Algorithmen und Datenstrukturen
Kapitel 3:
Divide & Conquer

Frank Heitmann
heitmann@informatik.uni-hamburg.de

28. Oktober 2015

Organisatorisches

1. Raumwechsel: Gr. 15 (Do 10-12, F-235) ab sofort in G-021.
2. Studie zum Arbeitsverhalten von Studierenden unter Leitung von Prof. Maria Knobelsdorf.

Divide & Conquer

Divide & Conquer ist eine Algorithmentechnik, die i.A. rekursiv arbeitet. Die Laufzeit solcher Algorithmen kann daher i.A. durch Rekurrenzgleichungen analysiert werden.

Die Technik kann wie folgt beschrieben werden: Größere Probleme werden in kleinere zerlegt, die dann seperat (und i.A. rekursiv) nach der gleichen Methode (!) gelöst werden. Die Teilergebnisse werden dann passend zur Lösung des Problems zusammengesetzt.

- Da auf kleineren Instanzen rekursiv gearbeitet wird, nutzt man Rekurrenzgleichungen zur Analyse. (Wie dort gibt es meist einen einfachen Basisfall.)
- Ein naher Verwandter ist die *dynamische Programmierung*, die wir später noch kennenlernen werden.

D & C - Laufzeitverbesserung

Wenn D & C Algorithmen möglich sind, dann ist die Problemstellung oft schon so, dass bereits der brute-force Algorithmus eine polynomielle Laufzeit hat. *Durch D & C wird diese Laufzeit dann weiter reduziert.*

Bringt das viel? Z.B. bei einem $O(n^2)$ und einem $O(n \cdot \log n)$ Algorithmus?

Sei $n = 1024 = 2^{10}$ dann braucht

- der $O(n^2)$ Algorithmus $c \cdot 2^{20} \approx c \cdot 1.000.000$ Schritte
- der $O(n \cdot \log n)$ aber nur $c \cdot 2^{10} \cdot 10 \approx c \cdot 10.000$ Schritte

Bei $n = 2^{20} \approx 1.000.000$ wird dies noch sehr viel deutlicher.

Ablauf

Der Plan für heute:

1. MergeSort als klassisches D & C Verfahren
2. Wiederholung und Vertiefung zum Mastertheorem
3. QuickSort als weiteres klassisches D & C Verfahren

Sortieren

Definition (Das Sortierproblem)

Eingabe: Eine Sequenz $< a_1, a_2, \ldots, a_n >$ von n Zahlen.
Gesucht: Eine Permutation $< a'_1, a'_2, \ldots, a'_n >$ der Eingabesequenz mit $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Anmerkung

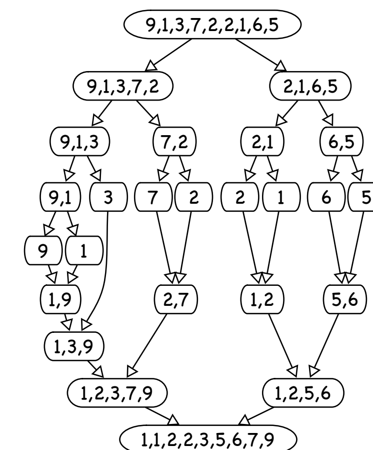
Ob die Reihenfolge zweier Elemente a_i, a_j $(i < j)$ mit $a_i = a_j$ beibehalten werden soll oder nicht, ist nicht gesagt. Bleibt sie erhalten, d.h. ist $a'_p = a_i$, $a'_q = a_j$ und $p < q$, so nennt man das Verfahren *stabil*.
Ferner heißt ein Sortierverfahren *in-place* (oder *in situ*), wenn der zusätzlich benötigte Speicherbedarf unabhängig von der gegebenen Sequenz ist.

Mergesort - Idee

Die bisherigen Verfahren (MaxSort, InsertionSort) sind in $O(n^2)$.
Das können wir besser...

Die Idee:

- Teile die Eingabe in zwei Teile gleicher (ca.) Größe.
- Löse die zwei Teilprobleme unabhängig und rekursiv.
- Kombiniere die zwei Ergebnisse in eine Gesamtlösung, wobei wir nur *lineare viel Zeit für die anfängliche Teilung und die abschließende Zusammenführung* nutzen wollen.



Mergesort - Algorithmus

Algorithmus 1 mergesort(A,l,r)

1: **if** $l < r$ **then**
2: $q = (l + r)/2$
3: $mergesort(A, l, q)$
4: $mergesort(A, q + 1, r)$
5: $merge(A, l, q, r)$
6: **end if**

Mergen

- Gegeben zwei sortierte Listen A und B.
- cur_A und cur_B seien Zeiger auf Elemente in A bzw. B. Am Anfang zeigen sie auf das erste Element der jeweiligen Liste.
- Solange beide Listen nicht leer sind:
  - Seien a_i und b_j die Elemente auf die cur_A und cur_B zeigen.
  - Füge das kleinere Element der Ausgabeliste hinzu.
  - Erhöhe den Zeiger der Liste dessen Element gewählt wurde.
- Ist eine Liste leer, füge die andere an die Ausgabeliste an.

MergeSort - Korrektheit

Es wäre nun noch

1. für Merge PseudoCode anzugeben
2. Merge als korrekt nachzuweisen (Schleifeninvariante!)
3. MergeSort als korrekt nachzuweisen (Induktion?)

$\Rightarrow$ Zur Übung

Analyse

Wir können für Mergesort nun die folgende Rekurrenzgleichung aufstellen:

$$T(n) = \begin{cases} d & \text{, falls } n = 1 \\ 2T(\frac{n}{2}) + cn & \text{, falls } n \geq 2 \end{cases}$$

Diese können wir durch Abwicklung oder das Mastertheorem lösen.

Das Mastertheorem

Definition (Mastertheorem)

Das Mastertheorem gilt für Rekurrenzen der Form:

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n) & \text{falls } n > 1 \end{cases}$$

Hierbei sind $a \geq 1, b > 1$ und c Konstanten, $\frac{n}{b}$ wird als $\lfloor \frac{n}{b} \rfloor$ oder $\lceil \frac{n}{b} \rceil$ verstanden und $T(n)$ über $\mathbb{N}$ definiert. Es gilt:

1. $T(n) \in \Theta(n^{\log_b(a)})$, falls $f(n) \in O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$.
2. $T(n) \in \Theta(n^{\log_b(a)} \log_2(n))$, falls $f(n) \in \Theta(n^{\log_b(a)})$.
3. $T(n) \in \Theta(f(n))$, falls $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ **und** $a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n)$ für ein $\delta < 1$ und große n.

Das Mastertheorem und Divide & Conquer

Hat man eine Gleichung

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n) & \text{falls } n > 1 \end{cases}$$

für einen Divide & Conquer Algorithmus, so gibt

- das a die Anzahl der Teilprobleme wieder,
- das $\frac{n}{b}$ die Größe der Teilprobleme und
- das $f(n)$ den Zeitbedarf beim Teilen des Problems und/oder beim Zusammenfügen der Teillösungen.

Mastertheorem - Mergesort (Beispiel Fall 2)

Fall 2

$T(n) \in \Theta(n^{\log_b(a)} \log_2(n))$, falls $f(n) \in \Theta(n^{\log_b(a)})$

$$T(n) = \begin{cases} d & \text{, falls } n = 1 \\ 2T(\frac{n}{2}) + cn & \text{, falls } n \geq 2 \end{cases}$$

Wegen $\log_b(a) = \log_2(2) = 1$ und $f(n) = cn$ ist $f(n) \in \Theta(n^1)$.

Es liegt also Fall 2 vor und dies ergibt $T(n) \in \Theta(n \cdot \log n)$.

Ergebnis

Mergesort sortiert eine Sequenz von n Zahlen in $\Theta(n \cdot \log n)$ Zeit.

Mastertheorem

Wir betrachten nachfolgend noch

- weitere Beispiele für die einzelnen Fälle des Mastertheorems und
- ein Hilfsmittel für die Benutzung des Mastertheorems

Mastertheorem - Beispiel Fall 1

Fall 1

$T(n) \in \Theta(n^{\log_b(a)})$, falls $f(n) \in O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$

$$T(n) = \begin{cases} c' \\ 3T(n/3) + 1/n \end{cases} \qquad T(n) = \begin{cases} c \\ a \cdot T(\frac{n}{b}) + f(n) \end{cases}$$

Wegen $\log_b(a) = \log_3(3) = 1$ und $f(n) = \frac{1}{n} = n^{-1}$ ist $f(n) \in O(n^{1-\epsilon})$ bei Wahl von z.B. $\epsilon = 2 > 0$.

Es liegt also Fall 1 vor und dies ergibt, wie letztes Mal durch Abwickeln, $T(n) \in \Theta(n)$.

Mastertheorem - Beispiel 3

Fall 3

$T(n) \in \Theta(f(n))$, falls $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ **und** $a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n)$ für ein $\delta < 1$ und große n

$$T(n) = \begin{cases} c & \text{, falls } n = 1 \\ 3T(\frac{n}{4}) + n \cdot \log n & \text{, falls } n \geq 2 \end{cases}$$

Hier ist $\log_b(a) = \log_4(3) \approx 0,793$ und $f(n) = n \cdot \log n$. Da $f(n) \in \Omega(n^{\log_4(3)+\epsilon})$ mit $\epsilon = 0,2$ gilt, kommt Fall 3 zur Anwendung, **wenn** die Regulartitätsbedingung ($af(n/b) \leq \delta f(n)$ für ein $\delta < 1$ und hinreichend große n) erfüllt ist, falls also

$$3(n/4)\log(n/4) = 3/4 n \log(n/4) \leq \delta n \log n$$

möglich ist. Für genügend große n gilt tatsächlich $3/4 n \log(n/4) \leq 3/4 n \log n = \delta f(n)$ mit $\delta = 3/4$, also $T(n) \in \Theta(n \cdot \log n)$.

Mastertheorem - Anmerkung

Bemerkung

Das Mastertheorem ist die **dritte Methode**, die wir zur Lösung von Rekurrenzgleichungen nutzen werden.
Das Mastertheorem ist ein mächtiges Hilfsmittel. Es ist aber nicht immer einsetzbar:

- Z.B. bei $T(n) = 2 \cdot T(n/2) + n \cdot \log n$ nicht.
- Hier ist $f(n) = n \cdot \log n$ und $log_b(a) = log_2(2) = 1$. Wir vergleichen also $n \cdot \log n$ mit n.
- Es sieht so aus, als ob der dritte Fall zutreffen könnte ($f(n) \in \Omega(n^{1+\epsilon})$)
- Aber man wird kein so ein $\epsilon > 0$ finden. $f(n)$ ist nämlich nicht *polynomiell* größer als n^1.

Mastertheorem - Anmerkung

Bemerkung

Der Fall von eben liegt also zwischen den Fällen 2 und 3 im Mastertheorem. Ebenso kann es Fälle geben, die mit ähnlicher Argumentation zwischen den Fällen 1 und 2 des Mastertheorems liegen. Dann kann es noch Fälle geben, die gegen die Regulartitätsbedingung von Fall 3 des Mastertheorems verstossen. Bei all diesen Fällen ist das Mastertheorem nicht anwendbar.

Hilfsmittel Variablentransformation

Bei Rekurrenzgleichungen sind oft **Variablentransformationen** hilfreich. Sei

$$T(n) = 2T(\sqrt{n}) + \log n$$

so ist das Mastertheorem zunächst nicht anwendbar. Die Substitution $m = \log n$ ($2^m = n$ und $2^{m/2} = n^{1/2} = \sqrt{n}$) führt zu

$$T(2^m) = 2T(2^{m/2}) + m$$

Sei nun $S(m) = T(2^m)$ so erhalten wir

$$S(m) = 2S(m/2) + m$$

Das sieht vertraut aus...

Hilfsmittel Variablentransformation

Bisher:

- $T(n) = 2T(\sqrt{n}) + \log n$
- $T(2^m) = 2T(2^{m/2}) + m$ (mit $m = \log n$, $2^m = n$)
- $S(m) = 2S(m/2) + m$ (mit $S(m) = T(2^m)$)

Mit $S(m) = \Theta(m \log m)$ folgt nun

$$T(n) = T(2^m) = S(m) = \Theta(m \log m) = \Theta(\log n \log\log n)$$

Und wir haben eine Abschätzung für $T(n)$.

Anmerkung

Wir haben hier $S(m) = \Theta(\dots)$ statt $S(m) \in \Theta(\dots)$ geschrieben, was nur von links nach recht gelesen wird und bei der O-Notation verbreitet ist.

Spezialfälle

Eine Verallgemeinerung der Rekurrenzgleichung von Mergesort ist

$$T(n) = \begin{cases} d & \text{, falls } n = 1 \\ qT(\frac{n}{2}) + cn & \text{, falls } n \geq 2 \end{cases}$$

- Ist $q > 2$, so erhalten wir $T(n) \in \Theta(n^{\log_2 q})$. Die rekursiven Aufrufe erzeugen mehr Arbeit bei wachsendem q.
- Ist $q = 1$, so erhalten wir $T(n) \in \Theta(n)$. Meiste Arbeit auf oberster Ebene.

Zusammenfassung

Die drei Schritte des Divide & Conquer (Teile-und-Beherrsche) Paradigmas

1. **Teile** das Problem in n Teilprobleme auf
2. **Beherrsche** die Teilprobleme durch rekursives Lösen (d.h. teile sie weiter; sind die Teilprobleme hinreichend klein, löse sie direkt)
3. **Verbinde** die Lösungen der Teilprobleme zur Lösung des übergeordneten Problems und letztendlich des ursprünglichen Problems.

Die Schritte finden auf jeder Rekursionsebene statt.

Anmerkungen

Rekursive Verfahren können auch iterativ implementiert werden. Das ist manchmal sinnvoll, manchmal nicht. (Siehe Anmerkungen im letzten Foliensatz.)

QuickSort

QuickSort ist ein *Divide-&-Conquer*-Algorithmus, der

- im schlechtesten Fall in $\Theta(n^2)$ arbeitet,
- dessen *erwartete Laufzeit* aber in $\Theta(n \cdot \log n)$ liegt (mit nur einer kleinen versteckten Konstanten!),

d.h. in der Praxis *ist er oft die beste Wahl für das Sortieren*.

QuickSort - Die Idee

- Wähle ein Elemente x aus A aus.
- Teile: Das Feld $A[p..r]$ wird in zwei Teilfelder $A[p..q-1]$ und $A[q+1..r]$ zerlegt, wobei alle Elemente im ersten Teilfeld kleiner oder gleich $x = A[q]$ sind und alle Elemente im zweiten Teilfeld größer oder gleich $A[q]$ sind. Der Index q (für x) wird in der Zerlegungsprozedur berechnet.
- Beherrsche: Sortiere die beiden Teilfelder durch rekursive Aufrufe.
- Verbinde: Da die Teilfelder (rekursiv) in-place sortiert werden, brauchen sie nicht weiter verbunden werden. Das Feld $A[p..r]$ ist nun sortiert.

QuickSort - Psuedocode (1/2)

Algorithmus 2 QuickSort(A, p, r)

```
1: if p < r then
2:     q = Partition(A, p, r)
3:     QuickSort(A, p, q − 1)
4:     QuickSort(A, q + 1, r)
5: end if
```

Um ein Array A zu sortieren rufen wir

$$\texttt{QuickSort}(A, 1, \text{länge}[A])$$

auf.

QuickSort - Pseudocode (2/2)

Algorithmus 3 Partition(A, p, r)

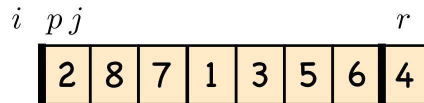
```
1: x = A[r]     // dies ist das Pivotelement
2: i = p − 1
3: for j = p to r − 1 do
4:     if A[j] ≤ x then
5:         i = i + 1
6:         swap(A[i], A[j])
7:     end if
8: end for
9: swap(A[i + 1], A[r])
10: return i + 1
```

Nachfolgend ohne Zeile 1, 2 und 10! (Wegen Platzmangel...)

QuickSort - Beispiel

Algorithmus 4 Partition(A, p, r)

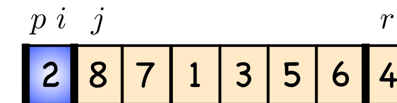
1: **for** $j = p$ to $r - 1$ **do**
2: **if** $A[j] \leq x$ **then**
3: $i = i + 1$
4: swap$(A[i], A[j])$
5: **end if**
6: **end for**
7: swap$(A[i + 1], A[r])$



QuickSort - Beispiel

Algorithmus 5 Partition(A, p, r)

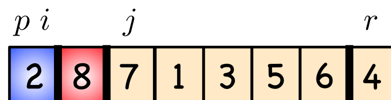
1: **for** $j = p$ to $r - 1$ **do**
2: **if** $A[j] \leq x$ **then**
3: $i = i + 1$
4: swap$(A[i], A[j])$
5: **end if**
6: **end for**
7: swap$(A[i + 1], A[r])$



QuickSort - Beispiel

Algorithmus 6 Partition(A, p, r)

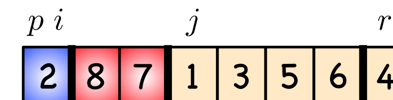
1: **for** $j = p$ to $r - 1$ **do**
2: **if** $A[j] \leq x$ **then**
3: $i = i + 1$
4: swap$(A[i], A[j])$
5: **end if**
6: **end for**
7: swap$(A[i + 1], A[r])$



QuickSort - Beispiel

Algorithmus 7 Partition(A, p, r)

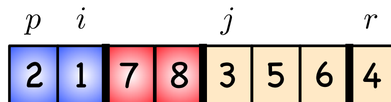
1: **for** $j = p$ to $r - 1$ **do**
2: **if** $A[j] \leq x$ **then**
3: $i = i + 1$
4: swap$(A[i], A[j])$
5: **end if**
6: **end for**
7: swap$(A[i + 1], A[r])$



QuickSort - Beispiel

Algorithmus 8 Partition(A, p, r)

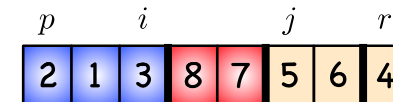
1: **for** $j = p$ to $r - 1$ **do**
2: **if** $A[j] \leq x$ **then**
3: $i = i + 1$
4: swap($A[i], A[j]$)
5: **end if**
6: **end for**
7: swap($A[i + 1], A[r]$)



QuickSort - Beispiel

Algorithmus 9 Partition(A, p, r)

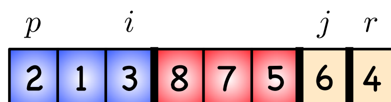
1: **for** $j = p$ to $r - 1$ **do**
2: **if** $A[j] \leq x$ **then**
3: $i = i + 1$
4: swap($A[i], A[j]$)
5: **end if**
6: **end for**
7: swap($A[i + 1], A[r]$)



QuickSort - Beispiel

Algorithmus 10 Partition(A, p, r)

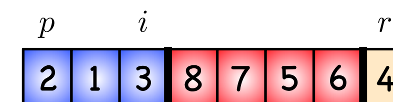
1: **for** $j = p$ to $r - 1$ **do**
2: **if** $A[j] \leq x$ **then**
3: $i = i + 1$
4: swap($A[i], A[j]$)
5: **end if**
6: **end for**
7: swap($A[i + 1], A[r]$)



QuickSort - Beispiel

Algorithmus 11 Partition(A, p, r)

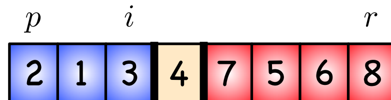
1: **for** $j = p$ to $r - 1$ **do**
2: **if** $A[j] \leq x$ **then**
3: $i = i + 1$
4: swap($A[i], A[j]$)
5: **end if**
6: **end for**
7: swap($A[i + 1], A[r]$)



QuickSort - Beispiel

Algorithmus 12 Partition(A, p, r)

1: **for** $j = p$ to $r - 1$ **do**
2: **if** $A[j] \leq x$ **then**
3: $i = i + 1$
4: swap($A[i], A[j]$)
5: **end if**
6: **end for**
7: swap($A[i+1], A[r]$)



Eine Schleifeninvariante

Das Feld ist während der Ausführung (von Partition) in vier Teile zerlegt. Diese lassen sich in einer Schleifeninvarianten formulieren:

Für jeden Feldindex k gilt (zu Beginn jeder Iteration der for-Schleife):

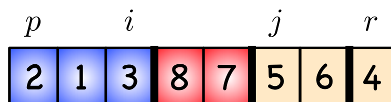
1. *Ist $p \leq k \leq i$, so gilt $A[k] \leq x$.*
2. *Ist $i + 1 \leq k \leq j - 1$, so gilt $A[k] > x$.*
3. *Ist $k = r$, so gilt $A[k] = x$.*

(Der vierte Teil sind die Indizes zwischen j und $r - 1$, deren Werte noch kein definiertes Verhältnis zu x haben.)

Zur Illustration der Schleifeninvarianten

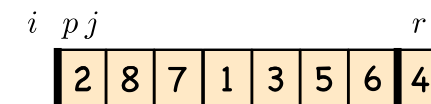
Für jeden Feldindex k gilt (zu Beginn jeder Iteration der for-Schleife):

1. *Ist $p \leq k \leq i$, so gilt $A[k] \leq x$.*
2. *Ist $i + 1 \leq k \leq j - 1$, so gilt $A[k] > x$.*
3. *Ist $k = r$, so gilt $A[k] = x$.*



Schleifeninvariante - Initialisierung

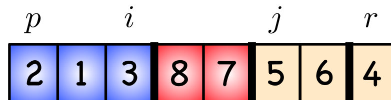
1. Ist $p \leq k \leq i$, so gilt $A[k] \leq x$.
2. Ist $i + 1 \leq k \leq j - 1$, so gilt $A[k] > x$.
3. Ist $k = r$, so gilt $A[k] = x$.



Initialisierung. $i = p - 1$ und $j = p$. 1 und 2 gelten sofort. 3 ist auch erfüllt. (Wegen Zeile 1 im Sourcecode.)

Schleifeninvariante - Fortsetzung 1

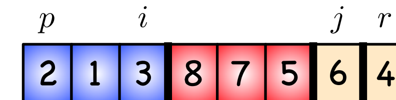
1. Ist $p \leq k \leq i$, so gilt $A[k] \leq x$.
2. Ist $i + 1 \leq k \leq j - 1$, so gilt $A[k] > x$.
3. Ist $k = r$, so gilt $A[k] = x$.



Fortsetzung 1. Ist $A[j] > x$, so wird in der Schleife nur j inkrementiert, danach gilt Bedingung 2 für $A[j-1]$ und alles andere bleibt wie vorher weiter erfüllt.

Schleifeninvariante - Fortsetzung 1

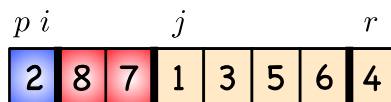
1. Ist $p \leq k \leq i$, so gilt $A[k] \leq x$.
2. Ist $i + 1 \leq k \leq j - 1$, so gilt $A[k] > x$.
3. Ist $k = r$, so gilt $A[k] = x$.



Fortsetzung 1. Ist $A[j] > x$, so wird in der Schleife nur j inkrementiert, danach gilt Bedingung 2 für $A[j-1]$ und alles andere bleibt wie vorher weiter erfüllt.

Schleifeninvariante - Fortsetzung 2

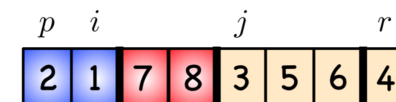
1. Ist $p \leq k \leq i$, so gilt $A[k] \leq x$.
2. Ist $i + 1 \leq k \leq j - 1$, so gilt $A[k] > x$.
3. Ist $k = r$, so gilt $A[k] = x$.



Fortsetzung 2. Ist $A[j] \leq x$, so wird i inkrementiert, $A[i]$ und $A[j]$ vertauscht und dann j inkrementiert. Wegen des Tauschens ist nun (i und j sind inkrementiert!) $A[i] \leq x$ (Bedingung 1 erfüllt) und $A[j-1] > x$ wegen der Schleifeninvarianten (Bedingung 2 erfüllt).

Schleifeninvariante - Fortsetzung 2

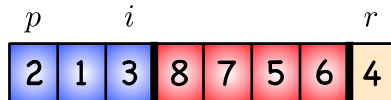
1. Ist $p \leq k \leq i$, so gilt $A[k] \leq x$.
2. Ist $i + 1 \leq k \leq j - 1$, so gilt $A[k] > x$.
3. Ist $k = r$, so gilt $A[k] = x$.



Fortsetzung 2. Ist $A[j] \leq x$, so wird i inkrementiert, $A[i]$ und $A[j]$ vertauscht und dann j inkrementiert. Wegen des Tauschens ist nun (i und j sind inkrementiert!) $A[i] \leq x$ (Bedingung 1 erfüllt) und $A[j-1] > x$ wegen der Schleifeninvarianten (Bedingung 2 erfüllt).

Schleifeninvariante - Terminierung

1. Ist $p \leq k \leq i$, so gilt $A[k] \leq x$.
2. Ist $i + 1 \leq k \leq j - 1$, so gilt $A[k] > x$.
3. Ist $k = r$, so gilt $A[k] = x$.



Terminierung. Jetzt ist $j = r$, also gehört jedes Element des Arrays zu einer der drei Mengen.

Korrektheit von Quicksort

Als letztes wird noch das Pivotelement $A[r]$ an seinen richtigen Platz im Array gesetzt. Partition ist damit korrekt und daraus folgt dann auch die Korrektheit von Quicksort.

Die Laufzeitanalyse ist komplizierter...

... und wir werden das Thema hier nur ankratzen.

Worst-Case-Laufzeit

Die Laufzeit von QuickSort hängt stark davon ab, ob die Zerlegung balanciert (die Teilarrays haben ungefähr die gleiche Größe) oder unbalanciert (ein Teilarray sehr viel größer als das andere) ist.

$\Rightarrow$ Balanciertheit führt zu $O(n \cdot \log n)$.

$\Rightarrow$ Unbalanciertheit führt zu $O(n^2)$.

Unbalanciert

Im schlechtesten Fall erzeugt die Zerlegungsroutine ein Teilproblem mit $n - 1$ Elementen. Dies führt zu einer Rekurrenzgleichung der Art

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

Was zu $T(n) \in \Theta(n^2)$ führt.

Dieser Fall tritt tatsächlich auf (z.B. auch bei einem sortierten Array!), so dass QuickSort im worst-case tatsächlich eine Laufzeit von $\Theta(n^2)$ hat.

Balanciert

Teilt Partition hingegen gleichmässig, so erhalten wir eine Rekurrenzgleichung der Art

$$T(n) = 2T(n/2) + cn$$

Was (Mastertheorem!) zu $T(n) \in \Theta(n \cdot \log n)$ führt. (Die worst-case-Laufzeit von QuickSort liegt aber dennoch in $\Theta(n^2)$!)

Abschließende Bemerkungen

Anmerkung 1

Tatsächlich tritt der Fall einer balancierten Zerlegung meist auf, was unter anderem daran liegt, dass 'ein bißchen balanciert zu sein' genügt. Die mittlerer Laufzeit (die 'normalerweise auftretende Laufzeit') von QuickSort ist also viel näher an der des besten Falls als an der des schlechtesten Falls, daher ist QuickSort oft die beste Wahl für das Sortieren und in der Praxis häufig anzutreffen. (Insb. wegen des nur kleinen konstanten Faktors.)
Um dies näher auszuführen braucht man Hilfsmittel aus der Stochastik, daher verzichten wir hier darauf. Näheres findet man in Kapitel 7.4.2 im [Cormen].

Anmerkung 2

In der Literatur findet man viele QuickSort-Varianten, die sich insb. in der Wahl des Pivotelementes unterscheiden.

QuickSort - Zusammenfassung

- QuickSort sortiert ein Array von Zahlen, indem rekursiv das Array in zwei kleinere Arrays geteilt wird, wobei in dem einen Teilarray die kleineren in dem anderen die größeren Elemente bzgl. eines gewählten Pivotelementes sind.
- QuickSort sortiert inplace.
- Die worst-case-Laufzeit von QuickSort ist in $\Theta(n^2)$.
- Die mittlere Laufzeit von QuickSort ist in $\Theta(n \cdot \log n)$.
- Die bessere Grenze von $O(n \cdot \log n)$ tritt sehr viel häufiger auf als die schlechtere. QuickSort ist daher wegen seiner kleinen versteckten Konstanten (trotz der schlechten worst-case-Laufzeit) ein sehr gutes Sortierverfahren.

Zusammenfassung und Wiederholung

Wir haben heute mit *Divide & Conquer* eine neue Algorithmentechnik (ein Muster, ein Paradigma) kennengelernt.

$\Rightarrow$ Größere Probleme werden in kleinere zerlegt, die dann seperat (und i.A. rekursiv) nach der gleichen Methode (!) gelöst werden. Die Teilergebnisse werden dann passend zur Lösung des Problems zusammengesetzt.

Divide & Conquer

Die drei Schritte des Divide & Conquer (Teile-und-Beherrsche) Paradigmas

1. **Teile** das Problem in n Teilprobleme auf
2. **Beherrsche** die Teilprobleme durch rekursives Lösen (d.h. teile sie weiter; sind die Teilprobleme hinreichend klein, löse sie direkt)
3. **Verbinde** die Lösungen der Teilprobleme zur Lösung des übergeordneten Problems und letztendlich des ursprünglichen Problems.

Die Schritte finden auf jeder Rekursionsebene statt.

Anmerkungen

Rekursive Verfahren können auch iterativ implementiert werden. Das ist manchmal sinnvoll, manchmal nicht. (Siehe Anmerkungen im letzten Foliensatz.)

Das Mastertheorem und Divide & Conquer

Hat man eine Gleichung

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n) & \text{falls } n > 1 \end{cases}$$

für einen Divide & Conquer Algorithmus, so gibt

- das a die Anzahl der Teilprobleme wieder,
- das $\frac{n}{b}$ die Größe der Teilprobleme und
- das $f(n)$ den Zeitbedarf beim Teilen des Problems und/oder beim Zusammenfügen der Teillösungen.

Sortierverfahren

Wir haben dann

- MergeSort und
- QuickSort

kennengelernt, zwei klassische Sortierverfahren.

Idee der Verfahren? Ungefährer Ablauf? Idee beim Korrektheitsbeweis? Idee bei der Laufzeitanalyse? Laufzeit?

Herausforderung für Interessierte

Wie kann ein inplace MergeSort realisiert werden?

Zum Algorithmenentwurf

Will man ein D & C Algorithmus entwerfen:

1. Verstehe das Problem (evtl. hilft es einen Brute-Force-Algorithmus zu entwickeln).
2. Wie kann das Problem aufgeteilt werden?
3. Wie können die Einzelteile zusammengesetzt werden?

$\Rightarrow$ Je nach Anzahl der Aufteilung, Größe der dabei enstehenden Instanzen und benötigte Zeit für die Aufteilung und Zusammenführung entstehen andere Rekurrenzgleichungen und damit oft auch andere Laufzeiten. Führt also eine Aufteilung nicht zum Ziel, kann man versuchen eine andere zu finden, was dann oft verschiedene Algorithmen zum Kombinieren nach sich zieht.

Ausblick

(Über-)Nächstes Mal:

- Klassische Datenstrukturen (Queue, Stack, Baum, Heap, ...)
- Können diese für schnelleres Sortieren benutzt werden?
- HeapSort
- Können wir schneller sortieren als bei MergeSort?