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

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 6

Abgabe: Montag, **10.06.2013**, 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 (Summe von Array Werten [**10 Punkte**])

In dieser Aufgabe geht es darum, einen Algorithmus zu entwerfen, der in einem Byte-Array zwei Werte sucht, deren Summe einen bestimmten Betrag hat.
Sei A ein Byte-Array der Länge n, und sei m der gesuchte Betrag.
Entwerfen Sie einen Algorithmus, der linear viele Schritte benötigt, um zu entscheiden, ob in dem Array zwei Indizes i und j existieren, so dass $A[i] + A[j] = m$ gilt. Falls solche Indizes existieren, soll der Algorithmus **True** zurückgeben, ansonsten **False**.

(a) Beschreiben Sie Ihre Idee für den Algorithmus und geben Sie den Algorithmus in Pseudocode an.

(b) Beweisen Sie, dass Ihr Algorithmus die Laufzeitkomplexität $O(n)$ hat.

Lösungsvorschlag

(a)
```
function A1(A, m)
    M ← Boolean[256]  ▷ M markiert, welche Werte bisher gesehen wurden.
    for i ← 0, 255 do
        M[i] ← False
    end for
    for i ← 0, n − 1 do
        M[A[i]] ← True        ▷ Markieren, dass der Wert A[i] gesehen wurde.
        j ← m − A[i]
        if 0 ≤ j ≤ 255 ∧ M[j] then
            return True
        end if
    end for
    return False
end function
```

(b) Die erste Schleife im Algorithmus $A1$ hat eine konstante Zahl an Durchläufen. Die zweite Schleife hat n Durchläufe, die jeweils maximal eine konstante Zahl an Schritten benötigen. Somit benötigt der Algorithmus $O(n)$ viele Schritte zur Berechnung.

Aufgabe 2 (*Merge-Sort* [**10 Punkte**])

Der klassische Merge-Sort Algorithmus sortiert ein Array nach dem Divide-and-Conquer Prinzip, indem er das Array in zwei (optimalerweise) gleich große, kleinere Arrays aufteilt, diese rekursiv sortiert, und danach die beiden sortierten Teilarrays in linearer Zeit zu einem sortierten Array zusammenfügt.
Eine Variation dieses Algorithmus ist der *ternäre Merge-Sort*, welcher das Array nicht in zwei, sondern in drei kleinere Arrays aufteilt. In dieser Aufgabe soll der klassische binäre Merge-Sort hinsichtlich der Laufzeitkomplexität mit dem ternären Merge-Sort verglichen werden.

(a) Stellen Sie Rekursionsgleichungen für die Anzahl der Vergleiche zweier Array-Elemente des binären und des ternären Merge-Sort sowohl im *Worst-Case* als auch im *Best-Case* auf. Finden Sie explizite (nicht-rekursive) Darstellungen der Rekursionsgleichungen und vergleichen Sie. Welche Variante würden Sie vorziehen? [4 Punkte]

(b) In dem von uns bereitgestellten Code finden Sie zwei Klassen. Die Klasse `MergeSort.java` stellt alle notwendigen Methoden zur Verfügung, um ein Array `A` mit dem binären oder ternären Merge-Sort zu sortieren. Ihre Aufgabe ist es, die Rümpfe der Methoden `mergeSortBinary`, `mergeBinary`, `mergeSortTernary` und `mergeTernary` zu ergänzen.

Die Methode `mergeSortBinary(int l, int r)` sortiert das Teilarray `A[l..r]` rekursiv mit der binären Merge-Sort Variante. Sie verwendet die Methode `mergeBinary(int l, int m, in tr)`, um die beiden rekursiv sortierten Teilarrays `A[l..m-1]` und `A[m..r-1]` zu einem sortierten Gesamtarray zusammenzufügen.

Analog dazu sortiert die Methode `mergeSortTernary(int l, int r)` das Teilarray `A[l..r]` rekursiv mit der ternären Merge-Sort Variante. Sie verwendet die Methode `mergeTernary(int l, int m1, int m2, int r)`, um die drei rekursiv sortierten Teilarrays `A[l..m1-1]`, `A[m1..m2-1]` und `A[m2..r-1]` zu einem sortierten Gesamtarray zusammenzufügen.

Die Klasse `MainClass.java` erstellt ein Array mit 5 Millionen zufallsgenerierten Elementen, sortiert es einmal mit der binären Merge-Sort und einmal mit der ternären Merge-Sort-Variante. Anschließend gibt sie aus, wie lange die jeweilige Variante für die Sortierung des Arrays benötigt hat und ob das Array korrekt sortiert wurde. [4 Punkte]

(c) Messen Sie wiederholt die Zeiten ihrer Implementierung aus Teilaufgabe (b). Erklären Sie die gemessenen Zeitunterschiede und vergleichen Sie diese mit Ihren Resultaten aus Teilaufgabe (a). **Hinweis**: Stellen Sie Rekursionsgleichungen für die Anzahl der Aufrufe der jeweiligen `merge`-Methoden auf. [2 Punkte]

Lösungsvorschlag

(a) *Worst-Case binär:*

$$\begin{aligned}
T(1) &= 0 \\
T(n) &= 2T(n/2) + n - 1 \\
&= 2\left[2T(n/4) + n/2 - 1\right] + n - 1 \\
&= 4T(n/4) + 2n - 1 - 2 \\
&\quad\vdots \\
&= n\log_2 n - \sum_{i=0}^{\log_2 n - 1} 2^i \\
&= n\log_2 n - n + 1
\end{aligned}$$

Best-Case binär:

$$\begin{aligned}
T(1) &= 0 \\
T(n) &= 2T(n/2) + n/2 \\
&= 2\left[2T(n/4) + n/4\right] + n/2 \\
&= 4T(n/4) + n \\
&\quad\vdots \\
&= \frac{n}{2}\log_2 n
\end{aligned}$$

Worst-Case ternär:

$$\begin{aligned}
T(1) &= 0 \\
T(n) &= 3T(n/3) + 2(n-2) + 1 \\
&= 3\left[3T(n/9) + 2(n/3 - 2) + 1\right] + 2n - 4 + 1 \\
&= 9T(n/9) + (2n + 2n) - (4 + 12) + (1 + 3) \\
&\quad\vdots \\
&= 2n\log_3 n - 4\sum_{i=0}^{\log_3 n - 1} 3^i + \sum_{i=0}^{\log_3 n - 1} 3^i \\
&= 2n\log_3 n - 3\sum_{i=0}^{\log_3 n - 1} 3^i \\
&= 2n\log_3 n - 3\frac{n-1}{2} \\
&= 2n\log_3 n - 1.5n + 1.5
\end{aligned}$$

Best-Case ternär:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 3T(n/3) + 2n/3 + n/3 \\ &= 3\left[3T(n/9) + 2n/9 + n/9\right] + n \\ &= 9T(n/9) + n + n \\ &\vdots \\ &= n \log_3 n \end{aligned}$$

Im Best-Case sind bei der ternären Variante

$$\frac{n \log_3 n}{n/2 \log_2 n} = 2\frac{\ln n / \ln 3}{\ln n / \ln 2} = 2\frac{\ln 2}{\ln 3} \approx 1.26$$

mal soviele Vergleiche notwendig wie bei der binären Variante.

Im Worst-Case sind bei der ternären Variante für $n \to \infty$

$$\begin{aligned} &\lim_{n\to\infty} \frac{2n \log_3 n - 1.5n + 1.5}{n \log_2 n - n + 1} \\ \overset{\text{L'H.}}{=} &\lim_{n\to\infty} \frac{2 \log_3 n + 2/\ln 3 - 1.5}{\log_2 n + 1/\ln 2 - 1} \\ \overset{\text{L'H.}}{=} &\lim_{n\to\infty} 2\frac{\ln 2}{\ln 3} \approx 1.26 \end{aligned}$$

mal soviele Vergleiche notwendig wie bei der binären Variante.

Daher wäre aufgrund der geringeren Anzahl der Vergleiche die binäre Variante vorzuziehen.

(b) Siehe `MergeSort.java`.

(c) Nach 100 Wiederholungen benötigte der binäre Merge Sort durschnittlich 2.54 Sekunden, der ternäre Merge Sort benötigte durchschnittlich 2.07 Sekunden. Dies steht im Widerspruch zu den Resultaten aus Teilaufgabe (a), welche vermuten ließen, dass die binäre der ternären Variante überlegen ist.

Der Grund dafür ist, dass bei der ternären Variante deutlich weniger Aufrufe der `merge`-Methode stattfinden. Dies lässt sich auch durch Aufstellen von Rekursionsgleichungen für die Anzahl der Funktionsaufrufe verifizieren:

Funktionsaufrufe binär:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2T(n/2) + 1 \\ &= 2\left[2T(n/4) + 1\right] + 1 \\ &= 4T(n/4) + 2 + 1 \\ &\vdots \\ &= \sum_{i=0}^{\log_2 n - 1} 2^i \\ &= n - 1 \end{aligned}$$

Funktionsaufrufe ternär:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 3T(n/3) + 1 \\ &= 3\left[3T(n/9) + 1\right] + 1 \\ &= 9T(n/9) + 3 + 1 \\ &\vdots \\ &= \sum_{i=0}^{\log_3 n - 1} 3^i \\ &= \frac{n-1}{2} \end{aligned}$$

Die binäre Variante benötigt also doppelt so viele Aufrufe der `merge` Funktion wie die ternäre Variante. In jedem dieser Aufrufe wird Speicher für das Array `B` allokiert und zwei mal über den betrachteten Arraybereich iteriert (einmal beim zusammenfügen der beiden Teilarrays in das Array `B` und einmal beim Kopieren des Arrays `B` in das Array `A`). Einen noch detaillierteren Vergleich beider Varianten erhält man, wenn man die Gesamtanzahl der Arrayzugriffe ermittelt. Dies lässt sich erneut durch Aufstellen und Lösen von Rekursionsgleichungen (hier exemplarisch für den Best-case) bewerkstelligen:

Arrayzugriffe binär (Best-case):

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2T(n/2) + \frac{n}{2} \cdot 4 + \frac{n}{2} \cdot 2 + 2n \\ &= 2T(n/2) + 5n \\ &\vdots \\ &= 5n \cdot \log_2 n \end{aligned}$$

Arrayzugriffe ternär (Best-case):

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 3T(n/3) + \frac{n}{3} \cdot 6 + \frac{n}{3} \cdot 4 + \frac{n}{3} \cdot 2 + 2n \\ T(n) &= 3T(n/3) + 6n \\ &\vdots \\ &= 6n \cdot \log_3 n \end{aligned}$$

Damit sind bei der binären Variante $\frac{5 \ln 3}{6 \ln 2} \approx 1.32$ mal soviele Arrayzugriffe notwendig wie bei der ternären, wodurch die längere Laufzeit zu erklären ist.