Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn
M. Fuchs, G. Schmid

Algorithmen und Datenstrukturen
Sommersemester 2024
Musterlösung Übungsblatt 6

Abgabe: Dienstag, 4. Juni, 2024, 10:00 Uhr

Aufgabe 1: Minimaler Abstand zwischen zwei Werten *(10 Punkte)*

(a) Gegeben n Zahlen in einem Array A. Beschreiben Sie einen Algorithmus der die zwei Werte ausgibt die am nähesten zusammen liegen (kleinste Distanz haben). Formaler: Der Algorithmus findet Indizes $i \neq j$, so dass $|A[i] - A[j]|$ minimal für alle Indexpaare ist.
Argumentieren Sie, dass ihr Algorithmus korrekt ist und beweisen Sie dass er eine Laufzeit in $o(n^2)$ hat. [1] *(5 Punkte)*

(b) Nehmen Sie nun an, dass die n Zahlen in einem Binären Suchbaum B, anstatt in einem Array, gegeben sind. Beschreiben Sie wieder einen Algorithmus der Baumknoten $u \neq v$ findet, so dass $|\text{val}(v) - \text{val}(u)|$ minimal ist.
Argumentieren Sie, dass ihr Algorithmus korrekt ist und die Laufzeit in $O(n)$ liegt. *(5 Punkte)*

Musterlösung

(a) Wir sortieren A in Zeit $O(n \log n)$ (z.B. mit MergeSort) und iterieren anschließend über das sortierte Array mit Laufvariable k. Wir speichern das k welches $A[k+1] - A[k] (\geq 0)$ minimiert. Nach Aufgabenstellung sollen wir nun aber die Indizes aus dem orginalen (nicht sortieren) Array ausgeben. Eine elegante Möglichkeit ist es noch vor dem sortieren jeden Eintrag $A[i]$ druch ein Tupel $(A[i], i)$ zu ersetzten. Beim sortieren wird wie vorher auch nur nach dem Wert $A[i]$ sortiert, aber der sortierte Array hat somit die orginalen positionen gespeichert und diese können wir nun auch zurückgeben.

Korrektheit: Da bei jedem sortieren Array A für alle k gilt $... \leq A[k-2] \leq A[k-1] \leq A[k] \leq A[k+1] \leq A[k+2] \leq ...$ muss das größte Element das noch kleiner als $A[k]$ ist, gerade $A[k-1]$ sein (sonst wäre das Array ja nicht korrekt sortiert) und mit der gleichen Begründung muss das kleinste Elemente, das gerade so größer als $A[k]$ ist, $A[k+1]$ sein. Wir müssen also $A[k]$ nicht mit jedem anderen Element im Array vergleichen sondern nur mit diesen beiden *Nachbarn*. Da unser Algorithmus jedes Element mit seinen Nachbarn im sortierten Array vergleicht findet dieser auch die minimale Distanz.
Laufzeit: Sortieren in $O(n \log n)$ und dann anschließend einmal über das Array iterieren $O(n)$, also insgesamt $O(n \log n) \subset o(n^2)$. Wenn wir wie oben beschrieben erst alle Elemente in Tupel konvertieren, kostet uns dies zusätzliche Zeit von $O(n)$, was keine asymptotische Auswirkung hat.

(b) Wir benutzen die In-order Traversierung von Binären Suchbäumen. Diese gibt immer eine sortierte Reihenfolge aller Zahlen aus. Wir betrachten diese wie ein sortiertes Array A, so dass $A[i]$ die i-te Zahl ist die von der Traversierung ausgegeben wird. Der Rest ist wie oben.

[1] z.B. in dem Sie zeigen dass ihr Algorithmus eine Laufzeit von $O(n^{3/2})$, oder $O(n)$ hat.

Korrektheit: Folgt aus der Tatsache, dass die In-Order Traversierung den Baum in sortierter Reihenfolge durchgeht und dem Argument aus a)
Laufzeit: Die Baumtraversierung benötigt $\Theta(n)$ Zeit und die Vergleiche auch, also ist die insgesamte Laufzeit auch $O(n)$.

Aufgabe 2: Binärer Suchbaum - Schwerster Pfad *(10 Punkte)*

Gegeben einen Binären Suchbaum B mit n Zahlen, wir sagen ein Pfad $P = \{r, v_1, v_2, \ldots, b\}$ von der Wurzel r bis zu einem Blatt b hat Gewicht $w(P) = \sum_{v \in P} \text{val}(v)$. Beschreiben Sie einen Algorithmus der, in einem gegebenen Suchbaum, einen Wurzel-Blatt Pfad mit dem maximalen Gewicht findet. Argumentieren Sie, dass ihr Algorithmus korrekt ist und dass die Laufzeit in $O(n)$ liegt. *(10 Punkte).*

Musterlösung

Wir modifizieren die Post-Order Traversierung. Wenn ein Knoten v besucht wird, wurden also schon beide Kinder besucht. Wen wir einen Knoten besuchen, berechnen wir den schwersten Pfad der zu diesem Knoten führt. Der schwerste Pfad hat dann Gewicht

$$\text{val}(v) + \max_{u \text{ Kind von } v} \{w(P_u)\}$$

Wobei P_u der schwerste Pfad zu u ist.
Korrektheit und Wohldefiniertheit: Wir beweisen das jeder besuchte Knoten v bereits den Schwersten Pfad von seinem Teilbaum (mit Wurzel v) kennt.
Per Induktion über die Höhe des Teilbaums, sei also für den Induktionsanfang v ein Blatt. Für ein Blatt ist der schwerste Pfad nur der Pfad der v selbst enthält, also hat der schwerste Pfad bis v, Gewicht $val(v)$.
Sei nun v kein Blatt, also hat v mindestens 1 Kind. Weil wir in Post-Order den Baum traversieren wurden alle Kinder von v bereits besucht und kennen also per Induktionsannahme bereits ihren schwersten Pfad (Kinder haben einen kleineren Teilbaum also mit niedrigerer Höhe). Der schwerste Pfad ergibt sich also indem wir den schwersten Pfad bis zu einem Kind fortsetzen. Der schwerste Pfad zu v hat dann also Gewicht

$$\text{val}(v) + \max_{u \text{ Kind von } v} \{w(P_u)\}$$

Also sind alle Werte $w(P_u)$ immer bereits berechnet und der Algorithmus wohldefiniert.
Am Ende des Algorithmus ist auch die Wurzel von der Post-Order Traversierung besucht worden, also kennen wir auch den schwersten Pfad zu r und haben damit die Korrektheit bewiesen.
Laufzeit: Die Traversierung benötigt $\Theta(n)$ Zeit und bei jedem Knoten der Besuch wird, werden nur eine konstante Anzahl an Operationen durchgeführt (da in binären Suchbäumen jeder Knoten maximal 2 Kinder hat). Also ergibt sich insgesamt eine Laufzeit von $\Theta(n) \subset O(n)$.