Divide & Conquer

Divide and Conquer = Divide et impera = Teile und herrsche

Dieses Konzept kann man als Spezialfall des Rekursions-Schemas betrachten. Es läuft üblicherweise so ab:

(Die zu bearbeitenden Daten seien in einem Array $a_1, \ldots, a_n$ gegeben)

---

1) Teile $a_1, \ldots, a_n$ auf in $b_1, \ldots, b_m$ und $c_1, \ldots, c_{n-m}$
2) Führe rekursiv die geforderte Berechnung sukzessive auf beiden Teil-Arrays aus. Die Ergebnisse seien LSG_b und LSG_c.
3) Füge die Lösungen LSG_b und LSG_c zur Lösung LSG_a des ursprünglichen Problems zusammen.

(Beachte: Die Schritte 1) und 3) sollten möglichst in Linearzeit durchgeführt werden.)

Divide & Conquer (Forts.)

Für einen „guten“ Divide & Conquer Algorithmus ist es im Normalfall sehr wichtig, dass die beiden Teile, auf denen die rekursiven Aufrufe ausgeführt werden, etwa gleich groß sind, d.h. wir streben an, dass $m \approx \frac{n}{2}$ gilt.

Darüberhinaus ist es wichtig, dass die Schritte 1) und 3), die die rekursiven Aufrufe quasi „einrahmen“, nicht zu viel Zeit verbrauchen. Hier fordert man zweckmäßigerweise, dass diese Schritte jeweils mit linearer Laufzeit auskommen, d.h. die Gesamtlaufzeit eines Aufrufs soll (ohne die Rekursion aus Schritt 2) in $O(n)$ bzw. $\Theta(n)$ liegen.

Wir werden das am Beispiel von Sortierverfahren durchspielen.

Beispiel: Mergesort

Als erstes Beispiel betrachten wir ein Sortierverfahren, das man *Sortieren durch Mischen*, englisch *mergesort* nennt.

Eingabe ist ein Array $a_1, \ldots, a_n$ OBdA sei n gerade

AUFGABE: Sortiere $a_1, \ldots, a_n$ nach vorgegebenem Schlüssel!

```
mergesort(a_1, ..., a_n):
    IF n ≥ 2 THEN
        b_1, ..., b_{n/2} := mergesort(a_1, ..., a_{n/2});
        c_1, ..., c_{n/2} := mergesort(a_{n/2+1}, ..., a_n);
        a_1, ..., a_n := mische b_1, ..., b_{n/2} mit c_1, ..., c_{n/2};
    END
```

Bemerkungen

1) Der Aufteilungsschritt ist trivial – man muss nur den Index n halbieren und dann die Hälften kopieren, d.h. $O(n)$ reicht aus!
2) Die Aufteilung ist optimal, da $m = n/2$ gilt.
3) Das Mischen im letzten Schritt ist relativ einfach in Linearzeit realisierbar! (Details bitte selbst überlegen)

Was bedeutet das für die Gesamtlaufzeit?

Die sogenannte *Aufruftiefe* (auch *Rekursionstiefe*) ist $\log n$.

Je „Aufrufebene“ ist der Aufwand in $O(n)$.

Das ergibt eine Gesamtrechenzeit in $O(n \log n)$.

Nachteil von mergesort: Zusätzlicher Speicherbedarf!

mergesort ist kein „in situ“-Verfahren (in situ = in place)

Beispiel: Quicksort

Als zweites Beispiel betrachten wir ein weiteres Sortierverfahren, das unter dem Namen *quicksort* bekannt ist.

Eingabe ist wieder ein Array $a_1, \ldots, a_n$

AUFGABE: Sortiere $a_1, \ldots, a_n$ nach vorgegebenem Schlüssel!

```
quicksort(a_1, ..., a_n):
    IF n ≥ 2 THEN
        teile a_1, ..., a_n in b_1, ..., b_r und c_1, ..., c_s so, dass
                        r + s = n, b_i ≤ a_1 und c_j > a_1 für alle i, j.
        quicksort(b_1, ..., b_r);
        quicksort(c_1, ..., c_s);
        Lösung:  b_1, ..., b_r, c_1, ..., c_s
    END
```

Das Aufteilen geschieht mit Zeigern, die (gleichzeitig) von links nach einem Element $> a_1$ und von rechts nach einem Element $\leq a_1$ suchen – dann werden diese beiden Elemente vertauscht und es geht so weiter...

Laufzeit: im worst-case $\Theta(n^2)$, im average-case $O(n \log n)$.

Zur Laufzeit bei Divide & Conquer

Wir schauen uns zwei Extremfälle bei Divide & Conquer Algorithmen an:

1. Fall: In jedem Durchgang wird die Eingabe in zwei gleich große Teile geteilt, d.h. es gilt im ersten Schritt $m = n/2$, und so weiter.

Wir nehmen an, dass Schritte 1) und 3) jeweils in Linearzeit ausführbar sind. Dann ist die Rekursionstiefe entscheidend für die Rechenzeit.
Diese sei k, d.h. Abbruch erfolgt bei Eingabegröße $n/2^k$.
Daraus ergibt sich $k = \log n$ (mit Logarithmus zur Basis 2).
Die Gesamtrechenzeit ist daher in $O(n \log n)$.

2. Fall: In jedem Durchgang sei $m = 1$ (oder $m = n - 1$), d.h. die Aufteilung ergibt einen sehr kleinen und einen sehr großen Teil.

Man erhält eine Rekursionstiefe von n, die Laufzeit in jeder Aufrufebene ist im Mittel $n/2$, als Gesamtlaufzeit erhält man hier also $\Theta(n^2)$.

Weitere Laufzeiten

Angenommen, die Aufteilung ist in jedem Schritt so, dass ein Teil die Größe 3, der andere $N - 3$ hat – Anfangsgröße sei n.
Was ist die Rekursionstiefe? Antwort: $n/3$
Was ist die Gesamtlaufzeit? Antwort: $\Theta(n^2)$

Wie sieht es bei 100000 und $N - 100000$ aus?
Rekursionstiefe $n/100000$, Rechenzeit $\Theta(n^2)$

Und wie bei $0,0001N$ und $0,9999N$?

Allgemein gilt, dass bei einer Aufteilung in $\alpha \cdot N$ und $(1-\alpha)N$ die Rekursionstiefe logarithmisch und die Laufzeit in $O(n \log n)$ ist.

Wie sieht man das?

Die Zahl α erfülle $0.5 < \alpha < 1$, dann gilt $\lim_{n\to\infty} \alpha^n = 0$.
Also gibt es eine Zahl n_0 mit $\alpha^{n_0} < 0.5$, d.h. nach n_0 vielen Schritten ist die Eingabegröße halbiert, folglich Rekursionstiefe beschränkt durch $n_0 \cdot \log n \in O(\log n)$.

Multiplikation

Als letztes Beispiel betrachten wir die Multiplikation großer Zahlen.

INPUT: Zwei n-Bit Zahlen X und Y, dabei n gerade, etwa $n = 2k$.

OUTPUT: Das Produkt $X \cdot Y$ als $2n$-Bit Zahl.

Sei $X = \sum_{i=0}^{n-1} a_i \cdot 2^i$ und $Y = \sum_{i=0}^{n-1} b_i \cdot 2^i$.

Man kann X und Y jeweils in zweimal k Bit zerlegen, wobei wir die „obere Hälfte“ von X mit A, den Rest mit B bezeichnen, bei Y entsprechend C und D.
Es gilt also $X = A \cdot 2^k + B$ und $Y = C \cdot 2^k + D$ und damit

$$X \cdot Y = A \cdot C \cdot 2^{2k} + (AD + BC) \cdot 2^k + BD$$

Wir müssen also 4 Multiplikationen und einige Additionen durchführen.
Additionen können in Linearzeit ausgeführt werden, damit erhalten wir eine Rekursion mit Aufwand $T(n) = 4 \cdot T(n/2) + O(n)$. Das ergibt eine Laufzeit in $\Theta(n^2)$, also haben wir so keinen Gewinn gegenüber der Schulmethode!

ABER: Mit einem einfachen Trick genügen drei Multiplikationen.
(bitte selbst versuchen, wie das geht...)

Damit ist die Laufzeit in $o(n^{1,6})$.

Dynamisches Programmieren

Grundprinzip des Entwurfsprinzips *dynamisches Programmieren* ist das folgende:

Für das Gesamtproblem wird eine Lösung aus vorher ermittelten Teil-Lösungen generiert. Dazu sollen möglichst viele Teil-Lösungen in einer Tabelle organisiert aufbewahrt werden.

Diese Technik ist immer dann anwendbar, wenn das sogenannte Bellman'sche Optimalitätsprinzip erfüllt ist.

Aber Achtung: Dieses Prinzip gilt durchaus nicht immer!

Ein bekanntes und sehr einfaches Beispiel für eine Berechnung mit „dynamischem Programmieren“ ist die Ermittlung der Binomialkoeffizienten $\binom{n}{k}$ mit dem Pascal'schen Dreieck!

Der CYK-Algorithmus

Aus der Vorlesung *Theoretische Informatik I* (bzw. Formale Sprachen und Automatentheorie) kennen wir den CYK-Algorithmus zur effizienten Lösung des Wortproblems für kontextfreie Sprachen.

Benannt nach seinen Erfindern *Cocke-Younger-Kasami*.

Die kontextfreie Sprache L muss in Form einer Typ-2 Grammatik in Chomsky-Normalform (CNF) vorliegen, die Eingabe sei ein Wort w.

Man ermittelt sukzessive für alle Längen von 1 bis $|w|$ und für alle Teilwörter von w der gegebenen Länge, welche Variablen das entsprechende Teilwort erzeugen können. Durch die CNF ist es möglich, mit Hilfe der Ergebnisse für die kürzeren Längen diese Variablen effizient zu ermitteln.

Bitte informieren Sie sich, wenn Sie diesen Algorithmus nicht oder nicht mehr kennen...

Optimale Klammerung

Das Problem der optimalen Klammerung tritt auf, wenn man das Produkt von k Matrizen $M_1 \cdot \ldots \cdot M_k$ berechnen will, wobei M_i eine $(n_{i-1} \times n_i)$-Matrix ist $(i = 1, \ldots, k)$.

Wir sollen die optimale Ausführungsreihenfolge (= Klammerung) ermitteln, um möglichst wenige sogenannte *Skalarmultiplikationen* zu benötigen.

Wieso haben wir hier überhaupt eine Freiheit? Weil das Assoziativgesetz gilt!

Wieviele Skalarmultiplikation benötigt man für eine einzelne Matrixmultiplikation?

M sei $(\ell \times m)$-, N sei $(m \times n)$-Matrix, dann $\ell \cdot m \cdot n$.

Wir betrachten $M_1 \cdot M_2 \cdot M_3$, also $k = 3$.
Die Dimensionen seien (10×2), (2×8),(8×3).

Für $(M_1 \cdot M_2) \cdot M_3$ werden 400 skalare Multiplikationen benötigt.
Für $M_1 \cdot (M_2 \cdot M_3)$ werden 108 skalare Multiplikationen benötigt.

Plan für einen Algorithmus

Die grobe Struktur ist, wie bei fast allen Algorithmen dieses Typs, eine Tabelle mit k Zeilen und k Spalten. Allerdings wird jedes „Indexpaar“ $\{i, j\}$ nur einmal gebraucht, d.h. es ergibt sich eine Dreiecksstruktur:

		1	2	3	4	.	.	j	k
	1	0							
	2		0						
	3			0					
	4				0				
i	.					.		$T_{i,j}$	
	.						.		
	.							.	
	k								0

$T_{i,j}$ soll für $i < j$ die Kosten für die Teilmultiplikation

$$M_i \cdot \ldots \cdot M_j$$

enthalten.

Diese Kosten berechnen wir mit der Formel:

$$T_{i,j} = \min_{i \le m < j} (T_{i,m} + T_{m+1,j} + n_{i-1} \cdot n_m \cdot n_j)$$

Pseudo-Code

```
FOR i := 1 TO k DO                                   Initialisierung
        T[i,i] := 0;
        FOR j := i + 1 TO k DO T[i,j] := ∞ ENDFOR
ENDFOR;
FOR d := 1 TO k − 1 DO                          d ist Differenz
        FOR i := 1 TO k − d DO                        zwischen i und j
                j := i + d;
                FOR m := i TO j − 1 DO          m ist der „Trennpunkt“
                        t := T[i,m]+T[m + 1,j] + n[i − 1] ∗ n[m] ∗ n[j];
                        IF t < T[i,j] THEN T[i,j] := t; B[i,j] := m ENDIF
                ENDFOR                                Trennpunkte in B
        ENDFOR                                              protokollieren
ENDFOR
```

Beispiel

Wir wollen die folgende Matrixmultiplikation ausführen:

$$M_1 \cdot M_2 \cdot M_3 \cdot M_4$$

M_1 ist eine (3×11)-, M_2 eine (11×5)-, M_3 eine (5×7)-, M_4 eine (7×2)-Matrix.

Bitte selbst durchführen...

Als Ergebnis sollten Sie die folgenden Matrizen erhalten:

$$T: \begin{matrix} 0 & 165 & 270 & 246 \\ & 0 & 385 & 180 \\ & & 0 & 70 \\ & & & 0 \end{matrix} \qquad B: \begin{matrix} - & 1 & 2 & 1 \\ & - & 2 & 2 \\ & & - & 3 \\ & & & - \end{matrix}$$

Also ist die optimale Klammerung: $M_1 \cdot (M_2 \cdot (M_3 \cdot M_4))$