Technische Universität München
Institut für Informatik I-16
Dr. Tobias Lasser
Jakob Vogel

WiSe 2012/13
Lösungsblatt 7
3. Dezember 2012

Algorithmen und Datenstrukturen

Aufgabe 1 Rechnen mit Landau-Symbolen (Beispiellösung)

Die Definitionen der beiden für uns relevanten Landau-Symbole waren:

$$\begin{aligned}\Theta(g) &:= \{f : \mathbb{R} \to \mathbb{R} \ : \text{ es existieren } c_1, c_2 > 0 \text{ und } n_0 \in \mathbb{N}, \text{ so dass für alle } n \geq n_0 \text{ gilt:} \\ &\qquad 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\} \\ O(g) &:= \{f : \mathbb{R} \to \mathbb{R} \ : \text{ es existieren } c > 0 \text{ und } n_0 \in \mathbb{N}, \text{ so dass für alle } n \geq n_0 \text{ gilt:} \\ &\qquad 0 \leq f(n) \leq c \cdot g(n)\}\end{aligned}$$

a) *Die Worst-Case-Komplexität eines Algorithmus sei $\Theta(n)$. Nachdem auch $2n = \Theta(n)$, entspricht die Dauer eines Worst-Case-Beispiels der Länge $2n$ ungefähr der Dauer eines Worst-Case-Beispiels der Länge n.*

Die Behauptung ist **falsch**, da konstante Faktoren der tatsächlichen Laufzeitfunktion bei Landau-Symbolen der Einfachheit halber entfallen, und nur das grundsätzliche Wachstum abgeschätzt wird.

Daher wird das doppelt so große Beispiel tatsächlich etwa doppelt so lange rechnen, wie das einfacher Größe.

b) *Sei die Komplexität einer Funktion f bestimmt als $O(n)$. Dann ist die n-malige Ausführung $O(n^2)$.*

Sei $T_f(n)$ die Laufzeitfunktion von f. Aus der Annahme folgt, dass es c und n_0 gibt, so dass

$$0 \leq T_f(n) \leq c \cdot n$$

für alle $n \geq n_0$. Bei n-maliger Ausführung muss einfach multipliziert werden:

$$0 \leq n \cdot T_f(n) \leq n \cdot c \cdot n = c \cdot n^2$$

Das ist direkt die Forderung aus der Definition von $O(n^2)$, die Aussage ist **korrekt**! □

c) *Falls $f(n) = \Theta(g(n))$, dann folgt $2^{f(n)} = \Theta(2^{g(n)})$.*

Wähle $f(n) = n$ und $g(n) = 2 \cdot n$, dann gilt offensichtlich $f(n) = \Theta(g(n))$:

$$0 \leq c_1 \cdot 2 \cdot n \leq n \leq c_2 \cdot 2 \cdot n \ ,$$

etwa mit $c_1 = c_2 = \frac{1}{2}$ und $n_0 = 1$. Nehmen wir nun an, dass die Aussage korrekt ist, und wir erhalten aus $2^{f(n)} = \Theta(2^{g(n)})$ die Ungleichung

$$0 \leq c_1 \cdot 2^{2 \cdot n} \leq 2^n \leq c_2 \cdot 2^{2 \cdot n} \ .$$

Vereinfachen wir diese Formel nun durch Teilung durch 2^n, was wegen $n > 0$ ohne Umkehr der Vorzeichen problemlos möglich ist:

$$0 \leq c_1 \cdot 2^n \leq 1 \leq c_2 \cdot 2^n \ .$$

Dies ist aber ein Widerspruch, da wir zwar ein c_2, nicht aber ein c_1 finden können, da die Folge 2^n unbeschränkt ist.

Damit ist die komplette Aussage **falsch**, da mit der getroffenen Auswahl für f und g ein Gegenbeispiel gefunden wurde. □

d) $n^n = O(2^n)$

Nehmen wir wieder an, dass die Aussage wahr ist. Dann gilt aus der Definition von O die Ungleichung

$$0 \leq n^n \leq c \cdot 2^n \ ,$$

was sich (wieder wegen $2^n > 0$ für $n > 0$) umformen lässt zu

$$\left(\frac{n}{2}\right)^n \leq c \ .$$

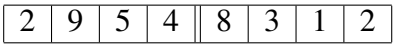
Auch diese Folge ist unbeschränkt und wächst beliebig für große Werte von n. Damit kann keine Konstante c im Sinn einer oberen Schranke existieren.

Damit ist die Aussage als **falsch** widerlegt! □

Aufgabe 2 Sortieren mittels Divide-and-Conquer-Ansätzen (Beispiellösung)

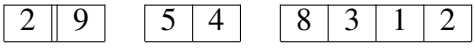
a) Die Idee hinter Merge Sort war ja, das Array mittig aufzuspalten, die Teile rekursiv sortieren zu lassen, und dann aus den sortierten Teilen das Ergebnis zusammenzufügen.

Ausgangsdaten

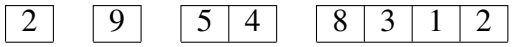


Divide

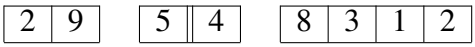
Divide



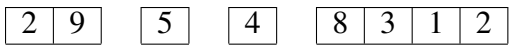
Divide



Conquer/Merge

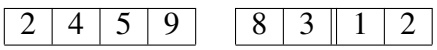


Divide

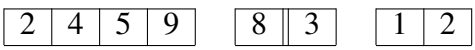


Conquer/Merge

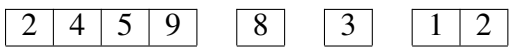
Conquer/Merge



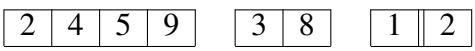
Divide



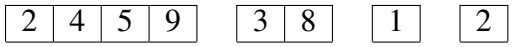
Divide



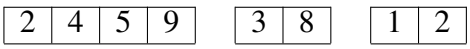
Conquer/Merge



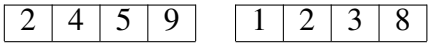
Divide



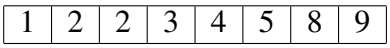
Conquer/Merge



Conquer/Merge



Conquer/Merge

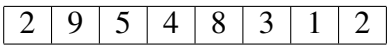


Ergebnis

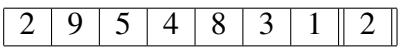
b) Beim Quick Sort ist die Strategie etwas anders. Zwar wird auch hier unterteilt, doch geschieht dies nicht mittig, sondern basierend auf einem Pivot-Element[1], bezüglich dessen in einen vorderen Teil mit kleineren Elementen und einen hinteren Teil mit größeren Elementen sortiert wird. Es gibt verschiedene Auswahlstrategien für den Pivot, und wir wählen hier immer das *letzte* Element.

Der Quick Sort ist zwar ein Divide-and-Conquer-Algorithmus, aber ein offensichtlicher Conquer-Schritt fehlt. Der Grund ist der, dass die rekursiv aufgerufene Funktion die Daten bereits „richtig“ abliefert, und so die Zusammenfügung implizit erfolgt ist. Im folgenden Schaubild sind derartige, nur zum Verständnis aufgeführte, Schritte durch drei Punkte gekennzeichnet.

Ausgangsdaten

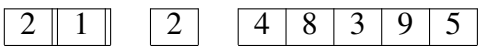


Pivot-Wahl



Partitionierung

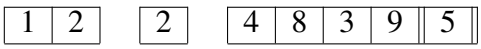
Pivot-Wahl



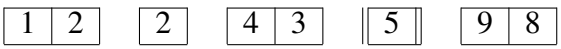
Partitionierung

Trivial-Schritt

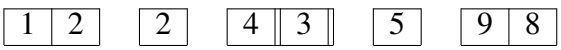
Pivot-Wahl



Partitionierung



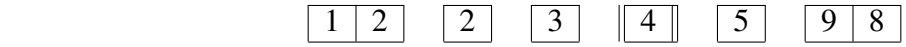
Pivot-Wahl



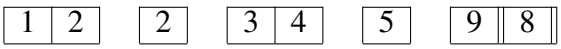
Partitionierung

[1] Dreh-, Angelpunkt

Trivial-Schritt

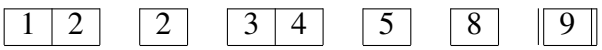


Pivot-Wahl

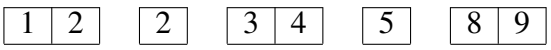


Partitionierung

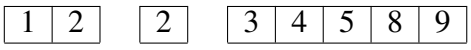
Trivial-Schritt



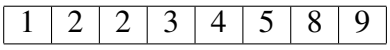
...



...



...



Ergebnis

Aufgabe 3 Stabilität von Sortierverfahren (Beispiellösung)

Wir bezeichnen ein Sortierverfahren als **stabil**, wenn bei gleichrangigen Elementen während der Sortierung die vorherige Ordnung erhalten bleibt.

- Der *Insertion Sort* ist **stabil**.

  Dies ist offensichtlich, da der Algorithmus den unsortierten Teil der Reihe nach durchgeht, und das Element (von hinten her Platz schaffend) in den sortierten Teil einfügt. Sollte also ein gleichrangiges Element vorhanden sein, so wird das neue Element als dessen Nachfolger einsortiert.

- Der *Selection Sort* ist **nicht stabil**.

  Es wird zwar stets aus dem unsortierten Teil das Minimum gesucht und eingefügt, aber der Platz wird nicht durch „Rücken“, sonder durch Vertauschen geschaffen. Insofern kann sich hier die Reihenfolge gleichrangiger Elemente ändern.

  Man könnte den Selection Sort stabil machen, indem der Platz eben doch durch schrittweises Verrücken der Elemente geschaffen würde. Dies würde jedoch erheblichen Aufwand bedeuten.

- Der *Merge Sort* ist in der gegebenen Formulierung **nicht stabil**.

  Der einzige Vertauschungsfall kann dabei passieren, wenn in der Merge-Funktion im Falle gleichrangiger Anfänge der Teillisten aus dem hinteren Teil gelesen würde. Wenn wir jedoch explizit fordern, dass in diesem Falle immer nur vorne ausgelesen wird, dann ist der Merge Sort stabil.

- Der *Quick Sort* ist **nicht stabil**.

  Die Vertauschungen können laufend passieren, wenn nach der Pivot-Wahl die Menge partitioniert wird. Man kann Quick Sort auch stabil implementieren, doch vernachlässigen wir diese Möglichkeit wegen ihrer Komplexität.

Aufgabe 4 Komplexität der Polynom-Auswertung (Beispiellösung)

In dieser Aufgabe gehen wir etwas anders vor, und zählen nicht wie im RAM-Modell alle Rechenschritte, sondern begnügen uns damit, „teure“ arithmetische Operationen zu zählen. Im vorliegenden Fall sind dies vor allem die Multiplikationen.

Vergegenwärtigen Sie sich, dass wir bei einem Polynom des Grads n von $n+1$ Koeffizienten ausgehen (auch wenn diese dann den Wert 0 haben können).

a) Betrachten wir nochmals den Code der einfachen Implementierung:

```
1       p = 0;
2       for i = n down to 0 {
3           m = 1;
4           for j = 1 to i {
5               m = m · x;
            }

6           m = a[i] · m;
7           p = p + m;
        }
```

Im Folgenden vernachlässigen wir die beiden Initialisierungen in den Zeilen 1 und 3. Für jeden Durchlauf i der Schleife (ab Zeile 2) haben wir also folgenden Aufwand:

- i Multiplikationen für die i-te Potenz in Zeilen 4 und 5
  Beachten Sie dabei, dass die letzte Iteration für $i = 0$ diesen Schritt überspringt!
- 1 Multiplikation für den Koeffizienten in Zeile 6
- 1 Addition für die Summierung in Zeile 7

Damit kommen wir bei $n+1$ Iterationen also auf folgenden Gesamt-Aufwand:

- Multiplikationen:

$$\left(\sum_{i=1}^{n} i\right) + (n+1) = \left(\frac{n(n+1)}{2}\right) + (n+1) = \frac{1}{2}n^2 + \frac{3}{2}n + 1$$

- Additionen:

$$n+1$$

- Die Laufzeitfunktion ist damit mindestens so groß wie die Summe dieser beiden Zahlen, da wir ja zusätzlichen Aufwand wie hauptsächlich die Schleifenverwaltung (vom Aufwand $O(n)$) ignorieren. Also folgern wir:

$$T(n) = O(n^2)$$

b) Dies bringt uns zur nächsten Implementierung, bei der die Berechnung der Potenz inkrementell über die Hilfsvariable h erfolgt:

```
1       p = 0;
2       h = 1;
3       for i = 0 to n {
4           m = a[i] · h;
5           h = h · x;
6           p = p + m;
        }
```

Auch hier vernachlässigen wir wieder die Initialisierungen in Zeilen 1 und 2. Für jeden Durchlauf der Schleife (ab Zeile 3) ergibt sich also:

- 1 Multiplikation für den Koeffizienten in Zeile 4
- 1 Multiplikation für die Potenzierung in Zeile 5
- 1 Addition für die Summierung in Zeile 6

Somit lässt sich also der Gesamt-Aufwand bei $n+1$ Iterationen errechnen:

- Multiplikationen:
$$2(n+1) = 2n+2$$
Nachdem die Zeile 5 *ganz am Schluss einmal zu oft ausgeführt wird, ließe sich der Aufwand um eine Multiplikation reduzieren. Außerdem liessen sich zwei weitere Multiplikationen vermeiden, da ja im ersten Durchlauf (mit* $h = 1$*) zwei überflüssige Multiplikationen mit* 1 *stattfinden. Die minimale Anzahl an Multiplikationen ist daher* $2n-1$.
- Additionen:
$$n+1$$
- Die Laufzeitfunktion ist hier wieder mindestens so groß wie die Summe dieser beiden Zahlen, und wieder käme zusätzlicher Aufwand für die Schleifenverwaltung hinzu. Also:
$$T(n) = O(n)$$

c) Betrachten wir zuletzt das Horner-Schema:

```
1       p = a[n];
2       for i = n − 1 down to 0 {
3           p = p · x;
4           p = p + a[i]
        }
```

Dieser Code ist vollständig optimiert und kann durch die Zuweisung in Zeile 1 eine komplette Schleifeniteration einsparen. In einem einzigen Durchlauf haben wir dabei folgenden Aufwand:

- 1 Multiplikation mit einem ausgeklammerten Faktor der Potenz in Zeile 3
- 1 Addition für die Summierung in Zeile 4

Bei nur n Iterationen ergibt sich also dieser Gesamtaufwand:

- Multiplikationen:
$$n$$
- Additionen:
$$n$$
- Wie zuvor ergibt sich die Laufzeitfunktion als Summe der beiden Zahlen und muss durch Schleifenverwaltungsaufwand der Größe $O(n)$ ergänzt werden. Zusammen also:
$$T(n) = O(n)$$

d) Zuletzt waren die drei Implementationen zu vergleichen. Offensichtlich ist der naïve Ansatz wegen seiner $O(n^2)$ Komplexität der eindeutige Verlierer gegen die beiden anderen Methoden. Interessanter ist dagegen der Vergleich zwischen inkrementeller Berechnung der Potenz und dem Horner-Schema, da beide ja in der gleichen Komplexitätsklasse $O(n)$ sind.

Wie Sie sich erinnern, unterscheiden sich die Elemente einer Komplexitätsklasse durchaus um konstante Faktoren, lediglich das generelle Wachstumsverhalten ist ab einer bestimmten Problemgröße vergleichbar. Insofern können zwei $O(n)$ Lösungen doch unterschiedlich schnell sein, alleine der Beschleunigungsfaktor ist garantiert unabhängig von n.

Wenn man sich also auf die Zählung von Multiplikationen beschränkt – die auf klassischen Systemen den größten Aufwand erzeugen –, so wird das Horner-Schema mit seinen n Multiplikationen gegenüber der inkrementellen Berechnung mit ihren minimal $2n - 1$ Multiplikationen stets etwa doppelt so schnell sein. Damit ist im Hinblick auf Geschwindigkeitsoptimalität das Horner-Schema der eindeutige Gewinner.