Technische Universität München
Institut für Informatik I-16
Dr. Tobias Lasser
Matthias Wieczorek

SoSe 2015
Lösungsblatt 4
20. Mai 2015

Algorithmen und Datenstrukturen

Aufgabe 1 Komplexität – Berechnung der Fibonacci-Zahlen (Beispiellösung)

a) Betrachten wir nochmals den Code der rekursiven Implementierung:

```
1   Input: Index n der Fibonacci Folge
2   Output: Wert f_n
3   fib(n):
4   if (n == 1 || n == 2) {
5       return 1;
6   }
7   else {
8       return fib(n − 1) + fib(n − 2);
9   }
```

Wie sehen, dass $T(1) = T(2) = 1$ gilt, da der Ausgabewert direkt zurück gegeben werden kann. Für $n \geq 3$ erhalten wir eine Rekursionsformel für $T(n)$, welche der Formel der Fibonacci-Zahlen stark ähnelt:

$$T(n) = 1 + T(n-1) + T(n-2)$$

Da wir eine Addition und zwei weitere Aufrufe der Funktion benötigen.

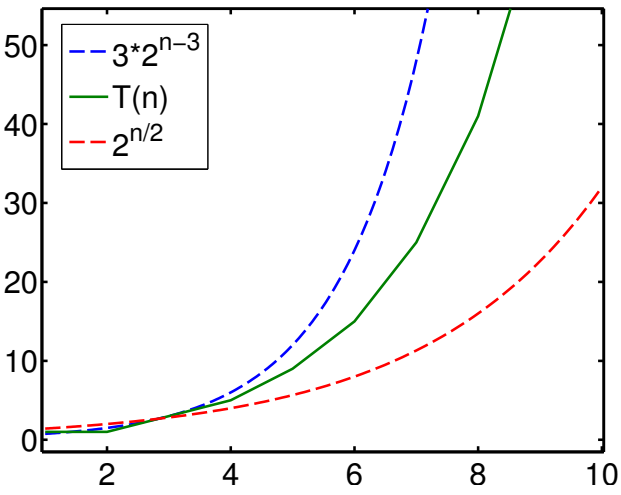
Weiter gilt für $n \geq 4$: $1 + T(n-1) + T(n-2) \leq 2 * T(n-1)$ und damit:

$$T(n) \leq 2 * T(n-1) \leq \cdots \leq 2^{n-3} * T(3) = 2^{n-3} * 3$$

Damit folgt $T = O(2^n)$.

Darüber hinaus gilt für $n \geq 3$: $1 + T(n-1) + T(n-2) \geq 2 * T(n-2)$, woraus $T(n) \geq 2^{n/2}$ folgt.

Zur Veranschaulichung dieser Abschätzungen betrachten Sie bitte den folgenden Graphen.



b) Im Falle der Implementierung mittels Dynamischer Programmierung:

```
1   Input: Index n der Fibonacci Folge
2   Output: Wert f_n
3   fibDyn(n):
4   fib = leeres Feld Größe n + 1;
5   fib[1] = 1;
6   fib[2] = 1;
7   for k = 3 to n {
8       fib[k] = fib[k-1] + fib[k-2];
9   }
```

In diesem Fall gilt:

$$T(n) = 2 + (n-2) = n$$

woraus $T = O(n)$ folgt.

c) Für die rekursive Berechnung haben wir zuvor folgende Abschätzung hergeleitet (für $n \geq 4$, was für $n = 100$ natürlich gegeben ist):

$$3 * 2^{n-3} \leq T(n) \leq 2^{n/2}$$

Im Falle des angegebenen Rechners erhalten wir also eine obere Schranke von ($1ns = 10^{-9}s; 1a = 365 * 86400s$):

$$0.4 * 3 * 2^{97} ns \approx 1.9015 * 10^{20} s \approx 6 * 10^{12} a$$

und eine untere Schranke von

$$0.4 * 2^{50} ns \approx 5d$$

Im Gegensatz hierzu erhalten wir im Fall der Implementierung mittels Dynamischer Programmierung eine Laufzeit von $40ns$:

Hinweis: Es geht noch besser! Die Rekursionsformel lässt sich mathematisch lösen, so dass eine Fibonacci-Zahl in $O(1)$ berechnet werden kann.

Aufgabe 2 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.

Aufgabe 3 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

2	9	5	4	8	3	1	2

Divide

2	9	5	4

8	3	1	2

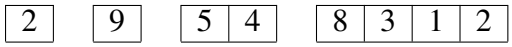
Divide

2	9

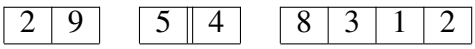
5	4

8	3	1	2

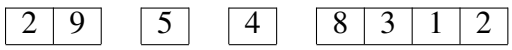
Divide



Conquer/Merge

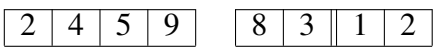


Divide

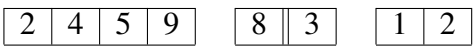


Conquer/Merge

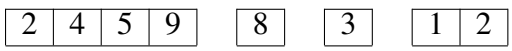
Conquer/Merge



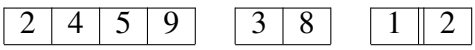
Divide



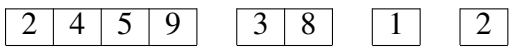
Divide



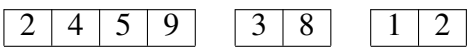
Conquer/Merge



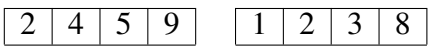
Divide



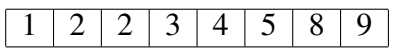
Conquer/Merge



Conquer/Merge



Conquer/Merge

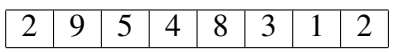


Ergebnis

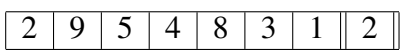
b) Da es unterschiedliche Auswahlstrategien für das Pivot-Element gibt, wählen wir hier immer das *letzte* Element.

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.

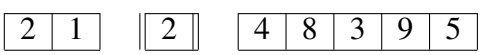
Ausgangsdaten



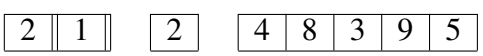
Pivot-Wahl



Partitionierung

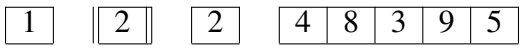


Pivot-Wahl

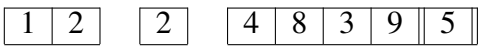


Partitionierung

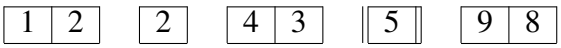
Trivial-Schritt



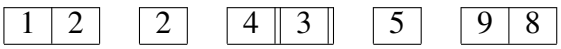
Pivot-Wahl



Partitionierung



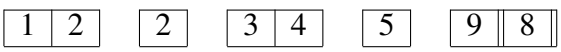
Pivot-Wahl



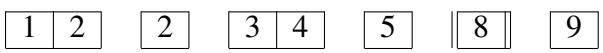
Partitionierung

Trivial-Schritt

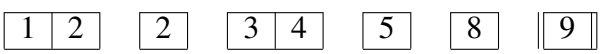
Pivot-Wahl



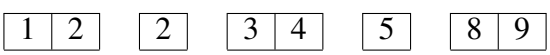
Partitionierung



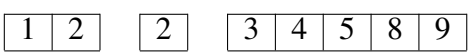
Trivial-Schritt



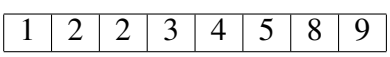
...



...



...



Ergebnis

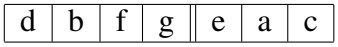
Aufgabe 4 Merge Sort (Beispiellösung)

a) Es fallen uns zwei Besonderheiten auf. Erstens handelt es sich bei der Eingabelänge n nicht um eine Zweier-Potenz weswegen bei den Divide-Schritten Teilfelder entstehen, welche eine ungerade Länge haben. Um auch solche Arrays mittels Merge Sort sortieren zu können muss die Divide Strategie derart angepasst werden, dass sie auch auf ungerade Längen anwendbar ist. Wir wählen für n ungerade folgende Strategie:

$$A = \{a_1, \ldots, a_n\} \rightarrow A_{\text{left}} = \{a_1, \ldots, a_{\frac{n+1}{2}}\}, A_{\text{right}} = \{a_{\frac{n+1}{2}+1}, \ldots, a_n\}$$

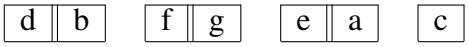
Die zweite Besonderheit ist, dass es sich bei den Elementen des Arrays um Buchstaben anstelle von Zahlen handelt. Um diese sortieren zu können müssen wir also eine Ordnung auf diesen Elementen definieren. Wir wählen hierzu die Ordnung des Alphabets (lexikographisch).

b) Ausgangsdaten



Divide

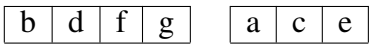
Divide



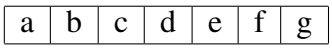
Divide

Conquer/Merge

Conquer/Merge



Conquer/Merge



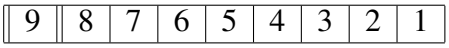
Ergebnis

Aufgabe 5 Quick Sort – Pivoting (Beispiellösung)

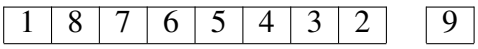
a) Zuerst wählen wir das Pivotelement stets als das erste Element aus dem Teilarray:

Ausgangsdaten

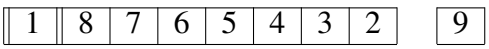
Pivot-Wahl



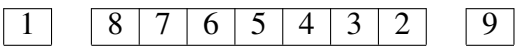
Partitionierung



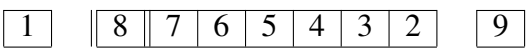
Pivot-Wahl



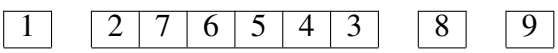
Partitionierung



Pivot-Wahl



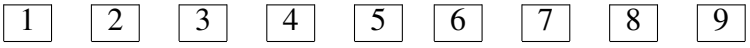
Partitionierung



...

...

...



Trivial-Schritt

Ergebnis

b) Nun wählen wir statt dem ersten Element stets das mittlere Element (bzw. das Element mit Index $\frac{n}{2}+1$ bei gerader Eingabelänge):

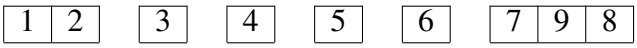
Ausgangsdaten

Pivot-Wahl

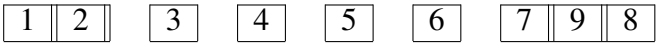
Partitionierung

Pivot-Wahl

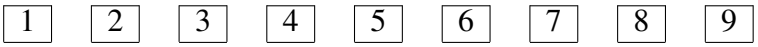
Partitionierung



Pivot-Wahl



Partitionierung



Trivial-Schritt

Ergebnis

c) Wir stellen fest, dass wir im Fall der ersten Pivot Strategie (immer das erste Element) die Worst-case Laufzeit erhalten, da die Arraygröße bei jeder Partition lediglich um Eins reduziert wird. Wenn wir hingegen die zweite Strategie wählen, erhalten wir eine ausgewogenere Partitionierung und eine deutlich verbesserte Laufzeit.

Hinweis: Bitte beachten Sie, dass es theoretisch für jedes Eingabearray eine Pivotstrategie gibt, so dass die Partitionierungen ausgewogen sind. Allerdings sind diese Lösungen individueller Natur! Es lässt sich keine Strategie finden, welche dieses Problem im Allgemeinen löst. Anders gesagt gibt es zu jeder Pivotstrategie ein Eingabearray, so dass ein derartiger Worst-Case auftritt. Daher ergibt sich eine Worst-Case Laufzeit von $O(n^2)$.