0, 1, 2, 3\}$, $m = 3$, und Array

$$A = [103, 202, 101, 231, 022, 031, 030, 233, 201]$$

durch. Notieren Sie nach jedem Durchlauf der Schleife mit Laufvariable i das Array A als A_i. Markieren Sie wie in den VL-Folien mit vertikalen Strichen, welche Elemente von A_i sich in dieser Iteration in einem Bucket befanden.

Aufgabe 2 (Sortieren in Linearzeit) **$5 \cdot 2 + 4 \cdot 2 = 18$ Punkte**

Betrachten Sie für diese Aufgabe die am Ende von Aufgabenteil (a) gegebenen Sortierverfahren **Linearsort**, **Bubblesort** und **Genericsort**.

(a)
1. Erklären Sie, was bei **Linearsort** nach der ersten und nach der zweiten **for**-Schleife im Array B steht.
2. Analysieren Sie die Laufzeit von **Linearsort** in Abhängigkeit von n und z. Geben Sie also eine möglichst gute obere Schranke der Form $\mathcal{O}(f)$ an, wobei f nur von n und z abhängt.
3. Zeigen Sie, dass für jede Konstante $c \in \mathbb{N}_{>0}$ **Linearsort** n Zahlen im Bereich von 1 bis cn in $\mathcal{O}(n)$ Schritten sortiert.
4. Sei $l > 1$ eine beliebige ganzzahlige Konstante. Zeigen Sie, dass **Bubblesort** zum Sortieren einer n-elementigen Folge, welche genau l unterschiedliche Werte besitzt, im "worst case" $\Omega(n^2)$ Vertauschungen (Zeile 5-7) benötigt.
5. Analysieren Sie die Laufzeit von **Genericsort** mit **Bubblesort** als Subroutine X.

Linearsort(Array A)

Input: Array A von n ganzen Zahlen im Bereich von 1 bis z.
Output: Array A aufsteigend sortiert.

```
1:  Initialisiere Array B der Länge z, welches überall auf 0 gesetzt ist
2:  n ← |A|
3:  Initialisiere Array C der Länge n
4:  for i = 1 to n do
5:      B[A[i]] ← B[A[i]] + 1
6:  end for
7:  for j = 2 to z do
8:      B[j] ← B[j] + B[j − 1]
9:  end for
10: for i = n downto 1 do
11:     C[B[A[i]]] ← A[i]
12:     B[A[i]] ← B[A[i]] − 1
13: end for
14: return C
```

Bubblesort(Array A)

Input: Array A von n ganzen Zahlen.
Output: Array A aufsteigend sortiert.

```
1:  repeat
2:      swapped := false
3:      for i := 1 to n − 1 do
4:          if A[i] > A[i + 1] then
5:              temp := A[i]
6:              A[i] := A[i + 1]
7:              A[i + 1] := temp
8:              swapped := true
9:          end if
10:     end for
11: until not swapped
```

Sei A ein Array mit n ganzen Dezimalzahlen, welche alle genau k Stellen haben. Sei $A[i]_j$ die j-te Stelle des Elements $A[i]$. Wir sagen, dass A *nach Stelle j* aufsteigend sortiert ist, wenn $A[1]_j \leq A[2]_j \leq \cdots \leq A[n]_j$ gilt.

Genericsort(Array A)

Input: Array A von ganzen Dezimalzahlen mit jeweils k Stellen.
Output: Array A aufsteigend sortiert.

```
1: for j := k downto 1 do          # die k-te Stelle ist die Stelle ganz rechts
2:     A ← A nach Stelle j mit Sortierverfahren X aufsteigend sortiert
3: end for
4: return A
```

(b) Im Folgenden verwenden wir den Begriff eines *stabilen Sortierverfahrens*. Hierfür betrachten wir Arrays mit Einträgen des abstrakten Datentyps Element. Grundlage dafür bilden eine Menge $\mathbb{K}$ von *Schlüsseln*, eine Menge $\mathbb{V}$ von *Werten* und eine lineare Ordnung $\leq$ auf $\mathbb{K}$. Ein Element e hat einen *Schlüssel* $e.\text{key} \in \mathbb{K}$ und einen *Wert* $e.\text{val} \in \mathbb{V}$. Wir notieren ein Element e auch als Paar $(e.\text{key}, e.\text{val})$. Ist beispielsweise $\mathbb{K} = \{\text{a}, \text{b}\}$ und $\mathbb{V} = \mathbb{N}$, dann schreiben wir $(\text{a}, 17)$ für ein Element e mit $e.\text{key} = \text{a}$ und $e.\text{val} = 17$.

Sind e_1 und e_2 vom Typ Element, können wir e_1 und e_2 auf Basis ihrer Schlüssel vergleichen: Es gelte genau dann $e_1 \leq e_2$, wenn $e_1.\text{key} \leq e_2.\text{key}$. Ist beispielsweise $\mathbb{K} = \mathbb{N}$ und $\mathbb{V} = \{a, b, c\}$ und $\leq$ die natürliche Ordnung auf $\mathbb{N}$, dann gilt $(2, c) \leq (4, a)$.

Ein Sortierverfahren heißt *stabil*, wenn Elemente mit gleichen Schlüsseln nach der Sortierung in der gleichen Reihenfolge aufeinander folgen wie vor der Sortierung.

Die Sortierverfahren aus (a) lassen sich auf Arrays A mit Einträgen des Typs Element übertragen, bei denen die Schlüsselmenge $\mathbb{K}$ den ganzen Zahlen (im Bereich von 1 bis z) bzw. den ganzen Dezimalzahlen mit genau k Stellen entspricht.

1. Ist **Bubblesort** ein stabiles Sortierverfahren? Beweisen Sie Ihre Antwort.
2. Zeigen Sie, dass **Linearsort** ein stabiles Sortierverfahren ist.
3. Ist Linearsort immer noch stabil, falls die dritte **for**-Schleife von 1 bis n (aufsteigend) läuft? Begründen Sie ihre Antwort.
4. Zeigen Sie mittels vollständiger Induktion, dass das Sortierverfahren **Genericsort** korrekt ist, falls Sortierverfahren X korrekt und stabil ist.

Aufgabe 3 (Sortierung spezieller Arrays) **2 + 2 + 2 + 2 + 2 + 3 = 13 Punkte**

Beweisen oder widerlegen Sie: Es gibt ein Sortierverfahren, welches ein Array von n beliebigen (in konstanter Zeit vergleichbaren) Elementen im "worst case" in Laufzeit $\mathcal{O}(n)$ sortiert, falls...

(a) ... das Array so vorsortiert ist, dass alle Elemente an einer Position mit geradem Index untereinander bereits aufsteigend sortiert sind und alle Element an einer Position mit ungeradem Index untereinander bereits absteigend sortiert sind.

(b) ... ein Element im Array mehr als $\frac{n}{2}$-mal vorkommt.

(c) ... im Array nur Zahlen von 1 bis n stehen.

(d) ... 99% aller Elemente im Array gleich sind.

(e) ... das Array so vorsortiert ist, dass alle Elemente der ersten Hälfte des Arrays kleiner sind als alle Elemente der zweiten Hälfte.

(f) ... im Array nur höchstens $k \in \mathbb{N}_{>0}$ viele unterschiedliche Elemente vorkommen. Hierbei sei k eine Konstante.

Hinweis: Für den Fall, dass es ein solches Sortierverfahren gibt, können Sie als Beweis die Idee eines konkreten Verfahrens beschreiben.

Aufgabe 4 (Untere Schranke für merge) **6 Punkte**

Gegeben seien zwei aufsteigend sortierte Arrays X und Y mit jeweils n in konstanter Zeit vergleichbaren Elementen.

Zeigen Sie: Lässt man als einzige Operation Vergleiche von je zwei Elementen zu, so benötigt jeder deterministische Algorithmus im schlechtesten Fall mindestens $2n-1$ Vergleiche, um beide Arrays zu einem aufsteigend sortierten Gesamtarray Z der Länge $2n$ zu verschmelzen.

*Hinweis: Überlegen Sie sich zunächst eine Instanz, in der beim Verschmelzen (***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.*