

Algorithmen und Datenstrukturen
SS 2018

Übungsblatt 2: Grundlagen

Tutorien: 23.04-27.04.2018

Aufgabe 2-1 *Laufzeitanalyse*

Einfügen in binären Baum mit n Elementen

```
insert(int new, tree t){
t.current = root;
while(t.currentIsLeaf == false){
    if(new > t.currentValue){
        t.goRight();
    }
    else{t.goLeft();}
}
t.insertHere(new);
}
```

Suchen gleicher Elemente in zwei Listen

```
int compare(char[] a, char[] b){
int count = 0;
foreach(char c: a){
    foreach(char d: b){
        if(c == d) count++;
    }
}
return count;
}
```

(a) Betrachten Sie die Programme a - e auf dem ersten Übungsblatt sowie die beiden obigen Algorithmen und bearbeiten Sie für alle Programme die folgenden Punkte:

- Welche Komponenten haben konstante Laufzeit?
- Welche Komponenten haben eine Laufzeit von $O(n)$?
- Welche Komponenten haben andere Laufzeiten, und welche Laufzeiten sind das?

sollten Funktionen nur unter Umständen terminieren, formulieren Sie die Voraussetzungen.

(b) Erstellen Sie zu jeder Aufgabe eine ausformulierte Laufzeitformel, welche sämtliche vorhandenen Komponenten beinhaltet.

(c) Formulieren Sie nun aus den erstellten Formeln gültige O-Notationen.

Lösungsvorschlag:

für a):

$O(1)$: if, return true/false, return $f_0(-a)$ (kommt genau einmal vor)

$O(n)$: return $f_0(a-2)$; wird $a/2$ mal aufgerufen

keine andere Laufzeiten.

Formel: $O(1) + O(n) = O(n)$

für b):

$O(1)$: Arrayzuweisungen, Arrayvergleich (immer max Value lang), return true

$O(n)$: For-schleifen für str1 und str2 (abhängig von Stringlänge)

keine anderen Laufzeiten

Formel: $O(1) + 2 * O(n) = O(n)$

für c):

$O(1)$: Wertdefinierung, Return

$O(n)$: Schleife

keine anderen Laufzeiten

Formel: $O(1) + O(n) = O(n)$

terminiert nur für positive Zahlen

für d):

$O(1)$: Wertdefinierung, return

$O(n)$: Schleife

keine anderen Laufzeiten

Formel: $O(1) + O(n) = O(n)$

für e):

$O(1)$: if, return a

$O(n)$: return $a * f_4(a, b - 1)$

keine anderen Laufzeiten

Formel: $O(1) + O(n) = O(n)$

terminiert nur für positive b

für Einfügen in binären Baum mit n Elementen: $O(1)$: setzen current, insertHere $O(n)$: keine

$O(\log_2(n))$: Suchen in binärbaum hat $\log_2(n)$

Formel: $O(1) + O(\log_2(n)) = O(\log_2(n))$

Anmerkung: n-maliges Einfügen hat $O(n * \log_2(n))$

für Suchen gleicher Elemente in zwei Liste: $O(1)$: Wertdefinierung, return

$O(n)$: keine

$O(n^2)$: geschachteltes foreach

Formel: $O(1) + O(n^2) = O(n^2)$

Aufgabe 2-2 *O*-Notation

Zeigen oder widerlegen Sie:

(a) $O(n) \in O(n^2)$

(b) $\log_{10}(x) \notin \log_2(x)$

(c) $O(n^2) \in O(n * \log_2(n))$

(d) $20000 * g(n) \in O(g(n))$

(e) $g(n) * n^2 \in O(n^2)$

Lösungsvorschlag:

- (a) $O(n) \in O(n^2)$ stimmt ($\lim_{n \rightarrow \infty} n/n^2 = 0$)
- (b) $\log_{10}(x) \notin O(\log_2(x))$ falsch - ist darin (siehe VL-Folien)
- (c) $O(n^2) \in O(n * \log_2(n))$ falsch - $\lim_{n \rightarrow \infty} \frac{n^2}{n * \log(n)} \neq 0$
- (d) $20000 * g(n) \in O(g(n))$ stimmt (konstante faktoren können vernachlässigt werden)
- (e) $g(n) * n^2 \in O(n^2)$ stimmt, wenn $g(n)$ in $O(1)$ ist, sonst nicht.

Aufgabe 2-3 Rekursionsgleichungen und Mastertheorem

a) Der Algorithmus, um die Türme von Hanoi zu lösen, hat als Java-Code folgende Form:

```
public static void Hanoi(int Scheibenindex, int start, int ziel, int temp){  
    if(Scheibenindex == 1) {  
        VersetzeObersteScheibe(start, ziel);  
    }  
    else {  
        Hanoi(Scheibenindex - 1, start, temp, ziel);  
        VersetzeObersteScheibe(start, ziel);  
        Hanoi(Scheibenindex - 1, temp, ziel, start);  
    }  
}
```

Die Zeitkomplexität der Funktion `VersetzeObersteScheibe(...)` sei $\mathcal{O}(1)$. Stelle die Rekursionsgleichung für diesen Algorithmus auf und verwende das Mastertheorem, um die Zeitkomplexität in Abhängigkeit des Parameters *Scheibenindex* zu ermitteln.

b) Löse die folgenden Rekursionsgleichungen mithilfe des Mastertheorems.

(i) $T(n) = 2T(\frac{n}{2}) + n^3$

(ii) $T(n) = T(\sqrt{n}) + 1$

(iii) $T(n) = a \cdot T(\sqrt[n]{n}) + \log_y(n)$ mit $x, y > 1$ und $\log_x(a) < 1$.

Hinweis für (iii): Substituiere $m = \log_y(n)$.

Lösungsvorschlag:

a) Die Rekursionsgleichung lautet: $T(n) = T(n-1)+1+T(n-1) = 2T(n-1)+1$. Nach dem Mastertheorem für Rekursionsgleichungen der Form $T(n) = aT(n-b) + f(n)$ ergibt das eine Laufzeitkomplexität von $\mathcal{O}(2^n)$.

b) Alle drei Teilaufgaben können mit dem Mastertheorem für Rekursionsgleichungen der Form $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ gelöst werden:

(i) Da $\log_2(2) = 1 < 3$ folgt $\mathcal{O}(n^3)$

(ii) Substituieren mit $m = \log_2(n) \Rightarrow 2^m = n$. Es folgt $T(2^m) = T(2^{\frac{m}{2}})+1$. Aus $S(m) = T(2^m)$ folgt $S(m) = S(\frac{m}{2})+1$. Lösen mithilfe des Mastertheorem ergibt: $S(m) \in \mathcal{O}(\log(m))$. Rücksubstituieren ergibt: $T(n) \in \mathcal{O}(\log(\log(n)))$

(iii) Auch durch Substitution lösbar. Setze $m = \log_y(n) \Rightarrow y^m = n$ um $T(y^m) = a \cdot T(y^{\frac{m}{x}}) + m$ zu erhalten. Wie zuvor ergibt sich mit $S(m) = T(y^m)$ folgende Rekursionsformel: $S(m) = a \cdot S(\frac{m}{x}) + m$. Aus der Bedingung $\log_x(a) < 1$, ergibt sich, welcher Fall des Mastertheorem für diese Gleichung relevant ist. Es gilt also $S(m) \in \mathcal{O}(m)$. Durch Rücksubstituieren erhält man $T(n) \in \mathcal{O}(\log_y(n))$.