Technische Universität München
Institut für Informatik
Lehrstuhl für Effiziente Algorithmen
Dr. Michal Mnuk

SS 1998
Übungsblatt 5
17. Juni 1998

Grundlegende Algorithmen

Abgabe bis 24. Juni 1998 in der Übung

Aufgabe 13

In der Vorlesung wurde eine Version von `is_member` vorgestellt und analysiert, die auf Exponential–Search basiert. Gegeben eine sortierte Folge $(a_i)_{i=0,\ldots,n-1}$ der Länge n von paarweise verschiedenen Zahlen (ohne wesentliche Einschränkung sei im Folgenden n eine Zweierpotenz, sonst Gaußklammern). Wenn wir die Zusatzbedingung haben, daß wir sicherlich auf ein Element $b \in \bigcup_i \{a_i\}$ mit Rang höchstens k zugreifen wollen, dann läßt sich `is_member` mit $O(\log(k))$ Vergleichen entscheiden. Wir teilen die Folge in Teilintervalle, indem wir die Elemente a_{2^i} markieren. Da wir wissen, daß der Rang von b höchstens k ist, müssen wir nur $O(\log k)$ Vergleiche durchführen, um das richtige Intervall zu finden. Die Breite des Intervalls ist aber nur $O(k)$, und wir haben b deshalb nach $O(\log k)$ weiteren Vergleichen mittels Binary–Search lokalisiert. Genauere Rechnung ergibt, daß man $2 \log k$ Vergleiche braucht.

Betrachten Sie nun eine andere Zusatzbedingung. Es soll garantiert sein, daß der Rang des Elements b in $[n/2 - \log n, n/2 + \log n]$ liegt. Entwerfen und analysieren Sie einen Algorithmus, der diese Zusatzbedingung geschickt ausnützt und deshalb mit $O(\log \log n)$ Vergleichen auskommt.

Aufgabe 14

Die Idee von BUCKETSORT war: Bestimme das Maximum $a_{\max}$, alloziere ein Array der Länge $a_{\max}$ von anfangs leeren zyklischen Listen (den Buckets), ordne das Eingabearray in die Buckets ein (a_i wird in die zyklische Liste a_i eingehängt), durchlaufe zuletzt alle Buckets in aufsteigender Ordnung und sammle die Folgenglieder ein.
In der letzten Übung haben wir uns gefragt, warum man beim Sortieren nicht immer BUCKETSORT mit (intuitiv) „linearer Komplexität“ verwendet, und wie dies zur in der Vorlesung bewiesenen unteren Schranke $\Omega(n \log n)$ fürs Sortieren paßt.
Die erste Frage soll in dieser Aufgabe geklärt werden. Man erinnere sich, daß die Komplexität eines Algorithmus natürlich vom Algorithmus selbst, aber auch von der Definition der Eingabelänge und dem gewählten Kostenmaß abhängt. Weil man insbesondere bei der Definition der Eingablänge oft mehrere Möglichkeiten hat, und diese häufig als intuitiv klar (?) implizit festgelegt wird, kommt es leicht zu Verwirrung. Außerdem ist natürlich die Wahl der möglichen Eingaben relevant. Zusatzbedingungen verkleinern den Spielraum des Gegenspielers und können deshalb die Worst–Case–Laufzeit verbessern.
Die zweite Frage wird dadurch beantwortet, daß die untere Schranke für das Kostenmaß *Anzahl der Vergleiche* gilt, und daß außerdem die Algorithmen, für die untere Schranke gilt, *vergleichsbasiert* sein müssen. Da BUCKETSORT nicht vergleichsbasiert ist, gilt die Schranke nicht, und man sollte deshalb als Kostenmaß besser die *Laufzeit* verwenden.

Bestimmen Sie die asymptotische Laufzeitkomplexität von BUCKETSORT bezüglich verschiedener Eingabelängen und Zusatzbedingungen.

a) $|(a_1, \ldots, a_r)|_a := r$

b) $|(a_1, \ldots, a_r)|_b := \max\{r, \max_{i=1,\ldots,r}\{|a_i|\}\}$

c) Nehmen Sie nun zusätzlich an, daß die a_i aus einem endlichen Universum stammen, daß die Bitlängen $|a_i|$ also durch eine gemeinsame Konstante beschränkt sind.

Bemerkungen: Es darf auch sein, daß die Worst–Case–Komplexität unendlich ist. Unter b) müssen Eingabefolgen mit Eingabelänge n nicht unbedingt n Elemente haben. In c) fallen die Eingabelängen aus a) und b) für hinreichend lange Folgen zusammen.

Aufgabe 15

Analysieren Sie die im Worst–Case von QUICKSELECT benötigte Anzahl von Vergleichen, indem Sie eine Rekursionsgleichung aufstellen und lösen.
Wie sähen die entsprechenden Rekursionsgleichungen und deren Lösung für QUICKSELECT und QUICKSORT aus, wenn PARTITION die Arrays immer in zwei gleich große Teile aufteilen könnte? Warum ist hier die Lösung für QUICKSELECT besser? (Tip: Verwenden Sie die Lösungsformeln aus der Vorlesung, siehe auch Aufgabe 12b.)

Eingabe: Array A mit n Elementen, Index k
Ausgabe: Das Element mit dem Rang k
QUICKSELECT(A, n, k)

$(B, C) :=$PARTITION(A, n);

if (length$(B) + 1 = k$) return $C[1]$;

else if (length$(B) + 1 > k$) return QUICKSELECT$(B, \text{length}(B), k)$;

else return QUICKSELECT$(C, \text{length}(C), k - \text{length}(B))$;

Bemerkungen: Ein Element der Eingabefolge *hat den Rang k*, wenn $(k-1)$ Folgenglieder kleiner sind. PARTITION soll das Element am rechten Ende des Eingabearrays als Pivotelement verwenden und dieses zuletzt an den Anfang der „größeren Hälfte“ C stellen. PARTITION(A, n) braucht $n - 1$ Vergleiche.