Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. Thomas Seidl
Janina Sontheim, Florian Richter

München, 14.05.2019

Algorithmen und Datenstrukturen

SS 2019

Übungsblatt 3: Grundlagen

Aufgabe 3-1 *Mastertheorem*

Nutzen Sie das Mastertheorems um eine Laufzeitabschätzung zu geben. Sollte dies nicht möglich sein, begründen Sie warum.

(a) $T(n) = 8T(n/2) + n * log(n)$

(b) $T(n) = 7T(n/3) + n^2 + 4n + 1$

(c) $T(n) = n * T(n/2) + 1$

Lösungsvorschlag:

a)
$a = 8, b = 2, f(n) = n * log(n) \rightarrow 1 < d < 2$
Da $log_b(a) = log_2 8 = 3 > (2 > d > 1)$ folgt: $T(n) \in O(n^{log_2(8)}) = O(n^3)$
b)
$a = 7, b = 3, f(n) = n^2 + 4n + 1 \in O(n^2) \rightarrow d = 2$
$d > log_3(7) \rightarrow T(n) \in O(n^d) = O(n^2)$
c)
nicht mit dem Mastertheorem lösbar, da a = n und damit ist a keine Konstante.

Aufgabe 3-2 *Laufzeit*

Gegeben sei folgender Algorithmus, der das Maximum Subarray Problem löst:

```
public static int msp(int[] array) {
        int result = 0;
        int n = 0;

        for (int i = 0; i < array.length; i++){
            n = 0;
            for (int j = i; j < array.length; j++){
                n += array[j];
                if(n > result){
                    result = n;
                }
            }
        }
        return result;
}
```

(a) Welche worst-case Laufzeitkomplexität hat dieser Algorithmus?

(b) **Bonus:** Überlegen Sie sich, ob sie den Algorithmus auch effizienter gestalten können.

Lösungsvorschlag:

(a) Sei n die Länge des Input-Arrays.
Die erste for-Schleife läuft über das gesamte Array, man hat also n Zugriffe.
Die zweite (verschachtelte) for-Schleife läuft vom Stand der ersten Schleife bis zum Ende des Arrays, hat also im Durchschnitt $\frac{n}{2}$ Zugriffe.
Durch die Verschachtelung müssen bei der Bestimmung der Gesamtlaufzeit die Laufzeiten der for-Schleifen multipliziert werden, man erhält also: $O(n \cdot \frac{n}{2}) = O(\frac{n^2}{2}) = O(n^2)$ (quadratische Laufzeit).

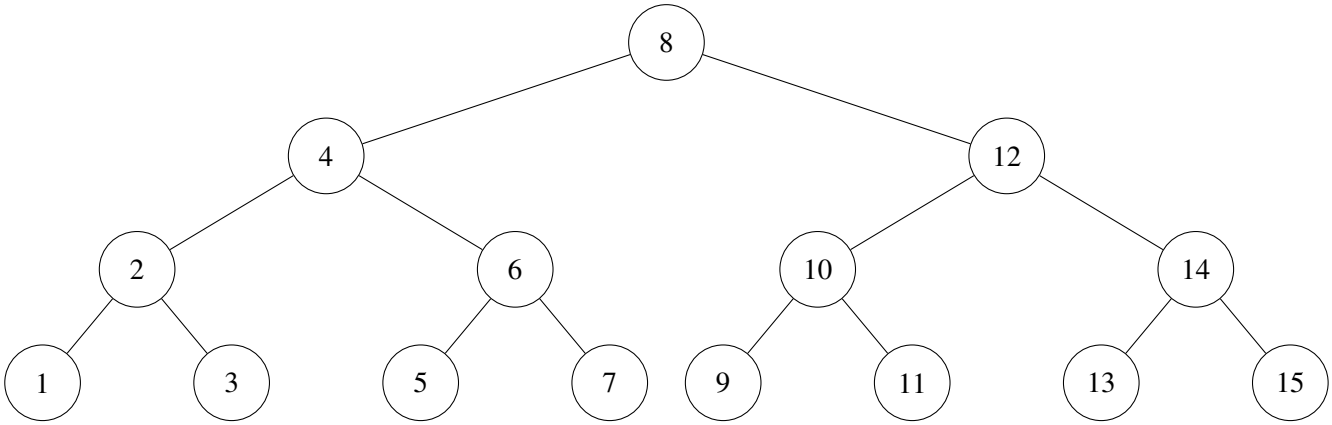
(b) Eine grundlegende Idee wäre, das Array von vorne bis hinten einmal durchzulaufen, und Schritt für Schritt die Summe s zu bilden. Falls s unter Null fällt, wäre es ja offensichtlich besser, mit der Berechnung der Summe neu zu starten. Wenn das Ende des Arrays erreicht ist, wird der aktuelle Wert für s zurückgegeben.
Mit diesem Ansatz erhält man eine Laufzeit von $\mathcal{O}(n)$
Beispielimplementierung:

```
private static int msp(int[] array){
        int currSum = 0;
        int maxSum = Integer.MIN_VALUE;

        for(int i = 0; i < array.length; i++){
            // "Neustart" falls die Summe negativ wird
            if(currSum <= 0){
                currSum = 0;
            }
            currSum += array[i];
            if (currSum > maxSum){
                maxSum = currSum;
            }
        }
        return maxSum;
    }
```

Aufgabe 3-3 *Eigenschaften von Bäumen*

Gegeben ist folgender Baum:



Hinweis: Die Höhe h wird wie in der Vorlesung definiert!
Beantworten Sie nun folgende Fragen. Achten Sie wo nötig auf eine formal korrekte Herleitung!

(a) Um welche Art von Baum handelt es sich? Welche Arität (=Stelligkeit) besitzt er?

(b) Welche Höhe h hat dieser Baum? Wie viele Knoten k und wie viele Blätter b besitzt er?

(c) Angenommen man würde die Höhe um 2 erhöhen, wie viele Knoten k muss der Baum nun mindestens haben und wie viele kann er maximal haben, wenn sich die bestehenden Einträge des Baumes dabei nicht ändern?

(d) Wie viele Knoten k hat der Baum mindestens und maximal, wenn die Höhe nun um n erhöht wird?

(e) Wie ändern sich die Formeln aus der Vorlesung, wenn wir anstelle eines Baums, in denen jeder Knoten maximal 2 Nachfolger hat, einen Baum mit Arität n annehmen? Geben Sie die allgemeinen Formeln in Abhängigkeit der Höhe h an, für:

  (i) Die minimale Knotenanzahl k_{min}

  (ii) Die maximale Knotenanzahl k_{max}

  (iii) Die maximale Anzahl der inneren Knoten k_{in}

  (iv) Die maximale Anzahl der Blätter b

(f) **Bonus:** Die in der Vorlesung vorgestellten Baumtraversierungsmöglichkeiten haben alle lineare Laufzeit in Bezug auf die Knotenanzahl k des Baumes. Begründen Sie warum. Wie viele Schritte brauchen Sie im Durchschnitt, wenn Sie ein Element des Baumes auf diese Weise suchen wollen?

(g) **Bonus:** Angenommen alle Einträge sind wie im obigen Baum der Größe nach sortiert, wie könnte dann ein effizienter Algorithmus aussehen, um ein beliebiges Element des Baumes zu suchen? Welche Laufzeit hätte der Algorithmus? Und wie ändert sich die Laufzeit bei einem allgemeinen Baum mit Arität n?

Lösungsvorschlag:

(a) Es handelt sich um einen vollständigen (ausgeglichenen) Binärbaum mit Arität 2. Dieser Baum ist ebenfalls ein binärer Suchbaum (Binary Search Tree)(siehe Vorlesung in späteren Kapiteln)

(b) Höhe: $h = 3$, Knoten: $k = 15$, Blätter: $b = 8$

(c) Neue Höhe: $h_{neu} = h_{alt} + 2 = 3 + 2 = 5$,
Knoten: $k_{alt} + 2 \leq k \leq 2^{h_{neu}+1} - 1 \Leftrightarrow 15 + 2 \leq k \leq 2^{5+1} - 1 \Leftrightarrow 17 \leq k \leq 63$

(d) Neue Höhe: $h_{neu} = n + 3$,
Knoten: $k_{alt} + n \leq k \leq 2^{(n+3)+1} - 1 \Leftrightarrow n + 15 \leq k \leq 2^{n+4} - 1 \Leftrightarrow n + 15 \leq k \leq 16 \cdot 2^n - 1$

(e) (i) Es gilt weiterhin: $k_{min} = h + 1$

(ii) Für die maximale Knotenzahl gilt, dass für jede inkrementelle Vergrößerung von h n Knoten als Kinder pro bisherigen Blatt-Knoten hinzukommen. Für Höhe 0 ist das also n^0 Knoten (die Wurzel), in Höhe 1 befinden sich maximal n^1 Knoten, in Höhe 2 dann maximal n^2 usw.. Da uns die Summe aller Knoten interessiert, ergibt sich damit für Höhe h und Arität n: $k_{max} = \sum_{i=0}^{h} n^i$. Das Ganze lässt sich noch kürzen: $k_{max} = \sum_{i=0}^{h} n^i = 1 + n + n^2 + \cdots + n^h = \frac{(1+n+n^2+\cdots+n^h)\cdot(n-1)}{n-1} = \frac{(n+n^2+n^3+\cdots+n^{h+1})-(1+n+n^2+\cdots+n^h)}{n-1} = \frac{n^{h+1}+(n^h-n^h)+(n^{h-1}-n^{h-1})+\cdots+(n^2-n^2)+(n-n)-1}{n-1} = \frac{n^{h+1}-1}{n-1}$
(Zum Vergleich binärer Baum: $K = \frac{2^{h+1}-1}{2-1}$)

(iii) Es gilt offensichtlich, dass die Anzahl der inneren Knoten (also aller Knoten ohne die Blätter) für Höhe h der Anzahl aller Knoten für Höhe $h-1$ entspricht. Wir nutzen diese Erkenntnis und setzen in die Formel aus obiger Teilaufgabe ein. Damit ergibt sich: $k_{in} = \frac{n^{(h-1)+1}-1}{n-1} = \frac{n^h-1}{n-1}$

(iv) Die Anzahl der Blätter liegt bei: $b = n^h$

(f) **Bonus:** Wenn wir den Inhalt eines Arrays mit k Elementen in einem Baum speichern, bleibt uns die Anzahl der Elemente des Arrays erhalten. Beim Traversieren des Baumes geht es darum, diese Elemente wieder in eine neue Liste bzw. Array einzuordnen. Die Reihenfolge der Einträge dieses Arrays hängt davon ab, welche Traversierungsstrategie (Inorder, Preorder, Postorder etc.) gewählt wurde. Die Anzahl k der Elemente des Baumes bleibt jedoch erhalten. Das Traversieren des Baumes braucht umso länger je mehr Elemente der Baum besitzt. Da die Elemente einzeln untersucht bzw. abgearbeitet werden - und das ohne weitere Berechnungen - ist die Komplexität für diesen Vorgang offensichtlich linear in Bezug auf die Anzahl der Elemente k des Baumes, also $O(k)$.
Wenn wir nun ein Element des Baumes suchen wollen, können wir das hier nur auf die folgene Art und Weise machen:

(i) Traversiere den Baum und speichere das Ergebnis der Traversierung in einer neuen Liste je nach Traversierungsstrategie

(ii) Führe eine lineare Suche aus: Iteriere über die Liste und überprüfe, ob das gesuchte Element mit dem aktuell geprüften Element übereinstimmt. Tue das solange, bis das gesuchte Element gefunden wurde oder melde es als nicht gefunden, wenn über die gesamte Liste iteriert wurde.

Wenn wir nun annehmen, dass die Daten des Baumes zufallsverteilt sind, benötigt man im besten Fall eine Vergleichsoperation und im schlechtesten Fall k Vergleichsoperationen. Für den Mittel-/Durchschnittswert ergibt sich: $\frac{1}{k}\sum_{i=1}^{k} i \overset{Gaußsche Summenformel}{=} \frac{1}{k} \cdot \frac{(k+1)k}{2} = \frac{k+1}{2}$

Lösungsvorschlag:

(g) **Bonus:** Wenn die Elemente wie im obigen Beispiel sortiert sind, spricht man von einem balancierten binären Suchbaum. Wie der Name schon erkennen lässt, ist diese Anordnung sehr gut dafür geeignet, Elemente innerhalb des Baumes zu suchen. Ein effizienter Algorithmus könnte folgendermaßen aussehen:

(i) Beginne mit der Wurzel: Überprüfe, ob das gesuchte Element dem Element der Wurzel entspricht.

(ii) Wenn ja: Fertig!

(iii) Wenn nein: Überprüfe, ob gesuchtes Element größer oder kleiner als Element der Wurzel ist.

(iv) Wenn das gesuchte Element größer als das untersuchte ist, prüfe, ob rechter Teilbaum existiert.

(v) Wenn rechter Teilbaum existiert: Gehe zu rechtem Teilbaum und führe dort den Algorithmus durch (Von Schritt 1 an).

(vi) Wenn rechter Teilbaum nicht existiert: Element wurde nicht gefunden!

(vii) Wenn das gesuchte Element kleiner als das untersuchte ist, prüfe, ob linker Teilbaum existiert.

(viii) Wenn linker Teilbaum existiert: Gehe zu linkem Teilbaum und führe dort den Algorithmus durch (Von Schritt 1 an).

(ix) Wenn linker Teilbaum nicht existiert: Element wurde nicht gefunden!

Für die Komplexität ergibt sich:

- Für ein Element auf einem Level des Baumes werden maximal 3 Vergleiche durchgeführt:
  (i) Ist das gesuchte Element gleich der Wurzel?
  (ii) Ist das gesuchte Element größer als das untersuchte? (Kleiner muss nicht mehr explizit geprüft werden, da das gesuchte Element automatisch kleiner sein muss, wenn es nicht größer oder gleich ist dem untersuchten Element ist).
  (iii) Existiert der Teilbaum? (Entweder links oder rechts je nachdem, ob das Element größer oder kleiner ist.)

  Damit hat eine Iteration des Algorithmus eine Komplexität von $O(3)$
- Es wird maximal die Anzahl der Höhe h an Iterationen benötigt. Die Höhe h entspricht aufgerundet $\log_2 k$ (Herleitung offensichtlich). Für die Komplexität ergibt sich damit: $O(\log_2 k)$
- Da $O(3) \subset O(\log_2 k)$ ergibt sich für die Komplexität insgesamt $O(\log_2 k)$.

Wenn wir nun einen sortierten Baum mit Arität n haben, müssen wir für eine einzelne Iteration nicht nur konstant 3 Vergleiche durchführen, sondern maximal $n+2$ Vergleiche, wenn wir von n Partitionen für die Kinder eines Knotens ausgehen. Damit ergibt sich bereits für eine einzelne Iteration eine Komplexiät von $O(n)$. Für maximal $\log_2 k$ Iterationen wäre das theoretisch eine Komplexität von $O(n \cdot \log_2 k)$. Je nach Wahl von n handelt es sich bei großen n jedoch mehr um eine Liste als um einen Baum. Wenn sich alle Elemente bereits in der Wurzel befinden ($n \geq k$) ist es sogar eine Liste. Da sich während des Traversierens auch gleich Elemente suchen lassen, kann man also hier den Traversierungsalgorithmus mit einer kombinierten Suche in Laufzeit $O(k)$ durchführen und ist in diesem Fall schneller. Die Struktur eines solchen Baumes mit Arität n ist jedoch wie gezeigt nicht sehr sinnvoll und wird daher in der Praxis nicht verwendet.