Technische Universität München
Fakultät für Informatik, I-16
Dr. Stefanie Demirci
Rüdiger Göbl, Mai Bui

SoSe 2017
Übungsblatt 4
31. Mai 2017

Algorithmen und Datenstrukturen

Aufgabe 1 Komplexität – Berechnung der Fibonacci-Zahlen

Wie untersuchen im Folgenden die Komplexität der Berechnung der Fibonacci-Zahlen f_n, welche für $n \geq 3$ wie folgt rekursiv definiert sind:

$$f_n = f_{n-1} + f_{n-2}$$

mit den Anfangswerten $f_1 = 1, f_2 = 1$. Hierzu betrachten wir die zwei Implementationsvarianten aus der Vorlesung:

a) Berechnung mittels Rekursion:

```
Input: Index n der Fibonacci Folge
Output: Wert f_n
fib(n):
if (n == 1 || n == 2) {
        return 1;
}
else {
        return fib(n − 1) + fib(n − 2);
}
```

Bestimmen Sie die Laufzeitfunktion dieser Implementierung! In welcher Effizienzklasse in O-Notation ist dieser Code?

b) Berechnung mittels Dynamischer Programmierung:

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

Bestimmen Sie die Laufzeitfunktion dieser Implementierung! In welcher Effizienzklasse in O-Notation ist dieser Code?

c) Nehmen Sie nun an, dass Sie einen Rechner mit 2.5GHz Taktfrequenz zur Verfügung haben. Weiter nehmen wir an, dass jede Operation exakt 1 Takt benötigt, d.h. eine Operation benötigt $1/2.5ns = 0.4ns$ zur Ausführung. Schätzen sie (mathematisch!) die Berechnungsdauer der 100sten Fibonacci-Zahl ab.

Aufgabe 2 Komplexität der Polynom-Auswertung

In allgemeiner Form ist ein *Polynom* gegeben durch

$$p(x) := \sum_{i=0}^{n} a_i x^i = a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x + a_0 \ ,$$

wobei $a_i \in \mathbb{R}$ der Koeffizient für die i-te Potenz x^i ist. Die höchste Potenz n ist dabei der *Grad des Polynoms*. Einen einzigen Summanden $a_i x^i$ bezeichnet man als *Monom*.

Wir untersuchen nun drei verschiedene Methoden, ein Polynom auszuwerten. Gegeben sei dazu ein Array a der Länge $n+1$, wobei $a[i]$ den Koeffizienten der Potenz x^i enthält. Die allgemeine Signatur der Funktion in Pseudo-Code ist also wie folgt:

Input: Koeffizienten-Array a der Länge $n+1$, Auswertungspunkt x
Output: Ergebnis $p = a[n] \cdot x^n + ... + a[1] \cdot x + a[0]$
EvaluatePolynomial(a,x)**:**

```
p = ...
return p;
```

a) In einer ersten Implementation gehen wir vor, wie dies auf dem Papier erfolgen würde. Wir beginnen also bei der höchsten Potenz, berechnen jeweils die Monome und addieren diese dann auf:

```
p = 0;
for i = n down to 0 {
    // Berechne i-te Potenz
    m = 1;
    for j = 1 to i {
        m = m · x;
    }

    // Berechne i-tes Monom durch Multiplikation der Potenz mit dem Koeffizienten
    m = a[i] · m;

    // Aktualisiere das Polynom durch Addition des Monoms
    p = p + m;
}
```

Bestimmen Sie die Laufzeitfunktion dieser Implementierung! In welcher Effizienzklasse in O-Notation ist dieser Code?

b) Offensichtlich ist in diesem Code die Berechnung der Potenzen redundant. Wir stellen den Code nun so um, dass die Potenzen inkrementell auf Basis vorheriger Iterationen errechnet werden:

```
p = 0;
h = 1;
for i = 0 to n {
    // Berechne i-tes Monom durch Multiplikation der (vorberechneten) Potenz mit
    // dem Koeffizienten
    m = a[i] · h;

    // Aktualisiere die Potenz für den nächsten Schritt i + 1
    h = h · x;

    // Aktualisiere das Polynom durch Addition des Monoms
```

$p = p + m;$
}

Bestimmen Sie auch die Laufzeitfunktion dieser neuen Implementierung, sowohl exakt wie auch in O-Notation!

c) Zuletzt untersuchen wir das sogenannte *Horner-Schema*. Die Idee ist, das Polynom umzuformen:

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x + a_0 = (((...) \cdot x + a_2) \cdot x + a_1) \cdot x + a_0$$

Dies kommt dadurch zustande, dass man immer wieder jeweils x aus den höheren Potenzen ausklammert. In Pseudo-Code lautet eine entsprechende Implementierung dann wie folgt:

$p = a[n];$
for $i = n - 1$ **down to** 0 {
// Multipliziere ausgeklammertes x an den bereits ausgewerteten Teil
$p = p \cdot x;$

// Addiere den nächsten Koeffizienten
$p = p + a[i]$
}

Bestimmen Sie zuletzt auch die Laufzeitfunktion dieser Implementierung, sowohl genau wie auch in O-Notation!

d) Basierend auf Ihren vorherigen Ergebnissen, welche Methode halten Sie für die beste?

Aufgabe 3 Sortieren mittels Divide-and-Conquer-Ansätzen

In der Vorlesung haben Sie mittlerweile zwei weitere Sortieralgorithmen kennengelernt, *Merge Sort* und *Quick Sort*, die beide nach dem *Divide-and-Conquer*-Prinzip funktionieren. Vergegenwärtigen Sie sich nochmals deren Ablauf!

Die Idee bei *Divide-and-Conquer* ist stets, das eigentliche Problem in kleinere Teilprobleme gleicher Art aufzuspalten, und diese dann separat zu lösen. Aus den Ergebnissen der Teile lässt sich dann einfach die Gesamtlösung erzeugen. Wichtig ist, dass dann auch die Teilprobleme entweder trivial oder ihrerseits aufspaltbar sind, so dass der *Divide-and-Conquer*-Ansatz rekursiv arbeitet.

Gegeben sei nun das Array

$$A = \{2,9,5,4,8,3,1,2\} .$$

a) Gehen Sie davon aus, dass A mittels *Merge Sort* sortiert wird. Zeichnen Sie die Zwischenschritte der Sortierung auf, so dass die Aufteilungen und Rekombinationen ersichtlich werden.

b) Zeichnen Sie nun alle Zwischenschritte auf, die sich bei der Sortierung mittels *Quick Sort* ergeben.

Aufgabe 4 Merge Sort

Gegeben sei das Array

$$A = \{d,b,f,g,e,a,c\} .$$

a) Welche Besonderheiten fallen Ihnen in Bezug auf A mit Hinblick auf *Merge Sort* auf?

b) Sortieren sie das Array A lexikographisch mittels *Merge Sort*.

Aufgabe 5 Quick Sort – Pivoting

Gegeben sei das Array

$$A = \{9,8,7,6,5,4,3,2,1\} .$$

a) Sortieren sie das Array A mit Quick Sort wobei Sie als Pivot-Element stets das erste Element aus dem Teilarray wählen.

b) Sortieren sie das Array A ein weiters Mal wobei Sie dieses Mal stets das mittlere Element (bzw. das Element mit Index $\frac{n}{2}+1$ bei gerader Eingabelänge) wählen.

c) Welche Beobachtungen machen Sie?