Iteration, Rekursion, Divide and Conquer*

Hans Jürgen Ohlbach

23. März 2018

Keywords: Algorithmen, Iteration, Divide and Conquer, Rekursion, Endrekursion, Komplexität, Master-Theorem

Empfohlene Vorkenntnisse: $\mathcal{O}$-Notation, Logarithmus, Java ist hilfreich, aber nicht notwendig

Inhaltsverzeichnis

*Dieser Text ist Teil einer Sammlung von Miniskripten zur Einführung in die Informatik. Er ist in erster Linien für Nichtinformatiker gedacht, kann aber natürlich auch als erste Einführung für Informatiker nützlich sein.

1 Einleitung

Algorithmen kann man auf die unterschiedlichste Weise strukturieren. Einige Strukturierungsprinzipien für Algorithmen haben sich aber als besonders nützlich herausgestellt. Dazu gehören Iteration, Divide and Conquer und Rekursion. In diesem Miniskript werden diese Verfahren vorgestellt, und insbesondere auch Methoden zur Komplexitätsanalyse dieser Algorithmen.

2 Iteration

Iteration ist das einfachste dieser Prinzipien: man hat eine geordnete Datenstruktur, z.B. eine Liste von Objekten, oder auch nur einen Zahlenbereich. Diese arbeitet man eins nach dem anderen ab. Man sagt dafür, man „iteriert“ durch die Liste oder über die Zahlen. Das hat man schon in der Grundschule kennen gelernt, z.B. bei der Methode zur Addition zweier positiver Zahlen:

- schreibe die Zahlen untereinander,
- gehe von rechts nach links durch die Ziffern,
- addiere jeweils die beiden Ziffern plus einen Übertrag (der zu Beginn 0 ist),
- schreibe die Einerstelle vor die bisherigen Ergebnisziffern,
- merke Dir die Zehnerstelle als Übertrag.

Die Iteration steckt dabei in der Zeile „gehe von rechts nach links durch die Ziffern“.

In der Informatik ist Iteration die vielleicht häufigste Vorgehensweise überhaupt. Folgende Java Methode berechnet die Fakultätsfunktion $n! = 1 \cdot \ldots \cdot n$ *iterativ*:

```
static long fakultaet(int n) {
   long fak = 1;
   for(int i = 2; i <= n; ++i) {fak *= i;}
   return fak;}
```

Die Iteration steckt dabei in der for-Schleife. Sie läuft von 2 bis n und multipliziert die Variable i auf die Variable fak.

Nicht immer ist die Anzahl der Iterationen beim Eintritt in die Schleife bekannt. Das Ende der Iterationen kann sich u.U. erst durch eine Bedingung während der Abarbeitung ergeben. Will man z.B. testen, ob eine bestimmte Zahl in einem Array von Zahlen ist, oder nicht, dann kann man es z.B. so machen.

```
static boolean enthaelt(int m, int[] liste) {
   for(int k:liste) {
      if(k == m) {return true;}}
   return false;}
```

Die Iteration läuft durch das Array und bricht ab, sobald das Element gefunden wurde.

Eine Alternative zur for-Schleife wäre die while-Schleife.

```
static boolean enthaelt(int m, int[] liste) {
    int i = 0;
    while(i < liste.length && liste[i] != m) {++i;}
    return i < liste.length;
```

`while(i < liste.length && liste[i] != m) {++i;}` erhöht die Zahl i so lange, bis entweder das Ende der Liste erreicht wurde (`i < liste.length` wurde falsch), oder die Zahl n gefunden wurde (`liste[i] == m` wurde wahr).

Die for-Schleife eignet sich eher für eine vorher bekannte Anzahl von Iterationen, während die while-Schleife benutzt wird, wenn das Ende der Iteration durch eine Bedingung erreicht wird. Schreibt man die Bedingung, die das Ende der Iteration erzwingt, als while-Bedingung in der while-Schleife auf, wird die Struktur der Iteration deutlicher, und daher das Programm verständlicher als wenn man mit Abbrüchen an irgendwelchen Stellen den Kontrollfluss verändert.

2.1 Die Komplexität der Iteration

Die Zeitkomplexität der Iteration ergibt sich zunächst aus der Anzahl der Iterationen mal der Komplexität der Einzeloperationen innerhalb der Iteration. Ist die Anzahl der Iterationen n, und die Komplexität der Einzeloperationen z.B. $\mathcal{O}(m^2)$, wobei m die Größe der Objekte angibt, die in jedem Schritt bearbeitet werden, dann ist die Gesamtkomplexität $\mathcal{O}(n \cdot m^2)$.

Für die obige Fakultätsfunktion ist die Komplexität der Einzeloperation (nämlich nur die Multiplikation) konstant, d.h. $\mathcal{O}(1)$. Die Schleife wird $n-2$ mal durchlaufen. Daher ist die Gesamtkomplexität $\mathcal{O}((n-2)\cdot 1) = \mathcal{O}(n)$ (die -2 wird ignoriert).

Bei einer while-Schleife kommt noch die Komplexität des Tests der Bedingung hinzu. Man muss also die Anzahl der Schleifendurchgänge mal der Komplexität des Tests plus der Komplexität des Schleifenrumpfs berechnen. Oft weiss man aber nicht, wieviele Schleifendurchgänge nötig sind, bis die Bedingung den Abbruch erzwingt. In diesem Fall berechnet man die „worst case“-Komplexität, d.h. man schätzt die *maximal mögliche* Anzahl von Schleifendurchgängen. Für bestimmte Anwendungen könnte auch die *durchschnittliche* Anzahl von Schleifendurchgängen interessant sein. Daraus berechnet man die „average case“-Komplexität.

Bei dem `enthaelt`- Beispiel mit der while-Schleife kann die Bedingung in konstanter Zeit getestet werden, und der Schleifenrumpf braucht auch nur konstante Zeit. Die maximale Anzahl von Schleifendurchgängen ergibt sich aus der Länge der Liste, sagen wir n. Damit erhält man für die Gesamtkomplexität: $n \cdot (\mathcal{O}(1) + \mathcal{O}(1)) = \mathcal{O}(n)$.

Es gibt allerdings auch Fälle, wo die maximale Anzahl der Schleifendurchgänge in einer While-Schleife nicht abzuschätzen ist. Ein berüchtigtes Beispiel ist die *Collatz-Funktion*. Eine rekursive Darstellung (siehe Kap.4) ist:

$$\text{Collatz}(n) = \begin{cases} 1 & \text{falls } n = 1 \\ \text{Collatz}(n/2) & \text{falls } n \text{ gerade ist} \\ \text{Collatz}(3n+1) & \text{falls } n \text{ ungerade ist} \end{cases}$$

Eine iterative Implementierung davon ist:

```
static int collatz(int n) {
    while(n != 1) {
        if((n % 2 ) == 0) {n = n/2;}
        else {n = 3*n+1;}}
    return n;}
```

`(n % 2 ) == 0` ist der Test, ob n gerade ist.
Wenn man jetzt z.B. `collatz(5)` aufruft, nimmt n die Wert an: 5, 16, 8, 4, 2, 1.
Bei `collatz(19)` ergibt sich für n: 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
Bei `collatz(31)` ergibt sich für n sogar: 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1.

Egal, für welche Zahl n man die Funktion aufruft, die Schleife kann wenige oder ganz viele Iterationen durchlaufen, endet aber überraschenderweise immer bei $n = 1$. Da man das aber nicht für alle unendlich vielen natürlichen Zahlen testen kann, ist das nicht wirkliche garantiert.

In der Tat ist es noch ein offenes Problem der Mathematik: endet die Collatz-Folge immer mit 1?

Für unsere Komplexitätsberechnungen bedeutet das, dass es Fälle gibt, wo eine Abschätzung der Zeitkomplexität für die Iteration extrem schwierig bis unmöglich ist.

3 Divide and Conquer (Teile und Herrsche)

Während man Iteration schon in der Schule an vielen Beispielen kennen gelernt hat, ist Divide and Conquer wohl selten explizit vorgekommen, sondern eher versteckt. Um das zu illustrieren betrachten wir mal folgendes Beispiel:

Die Kanzlerin/der Kanzler möchte eine Volkszählung machen. Es gibt zwei Möglichkeiten:

Volkszählung per Iteration:
Sie besorgt sich einen sehr großen LKW, lädt darauf 80 Millionen Fragebögen, und fährt damit von Haus zu Haus, um die Fragebögen ausfüllen zu lassen. Nach ein paar Jahren kommt sie wieder nach Berlin zurück, und kippt den ganzen LKW voller ausgefüllter Fragebögen ihren Statistikern vor die Füße, damit sie die auswerten.

Volkszählung per Divide and Conquer:
Sie beauftragt die Ministerpräsidenten der Bundesländer: macht eine Volkszählung in eurem Bundesland, und schickt mir die Auswertung.

Diese beauftragen die Landräte der Landkreise: macht eine Volkszählung in eurem Landkreis, und schickt mir die Auswertung.

Diese beauftragen die Bürgermeister der Kommunen: macht eine Volkszählung in eurer Kommune, und schickt mir die Auswertung.

Diese heuern genügend Studenten an, die von Haus zu Haus gehen und die Fragebögen ausfüllen lassen. Anschließend wird die statistische Auswertung gemacht und an den jeweiligen Landrat geschickt.

Die Landräte sammeln die Auswertungen der Bürgermeister, fassen sie zu einer Gesamtauswertung zusammen und schicken sie an die jeweiligen Ministerpräsidenten

Die Ministerpräsidenten sammeln die Auswertungen der Landräte, fassen sie zu einer Gesamtauswertung zusammen und schicken sie nach Berlin.

Die Kanzlerin sammelt die Auswertungen der Ministerpräsidenten, fasst sie zu einer Gesamtauswertung zusammen und ist fertig.

An diesem Beispiel haben wir alle Bestandteile einer Vorgehensweise per Divide and Conquer: Eine große Aufgabe wird in kleinere Aufgaben zerlegt, diese wiederum in noch kleinere Aufgaben, usw. bis die Aufgaben klein genug sind, so dass sie direkt gelöst werden können (das machen die Studenten in dem Beispiel). Anschließende werden die Ergebnisse in den gleichen Stufen wo die Aufgaben zerlegt wurden, wieder nach oben gereicht, dort zusammengefasst und wieder weiter nach oben gereicht, usw. bis sie irgendwann ganz oben angekommen sind.

Es gibt viele weitere Alltagsbeispiele, die so funktionieren. Ein Haus baut man nicht iterativ, sondern zerlegt die Arbeit in kleinere Teile, die wiederum in kleinere Teilaufgaben zerlegt werden, bis die

Aufgaben so klein sind, dass man sie ausführen kann. Manche davon kann man nur nacheinander machen, manche aber auch parallel.

4 Rekursion

Rekursion ist die Umsetzung des Divide and Conquer-Prinzips in Algorithmen, und insbesondere in Programmiersprachen. Dabei werden Funktionen oder Methoden definiert, die sich selbst direkt oder indirekt wieder aufrufen, allerdings mit Parametern, die immer näher an einen Zustand kommen müssen, wo die Arbeit direkt ausgeführt werden kann. Das ist die *Rekursionsbasis*. Auf dem Weg zur Rekursionsbasis spricht man von *rekursiven Aufrufen*.

Zwei Beispiele sollen das illustrieren.

Rekursive Implementierung der Fakultätsfunktion:
Für die Fakultätsfunktion gilt

$$n! = 1 \cdot \ldots (n-1) \cdot n = (n-1)! \cdot n$$

Um $n!$ zu berechnen wenn $n > 1$ ist, könnte man also zunächst $(n-1)!$ berechnen, und dann noch mit n multiplizieren. Für $n = 1$ ergibt sich sofort $n! = 1$. Das ist die Rekursionsbasis.

Eine Implementierung davon wäre:

```
static long fakultaet(int n) {
   if(n == 1) {return 1;}
   else       {return fakultaet(n-1)*n;}}
```

Die Aufrufreihenfolge für z.B. `fakultaet(5)` ist dann:

```
fakultaet(5)
fakultaet(4) * 5
(fakultaet(3) * 4) * 5
((fakultaet(2) * 3) * 4) * 5
(((fakultaet(1) * 2) * 3) * 4) * 5
(((1 * 2) * 3) * 4) * 5
((2 * 3) * 4) * 5
(6 * 4) * 5
24 * 5
120
```

Man sieht hier deutlich die *Abstiegsphase* der rekursiven Aufrufe, gefolgt von der *Aufstiegsphase*, wo die Ergebnisse der Abstiegsphase weiter verarbeitet werden. Die Ebene, wo die Abstiegsphase in die Aufstiegsphase übergeht, nennt man auch die *maximale Rekursionstiefe*. Im Beispiel ist sie 5.

Rekursive Implementierung der Collatz-Funktion:
Hier nutzen wir die ursprüngliche Definition:

$$\text{Collatz}(n) = \begin{cases} 1 & \text{falls } n = 1 \\ \text{Collatz}(n/2) & \text{falls } n \text{ gerade ist} \\ \text{Collatz}(3n+1) & \text{falls } n \text{ ungerade ist} \end{cases}$$

die man rekursiv direkt implementieren kann:

```
static int collatz(int n) {
    if(n == 1)          {return 1;}
    if((n % 2 ) == 0)   {return collatz(n/2);}
    else                {return collatz(3*n+1);}}
```

In beiden Beispielen wird innerhalb der neu definierten Funktion eben diese Funktion wieder aufgerufen. Das ist typisch für eine rekursive Implementierung: Funktionen rufen sich selbst auf. Allerdings muss nicht immer eine Funktion f sich direkt selbst aufrufen. Es kann auch sein, dass eine Funktion f die Funktion g aufruft, die wiederum f aufruft, oder noch indirekter: f ruft g_1 auf, diese ruft g_2 auf usw. bis die Funktion g_n wieder f aufruft. Das ist dann eine sehr indirekte Rekursion.

Ganz wichtig ist aber die Rekursionsbasis, an der die rekursiven Aufrufe enden. Vergisst man diese, dann terminiert die Rekursion entweder gar nicht, oder mit einer Fehlermeldung (z.B. Stack Overflow).

In der rekursiven Implementierung der Fakultätsfunktion sieht man sofort, dass die Funktion rekursiv mit immer kleineren Zahlen aufgerufen wird, solange bis die Zahl = 1 geworden ist, wo die Rekursion endet.

Bei der rekursiven Implementierung der Collatz-Funktion gibt es auch eine Rekursionsbasis. Aber ob die immer erreicht wird, und die Rekursion dann endet, ist nicht klar. Das Argument beim rekursiven Aufruf wird halt mal kleiner und mal wieder größer. Obwohl es immer mal wieder kleiner wird, könnte es tendentiell trotzdem immer größer werden, oder es könnte auch in einen Zyklus geraten, so dass die Rekursionsbasis nie erreicht wird. Nichts davon hat sich bisher bei Startwerten bis 10^{20} gezeigt. Aber der Beweis der Terminierung für alle Zahlen scheint ungeheuer schwierig, oder gar unmöglich zu sein.

4.1 Endrekursion

Eine Aufrufreihenfolge der rekursiven Implementierung der Fakultätsfunktion ist z.B.

```
fakultaet(5)
fakultaet(4) * 5
(fakultaet(3) * 4) * 5
((fakultaet(2) * 3) * 4) * 5
(((fakultaet(1) * 2) * 3) * 4) * 5
(((1 * 2) * 3) * 4) * 5
((2 * 3) * 4) * 5
(6 * 4) * 5
```

```
24 * 5
120
```

Das Beispiel zeigt eines der Probleme des Rekursionsprinzips: Um das Ergebnis des rekursiven Aufrufs weiter zu verarbeiten, muss die ganze Sequenz der Zahlen zwischengespeichert werden, solange bis der jeweilige rekursive Aufruf fertig ist, und dann multipliziert werden kann. Die Zwischenspeicherung passiert automatisch auf dem sog. Aufrufstack. Weder als Programmierer noch als Anwender bekommt man etwas davon zu sehen. Trotzdem kann es den Rechner enorm belasten und das Programm verlangsamen.

Will man das Programm effizienter machen, und trotzdem rekursiv implementieren, dann muss man die Zwischenspeicherung auf dem Stack vermeiden. Bei der rekursiven Implementierung der Fakultätsfunktion kommt das Problem aus dem Codefragment `fakultaet(n-1)*n`, wo der Teil `*n` erfordert, dass das `n` zwischengespeichert wird, bis `fakultaet(n-1)` fertig gerechnet hat.

Folgende Implementierung vermeidet das Problem:

```
static long fakultaet(int n, long akk) {
   if(n == 1) {return akk;}
   else {return fakultaet(n-1,akk*n);}}
```

Hierbei wird ein zweiter Parameter `akk` (für Akkumulator) eingeführt. Um $n!$ zu berechnen wird die Funktion mit `fakultaet(n,1)` aufgerufen. Die Aufrufreihenfolge für z.B. n = 5 ist dann:

```
fakultaet(5,1)
fakultaet(4,5)
fakultaet(3,20)
fakultaet(2,60)
fakultaet(1,120)
120
```

Offensichtlich muss hier nichts zwischengespeichert werden. Die Multiplikationen werden direkt beim rekursiven Aufruf durchgeführt.

Man nennt diese Art der Rekursion *Endrekursion*.

Eine rekursive Funktion ist dann endrekursiv, wenn der rekursive Aufruf so ist, dass nach Beendigung des rekursiven Aufrufs keine weitere Berechnung mehr notwendig ist.

In der ersten Version der Fakultätsfunktion war in `fakultaet(n-1)*n` nach Beendigung des rekursiven Aufrufs noch eine Multiplikation notwendig. In der endrekursiven Version war nach Beendigung von `fakultaet(n-1,akk*n)` nichts mehr zu tun.

Endrekursive Funktionen brauchen während der rekursiven Aufrufe keine Daten zwischenzuspeichern. Sie sind daher effizienter abzuarbeiten als welche mit Zwischenspeicherung. Ganz schlaue Compiler schaffen es sogar, Endrekursionen in Iterationen zu übersetzen, was dann die effizienteste Art der Abarbeitung erlaubt.

4.2 Dynamisches Programmieren (dynamic programming)

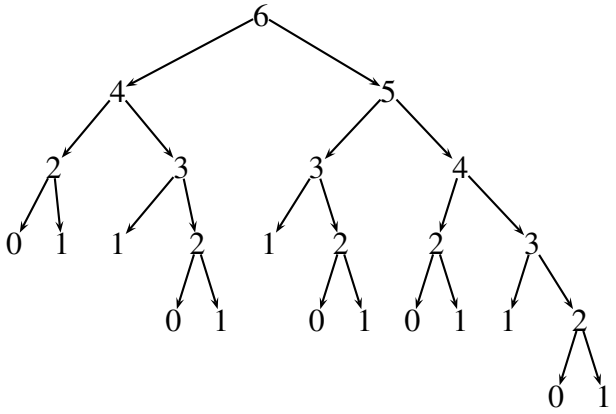
Rekursives Programmieren ist zwar eine sehr elegante und, wenn man damit vertraut ist, auch sehr übersichtliche Art des Programmierens. Es hat aber auch seine Tücken, wie wir an der nicht-endrekursiven Version der Fakultätsfunktion gesehen haben. Eine weitere Tücke besteht darin, dass sich rekursive Aufrufe u.U. sehr oft wiederholen können, und immer wieder dasselbe berechnen. Ein Beispiel, an dem man das deutlich sieht ist die *Fibonacci-Funktion*:

$$\mathit{fibonacci}(n) = \begin{cases} n & \text{falls } n < 2 \\ \mathit{fibonacci}(n-2) + \mathit{fibonacci}(n-1) & \text{sonst} \end{cases}$$

Die folgende Tabelle zeigt die ersten 16 Fibonacci-Zahlen.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
fibonacci(n)	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597

Der Baum zeigt die Aufrufhierarchie beim Aufruf von *fibonacci*(6). Nur das Argument n in den rekursiven Aufrufen ist dargestellt.



Man sieht, wie oft *fibonacci*(n) für dasselbe n aufgerufen wird. Wenn die Funktion einigermaßen effizient implementiert werden soll, müssen diese wiederholten Aufrufe für dasselbe Argument *unbedingt* vermieden werden.

Die einfachste Idee, solche wiederholten Aufrufe zu vermieden, ist, sich die *Zwischenergebnisse zu merken* und bei den wiederholten Aufrufen *einfach abzurufen*, statt sie noch einmal zu berechnen.

Genau das nennt man *dynamisches Programmieren*.

Die folgende Java-Implementierung nutzt eine HashMap, um sich die Ergebnisse der rekursiven Aufrufe zu merken. Mit `fibs.put(n,fib)` wird das aktuelle Ergebnis des rekursiven Aufrufs abgespeichert. Mit `fibs.get(n)` kann man nachsehen, ob für `n` schon früher ein Ergebnis berechnet wurde. Das kann man dann wieder verwenden.

```
static HashMap<Integer, Integer> fibs = new HashMap<Integer,Integer>();

static int fibonacci(int n) {
    if(n < 2) {return n;}
    Integer fib = fibs.get(n);
    if(fib != null) {return fib;}
    fib = fibonacci(n-2)+fibonacci(n-1);
    fibs.put(n,fib);
    return fib;}
```

Dynamisches Programmieren reduziert in diesem Fall die Anzahl der rekursiven Aufrufe von exponentiell auf linear. Für $fibonacci(40)$, z.B., reduziert sich die Anzahl rekursiver Aufrufe von 331160281 auf 41.

Die Fibonacci-Zahlen eignen sich gut, um das Prinzip der dynamischen Programmierung zu zeigen. Wenn es nur um diese Zahlen geht, würde man sie natürlich viel einfacher iterativ implementieren, indem man sich jeweils das vorletzte und das letzte Ergebnis merkt. Eine mögliche Implementierung wäre:

```
static int fibonacci(int n) {
    if(n < 2) {return n;}
    int fib = 0;
    int fib1 = 1;
    int fib2 = 0;
    for(int i = 2; i <= n; ++i) {
        fib = fib2 + fib1;
        fib2 = fib1;
        fib1 = fib;}
    return fib;}
```

Nicht alle rekursiven Funktionen sind jedoch so einfach iterativ zu implementieren. Um wiederholte rekursive Aufrufe mit denselben Argumenten zu vermieden, hilft dynamisches Programmieren.

5 Komplexitätsanalyse bei Divide and Conquer-Verfahren

Divide and Conquer-Verfahren spalten große Aufgaben in Einzelaufgaben um, die wiederum in Einzelaufgaben aufgespalten werden, usw. Das kann überhaupt nur funktionieren, wenn die Einzelaufgaben, in die eine Aufgabe aufgespalten wird, *kleiner* sind, als die aufgespaltete Aufgabe. Das muss im Prinzip nicht unmittelbar passieren, aber *tendenziell* müssen die aufgespalteten Aufgaben kleiner werden, so dass sie irgendwann klein genug sind, um direkt gelöst zu werden. Produziert die Aufspaltung immer sofort kleinere Teilaufgaben, dann ist es offensichtlich, dass das Verfahren endet, und eine Komplexitätsanalyse ist möglich. Die Collatz-Funktion ist jedoch ein Beispiel, wo nicht immer sofort kleinere Teilaufgaben produziert werden. Die Zahlen können wachsen, bis sie eine Zweierpotenz treffen, ab der der Abstieg bis zur 1 zwingend ist.

Im Folgenden betrachten wir nur Probleme, wo die Aufspaltung *direkt* kleinere Teilaufgaben produ-

ziert. Die Terminierung ist damit garantiert, und wir können eine Komplexitätsanalyse versuchen.

Die Gesamtkomplexität eines solchen Verfahrens ergibt sich dann aus folgenden Aspekten:

- in wieviele Teilaufgaben jeweils aufgespalten wird,
- wie die Größe der Teilaufgaben jeweils reduziert wird (daraus ergibt sich die Rekursionstiefe),
- wie groß der Aufwand der Aufspaltung und der Zusammenfassung der Ergebnisse der Teilaufgaben ist.

Bei der rekursiven Implementierung der Fakultätsfunktion ist das einfach:
- es wird in *eine* Teilaufgabe aufgespalten (`fakultaet(n-1)`)
- die Größe reduziert sich jeweils um 1
- der Aufwand der Subtraktion und der anschließenden Multiplikation ist konstant ($\mathcal{O}(1)$).

Da sich die Größe um 1 reduziert, ist die Rekursionstiefe n. Als Gesamtkomplexität ergibt sich damit $1 \cdot n \cdot \mathcal{O}(1) = \mathcal{O}(n)$.

Jetzt betrachten wir ein komplexeres Verfahren: Mergesort.

Beispiel: Mergesort
Dies ist ein Verfahren, um ein ein Array vergleichsbasiert zu sortieren. Es funktioniert folgendermaßen:

- Das Array wird in zwei möglichst gleich große Teilarrays zerlegt.
- Die Teilarrays werden mit Mergesort rekursiv sortiert.
- Die beiden sortierten Teilarrays werden im Reißverschlussverfahren zu einem sortierten Array zusammengefasst (engl. *merging*).

Die Basis der Rekursion ist der Fall, dass das Array nur ein oder zwei Elemente enthält. Die können direkt sortiert werden.

Als Beispiel sortieren wir (5,3,10,7,1,9,2,8).

Aufspaltung:	(5,3,10,7)	(1,9,2,8).	Reißverschluss:	(1) Rest (3,5,7,10) (2,8,9)
Aufspaltung:	(5,3) | (10,7)	(1,9) | (2,8).		(1,2) Rest (3,5,7,10) (8,9)
Sortierung:	(3,5) | (7,10)	(1,9) | (2,8)		(1,2,3) Rest (5,7,10) (8,9)
Reißverschluss:	(3) Rest (5) (7,10)	(1) Rest (9) (2,8)		(1,2,3,5) Rest (7,10) (8,9)
	(3,5) Rest () (7,10)	(1,2) Rest (9) (8)		(1,2,3,5,7) Rest (10) (8,9)
	(3,5,7) Rest () (10)	(1,2,8) Rest (9) ()		(1,2,3,5,7,8) Rest (10) (9)
	(3,5,7,10)	(1,2,8,9)		(1,2,3,5,7,8,9) Rest (10) ()
				(1,2,3,5,7,8,9,10)

Das Beispiel verdeutlicht auch, dass für jede Rekursionstiefe der Gesamtaufwand für die Aufspaltung und das Reißverschlussverfahren immer so groß ist wie die Originalliste lang ist.

Die Komplexität ergibt sich nun aus:
- es wird in *zwei* Teilaufgaben aufgespalten,
- die Größe der Listen reduziert sich auf die Hälfte,
- der Aufwand der Teilung und des Reißverschlussverfahrens ist linear ($\mathcal{O}(n)$).

Um die Rekursionstiefe zu bestimmen, muss man ermitteln, wie oft man eine Liste aus n Elementen halbieren kann. Ein Beispiel illustriert das. Wir halbieren eine Liste der Länge 256:

$$\left.\begin{array}{l}128\\64\\32\\16\\8\\4\\2\\1\end{array}\right\} 8 = \log_2(256)$$

Eine Liste der Länge n kann man also ungefähr $\log_2(n)$ mal halbieren. Daher ist die Rekursionstiefe bei Mergesort $\log_2(n)$. Für die Gesamtkomplexität erhalten wir also $2 \cdot log_2(n) \cdot \mathcal{O}(n) = \mathcal{O}(n \log(n))$.

5.1 Das Master-Theorem

Das Mergesort-Beispiel illustriert an einem noch einigermaßen übersichtlichen Beispiel die Vorgehensweise bei der Komplexitätsanalyse eines Divide and Conquer-Verfahrens. Für andere Probleme ist es die Analyse allerdings nicht so einfach. Hierbei hilft das sog. *Master Theorem*. Mit ihm kann man die Lösung von sog. *Rekursionsgleichungen* bestimmen. Rekursionsgleichungen haben die Form:

$$T(n) = a \cdot T(n/b) + f(n)$$

- $T(n)$ ist der Gesamtaufwand für ein Divide and Conquer-Verfahren für Eingaben der Größe n.
- a ist die Anzahl von Unterproblemen, in die das Problem aufgespalten wird.
- b ist der Bruchteil, mit dem die Größe des Problems bei den Unterproblemen reduziert wird. b bestimmt die Tiefe der Rekursion. $b = 2$ (Halbierung) bewirkt eine Tiefe von $\log_2(n)$. $b = 4$ (Viertelung) bewirkt eine Tiefe von $\log_4(n)$. Allgemein ist dann die Tiefe der Rekursion $\log_b(n)$.
- $f(n)$ bestimmt den Aufwand für das Zerlegen in Unterprobleme und das Zusammenbauen der Unterergebnisse.
- $T(n/b)$ ist damit der Aufwand für jedes einzeln gelöste Unterproblem.

Für Mergesort würde gelten:
$a = 2$ (zwei Unterprobleme)

$b = 2$ (Halbierung)
$f(n) \in \mathcal{O}(n)$ (linearer Aufwand für Aufspaltung und Reißverschlussverfahren).

Also $T(n) = 2 \cdot T(n/2) + \mathcal{O}(n)$ (mit der Lösung $T(n) \in \mathcal{O}(n \log(n))$).

Um das Master-Theorem zu verstehen, ist folgende Betrachtung hilfreich:

Im Mergesort Beispiel haben wir gesehen, dass der Gesamtaufwand für Splitting und Reißverschlussverfahren auf jeder Rekursionsebene gleich ist. Daher ergab sich dort für jede Rekursionsebene ein Gesamtaufwand von $\mathcal{O}(n)$. Bei der maximalen Rekursionstiefe von $\log_2(n)$ hat man daher den Gesamtaufwand $\mathcal{O}(n \log(n))$.

Bei anderen Aufteilungen in Unterprobleme als zweimal halbieren ist der Fall, dass der Gesamtaufwand auf jeder Rekursionsebene gleich ist, ebenfalls von besonderem Interesse.

Wenn, wie oben definiert, a die Anzahl von Unterproblemen ist, und b die der Bruchteil, in den die Probleme aufgespalten werden, dann können wir zeigen, wenn der Aufwand $f(n)$ für Splitting und Zusammenbau gerade $f(n) = n^{\log_b(a)}$ ist, dann bleibt der Gesamtaufwand in jeder Rekursionsebene gleich. Für Mergesort war $a = b = 2$, und daher $\log_b(a) = \log_2(2) = 1$ und damit $f(n) = n$.

Um das zu illustrieren betrachten wir die einzelnen Rekursionsebenen:

Ebene	Aufwand
1	$n^{\log_b(a)}$
2	$a \cdot (n/b)^{\log_b(a)}$
3	$a^2 \cdot (n/b^2)^{\log_b(a)}$
$\vdots$	
k	$a^k \cdot (n/b^k)^{\log_b(a)}$

Das geht weiter bis zur Ebene $\log_b(n)$.

Nun ist

$$\begin{aligned} a^k \cdot (n/b^k)^{\log_b(a)} &= a^k \cdot (n^{\log_b(a)}/(b^k)^{\log_b(a)}) \\ &= a^k \cdot (n^{\log_b(a)}/b^{k \log_b(a)} \\ &= a^k \cdot (n^{\log_b(a)}/(b^{(\log_b(a^k))}) \\ &= a^k \cdot (n^{\log_b(a)}/a^k) \\ &= n^{\log_b(a)} \end{aligned}$$

Tatsächlich ist auf jeder Ebene der Gesamtaufwand gleich groß, nämlich $n^{\log_b(a)}$.
Da es $\log_b(n)$ Ebenen gibt, macht das insgesamt einen Aufwand von $\log_b(n) \cdot n^{\log_b(a)}$.

$\log_b(a)$ ist gerade der *kritische Exponent* für die Aufwandsfunktion $f(n)$, bei dem der Gesamtaufwand bei jeder Rekursionsebene gleich bleibt.

Das Master-Theorem unterscheidet nun drei Fälle:

Fall 1: Der Gesamtaufwand auf jeder Rekursionsebene verringert sich drastisch.

Fall 2: Der Gesamtaufwand auf jeder Rekursionsebene bleibt ungefähr gleich.

Fall 3: Der Gesamtaufwand auf jeder Rekursionsebene erhöht sich drastisch.

Die Fälle werden danach unterschieden, wie sich die Aufwandsfunktion $f(n)$ relativ zum kritischen Exponenten $\log_b(a)$ verhält.

Die Resultate im Master-Theorem sind:

Fall 1: Bedingung: $f(n) \in \mathcal{O}(n^c)$ mit $c < \log_b(a)$.
Aufwand: $T(n) \in \Theta(n^{\log_b(a)})$[1].
Da $n^{\log_b(a)}$ die maximale Rekursionstiefe ist, heißt das, dass nur die Rekursionstiefe relevant ist, nicht aber der Einzelaufwand in den einzelnen Rekursionsebenen.

Fall 2: Bedingung: $f(n) \in \Theta(n^{\log_b(a)} \cdot \log^k(n))$ mit $k \geq 0$.
Aufwand: $T(n) \in \Theta(n^{\log_b(a)} \cdot \log^{k+1}(n)))$.
In der Betrachtung oben hatten wir den Fall $f(n) = n^{\log_b(a)}$ betrachtet. Der zusätzliche logarithmische Faktor erlaubt eine kleine Modifikation der Aufwandsfunktion, wo der Gesamtaufwand in jeder Rekursionstiefe nicht exakt gleich bleibt, sondern leicht ansteigen darf. $f(n) \in \Theta(\ldots)$ schließlich bedeutet, dass f asymptotisch um konstante Faktoren von $n^{\log_b(a)} \cdot \log^k(n)))$ abweichen darf.

Fall 3: Bedingung: $f(n) \in \Omega(n^c)$ mit $c > \log_b(a)$,[2]
und zusätzlich aber $af(n/b) \leq kf(n)$ für ein $k < 1$ und hinreichend große n.
Aufwand: $T(n) \in \Theta(f(n))$.
D.h. die Rekursionstiefe ist irrelevant.

Hier wurden nur die Ergebnisse des Master-Theorems vorgestellt. Die Einzelheiten der Beweise findet man in entsprechenden Büchern.

Beispiele:

Fall 1: $T(n) = 8T(n/2) + 1000n^2$.
D.h. $a = 8, b = 2, \log_b(a) = 3$. Daher ist $n^2 \in \mathcal{O}(n^3)$.
Der Aufwand ist damit $T(n) \in \Theta(n^3)$.

Fall 2: Mergesort: $T(n) = 2T(n/2) + \Theta(n)$ fällt in diese Fall.
$a = 2, b = 2, k = 0, \log_a(b) = 1$.
Der Aufwand ist damit $T(n) \in \Theta(n \log(n))$.

Fall 3: $T(n) = 2T(n/2) + n^2$.
D.h. $a = 2, b = 2, \log_b(a) = 1$.
Für die Bedingung $af(n/b) \leq kf(n)$ wählen wir $k = 1/2$ und erhalten: $2(n/2)^2 \leq (1/2)n^2$.
Der Aufwand ist damit $T(n) \in \Omega(n^2)$.

Leider fallen nicht alle Divide and Conquer-Verfahren in diese drei Fälle. Eine Erweiterung des Master-Theorems ist die Akra–Bazzi-Methode, die aber hier nicht thematisiert wird.

[1] Eine Funktion f ist in $\Theta(g)$ falls f asymptotisch in einem Korridor um g liegt, d.h. für genügend große n gilt $k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$ mit $k_1, k_2 > 0$.

[2] Eine Funktion f ist in $\Omega(g)$ falls f asymptotisch stärker wächst als g, d.h. $\lim_{n->\infty} |f(n)/g(n)| > 0$.

6 Ausblick

Der Programmieren einer Funktion, insbesondere aber einer rekursiven Funktion, muss sich u.A. um drei Probleme kümmern:

Korrektheit: Tut die Implementierung der Funktion überhaupt das, was sie soll?

Terminierung: Terminiert die Implementierung der Funktion eigentlich für jede zulässige Eingabe?

Komplexität: Wie aufwendig ist die Implementierung der Funktion?

Die Korrektheit kann man meistens mit einem Induktionsbeweis zeigen. Für die Terminierung muss man eine sog. Abstiegsfunktion finden, d.h. einen Wert, von dem man zeigen kann, dass er bei jedem rekursiven Aufruf näher an die Rekursionsbasis kommt. Beides ist in einfachen Fällen offensichtlich. Wie die Collatz-Funktion zeigt, kann es aber beliebig kompliziert werden. Die Automatisierung von Korrektheits- und Terminierungsbeweisen ist ein eigenes Forschungsgebiet, auf das in diesem Miniskript nicht eingegangen werden kann.

Für die Komplexitätsanalyse hilft meistens, aber nicht immer, das Master-Theorem.

Stichwortverzeichnis