Übungen zur Vorlesung

Algorithmen und Datenstrukturen

Übungsblatt 03

Cryptoplexity
Cryptography & Complexity Theory
Technische Universität Darmstadt
www.cryptoplexity.de

Arbeitsgruppe Kryptographie und Komplexitätstheorie
Prof. Dr. Marc Fischlin
Moritz Huppert
Tobias Schmalz

Sommersemester 2024
Veröffentlicht: 03.05.2024, 14:00 Uhr MESZ
Letzte Änderung: 06.05.2024, 17:08 Uhr MESZ
Abgabe: 17.05.2024, 23:59 Uhr MESZ

G1 Gruppendiskussion

Nehmen Sie sich etwas Zeit, um die folgenden Fachbegriffe in einer Kleingruppe zu besprechen, sodass Sie anschließend in der Lage sind, die Begriffe dem Rest der Übungsgruppe zu erklären:

(a) Divide-and-Conquer Paradigma;
(b) Sortieralgorithmen InsertionSort, MergeSort und QuickSort;
(c) Mastermethode zur Laufzeitanalyse;
(d) Stabilität von Sortieralgorithmen.

G2 (Bubble-Sort)

In dieser Aufgabe führen wir BubbleSort ein, einen weiteren, bekannten Sortieralgorithmus. Betrachten Sie dazu den folgenden Pseudocode:

BubbleSort(A)

```
1 : n = length(A)
2 : for i = 0 to n − 2 do
3 :     for j = n − 1 to i + 1 do
4 :         if A[j] < A[j − 1] then
5 :             tmp = A[j]
6 :             A[j] = A[j − 1]
7 :             A[j − 1] = tmp
8 : return A
```

(a) Welche Eigenschaften müssen erfüllt sein, damit BubbleSort ein korrekter Sortieralgorithmus ist?
(b) Beweisen Sie die Korrektheit von BubbleSort.
Hinweis: Stellen Sie für die innere und äußere Schleife je eine Schleifeninvariante auf.
(c) Analysieren Sie die Laufzeit von BubbleSort.
(d) Können Sie sich vorstellen, wie der Algorithmus zu seinem Namen kam?

G3 Darstellung von Merge-Sort und Quick-Sort

Betrachten Sie das Array $A = [14, 9, 5, 8, 11, 4, 21, 7, 6]$.

(a) Illustrieren Sie die Operationen von MergeSort anhand von A. Verwenden Sie dabei den Algorithmus aus der Vorlesung. Zeichnen Sie dazu die Zerlegungen, die der Algorithmus vornimmt, und stellen Sie dar, wie die einzelnen Teilarrays zu einem sortierten Array zusammengeführt werden. Machen Sie außerdem kenntlich, welches Teilarray in einem gegebenen Schritt gerade bearbeitet wird, und welche Teilarrays bereits sortiert sind. Die einzelnen Zwischenschritte der Unterroutine Merge müssen nicht dargestellt werden, bloß die jeweiligen Eingaben und das Endergebnis.

(b) Illustrieren Sie die Operationen von QuickSort anhand von A. Verwenden Sie dabei den Algorithmus aus der Vorlesung. Zeichnen Sie dazu das Array A nach jeder Tauschoperation, die von der Unterroutine Partition durchgeführt wird. Machen Sie außerdem kenntlich, welches das im jeweiligen Schritt betrachtete Pivotelement ist und welche Einträge bereits sortiert sind.

G4 Anwendungen des Mastertheorems

Begründen Sie für jede der folgenden Rekursionsgleichungen $T(n)$, ob Sie das Mastertheorem anwenden können oder nicht. Benutzen Sie gegebenenfalls das Mastertheorem, um eine asymptotische Schranke für $T(n)$ zu bestimmen. Die entsprechenden Anfangsbedingungenen (also die Werte $T(1)$ in allen Beispielen und, in (f), (h) und (i), zusätzlich $T(2)$) sind dabei vorgegebene Konstanten.

(a) $T(n) = 3T\left(\frac{n}{2}\right) + n^2$ (für $n > 1$);
(b) $T(n) = 4T\left(\frac{n}{2}\right) + n^2$ (für $n > 1$);
(c) $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$ (für $n > 1$);
(d) $T(n) = \frac{1}{2}T\left(\frac{n}{2}\right) + 1/n$ (für $n > 1$);
(e) $T(n) = \sqrt{2}T\left(\frac{n}{2}\right) + \log(n)$ (für $n > 1$);
(f) $T(n) = 2T\left(\frac{n}{\log(n)}\right) + n^2$ (für $n > 2$);
(g) $T(n) = 64T\left(\frac{n}{8}\right) - n^2\log(n)$ (für $n > 1$);
(h) $T(n) = 4T\left(\frac{n}{2}\right) + \frac{n}{\log(n)}$ (für $n > 2$);
(i) $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log(n)}$ (für $n > 2$);
(j) $T(n) = 6T\left(\frac{n}{3}\right) + n^2\log(n)$ (für $n > 1$);
(k) $T(n) = 2T\left(\frac{4n}{3}\right) + n$ (für $n > 1$);
(l) $T(n) = T\left(\frac{n}{2}\right) + 2T\left(\frac{n}{4}\right) + n$ (für $n > 1$).

G5* Max-Sort

Der Algorithmus MinSort sortiert die Einträge eines Arrays in aufsteigender Reihenfolge, indem er immer das erste unter den kleinsten Elementen im noch unsortierten Bereich mit dem ersten Element in diesem Bereich vertauscht. Hier sollen Sie den umgekehrten Ansatz untersuchen. (Hinweis: MinSort und MaxSort werden in der Literatur manchmal auch als SelectionSort oder ExchangeSort bezeichnet.)

(a) Entwerfen Sie einen Algorithmus MaxSort, der als Eingabe ein Array ganzer Zahlen bekommt, und es nach folgendem Prinzip in aufsteigender Reihenfolge sortiert:

> Es wird immer das erste unter den größten Elementen im noch unsortierten Bereich mit dem letzten Element in diesem Bereich vertauscht.

Beschreiben Sie Ihren Algorithmus kurz und stellen Sie ihn in Pseudocode dar.

(b) Beweisen Sie die Korrektheit Ihres Algorithmus. Bestimmen Sie dazu eine geeignete Schleifeninvariante und benutzen Sie diese, um die Korrektheit von MaxSort zu folgern.
(c) Analysieren Sie die Laufzeit von MaxSort.
(d) Ist MaxSort ein stabiler Sortieralgorithmus? Begründen Sie Ihre Angabe.
(e) Benutzen Sie MaxSort, um das Array $A = [6, 4, 1, 8, 3]$ zu sortieren. Geben Sie dabei alle Zwischenschritte an, jeweils vor jedem Durchlauf der äußeren Schleife.

Hausübungen

In diesem Bereich finden Sie die praktische Hausübung von Blatt 03. Bitte beachten Sie die allgemeinen Hinweise zu den Hausübungen und deren Abgabe im Moodle-Kurs.

Bitte reichen Sie Ihre Abgabe bis spätestens *Freitag, 17.05.2024, 23:59 Uhr MESZ* ein. Verspätete Abgaben können *nicht* berücksichtigt werden.

Wenn nicht anders angegeben, dürfen Sie keine Hilfsmethoden oder Datenstrukturen aus der Java Standardbibliothek verwenden[1]. Sie können davon ausgehen, dass übergebene Objekte nicht `null` sind.

H1 Comparators — 3 Punkte (1 + 2)

In dieser Hausübung werden Sie sich mit verschiedenen Sortieralgorithmen, welche Sie in der Vorlesung kennengelernt haben, beschäftigen und diese in Java implementieren.

Um Elemente sortieren zu können, müssen wir zunächst eine totale Ordnung für diese definieren. Dafür enthält die Java Standardbibliothek das generische, funktionale Interface `java.util.Comparator<T>`[2]. Es besitzt die funktionale Methode `int compare(T o1, T o2)`. Diese Methode vergleicht die beiden übergebenen Objekte und gibt genau dann eine Zahl kleiner 0 zurück, wenn $o1 < o2$. Wenn $o1 > o2$ wird eine Zahl größer 0 zurückgegeben und wenn $o1 = o2$ wird 0 zurückgegeben.

(a) Das Record `Card` im Package `p1.card` repräsentiert eine Pokerkarte, welche einen Wert (Zahl zwischen 2 und 13, beides inklusive) und eine Farbe (Kreuz, Herz, Pik, Karo) besitzt. Implementieren Sie im Package `p1.comparator` in der Klasse `CardComparator` die Methode `int compare(Card o1, Card o2)`, welche die beiden übergebenen Karten vergleicht und das Ergebnis wie oben beschrieben zurückgibt. Dabei werden zum Vergleichen zuerst die Werte der Karten verglichen. Falls die Karten denselben Wert besitzen, soll stattdessen die Farbe verglichen werden. Für den Vergleich der Werte gilt die natürliche Ordnung von Zahlen und für den Vergleich der Farben gilt Kreuz > Pik > Herz > Karo.

Beispiel: Karo 3 < Herz 4 < Herz 5 < Pik 5

Hinweis: Sie dürfen für diese Aufgabe Methoden aus den Klassen `java.util.Comparator`, `java.lang.Enum` und `java.lang.Integer` verwenden.

(b) Als Nächstes implementieren Sie eine Möglichkeit zu zählen, wie oft ein Comparator benutzt wurde um zwei Werte zu Vergleichen. Dies erlaubt einem z.B. zwei Sortieralgorithmen zu vergleichen. Vervollständigen Sie dafür die Klasse `CountingComparator` im Package `p1.comparator`. Diese besitzt ein Attribut `delegate`[3] vom Typ `Comparator<T>`, welches für das eigentliche Vergleichen zuständig ist. Die Klasse selber ist nur für das Zählen der durchgeführten Vergleiche zuständig. Dafür besitzt die Klasse die folgenden drei Methoden, welche Sie implementieren:

- `int compare(T o1, T o2)`: Diese Methode erhält zwei Objekte und vergleicht diese mit Hilfe des `delegate`. Danach wird die Anzahl der Vergleiche um eins erhöht und das Ergebnis des Vergleiches zurückgegeben.
- `void reset()`: Diese Methode setzt die Anzahl der Vergleiche auf 0 zurück.
- `int getComparisonsCount()`: Diese Methode gibt die Anzahl der Vergleiche seit dem letzten Aufruf von `reset`, bzw. seit der Erzeugung des Objektes, falls `reset` noch nicht aufgerufen wurde, zurück.

Hinweis: Wie Sie die Aufgabe lösen ist Ihnen überlassen. Das Verändern der Methodensignaturen ist allerdings nicht erlaubt.

[1] https://moodle.informatik.tu-darmstadt.de/mod/page/view.php?id=64695
[2] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Comparator.html
[3] https://java-design-patterns.com/patterns/delegation/

H2 Hybrid-Sort — 17 Punkte (6 + 6 + 5)

In dieser Aufgabe sollen Sie ein hybrides Sortierverfahren programmieren, welches zunächst genauso funktioniert wie MergeSort, jedoch innerhalb von MergeSort auf BubbleSort wechselt, sobald die Anzahl der Elemente, die in einem Unterschritt sortiert werden sollen, kleiner als k wird.

Dies setzten Sie in der Klasse `HybridSort<T>` im Package `p1.sort` um. Das Attribut k gibt dabei den Grenzwert an, ab welchem von MergeSort auf BubbleSort gewechselt wird und das Attribut `comparator` wird zum Vergleichen von zwei Werten verwendet.

Die zu sortierende Datenstruktur ist dabei eine `SortList<E>`, welche grundsätzlich wie ein Array funktioniert. Eine `SortList<E>` bietet die folgenden Methoden:

- `E get(int index)` liefert das Element am übergebenen Index zurück.
- `void set(int index, E value)` setzt das Element an der übergebenen Position auf den übergebenen Wert.
- `E remove(int)` setzt den Wert an der übergebenen Position auf `null` und gibt das Element zurück, welches zuvor an dieser Stelle gespeichert war.

Genauso wie ein Array hat eine `SortList` eine feste Größe, welche sich nicht verändern lässt. Die Methode `int getSize()` gibt die Größe der Liste zurück. Eine genauere Dokumentation finden Sie in der Vorlage im gleichnamigen Interface im Package `p1.sort`. Dort finden Sie in der Klasse `ArraySortList` ebenfalls eine Implementation dieses Interfaces.

Hinweis: Achten Sie beim Umsetzten beim Pseudocode darauf, dass sie diesen äquivalent umsetzten. Das heißt im Besonderen, dass Sie die Lese- und Schreiboperationen in derselben Reihenfolge durchführen müssen, wie sie im Pseudocode ausgeführt werden, und keine neuen Variablen hinzufügen um Werte zwischenzuspeichern.

(a) Implementieren Sie in der Klasse `HybridSort` die Methode `bubbleSort`, welche die zu sortierende `Sortlist`, sowie zwei Indizes `left` und `right` übergeben bekommt. Diese sortiert die übergebene `Sortlist` zwischen den beiden Indizes `left` und `right` (beide inklusive). Werte außerhalb dieser Indizes werden dabei nicht verändert. Implementieren Sie den Algorithmus äquivalent zu dem Pseudocode zu BubbleSort aus der Vorlesung, angepasst an die Anforderungen an die Indizes innerhalb welchen sortiert werden soll. Sie finden den Pseudocode dazu in Foliensatz 02 ab Seite 50.

(b) Als Nächstes implementieren Sie nun den MergeSort Teil des hybriden Sortieralgorithmus. Vervollständigen Sie dazu die Methoden `merge` und `mergeSort`. Die beiden Methoden sollen äquivalent zum Pseudocode aus der Vorlesung implementiert werden. Der einzige Unterschied dabei ist, dass in `mergeSort` eine weitere Bedingung hinzugefügt werden soll, sodass anstelle von `mergeSort` und `merge` die Methode `bubbleSort` aufgerufen wird, sobald die Anzahl der Elemente in dem zu sortierendem Bereich kleiner als k ist. Nach dem Aufruf von `bubbleSort` soll die Methode `mergeSort` nichts Weiteres tun, da nun die gesamte Liste sortiert ist. Verwenden Sie für den temporären Zwischenspeicher in der `merge` Methode ein Objekt der Klasse `ArraySortList`.

(c) Die Klasse `HybridOptimizer` im Package `p1.sort` soll nun den optimalen Parameter k für gegebene Eingabedaten berechnen. Implementieren Sie dazu die Funktion `optimize`, welche ein Objekt der Klasse `HybridSort` und ein Array, welches die zu sortierenden Elemente enthält, als Parameter übergeben bekommt. Dabei soll die Summe der Lese- und Schreiboperationen minimiert werden. Sie fangen also mit $k = 0$ an und ermitteln für jeden k-Wert die zugehörige Anzahl an Lese- und Schreibzugriffe. Brechen Sie die Suche nach dem Minimum ab, sobald Sie ein lokales Minimum gefunden haben[4], oder ein Erhöhen von k keinen Unterschied machen kann, da bereits direkt auf BubbleSort gewechselt wird, ohne das MergeSort einmal ausgeführt wurde. Konkret können Sie die Anzahl der Lese- und Schreibzugriffe für einen k-Wert bestimmen, indem Sie mit dem übergeben Array ein Objekt vom Typen `ArraySortList` erstellen und dieses an die Methode `sort` der Klasse `HybridSort` übergeben. Setzten Sie davor mit der Methode `setK` der Klasse `HybridSort` den zu verwendenden k-Wert. Danach können Sie die Anzahl der Lese- und Schreiboperationen mit den Methoden `getReadCount()` und `getWriteCount()` der Klasse `ArraySortList` ermitteln. Falls mehrere aufeinanderfolgende Werte den Wert des lokalen Minimums haben, geben Sie den letzten Index davon zurück. Anbei ein Beispiel für die Optimierung, wobei R+W für die Anzahl der benötigten Lese- und Schreibzugriffe steht.

- Beispiel 1: Die Rückgabe wäre $k = 2$, da an diesem Index das erste lokale (und auch globale) Minimum liegt.

[4] Wir interessieren uns dabei nur für Situationen in denen das nächste Ergebnis echt größer ist. D.h. sie sollen so lange weitersuchen bis ein Erhöhen von k ein schlechteres Ergebnis erzeugt (siehe Beispiel 3).

k	R+W
0	10
1	9
2	8
3	11
4	12

Beispiel 1

k	R+W
0	10
1	9
2	8
3	7
4	6

Beispiel 2

k	R+W
0	10
1	10
2	8
3	11
4	7

Beispiel 3

k	R+W
0	10
1	9
2	8
3	8
4	11

Beispiel 4

- Beispiel 2: Die Rückgabe wäre $k = 4$, da an diesem Index das erste lokale (und auch globale) Minimum liegt.
- Beispiel 3: Die Rückgabe wäre $k = 2$, da an diesem Index das erste lokale Minimum liegt. Das globale Minimum liegt jedoch bei $k = 4$. Streng genommen wäre bei $k = 0$ ebenfalls ein lokales Minimum. Allerdings betrachten wir nur Situation in das nächste Ergebnis echt größer ist.
- Beispiel 4: Die Rückgabe wäre $k = 3$, da das lokale Minimum bei $k = 2$ und $k = 3$ liegt. Da wir aber den letzten Index zurückgeben, ist die Rückgabe $k = 3$.

H3 Radix-Sort — 10 Punkte (3 + 7)

In der Vorlesung haben Sie ebenfalls den Radix-Sort Algorithmus kennengelernt. Radix-Sort ist ein Sortierverfahren, welches darauf basiert, dass die zu sortierenden Elemente aus Zeichen aus einem endlichen Alphabet bestehen, auf welchem eine totale Ordnung definiert ist. Es sortiert Elemente, indem es die Werte zunächst anhand der ersten Ziffer in Buckets einsortiert. Wenn die Elemente mehrere Ziffern enthalten, wird dies, unter Beibehaltung der vorherigen Sortierung, solange wiederholt, bis alle Ziffern betrachtet wurde. Unter der Annahme, dass die maximale Länge der zu sortierenden Elemente bekannt ist, besitzt Radix-Sort eine lineare Laufzeit.

Alle für diese Aufgaben relevanten Klassen befinden sich im Package `p1.sort.radix`.

(a) Ein wichtiger Bestandteil von Radix-Sort ist das Extrahieren von dem Zeichen an einer bestimmten Stelle in der Eingabe (z.B. die zweite Ziffer in einer Zahl) und das Abbilden auf den Index des zugehörigen Buckets. Da wir mit unserem Algorithmus nicht nur Zahlen, sondern beliebige Elemente sortieren möchten, wird dieser Vorgang durch das Interface `RadixIndexExtractor` modelliert. Es enthält dafür zwei Methoden. Die erste Methode ist `int getRadix()` und gibt die Anzahl an zulässigen Indizes zurück, die von der nächsten Methode zurückgegeben werden können. Dies ist in der Regel die Anzahl an verwendeten Buckets, bzw. die Anzahl an verschiedenen Zeichen im zugrundeliegenden Alphabet. Die zweite Methode ist `int extractIndex(T value, int position)`. Diese bestimmt zunächst das Zeichen an der Stelle `position` in der Eingabe `value`. Position 0 ist dabei das Zeichen mit der niedrigsten Stelligkeit. Danach bildet sie dieses Zeichen auf einen Index im Intervall $[0, \texttt{getRadix()})$ ab und gibt diesen Index zurück. Eine Implementation für Zahlen finden Sie in der Klasse `IntegerIndexExtractor`.

Implementieren Sie nun die Klasse `LatinStringIndexExtractor`, welche eine Implementation für Strings darstellt. Die Position 0 entspricht dabei dem letzten Zeichen in dem übergebenen String. Die Zeichen `'a' - 'z'` werden dabei auf die Indizes $0 - 25$ abgebildet. Groß- und Kleinbuchstaben werden dabei gleich behandelt, d.h. das Zeichen `'B'` wird ebenfalls auf den Index 1 abgebildet. Andere Zeichen werden auf den Index 0 abgebildet. Sie können davon ausgehen, dass die zu sortierenden Strings alle gleichlang sind, d.h., dass `position` immer kleiner als die Anzahl an Zeichen im String sind. Sie werden sich in der letzten Aufgabe mit Strings verschiedener Länge beschäftigen.

Hinweis: Sie dürfen für diese Aufgabe Methoden der Klassen `String` und `Character` verwenden. Sie können auch erst die nächste Aufgabe erledigen und mit dieser sich den Effekt verschiedener Implementation anschauen.

(b) Als Letztes müssen wir noch den eigentlichen Radix-Sort Algorithmus implementieren. Dies geschieht in der Klasse `RadixSort`. Die zu verwendenden Buckets sind in dem Array `buckets` gespeichert. Ein einzelner Bucket wird von dem Interface `Bucket` modelliert und ist in der Klasse `BucketLinkedList` implementiert[5]. Schauen Sie sich kurz die Dokumentation des Interfaces an, um zu verstehen, welche Operationen es für einen Bucket gibt. Implementieren Sie

[5] Diese Buckets modellieren dabei eine Queue (FIFO - First In First Out) und sind mithilfe einer verketteten Liste implementiert. Weitere Informationen zu diesen Datenstrukturen finden Sie in Foliensatz 03.

nun die Methode `putBucket(T value, int position)`, welche zunächst mit dem Attribut `IndexExtractor` den Index bestimmt, welcher zu dem Zeichen an der übergebenen Position im übergebenen Wert gehört. Danach wird der übergebene Wert in den Bucket, welcher zu dem zuvor bestimmten Index gehört, eingefügt. Sie müssen an dieser Stelle nicht, wie in der Vorlesung, die Position im Bucket und dessen Größe selber verwalten, da dies bereits in der Klasse `BucketLinkedList` geschieht. Implementieren Sie zum Schluss die Methode `sort`, welche die übergebene `SortList` sortiert, analog zu dem in der Vorlesung vorgestellten Pseudocode von Radix-Sort. Das Attribut `maxInputLength` gibt dabei die Länge des längsten Elements in der `SortList` an und bestimmt die Anzahl der notwendigen Iterationen für die äußere Schleife.

(c) *Unbewertet:* Bisher sind wir davon ausgegangen, dass die zu sortierenden Strings alle dieselbe Länge haben. Dies ist in der Praxis allerdings i.d.R. nicht der Fall. Implementieren Sie daher in der *PaddingLatinStringIndexExtractor* einen IndexExtractor, welcher ebenfalls auf Strings basiert, aber auch mit Fällen umgehen kann, in denen die abgefragte Position außerhalb des Indexbereiches des übergebenen Strings liegt und Strings verschiedener Länge lexikographisch sortiert. Überlegen Sie sich welche Änderungen Sie vornehmen müssen um dies umsetzten. Das Attribut *maxInputLength* enthält dabei die maximale Länge, die ein String haben kann. Die Klasse *PaddingLatinStringIndexExtractor* erbt dabei von der zuvor implementierten Klasse *LatinStringIndexExtractor*. Sie können also Funktionalitäten aus dieser wiederverwenden.

Hinweis: Es bietet sich an kürzere Strings mit einem Padding Zeichen aufzufüllen, damit alle Strings die selbe Länge haben. Beim Sortieren von Zahlen werden diese z.B. vorne mit Nullen aufgefüllt. Überlegen Sie sich, wie sie die Idee eines Padding Zeichen sinnvoll für Strings umsetzten können.