Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Dr. Hanjo Täubig
Jeremias Weihmann

Sommersemester 2012
Übungsblatt 8
11. Juni 2012

Grundlagen: Algorithmen und Datenstrukturen

Abgabetermin: Jeweilige Tutorübung in der Woche vom 18. bis 22. Juni

Tutoraufgabe 1

In dieser Aufgabe betrachten wir Operationen auf einem sortierten Feld $F = [0, ..., n-1]$. Wir nehmen an, dass dieses Feld zu jedem Zeitpunkt maximal $n > 0$ Elemente enthält und zu Beginn bereits bezüglich der Schlüssel der Elemente sortiert ist. Auf F sollen nun folgende Operationen ausgeführt werden: `insert(e)` (Füge Element e ein), `remove(k)` (Entferne Element mit Schlüssel k) und `find(k)` (Gib Element mit Schlüssel k aus).

Nehmen Sie an, es gibt ein weiteres Feld $B = [0, ..., \lceil\sqrt{n}\rceil - 1]$ der Größe $\lceil\sqrt{n}\rceil$. Die Operationen `insert(e)` und `remove(k)` auf F werden *lazy* bearbeitet: solange das Feld B noch nicht voll ist, merken wir uns die Operationen der Reihe nach im Feld B, anstatt diese Operationen sofort auf F anzuwenden – das Feld F bleibt also unverändert. Wir nehmen an, dass eine solche Einfügeoperation auf B konstante Kosten $\mathcal{O}(1)$ hat. Sobald das B Feld aber voll ist, muss ein Update auf F ausgeführt werden, bei welchem alle in B gespeicherten Operationen auf das Feld F angewendet werden und B geleert wird. Natürlich soll das Feld danach wieder sortiert sein.

Wir nehmen an, ein solches Update verursacht Kosten maximal cn für eine (genügend groß gewählte) Konstante c. Die `find()` Operation sucht immer zuerst in F nach einem Element; danach wird in B überprüft, ob das Element bereits gelöscht oder noch nicht in F eingefügt wurde.

Zeigen Sie, dass der amortisierte Aufwand für die Operationen `insert(e)`, `remove(k)` und `find(k)` durch diese *lazy* Bearbeitung durch $\mathcal{O}(\sqrt{n})$ beschränkt ist.

Tutoraufgabe 2

Wir wollen in dieser Aufgabe den `QuickSort` Algorithmus näher analysieren.

a) Zeigen Sie, dass keine asymptotisch bessere obere Schranke als $O(n^2)$ für `QuickSort` existiert, indem Sie eine unendliche Familie von Eingabearrays angeben, sodass der `QuickSort` Algorithmus bezüglich dieser Familie Laufzeit $\Omega(n^2)$ hat.

b) Zeigen Sie: Wird in dem Algorithmus als Pivot-Element das k-größte Element ausgewählt (wobei $k \in \mathbb{N}$ eine beliebige aber feste Konstante ist), so liegt die Worst-Case-Laufzeit des modifizierten Algorithmus immer noch bei $\Omega(n^2)$.

   Ist $|A| < k$, so wird das Pivot-Element wie im Algorithmus der Vorlesung gewählt.

c) Beweisen Sie: Wählen wir als Pivot-Element das $\lfloor \frac{9}{10} n \rfloor$-größte Element aus `A` ($|A| = n$), so ist die Worst-Case Laufzeit $\mathcal{O}(n \log n)$. Hierbei kann verwendet werden, dass es Selektionsverfahren gibt die für beliebiges (nicht notwendigerweise konstantes) $k \leq n$ ein k-größtes Element in Zeit $O(n)$ finden können.

Tutoraufgabe 3

Seien A und B nicht leere Mengen und $\leq_A$ und $\leq_B$ totale Ordnungen auf A bzw. B.
Definieren Sie eine totale Ordnung auf $A \times B$.
Wozu brauchen wir den Begriff der totalen Ordnung im Zusammenhang der aktuellen Vorlesungsthemen Sortieren und Selektieren?
Hinweis: Eine lineare Ordnung $\leq$ auf einer Menge X ist

- transitiv: $\forall x, y, z \in X: \quad x \leq y \ \wedge \ y \leq z \ \Rightarrow \ x \leq z$
- antisymmetrisch: $\forall x, y \in X: \quad x \leq y \ \wedge \ y \leq x \ \Rightarrow \ x = y$
- linear (total): $\forall x, y \in X: \quad x \leq y \ \vee \ y \leq x$
- reflexiv: $\forall x \in X: \quad x \leq x$ (folgt aus der Linearität)

Hausaufgabe 1

Implementieren Sie in der Klasse `UIsqsArray` den QuickSort-Algorithmus, in der Funktion `sort`, der die Elemente in dem Feld `A` sortiert.
Verwenden Sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die Klasse `UIsqsArray`.
Achten Sie bei der Abgabe Ihrer Aufgabe darauf, dass Ihre Klasse `UIsqsArray` heißt und auf den Rechnern der Rayhalle (`rayhalle.informatik.tu-muenchen.de`) mit der bereitgestellten Datei `main_qs` kompiliert werden kann. Anderenfalls kann eine Korrektur nicht garantiert werden. Achten Sie darauf, dass Ihr Quelltext ausreichend kommentiert ist.
Schicken Sie die Lösung per Email mit dem Betreff `[GAD] Gruppe <Gruppennummer>` an Ihren Tutor.

Hausaufgabe 2

In der ersten Tutoraufgabe haben wir die *lazy* Bearbeitung eines Feldes der Größe n mittels einem Buffer der Größe $\lceil \sqrt{n} \rceil$ betrachtet. Wir haben mittels amortisierter Analyse gezeigt, dass die Laufzeit für die Operationen `insert`, `find` und `remove` in $\mathcal{O}(\sqrt{n})$ liegt.
Dabei haben wir vereinfachend vorausgesetzt, dass ein Update, also das Synchronisieren des Feldes mit dem Buffer, in Zeit maximal cn erfolgen kann.

(a) Zeigen sie nun etwas allgemeiner: Ein Buffer der Grösse k kann mit einem Feld der Größe $n > 0$ immer in $\mathcal{O}(n + \text{sort}(k))$ Schritten synchronisiert werden, wobei sort(k) die Worst-Case-Laufzeit eines beliebigen Sortieralgorithmus zum Sortieren eines Feldes der Größe k ist. Nehmen Sie dabei an, dass das Feld groß genug ist um alle Elemente aufnehmen zu können.

(b) Ist damit die Forderung aus der ersten Tutoraufgabe erfüllt, dass eine Konstante c existiert, sodass die Synchronisierung in Zeit maximal cn durchgeführt werden kann? Begründen Sie Ihre Antwort.

Hausaufgabe 3

Zeigen Sie mit (starker) vollständiger Induktion die folgenden Gesetze:

(a) Für eine Funktion T definiert durch

$$T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + c, & \text{falls } n > 1 \\ c', & \text{falls } n = 1 \end{cases}$$

wobei c, c' positive Konstanten sind, gilt $T(n) \leq dn - \max\{c, c'\}$ für eine geeignet gewählte Konstante d.

(b) Folgendes Gesetz ist eine Verallgemeinerung von (a). Sei $f : \mathbb{N} \to \mathbb{N}$ eine Funktion mit $1 \leq f(n) < n$. Für eine Funktion T definiert durch

$$T(n) = \begin{cases} T(f(n)) + T(n - f(n)) + c, & \text{falls } n > 1 \\ c', & \text{falls } n = 1 \end{cases}$$

wobei c, c' positive Konstanten sind, gilt $T(n) \leq dn - \max\{c, c'\}$ für eine geeignet gewählte Konstante d.

Anmerkung: Dies bedeutet im Wesentlichen, dass ein Divide-and-Conquer-Algorithmus, der eine Lösung in **konstanter Zeit** aus zwei Lösungen der beiden Teilprobleme zusammensetzt, lineare Laufzeit hat. Hierbei ist nur gefordert, dass die Eingabegröße der beiden Teilprobleme zusammen so groß (bzw. maximal so groß) wie die ursprüngliche Eingabe ist. Dies führt oft zu interessanten Linearzeitalgorithmen, bei denen die Aufspaltung einer Eingabe der Länge n in zwei Eingaben der Längen $n - 1$ und 1 erfolgt. Dies ermöglicht in vielen Fällen eine sehr einfache nichtrekursive Implementation, wobei ausgenutzt wird, dass einer der beiden rekursiven Aufrufe stets der Basisfall ist.

Hausaufgabe 4

Gegeben sei eine Folge $(a_1, \ldots, a_n)$ ganzer Zahlen ($\in \mathbb{Z}$). Bestimme eine zusammenhängende Teilfolge $(a_i, a_{i+1}, \ldots, a_j)$ mit $1 \leq i \leq j \leq n$, sodass $\sum_{k=i}^{j} a_k$ maximal ist.

Entwerfen Sie einen Divide-and-Conquer-Algorithmus der obiges Problem löst. Geben Sie eine Rekursionsgleichung für (eine obere Schranke für) die Laufzeit Ihres Algorithmus an.

Hinweis: Konstruieren Sie Ihren Algorithmus so, dass er nicht nur eine zusammenhängende Teilfolge mit maximal möglicher Elementsumme konstruiert, sondern gleichzeitig diejenigen beiden zusammenhängenden Teilfolgen $(a_1, \ldots, a_l)$ sowie $(a_r, \ldots, a_n)$ mit $l, r \in [1, n]$ und maximal möglicher Elementsumme findet.