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

WiSe 2012/13
Übungsblatt 7
28. November 2012

Algorithmen und Datenstrukturen

Aufgabe 1 Rechnen mit Landau-Symbolen

Treffen die folgenden Behauptungen zu? Beweisen sie deren Richtigkeit, oder widerlegen Sie sie, entweder mittels Beweis oder durch Angabe eines Gegenbeispiels!

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.

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

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

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

Aufgabe 2 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 deren Ablauf nochmals!

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 3 Stabilität von Sortierverfahren

Bisher haben wir ja stets Zahlen sortiert, und dazu die Ordnung verwendet, die durch Standard-Vergleichsoperationen wie $<$ definiert wird. Natürlich kann man statt Zahlen auch andere Objekte sortieren, muss dazu aber gegebenenfalls eine andere Vergleichsoperation wählen.

Betrachten wir nun einen komplexen Datentyp, der eine Musikdatei abstrahiert und – vereinfachend – aus zwei Eigenschaften besteht:

- Titel des Songs als String
- Länge des Songs in Sekunden

In einer Playlist seien mehrere solcher Musiktitel eingereiht, und zwar alphabetisch sortiert nach Titeln. Nachdem dazu eine Abfrage $song1 < song2$ nicht direkt möglich ist, hätte man dazu dann eine eigene Vergleichsroutine benötigt, etwa **ist_vorher_im_alphabet**$(song1, song2)$.

Gehen wir nun davon aus, dass zwei Songs davon genau die gleiche Länge haben, und jetzt die alphabetisch sortierte Liste nach Songdauern umsortiert werden soll, etwa mittels des Vergleiches $song1.dauer < song2.dauer$. Nachher werden die beiden gleich langen Titel genau hintereinander liegen, vielleicht in alphabetischer Reihenfolge, vielleicht aber auch umgekehrt.

Ein Sortierverfahren, dass in einem solchen Fall gleicher Ordnung *immer* (also in allen möglichen Fällen, aufgrund des Entwurfes des Algorithmus) die vorherige, hier die alphabetische, Ordnung der betroffenen Elemente erhält, nennt man *stabil*.

Untersuchen Sie, ob die vier bisher besprochenen Verfahren[1] in den angegebenen Varianten stabil sind, und skizzieren Sie gegebenenfalls, ob und wie man Stabilität erreichen könnte!

Aufgabe 4 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*[2].

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;
```

[1] Insertion Sort, Selection Sort, Merge Sort und Quick Sort

[2] Von altgriechisch μόνος („mónos“), deutsch „einzeln“, im Unterschied zum aus mehreren Summanden bestehenden Polynom, von altgriechisch πολύς („polýs“), deutsch „viele“.

```
        // 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:

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

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 · 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?