Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 5

Abgabe: Montag, **27.05.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (Dynamische Programmierung [**10 Punkte**])

In rekursiven Programmen werden Werte oft unnötigerweise mehrfach berechnet, so z.B. bei der naiven Implementierung einer Funktion zur Berechnung der Fibonacci-Zahlen:

```
fibonacci(n)
    if (n < 2) return n;
    return fibonacci(n-1) + fibonacci(n-2)
```

Man mache sich das an einem Beispiel klar! Bei der dynamischen Programmierung geht man daher umgekehrt vor: Ausgehend vom Basisfall der Rekursion berechnet man weitere Werte und speichert diese für spätere Verwendung ab:

```
fib[0] = 0; // Basisfall der
fib[1] = 1; // Rekursion
for (i = 2; i <= n; ++i)
    fib[i] = fib[i-1] + fib[i-2]
// Ergebnis steht in fib[n]
```

Überzeugen sie sich, dass hier jede Fibonacci-Zahl nur einmal berechnet wird! In der folgenden Aufgabe soll diese Technik nochmals eingeübt werden.
Gegeben seien n verschiedene, positive Münzwerte $a_1, \ldots, a_n \in N^+$ und ein Betrag $m \in N_0$. Gesucht ist die Zahl der möglichen n-Tupel $(k_1, \ldots, k_n)$ mit

$$\sum_{l=1}^{n} k_l a_l = m$$

also die Zahl der Möglichkeiten, m durch die gegebenen Münzwerte darzustellen. Hierbei wollen wir optimistischerweise annehmen, dass von jeder Münzsorte beliebig viele zur Verfügung stehen. Außerdem vereinbaren wir folgenden Sonderfall: Der Betrag $m = 0$ lässt sich immer genau einmal durch 0 Münzen darstellen.
Wir verallgemeinern zunächst das Problem und fragen stattdessen nach der Zahl der Möglichkeiten einen Wert $0 \leq j \leq m$ durch nur $0 \leq i \leq n$ Münzwerte darzustellen. Diese Zahl nennen wir s_{ij}. Offensichtlich erhalten wir dann die Lösung zum ursprünglichen Problem in s_{nm}. Die Werte s_{ij} lassen sich bequem in einer Tabelle speichern deren Zeilen mit i und deren Spalten mit j nummeriert werden.

1. Welchen Wert hat s_{0j} für $0 \leq j \leq m$, d.h. welche Einträge stehen in der ersten Zeile der Tabelle? (Beachten Sie den Sonderfall für $j = 0$.) [1 Punkt]

2. Welchen Wert hat s_{i0} für $1 \leq i \leq n$, d.h. welche Einträge stehen in der ersten Spalte der Tabelle? [1 Punkt]

3. Es seien $n = 3, m = 10, a_1 = 2, a_2 = 5$ und $a_3 = 3$. Stellen Sie die zugehörige Tabelle auf und geben Sie die verschiedenen Möglichkeiten, den Betrag 10 darzustellen an. [2 Punkte]

4. Überlegen Sie sich eine Rekursions-Formel für den (i, j)-ten Eintrag der Tabelle. Was können Sie als Basis-Fall der Rekursion verwenden? [4 Punkte]

5. Skizzieren Sie einen Algorithmus der mittels der Rekursionsformel die Tabelle vollständig ausfüllt. Welche Zeit- und Platzkomplexität hat dieser Algorithmus in Abhängigkeit von n und m? [2 Punkte]

Lösungsvorschlag

1. Es ist
$$s_{0j} = \begin{cases} 1 & \text{falls} \quad j = 0 \quad \text{(der Sonderfall)} \\ 0 & \text{sonst} \end{cases}$$
denn ohne Münzen kann man auch keinen positiven Betrag darstellen!

2. Ohne Münzen läßt sich nur die 0 darstellen, also $s_{i0} = 1$.

3. Wir haben

	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	1	0	1	0	1	0	1
2	1	0	1	0	1	1	1	1	1	1	2
3	1	0	1	1	1	2	2	2	3	3	4

und es ist
$$\begin{aligned} 10 &= 2+2+2+2+2 \\ &= 2+3+5 \\ &= 2+2+3+3 \\ &= 5+5 \end{aligned}$$

4. Für $i > 0$ ist
$$s_{ij} = s_{i-1,j} + \begin{cases} s_{i,j-a_i} & \text{falls} \quad j - a_i \geq 0 \\ 0 & \text{sonst} \end{cases}$$
Basisfall ist $s_{00} = 1$ und $s_{0j} = 0, j > 0$, also die oberste Zeile der Tabelle.

5. Der Algorithmus berechnet zunächst die oberste Zeile der Tabelle. Dann werden die übrigen Einträge mit Hilfe der Rekursionsformel von links nach rechts und von oben nach unten ("Leserichtung") aufgefüllt. Offensichtlich ist Zeitkomplexität = Platzkomplexität = Größe der Tabelle = $O(nm)$.

Aufgabe 2 (Sortieralgorithmen [**10 Punkte**])

(a) In der Vorlesung wurde das *Insertion-Sort*-Verfahren vorgestellt (Foliensatz 2.3.2, Folie 11 ff.). In dem zu dieser Übungsaufgabe bereitgestellten Quellcode finden Sie die Java-Klasse `InsertionSort.java`, die eine leere Methode `sort()` beinhaltet. Beim Instanziieren der Klasse wird das zu sortierende Array übergeben und in das Member-Array `int[] A` kopiert. Implementieren Sie die Methode `sort()` so, dass das Array `A` mit Hilfe des Insertion-Sort Algorithmus sortiert wird. Überprüfen Sie Ihre Ergebnisse durch Ausführen der von uns in der Klasse `MainClass` bereitgestellten `main`-Methode. [2 Punkte]

(b) Insertion-Sort hat eine *Worst-Case*- und *Average-Case*-Komplexität von $O(n^2)$, wobei n die Eingabegröße ist. Wie könnte man die Laufzeit des Algorithmus verbessern? Begründen Sie ihre Antwort und analysieren Sie die *Best-*, *Worst-* und *Average-Case*-Laufzeit. Was fällt Ihnen bezüglich der *Best-Case*-Laufzeit im Vergleich zur normalen Version des Algorithmus auf? [3 Punkte]

(c) Ein Sortieralgorithmus ist "stabil", wenn er die Reihenfolge zweier Elemente mit demselben Sortierschlüsselwert beim Sortieren erhält. Erklären Sie, warum Quick-Sort diese Eigenschaft nicht besitzt und erweitern Sie den Algorithmus so, dass er stabil ist. Sie brauchen hierzu nichts implementieren, sondern lediglich die notwendigen Erweiterungen in Worten erklären und ggf. in Form von Pseudocode notieren. [5 Punkte]

Lösungsvorschlag

(a) Siehe Quellcode.

(b) Das Suchen der Stelle innerhalb der sortierten Liste am Anfang des Arrays, an der das jeweils nächste Element eingefügt werden soll, geschieht in linearem Zeitaufwand, also $O(n)$. Dies ist leicht nachzuvollziehen: Die sortierte Liste am Anfang des Arrays hat zu Beginn die Länge eins und wächst mit jedem Durchlauf der äußeren `while`-Schleife um ein Element. Dementsprechend hat sie in Durchlauf k die Größe k.

Im besten Falle (*Best-Case*) ist das zu sortierende Array bereits sortiert und das einzufügende Element muss lediglich mit einem, dem in der sortierten Liste am weitesten rechts stehenden Element verglichen werden und wird dort direkt einsortiert. Dies hat offensichtlich konstante Laufzeitkomplexität, sprich $O(1)$. Bei n Elementen kommen wir in dem Falle also auf eine Laufzeit von $O(n)$. Im schlimmsten Falle (*Worst-Case*) ist die Liste invers sortiert. Dann muss das einzusortierende Element in Durchlauf k mit allen k Elementen in der sortierten Liste verglichen werden. Bei n zu sortierenden Elementen ergibt sich folgende Komplexität:

$$\sum_{i=1}^{n} i = \frac{(n+1)\cdot n}{2} \leq c \cdot n^2 \Rightarrow \sum_{i=1}^{n} i \in O(n^2).$$

Das Suchen der Stelle in der sortierten Liste, an der ein Element einsortiert werden soll, könnte nun beschleunigt werden, indem man anstatt einer linearen Suche eine Binärsuche verwendet. Die Binärsuche funktioniert wie folgt:

```
function BinarySearch(l, r, h)          ▷ Suche zwischen Position l und r
    if r < l then
        return Error
    end if
    if l = r − 1 then
        if Wert[l] < h then
            return r                                  ▷ Position gefunden
        else
            return l                                  ▷ Position gefunden
        end if
    end if
    m ← l + ⌊(r−l)/2⌋
    if Wert[m] < h then
        return BinarySearch(m, r, h)
    end if
    return BinarySearch(l, m, h)                      ▷ Wert[m] ≥ h
end function
```

In dem Fall wird der Suchraum mit jedem rekursiven Aufruf halbiert, bis das gesuchte Element (die gesuchte Position) gefunden ist. Allerdings bestünde jetzt das Problem, dass beim Einfügen des einzusortierenden Elements an der gewünschten im Array alle nachfolgenden Elemente um eine Stelle nach hinten kopiert werden müssten. Um dies zu vermeiden, könnte man überlegen, eine (doppelt) verkettete Liste als zugrunde liegende Datenstruktur zu verwenden. Allerdings kann man in dem Fall nicht in konstantem Zeitaufwand an eine beliebige Stelle in der Folge springen, da man immer den Links zum jeweiligen nächsten/vorherigen Element folgend muss, um an eine bestimmte Stelle zu gelangen. Von daher benutzen wir eine *zusätzliche* Datenstruktur, die die guten Eigenschaften beider Ansätze verbindet: wir fügen die Elemente der Reihe nach in einen **binären Suchbaum** ein. Der Speicheraufwand verdoppelt sich nun offensichtlich und beträgt $2n \in O(n)$. Der Zeitaufwand für das Einfügen aller Elemente in den Suchbaum beträgt

$$\sum_{i=0}^{n-1} \log_2 i \in O(n \log n),$$

Da der Baum zu Beginn Größe 0 hat und mit jeder Einfüge-Operation um ein Element wächst. Jedes Element, das in einen Suchbaum eingefügt wird, wird als Blatt an den Baum gehängt. Aus diesem Grund gilt die o.g. Laufzeitabschätzung für das Einfügen von Elementen in jedem Fall, unabhängig von der Sortierung der Elemente in der ursprünglichen Folge. Vergleichen wir unsere Erweiterung mit dem *Best-Case* der Originalimplementierung, so sehen wir, dass die Originalmethode eine bessere Laufzeit hat, nämlich $O(n)$. In allen anderen Fällen hat die erweiterte Variante eine bessere Laufzeit.

(c) In jedem Partitioniervorgang wird die interne Reihenfolge zweier gleichgroßer Elemente invertiert, da die beiden Zeiger jeweils von außen nach innen laufen und Elemente vertauschen, wenn das des linken Zeigers größer und das des rechten Zeigers kleiner ist, als das Pivotelement. Das bedeutet, dass die Stabilität des Algorithmus maßgeblich von der Anzahl der Partitioniervorgänge abhängt. Darauf haben wir allerdings keinen Einfluss. Aus diesem Grund könnte man den Quick-Sort-Algorithmus wie folgt erweitern: Man führe einen zweiten Sortierschlüssel ein, der die Position eines jeden Elements in der initialen Folge enthält. Nach dem eigentlichen Sortiervorgang wird auf jedes Intervall in der sortierten Folge, das aus Elementen desselben Wertes besteht, ein weiterer Sortierdurchlauf angewendet. Für diesen Vorgang wählen wir nun den neu eingeführten Sortierschlüssel für die Vergleiche. Im Worst-Case besteht die zu sortierende Folge aus n Elementen desselben Wertes. In dem Fall wird der Quick-Sort-Algorithmus genau zweimal auf die gesamte Folge angewendet: Einmal zum Sortieren nach dem Primärschlüssel und ein weiteres Mal zum Sortieren nach dem neu eingeführten Sekundärschlüssel. Es ergibt sich eine Gesamtkomplexität von $2 \cdot O(n \log n) = O(n \log n)$. Im besten Fall gibt es in der Folge keine Duplikate und wir sind nach dem initialen Sortiervorgang bereits fertig.