Grundzüge DS & Alg (WS14/15)

Lösungsvorschlag zu Aufgabenblatt 2

Aufgabe 1

(a)
Angenommen alle Schlüssel besitzen den Wert k, so ist auch das Pivot-Element gleich k. Laut Definition ordnet Quicksort Elemente, die größer als das Pivot-Element sind dahinter und Elemente, die kleiner oder gleich sind, davor ein.
Somit werden alle Schlüssel vor dem Pivot-Element eingeordnet. Danach wird Quicksort rekursiv auf den beiden Teillisten aufgerufen.
Die erste Teilliste ist nur um ein Element kürzer, während die zweite leer ist.
Die folgende Rekurrenz beschreibt dieses Verhalten: $T(n) = n + T(n-1)$
Die geschlossene Form dieser Rekurrenz ist $\frac{n^2+n}{2}$
Somit beträgt die Laufzeit in diesem Szenario $\Theta(n^2)$

(b)
Eine Variante dieses Problem zu lösen besteht darin eine zusätzliche Liste für Elemente, die gleich dem Pivot-Element sind einzuführen.
Echt kleinere Elemente werden in die erste Teilliste, gleiche in die zweite und echt größere Elemente in die dritte Teilliste verschoben.
Danach wird Quicksort auf der ersten und dritten Teilliste rekursiv aufgerufen.
Im vorliegenden Szenario erreichen wir mit diesem Verfahren eine Laufzeit von $O(n)$, da wir nur einmal die Elemente in die mittlere Teilliste verschieben und die Listen für echt größere und echt kleinere Elemente leer sind.

(c)

Wir verwenden die Implementierung aus Aufgabenteil b).
Wir nehmen an, dass das Pivot-Element unter den d Elementen mit gleichverteilter Wahrscheinlichkeit ausgewählt wird. Danach wird die Liste unterteilt in eine Liste der Länge i und eine der Länge $n-i$. Hierbei wird nicht berücksichtigt, dass Elemente gleich dem Pivot-Element sein können und somit in die mittlere Liste einsortiert werden. Sei die Anzahl der unterschiedlichen Elemente in der linken Liste d_l und in der rechten d_r. Da die Elemente mit dem Pivot-Element verglichen werden und dadurch eindeutig einer Liste zugeteilt werden, folgt $d_l + d_r = d$.
Im folgenden wird eine Rekurrenz in Abhängigkeit der Anzahl unterschiedlicher Schlüssel d für die worst-case Laufzeit von Quicksort aufgestellt.

$$g(n,d) \leq O(n) + \frac{1}{n} \sum_{1 \leq i < n} \max_{d_l + d_r = d} \{g(i, d_l) + g(n-i, d_r)\} \text{ für } n \geq d > 2$$

Annahme: $g(n,d) \in O(n \cdot \log(d))$

Quelle: `http://ecommons.cornell.edu/bitstream/1813/6417/1/83-577.pdf` Theorem 4.1

Grundzüge DS & Alg (WS14/15)

Lösungsvorschlag zu Aufgabenblatt 2

Beweis durch Induktion:
IA: Für $d = 2$ müssen wir Quicksort nicht rekursiv aufrufen, da einer der beiden Schlüssel als Pivot-Elemente gewählt wird. Alle gleichen Elemente werden in die mittlere Liste verschoben, während die Elemente, die ungleich dem Pivot-Element sind untereinander gleich sein müssen und somit in die linke oder rechte Liste eingeordnet werden. Dies benötigt $O(n)$ Schritte.
IV: $g(n', d') < c \cdot n' \cdot \log(d')$, für alle $n' < n$ und $d' < d$.
IS: Der Definition der O-Notation folgend, gilt es eine Konstante c zu finden, so dass $g(n, d) \leq c \cdot n \cdot \log(d)$ gilt.
Wir setzen die rekursive Definition von $g(n, d)$ ein ($b > 0$):

$$g(n,d) \leq bn + \frac{1}{n} \sum_{1 \leq i < n} \max_{d_l + d_r = d} \{c \cdot i \cdot \log(d_l) + c \cdot (n-i) \cdot \log(d_r)\}$$

Unabhängig von i wird die Summe durch folgende Werte maximal:
$d_l = i\frac{d}{n}$, bzw. $d_r = (n-i)\frac{d}{n}$
Einsetzen:

$$g(n,d) \leq bn + \frac{c}{n} \sum_{1 \leq i < n} \left(i \cdot \log\left(i\frac{d}{n}\right) + (n-i)\log\left((n-i)\frac{d}{n}\right)\right)$$

$$= bn + \frac{2c}{n} \sum_{1 \leq i < n} \left(i \cdot \log\left(i\frac{d}{n}\right)\right)$$

$$= bn + \frac{2c}{n} \log\left(\frac{d}{n}\right) \sum_{1 \leq i < n} i + \frac{2c}{n} \sum_{1 \leq i < n} i \cdot \log(i).$$

Wir vereinfachen die Summe mit:
$\sum_{1 \leq i < n} i \cdot \log(i) \leq \frac{1}{2}n^2 \log(n) - \frac{1}{4}n^2$
$\sum_{1 \leq i < n} i = \frac{1}{2}n(n+1)$
Somit ergibt sich:

$$g(n,d) \leq bn + c(n+1)\log d - c(n+1)\log n + cn\log n - \frac{c}{2}n$$

$$\leq bn - \frac{c}{2}n - c\log n + c(n+1)\log d.$$

Können wir annehmen, dass eine Konstante c existiert, so dass folgende Abschätzung gilt:
$bn - \frac{c}{2}n - c\log n + c\log d < 0$ für $n > 2$.

Somit vereinfachen wir zu:

$$g(n,d) < c \cdot n \log d$$

Damit ist der Induktionsschritt gezeigt.
Die Laufzeit beträgt $O(n \cdot \log(d))$.

Aufgabe 2

Die while-Schleife läuft genau $F(a)$ Mal durch und in jedem Durchlauf wird genau eine Fehlstelle korrigiert. Sei $a = \langle a_1, ..., a_i, a_{i+1}, ..., a_n \rangle$ eine Folge. Eine Vertauschung von a_i und a_{i+1} findet nur statt, wenn $a_i > a_{i+1}$, da die while-Schleife sonst zuvor abbrechen würde. Nach der Vertauschung wurde die Fehlstelle (a_i, a_{i+1}) korrigiert und die relative Reihenfolge anderer Schlüsselpaare zueinander hat sich nicht geändert. Nach Insertion-Sort ist $F(a) = 0$, also wurde die while-Schleife in der Summe genau $F(a)$ Mal durchlaufen. Die for-Schleife braucht unabhängig von $F(a)$ zusätzlich Laufzeit in $O(n)$. Daher ist die Laufzeit in $O(n + F(a))$.

Das n in der Laufzeitschranke ist insbesondere notwendig, da auch bei einer sortierten Folge die for-Schleife Laufzeit in $O(n)$ benötigt, während $F(a) = 0$ ist.

Aufgabe 3

Quelle: `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/merge/merge.htm`

```
void merge(int lo, int m, int hi)
{
    int i, j, k;
    i=0;
    j=lo;
    // vordere Hälfte von a in Hilfsarray b kopieren
    while (j<=m)
        b[i++]=a[j++];
    i=0;
    k=lo;
    // jeweils das nächstgrößte Element zurückkopieren bzw nach vorne kopieren
    while (k<j && j<=hi)
        if (b[i]<=a[j])
            a[k++]=b[i++];
        else
            a[k++]=a[j++];
    // Rest von b falls vorhanden zurückkopieren
    while (k<j)
        a[k++]=b[i++];
}
```

Der obige Quellcode zeigt eine abgewandelte Version von Merge. Mit dieser Version ist es möglich, den Speicherverbrauch für die Kopieroperationen zu halbieren. Weiterhin muss nur die Hälfte der Schlüssel kopiert werden:
Dadurch das immer nur ein Hilfsarray für die vordere Hälfte des zu mergenden Teils angelegt werden muss, ist der Speicherverbrauch für das angepasste Mergesort lediglich $1.5n$ (Array größe n um das zu sortierende Feld zu speichern und ein $n/2$ großes Array zum Speichern der ersten Hälfte für die Mergeaufgabe). Nun werden beide Teillisten, die sich im selbem Feld befinden, von links nach rechts "durchgegangen", das heißt von lo nach hi, an jeder Position wird geprüft, ob der Kopf der linken oder rechten Teilliste kleiner ist, und entsprechend eingefügt (äquivalent zum normalen Mergesort). Hier kommt nun der Trick zum Einsatz. Ist das Element der linken Teilliste kleiner, so wird dieses zurückkopiert in das original Array, ist das rechte Element kleiner, so wird dieses nach vorne kopiert. Dadurch spart man sich in der rechten Teilliste das Kopieren in ein Hilfsarray und somit ein Viertel der Kopieroperationen in jedem Schritt. Dabei kann es niemals passieren, dass Elemente der rechten Teilliste überschrieben werden, ohne dass diese vorher schon einsortiert wurden, da zu jedem Zeitpunkt die Anzahl der linken Elemente plus die Anzahl der schon kopierten rechten Elemente genau so groß ist, dass nur die schon kopierten rechten Elemente überschrieben werden könnten.

Aufgabe 4

Aufgabe 4

(a)
Man muss nun zwei Funktionen f und g finden, für die weder $f \in O(g)$ noch $g \in O(f)$ gelten darf. Also darf weder $f \leq_{\text{fü}} c \cdot g$ noch $g \leq_{\text{fü}} c \cdot f$ gelten. Ein Beispiel für zwei solche Funktionen ist:

$$f(x) = x$$
$$g(x) = x^{1+(-1)^x}$$

(Also $g(x) = x^2$, wenn x gerade, $g(x) = 1$, wenn x ungerade ist.)

Beweis durch Widerspruch:
Angenommen $\exists c : f \leq c \cdot g, \forall x \geq x_0$.

$$\Rightarrow f(x) \leq c \cdot g(x) \Leftrightarrow x \leq c \cdot x^{1+(-1)^x}$$

Wenn $x > x_0$ ungerade ist, folgt:

$$\Rightarrow x \leq c \cdot x^{1-1} = c$$

Dies ist ein Widerspruch!
Der Beweis, dass $g \leq_{\text{fü}} c \cdot f$ für die gegebenen Funktionen nicht gilt, folgt analog. *q.e.d.*

(b)

Die beiden Funktionen aus (a) kann man hier nicht verwenden, da g offensichtlich nicht monoton steigend ist. Man wählt nun folgende Funktionen:

$$f(x) = (2x)^{x^2}$$

$$g(x) = (2x + (-1)^x)^{x^2}$$

z.z.: g(x) und f(x) sind monoton steigend und $f \notin O(g)$ und $g \notin O(f)$
f ist offensichtlich monoton steigend. Für g gilt z.z. dass $\frac{g(x+1)}{g(x)} > 1$

$$\frac{g(x+1)}{g(x)} = \frac{(2(x+1) + (-1)^{x+1})^{(x+1)^2}}{(2x + (-1)^x)^{x^2}}$$

Man sieht, dass der Term minimal wird, wenn gerade ist, da in diesem Fall von der Basis im Zähler 1 abgezogen wird und zu der Basis im Nenner 1 addiert wird.

$$\Rightarrow \frac{(2x + 2 - 1)^{(x^2+2x+1)}}{(2x+1)^{x^2}} = (2x+1)^{2x+1} > 1$$

Folglich ist der Quotient auch für ungerade $x > 1$.
Es bleibt z.z., dass $f \notin O(g)$ und $g \notin O(f)$.
Auch bei diesem Beispiel ist der ausschlaggebende Punkt, dass g um f osziliert, da die Basis von g entweder um 1 kleiner oder größer als die Basis von f ist.
Beweis durch Widerspruch:
Angenommen $\exists c : f \leq c \cdot g, \forall x \geq x_0$.

$$f(x) \leq c \cdot g(x) \Leftrightarrow (2x)^{x^2} \leq c \cdot (2x + (-1)^x)^{x^2}$$

Man setzt also für x eine beliebige ungerade Zahl ein.

$$\Rightarrow (2x)^{x^2} \leq c \cdot (2x-1)^{x^2}$$

$$\Rightarrow c \geq \left(\frac{2x}{2x-1}\right)^{x^2} = \left(1 + \frac{1}{2x-1}\right)^{x^2} \overset{\text{(Bernoulli)}}{\geq} 1 + \frac{x^2}{2x-1} \geq 1 + \frac{x}{2}$$

Widerspruch! Der zweite Fall des Beweises erfolgt analog. *q.e.d.*