Kapitel 1

Analyse von Quicksort

Wir beginnen unseren Kurs über die Analyse von Algorithmen mit einem Beispiel. Dieses Beispiel ist ein effizienter Sortieralgorithmus und es wird sich zeigen, daß auch ein komplizierter Algorithmus mathematisch zufriedenstellend analysiert werden kann.

Unsere erste Analyse enthält alle wichtigen Elemente, die im allgemeinen auftauchen und welche im Rest der Vorlesung ausgearbeitet werden:

1. Zunächst ist ein genaues Verständnis der Arbeitsweise des Algorithmus unerläßlich; die Analyse des Laufzeitverhaltens ist sonst unmöglich. Haben wir begriffen, wozu jede Anweisung im Algorithmus dient, müssen wir eine anschauliche Beschreibung für *die Anzahl, wie oft die Anweisung ausgeführt wird,* finden. Ist ein Block von Anweisung nicht durch Sprunganweisungen unterbrochen, läßt sich dieser Block in diesem Sinn gemeinsam behandeln. Darüberhinaus gibt es weitere Möglichkeiten zur Vereinfachung, welche wir später kennenlernen werden.

2. Wollen wir eine *Analyse des Average Case* durchführen, also das *durchschnittliche* Verhalten des Algorithmus analysieren, müssen wir ein statistisches Modell für die Eingaben, welche der Algorithmus erhält, erstellen.

3. Mit Hilfe der bisher wohl noch mystischen *anschaulichen Beschreibungen* müssen wir dann eine geschlossene Formel für die Anzahl der Ausführungen jeder Anweisung finden. Dabei verwenden wir die Annahmen über die Wahrscheinlichkeitsverteilung der Eingaben. Oft ist es nicht möglich eine geschlossene Formel zu finden oder

erfordert zu hohen Aufwand und wir beschränken uns mit einer Näherungslösung.

4. Schließlich können wir die Zeiten für alle Anweisungen addieren und erhalten so die durchschnittliche Gesamtlaufzeit.

Quicksort ist als einführendes Beispiel gut geeignet, weil es nicht zu einfach ist und bereits gut bekannt ist. Wir verwenden keine vereinfachte, “akademische” Version, sondern eine realistische, optimierte Variante von Quicksort. Quicksort an sich hat den Nachteil, daß es für sehr kleine Arrays einfacheren Verfahren stark unterlegen ist. Wir verwenden daher eine Variante, welche nur neu partitioniert, wenn der zu sortierende Abschnitt länger als M ist. Am Ende folgt einmal *straight-insertion sort*, um die verbleibenden Stücke fertigzusortieren. Eine weitere Optimierung betrifft nicht die Laufzeit, sondern den benötigten Speicherplatz. Unser Algorithmus sortiert nach dem Partitionieren immer zuerst den kürzeren Teil. Durch diesen wohlbekannten Trick bleibt die Rekursionstiefe klein, da die zu sortierende Arraygröße sich immer mindestens halbiert. Aus Effizienzgründen vermeiden wir direkte rekursive Aufrufe der Prozedur und verwenden selbst einen Stack um die Rekursion zu simulieren. Abbildung 1.1 enthält ein vollständiges C-Programm für Quicksort. Lediglich für den Stack wurden drei Makros verwenden, welche sich schnell definieren lassen.

Wir wollen analysieren, wie oft jede der Anweisungen des Programms im Mittel ausgeführt wird, wenn die Eingabe aus N verschiedenen Zahlen besteht, wobei jede Permutation gleich wahrscheinlich ist. Dies ist eine übliche Annahme für das Sortierproblem. Die Eingabe befindet sich im Feld $a[1], \ldots, a[N]$ und die sortierte Folge wird dort wieder abgelegt.

Das ganze Programm findet sich nocheinmal in Abbildung 1.2, aber in einer anderen Form: Es handelt sich um eine Art Flußdiagramm. Dort sind Anweisungen, welche nicht durch Sprunganweisungen oder durch Ziele von Sprunganweisungen getrennt sind, in Blöcke zusammengefaßt und der Programmfluß durch Pfeile zwischen den Blöcken dargestellt. Neben jedem Block ist die Anzahl, wie oft die Anweisungen dieses Blocks ausgeführt werden, angeschrieben, im allgemeinen durch eine Variable oder, um nicht zu viele Variablen einzuführen, durch einen Ausdruck, welcher mehrere Variablen umfaßt.

Beginnen wir mit A in unserer weiteren Vorgehensweise. Die Variable A erscheint neben mehreren Blöcken, was natürlich bedeutet, daß diese Blöcke ebensooft ausgeführt werden.

```
void quicksort(int N)
{
  int i, j, l, r, k, t;
  l = 1; r = N;
  if (N > M)
    while (1) {
      i = l − 1; j = r; k = a[j];
      do {
        do { i++; } while (a[i] < k);
        do { j−−; } while (k < a[j]);
        t = a[i]; a[i] = a[j]; a[j] = t;
      } while (i < j);
      a[j] = a[i]; a[i] = a[r]; a[r] = t;
      if (r − i ≥ i − l) {
        if (i − l > M) { push(i + 1, r); r = i − 1; }
        else if (r − i > M) l = i + 1;
        else if (stack_is_empty) break;
        else pop(l, r);
      }
      else if (r − i > M) { push(l, i − 1); l = i + 1; }
      else if (i − l > M) r = i − 1;
      else if (stack_is_empty) break;
      else pop(l, r);
    }
  for (i = 2; i ≤ N; i++)
    if (a[i − 1] > a[i]) {
      k = a[i]; j = i;
      do { a[j] = a[j − 1]; j−−; } while (a[j − 1] > k);
      a[j] = k;
    }
}
```

Abbildung 1.1: C-Programm für Quicksort

Warum können wir die Blöcke $i = l-1;\ j = r;\ k = a[j]$ und $a[j] = a[r];\ldots$ mit der gleichen Variablen bezeichnen? Es gibt dafür folgende einleuchtende Erklärung: Betrachten wir eine Menge M von Blöcken, dann *fließt alles was in M hineinfließt wieder hinaus.* Der *Fluß* ist hier der Programmfluß, also praktisch die Bewegung des Programmzählers des Prozessors wie er sich über die Blöcke bewegt. Sooft der Programmzähler sich von außerhalb M's in M hineinbegibt, sooft verlässt er auch wieder M. Eine Analogie gibt es in der Physik in Form von quellenlosen Strömungen und in der Elektrotechnik in Form von elektrischen Strömen in einem Netzwerk.

Wenn Block $i = l - 1;\ j = r;\ k = a[j]$ gerade A mal ausgeführt wird, dann wird er auch A mal verlassen, wofür es aber nur einen Pfeil gibt, welcher in M hineinführt, wobei wir als M die Menge der Blöcke zwischen $i = l - 1;\ j = r;\ k = a[j]$ und $a[j] = a[r];\ldots$ wählen. Ebenso wird A mal die Blockmenge M verlassen, was wiederum nur durch einen Pfeil möglich ist, welcher aber gerade in den Block $a[j] = a[r];\ a[i] = a[j];\ a[j] = t$ führt. Also wird dieser ebenfalls A mal durchgeführt. Dies war eine relativ einfache Schlußfolgerung. Viele andere Beziehungen lassen sich allein durch solche Flußbeziehungen finden, z.B. $I' + I'' = 1$ oder $A' + A'' = A - 1$. Auf diese Weise reduziert sich die Anzahl der unabhängigen Variablen, welche dann immer noch analysiert werden müssen, erheblich.

In Kapitel 2 wird die Reduktion der Variablen mit Hilfe der Flußrelation systematisch betrachtet.

1.1 Die Anzahl der Partitionsphasen

Kommen wir zurück zu A. Was ist die anschauliche Bedeutung dieser Zahl? Die while-Schleife in Abbildung 1.1 wird gerade A mal durchgeführt. Jeder Durchlauf dieser Schleife bewirkt ein einmaliges Partitionieren. Wir können daher A als die Anzahl der Partionierungsphasen interpretieren. Diese anschauliche Bedeutung von A ist sehr hilfreich. In der folgenden Analyse von A sind wir nicht mehr an unser C-Programm gebunden, sondern können direkt den abstrakten Quicksortalgorithmus verwenden. Auch wenn wir die Implementierung ändern sollten, bleibt die Analyse von A gültig—es handelt sich ja um die Anzahl der Partitionierungsphasen, welche von der eigentlichen Implementierung unabhängig ist.

Sei A_N die durchschnittliche Anzahl von Partitionsphasen, wenn N Elemente durch Quicksort sortiert werden. Falls $N > M$, wird die Eingabe einmal partitioniert und die beiden Teile werden wieder sortiert. Haben

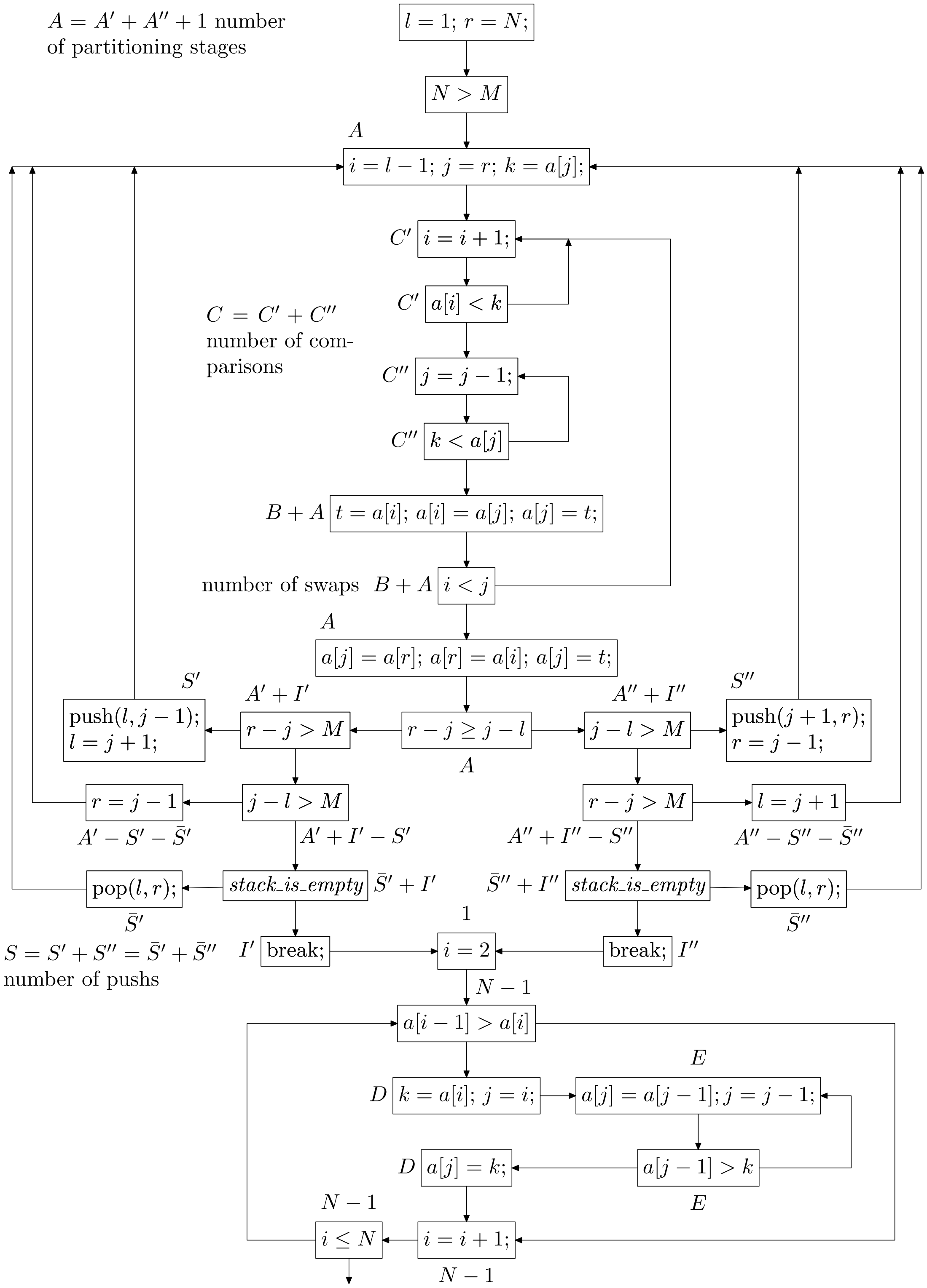

Abbildung 1.2: Programmflußdiagramm für Quicksort

die beiden Teile dann die Längen k und $N-1-k$ werden dann nochmals A_k und A_{N-1-k} Partitionierungen durchgeführt, wenn die beiden Teile wieder sortiert werden. Dabei bewegt sich k zwischen 0 und $N-1$ und nimmt jeden dieser Werte mit Wahrscheinlichkeit $1/N$ an, denn wir haben ja angenommen, daß jede Reihenfolge der N Zahlen gleichwahrscheinlich ist. Wir erhalten daher

$$\begin{aligned} A_N &= 1 + \frac{1}{N}\sum_{k=0}^{N-1}(A_k + A_{N-1-k}) \\ &= 1 + \frac{2}{N}\sum_{k=0}^{N-1} A_k, \quad \text{für } N > M \end{aligned}$$

Für $N \leq M$ gilt natürlich $A_N = 0$.

Bei der weiteren Analyse werden noch mehr Rekursionsgleichungen auftauchen, welche eine ähnliche Form haben. Sie alle können geschrieben werden als

$$X_N = \frac{2}{N}\sum_{k=0}^{N-1} X_k + f_N, \quad \text{fuer } N > M$$

für verschiedene Funktionen f_k. Im Fall von A_N war beispielsweise $f_k = 1$.

Diese Gleichungen lassen sich gar nicht so schwer lösen. Ein Problem ist, daß X_N eine Funktion von $X_0, \ldots, X_{N-1}$ ist, statt nur von wenigen früheren Werten abzuhängen. Zuerst wandeln wir die Rekursionsgleichung in endliche Ordnung um. Wir schaffen das durch Subtraktion von X_N und X_{N-1} nachdem wir die störenden Faktoren $1/N$ und $1/(N-1)$ beseitigt haben:

$$\begin{aligned} NX_N &= 2\sum_{k=0}^{N-1} X_k + Nf_N \\ (N-1)X_{N-1} &= 2\sum_{k=0}^{N-2} X_k + (N-1)f_{N-1} \end{aligned}$$

Subtraktion liefert

$$NX_N - (N-1)X_{N-1} = 2X_{N-1} + Nf_N - (N-1)f_{N-1}$$

beziehungsweise

$$NX_N = (N+1)X_{N-1} + Nf_N - (N-1)f_{N-1}, \quad \text{für } N > M+1.$$

Dies ist eine lineare Rekursionsgleichung erster Ordnung und läßt sich durch einen *Summationsfaktor* lösen, wie wir später sehen werden. Man muß hier mit $1/N(N+1)$ multiplizieren:

$$\frac{X_N}{N+1} = \frac{X_{N-1}}{N} + \frac{Nf_N - (N-1)f_{N-1}}{N(N+1)}$$

Mit den Substitutionen $Y_N = \frac{X_N}{N+1}$ und $g_N = \frac{Nf_N-(N-1)f_{N-1}}{N(N+1)}$ ergibt sich die einfache Gleichung

$$Y_N = Y_{N-1} + g_N,$$

welche für $N > M+1$ gilt. Daraus ergibt sich sofort

$$Y_N = Y_{M+1} + g_{M+2} + g_{M+3} + \cdots + g_N = Y_{M+1} + \sum_{k=M+2}^{N} g_k$$

und nach Rücksubstituierung schließlich die Lösung

$$X_N = \frac{N+1}{M+2} X_{M+1} + (N+1) \sum_{k=M+2}^{N} \frac{kf_k - (k-1)f_{k-1}}{k(k+1)}$$

Im Fall A_N hatten wir $f_k = 1$ und $A_{M+1} = 1$ und wir erhalten durch Einsetzen

$$\begin{aligned} A_N &= \frac{N+1}{M+2} + (N+1) \sum_{k=M+2}^{N} \frac{1}{k(k+1)} \\ &= \frac{N+1}{M+2} + (N+1)\left(\frac{N}{N+1} - \frac{M+1}{M+2}\right) = \frac{2N-M}{M+2} \end{aligned}$$

1.2 Die Anzahl der Vergleiche während dem Partitionieren

Wenn ein Subarray der Größe N partitioniert wird, befinden sich die Zeiger i und j zunächst am Anfang und Ende des Feldes. Wenn das Schleifenkriterium $i < j$ nicht mehr erfüllt ist, haben sich die Zeiger überkreuzt und die Partitionierungsphase ist zu Ende (wenigstens folgen keine Vergleiche mehr in dieser Phase). Vergleiche wurden durchgeführt jedesmal wenn i erhöht wurde und jedesmal wenn j erniedrigt wurde. Am Ende gilt $i = j+1$ (also $j - i = -1$) und am Anfang $j - i = N - 1$. Die Differenz zwischen j

und i wächst also gerade um $N + 1$. Dies ist dann gerade die Anzahl der durchgeführten Vergleiche. Wir erhalten so die Rekursionsgleichung

$$C_N = N + 1 + \frac{2}{N} \sum_{k=0}^{N-1} C_k,$$

welche wieder von der bekannten Form ist mit $f_k = k+1$ und $C_{M+1} = M+2$. Wir können also die Lösung leicht ausrechnen:

$$\begin{aligned} C_N = N + 1 + (N+1) \sum_{k=M+2}^{N} \frac{k(k+1) - (k-1)k}{k(k+1)} &= \\ = N + 1 + 2(N+1) \sum_{k=M+2}^{N} \frac{1}{k+1} & \\ &= N + 1 + 2(N+1)(H_{N+1} - H_{M+2}) \end{aligned}$$

1.3 Die Anzahl der Vertauschungen in der do-Schleife

Sei $B_N + 1$ die Anzahl der Tauschoperationen in der do-Schleife. Es wird eine Vertauschung zuviel ausgeführt, welche anschließend rückgängig gemacht wird. Wir analysieren B_N, die Anzahl der nicht rückgängig gemachten Tauschoperationen. (Das Rückgängigmachen einer Tauschoperation ist nötig, denn die do-Schleife wird ja mindestens einmal ausgeführt. Manchmal braucht man aber überhaupt keinen Tausch.)

Wenn das Feld in zwei Stücke der Größen k und $N - 1 - k$ zerteilt wird, dann müssen im Mittel $k(N - 1 - k)/(N - 1)$ Zahlen zwischen den Stücken ausgetauscht werden. Wir erhalten also die Rekursionsgleichung

$$B_N = \frac{1}{N} \sum_{k=0}^{N} \left(B_k + B_{N-1-l} + \frac{k(N-1-k)}{N-1} \right) = \frac{2}{N} \sum_{k=0}^{N} A_k + \frac{N-2}{6}.$$

Hier ist also $f_k = (N - 2)/6$ und $B_{M+1} = (M - 1)/6$. Die Lösung ist also

$$\begin{aligned} B_N &= \frac{(N+1)(M-1)}{6(M+2)} + \frac{(N+1)}{6} \sum_{k=M+2}^{N} \frac{k(k-2) - (k-1)(k-3)}{k(k+1)} \\ &= \frac{1}{6}(N+1) \left(2H_{N+1} - 2H_{M+2} + 1 - \frac{6}{M+2} \right) + \frac{1}{2}. \end{aligned}$$

1.4 Die Anzahl der Einfügungsphasen

Nehmen wir uns jetzt D_n vor. Wieder hilft es, erst einmal zu überlegen, was diese Größe anschaulich bedeutet. Immer wenn $a[i-1] > a[i]$ ist, wird $a[i]$ in das, zu diesem Zeitpunkt bereits sortierte, Array $a[1 \dots i-1]$ einsortiert. Die Größe D_n sagt nun genau, wie oft das passiert. Natürlich findet das Sortieren durch Einfügen erst im zweiten Teil des Algorithmus statt und daher erhalten wir die Rekursionsgleichung

$$D_N = \frac{2}{N} \sum_{k=0}^{N-1} D_k$$

für $N > M$. Wieder wird ja bei einer Partitionierung das Feld in drei Teile geteilt, von welchen zwei rekursiv weiterbearbeitet werden.

Falls aber $N \leq M$ gilt, dann wird keine Partitionierung durchgeführt und natürlich können wir wieder annehmen, daß jede Permutation der N Elemente gleich wahrscheinlich ist.

Wie groß ist dann die Wahrscheinlichkeit, daß $a[i-1] > a[i]$ in der `for`-Schleift gilt? An dieser Stelle ist, wie gesagt, $a[1], \dots, a[i-1]$ bereits sortiert und daher ist $a[i-1]$ das *größte* dieser Elemente. Es gilt also $a[i-1] > a[i]$ genau dann, wenn $a[i]$ nicht das größte Element in der Menge $\{a[1], \dots, a[i]\}$ ist. Dafür ist die Wahrscheinlichkeit genau $1 - 1/i$.

Durch Summieren erhalten wir

$$D_N = \sum_{i=2}^{N} (1 - 1/i) = N - H_N \text{ für } N \leq M.$$

Sehen wir uns einige kleine Werte an: $D_0 = 0$, $D_1 = 0$, $D_2 = 1/2$. Richtig, denn erst ab $N = 2$ wird der Rumpf der `for`-Schleife überhaupt durchlaufen. Falls $D = 2$ ist, kann lediglich $a[2] > a[1]$ ein Einfügen auslösen und das passiert in genau der Hälfte der Fälle.

Daraus können wir jetzt auch D_{M+1} berechnen:

$$D_{M+1} = \frac{2}{M+1} \sum_{k=0}^{M} D_k = \frac{2}{N} \sum_{k=0}^{M} (k - H_k) = M - 2H_{M+1} + 2$$

Für $N > M$ erhalten wir damit die geschlossene Form für D_N:

$$D_N = \frac{N+1}{M+2} D_{M+2} = \frac{N+1}{M+2}(M + 2 - 2H_{M+1}) = (N+1)\left(1 - \frac{2H_{M+1}}{M+2}\right)$$

Im wesentlichen erhalten wir also $D_N = \Theta(N)$, falls nur M eine Konstante ist. Das bedeutet, das eine lineare Anzahl von Elementen durch das „Nachsortieren" bewegt werden (außer natürlich, wenn $M = 1$ ist). Je größer wir M wählen, desto größer wird auch dieser Anteil. Das war auch zu erwarten: Elemente an den linken Rändern der partitionierten Bereiche werden ja nie nach unten verschoben, wogegen bei allen anderen Elementen ein Verschieben möglich ist—je weiter rechts sie in diesem Bereich liegen, desto größer ist auch die Wahrscheinlichkeit dafür.

1.5 Die Anzahl der Vertauschungen beim Einfügen

Der zweite Teil des Algorithmus ist ja *Sortieren durch Einfügen* und genau E-mal werden Elemente, die nicht in der richtigen Ordnung sind, vertauscht. Danach sind *diese beiden* in der richtigen Ordnung—und bleiben es dann auch. In Wirklichkeit werden die Elemente nicht einzeln vertauscht, sondern ein zyklisches Vertauschen durchgeführt. Das macht aber nichts, da das Ergebnis dasselbe ist.

Wann immer zwei Elemente nicht in der richtigen Ordnung ist, werden die E-Blöcke einmal ausgeführt. Das ist aber gerade die Anzahl der *Inversionen* der Permutation, also die Anzahl der Paare, welche nicht in der richtigen Reihenfolge sind.

Bei einer zufälligen Permutation von n Elementen gibt es $n(n-1)/2$ solche Paare und jedes Paar ist mit Wahrscheinlichkeit $1/2$ nicht in der richtigen Reihenfolge. Diese Wahrscheinlihchkeiten sind natürlich nicht voneinander unabhängig, was zu einer komplizierten Wahrscheinlichkeitsverteilung führt. Das macht natürlich gar nichts, denn wir interessieren uns ja nur für den Erwartungswert und der ist dann $n(n-1)/4$.

Wenn wir mit E_N die Anzahl der Inversionen bezeichnen, die zu Beginn des zweiten Teils des Algorithmus im Array a existieren, können wir diese Rekursionsgleichung aufstellen:

$$E_N = \begin{cases} \dfrac{2}{N} \displaystyle\sum_{k=0}^{N-1} E_k & \text{für } N > M \\ \dfrac{1}{2}\dbinom{N}{2} & \text{für } N \leq M \end{cases}$$

Wieder ist die Gleichung in der jetzt wohlbekannten Form wir können zunächst E_{M+1} bestimmen:

$$E_{M+1} = \frac{2}{M+1}\sum_{k=0}^{M}\frac{k^2}{4} = \frac{M(M-1)}{6}$$

Für $N > M$ erhalten wir das Ergebnis

$$E_N = \frac{N+1}{M+2}E_{M+1} = \frac{N+1}{M+2}\frac{M(M-1)}{6}.$$

1.6 Zusammenfassung

Folgende Tabelle zeigt noch einmal alle analysierten Größen gemeinsam:

$$\begin{aligned}
A_N &= \frac{2N-M}{M+2}\\
B_N &= \frac{1}{6}(N+1)\left(2H_{N+1} - 2H_{M+2} + 1 - \frac{6}{M+2}\right) + \frac{1}{2}\\
C_N &= N+1+2(N+1)(H_{N+1} - H_{M+2})\\
D_N &= (N+1)(1 - 2H_{M+1}/(M+2))\\
E_N &= \frac{1}{6}(N+1)M(M-1)/(M+2)\\
S_N &= (N+1)/(2M+3) - 1
\end{aligned}$$

Abbildung 1.3 enthält das aus dem C-Programm in Abbildung 1.1 gewonnene Assemblerprogramm für den DLX-Prozessor von Hennessy und Patterson. Wir werden diesen Prozessor einige Male für Beispiele heranziehen. Die Wahl für DLX fiel aus folgenden Gründen:

1. Es handelt sich um einen typischen RISC-Prozessor und er ist damit repräsentativ für die Prozessoren, welche in heute und in zumindest naher Zukunft vorherrschenden Computern verwendet werden.

2. Er wurde von Patterson und Hennessy für Lehrzwecke entworfen und enthält aus diesen Grund keine Sonderlösungen, wie Registerbänke oder Spezialbefehle; er ist vielmehr ein kleinster, gemeinsamer Kern aller RISC-Prozessoren. Der Entwurf von DLX ist einfach und die Befehle lassen sich in 10 Minuten lernen (zumindest für unsere Zwecke). Da das Buch von Patterson und Hennessy das Standardwerk für Rechnerarchitektur ist, ist DLX auch relativ gut bekannt; jedenfalls bekannter als jeder echte RISC-Prozessor.

3. Wir werden sehen, daß es nicht ausreichend ist, sich auf eine Hochsprache zu stützen, da dann nicht mehr feststeht wieviel Zeit jedes Kommando wirklich in Anspruch nimmt. Bei Assemblerbefehlen und insbesondere bei RISC-Prozessoren läßt sich dagegen ziemlich genau sagen, wie lange jeder Befehl dauert. Früher, sagen wir grob in den 80ern, ließ es sich *genau* sagen. Beim 6502 von National Semiconductor (später Commodore), einem der damals sehr verbreiteten Prozessor, gab es eine Tabelle, welche zu jedem Befehl genau angab, wieviele Taktzyklen er benötigt. Dies war vom Kontext unabhängig mit der Ausnahme, daß bedingte Sprungbefehle einen Taktzyklus mehr benötigten, wenn der Sprung tatsächlich ausgeführt wurde. Leider gibt es bei RISC-Prozessoren keine so einfache Zuordnung, da die Ausführungszeit von sehr vielen Faktoren abhängen kann. Ein Beispiel ist der Cache. Es ist sehr schwer, wenn nicht unmöglich, zu sagen, wann benötigte Daten oder Befehle im Cache sind und wann sie erst geladen werden müssen. Allerdings ist es eine sehr gute Näherung bei RISC-Prozessoren anzunehmen, daß jeder Befehl genau einen Taktzyklus benötigt.

Anhang ? enthält eine für uns ausreichende Beschreibung von DLX. Eine ausführliche Behandlung findet sich in Hennessy und Patterson.

Das Programm in Abbildung 1.3 ist ergänzt um die Anzahl der Ausführung jeder einzelnen Assembleranweisung. Dazu ist es notwendig jeden Assemblerbefehl zum richtigen Block in Abbildung 1.1 zuzuordnen. Stark optimierende Compiler machen uns diese Arbeit nicht unbedingt leicht. Um es uns am Anfang nicht zu schwer zu machen, ist das Programm in Abbildung 1.3 zwar auf höchster Stufe optimiert, aber es wurde auf das Füllen von *branch delay slots* verzichtet. RISC-Prozessoren führen nämlich mindestens den nächsten Befehl nach einem Sprungbefehl noch aus, selbst wenn der Sprung ausgeführt wird.

Andererseits werden wir meistens die Analyse nicht so genau machen wollen. Wenn doch, dann ist diese Arbeit zwar mühsam, aber trotzdem gegenüber der größeren Arbeit der mathematischen Analyse meist vernachlässigbar. Dennoch steht uns ein Algorithmus oft nicht implementiert zur Verfügung und wir können dann nur eine unvollständige Analyse durchführen. Für Quicksort und andere Sortieralgorithmen wird oft nur eine einzige Größe untersucht: Die Anzahl der Vergleiche. Auch diese eingeschränkte Analyse kann nützliche Ergebnisse liefern. Ist beispielsweise jeder Vergleich im Vergleich zu den anderen Operationen bekannterweise sehr teuer, dann ist C die dominierende Größe in der Laufzeit und ihre Analyse ist ausreichend. Wir werden jetzt sehen, daß dies in unserem Fall—dem Sortieren von Zahlen—aber keineswegs zutrifft:

Zählen wir nun alle Befehle (im C-Quellprogramm) zusammen. Die Gesamtzahl der ausgeführten Befehle ist dann

$$10A_N + 4B_N + 2C_N + 3D_N + 3E_N + 2S_N + 3N + 6.$$

Für sehr große N spielen zwar nur B_N und C_N eine Rolle, da die anderen Größen asymptotisch langsamer wachsen, aber wir interessieren uns hier nicht nur für große N, was zu praxisfern wäre. Mit unserer genauen Analyse können wir genaue Aussagen für jedes N machen. Darüberhinaus ist eine rein asymptotische Analyse auch aus einem anderen Grund gefährlich—meist wird nicht nachgerechnet, was *für große N* eigentlich quantitativ bedeutet, denn das ist ja gerade die Bedeutung von *asymptotischer* Analyse. Für welches M ist diese Zahl minimal? Offensichtlich kann die optimale Wahl von M auch von N abhängen, aber dies ist kaum spürbar. Es stellt sich heraus, das die Wahl von N nicht abhängt, außer für sehr kleine N. Dann wählt man aber sowieso lieber ein anderes Sortierverfahren. Abbildung **??** zeigt wie sich die Laufzeit für $N = 100$ abhängig von M verhält. Die "übliche" Wahl $M = 1$ in vielen Lehrbüchern ist also wirklich nicht gut.

Aufgaben

1.1 Zeigen Sie, daß der Block $r = j - 1$ gerade $A' - S' - \bar{S}'$ oft ausgeführt wird.

1.2 Die Beziehung $S' + S'' = \bar{S}' + \bar{S}''$ läßt sich *nicht* durch Flußüberlegungen aufstellen. Dennoch gilt sie und hilft wieder die Anzahl der unabhängigen Variablen zu reduzieren. Beweisen Sie diese Beziehung.
Hinweis: Betrachten Sie die Höhe des Kellers.

1.3 Ergänzen Sie das C-Programm aus Abbildung 1.1 um Definitionen für die Makros `push`, `pop` und `stack_is_empty`. Die ersten beiden Makros sollen zwei integers auf einen Keller legen oder von ihm nehmen und der letzte Makro prüfen, ob der Keller leeleerr ist. Ergänzen Sie das Programm um ein Hauptprogramm, welches die Funktion quicksort für geeignete, zufällige permutierte Eingaben, welche aus den Zahlen 1 bis N bestehen testet. Fügen Sie Programmcode hinzu, welcher den Wert von A bei jedem Aufruf von quicksort mitzählt. Wählen Sie ein für M einen Wert, den Sie für sinnvoll halten und bestimmen Sie A mehrmals für verschiedene Werte von N.

1.4 Analysieren Sie die verbleibende Größe S_N. Finden Sie zunächst eine anschauliche Definition dieser Größen, stellen Sie dann eine sie implizit definierende Rekursionsgleichung auf und lösen Sie diese schließlich.

```
quicksort:
 1 sw -4(r29),r30
 1 add r30,r0,r29
 1 sw -8(r29),r31
 1 subui r29,r29,#64
 1 sw 0(r29),r2
 1 sw 4(r29),r3
 1 sw 8(r29),r4
 1 sw 12(r29),r5
 1 sw 16(r29),r6
 1 sw 20(r29),r7
 1 sw 24(r29),r8
 1 sw 28(r29),r9
 1 sw 32(r29),r10
 1 sw 36(r29),r11
 1 sw 40(r29),r12
 1 sw 44(r29),r13
 1 sw 48(r29),r14
 1 sw 52(r29),r15
 1 lw r14,(r30)
 1 addi r9,r0,#1
 1 add r8,r0,r14
 1 sgti r1,r14,#3
 1 beqz r1,L8
 1 nop
 1 lhi r11,((_a)>>16)&0xffff
 1 addui r11,r11,(_a)&0xffff
 1 lhi r10,((_sp)>>16)&0xffff
 1 addui r10,r10,(_sp)&0xffff
 1 lhi r13,((_s1)>>16)&0xffff
 1 addui r13,r13,(_s1)&0xffff
 1 lhi r12,((_s2)>>16)&0xffff
 1 addui r12,r12,(_s2)&0xffff
L11:
 A addi r5,r9,#-1
 A add r4,r0,r8
 A slli r1,r4,#0x2
 A add r1,r1,r11
 A lw r6,(r1)
 A add r31,r0,r1
 A slli r1,r5,#0x2
 A add r3,r1,r11
L12:
 B+A slli r1,r5,#0x2
 B+A add r2,r1,r11
L15:
 C' addi r2,r2,#4
 C' addi r3,r3,#4
 C' addi r5,r5,#1
 C' lw r1,(r2)
 C' slt r1,r1,r6
 C' bnez r1,L15
 C' nop
 B+A slli r1,r4,#0x2
 B+A add r2,r1,r11
L19:
 C'' addi r2,r2,#-4
 C'' addi r31,r31,#-4
 C'' addi r4,r4,#-1
 C'' lw r1,(r2)
 C'' slt r1,r6,r1
 C'' bnez r1,L19
 C'' nop
 B+A lw r7,(r3)
 B+A lw r15,(r31)
 B+A sw (r3),r15
 B+A sw (r31),r7
 B+A slt r1,r5,r4
 B+A bnez r1,L12
 B+A nop
 A slli r1,r4,#0x2
 A add r3,r1,r11
 A slli r1,r5,#0x2
 A add r2,r1,r11
 A lw r15,(r2)
 A sw (r3),r15
 A slli r1,r8,#0x2
 A add r1,r1,r11
 A lw r15,(r1)
 A sw (r2),r15
 A sw (r1),r7
 A sub r3,r8,r5
 A sub r2,r5,r9
 A sge r1,r3,r2
 A beqz r1,L24
 A nop
 A''+I'' sgti r1,r2,#3
 A''+I'' beqz r1,L25
 A''+I'' nop
 S'' lw r3,(r10)
 S'' slli r4,r3,#0x2
 S'' add r2,r4,r13
 S'' addi r1,r5,#1
 S'' sw (r2),r1
 S'' addi r1,r3,#1
 S'' sw (r10),r1
 S'' add r1,r4,r12
 S'' sw (r1),r8
 S'' addi r8,r5,#-1
 S'' j L11
 S'' nop
L25:
 A''+I''-S'' sgti r1,r3,#3
 A''+I''-S'' beqz r1,L34
 A''+I''-S'' nop
 A''-S''-S̄'' addi r9,r5,#1
 A''-S''-S̄'' j L11
 A''-S''-S̄'' nop
L24:
 A'+I' sgti r1,r3,#3
 A'+I' beqz r1,L32
 A'+I' nop
 S' lw r2,(r10)
 S' slli r3,r2,#0x2
 S' add r1,r3,r13
 S' sw (r1),r9
 S' addi r1,r2,#1
 S' sw (r10),r1
 S' add r2,r3,r12
 S' addi r1,r5,#-1
 S' sw (r2),r1
 S' addi r9,r5,#1
 S' j L11
 S' nop
L32:
 A'+I'-S' sgti r1,r2,#3
 A'+I'-S' beqz r1,L34
 A'+I'-S' nop
 A'-S'-S̄' addi r8,r5,#-1
 A'-S'-S̄' j L11
 A'-S'-S̄' nop
L34:
 S+1 lw r2,(r10)
 S+1 snei r1,r2,#0
 S+1 beqz r1,L8
 S+1 nop
 S+1 addi r1,r2,#-1
 S+1 sw (r10),r1
 S+1 slli r2,r1,#0x2
 S+1 add r1,r2,r13
 S+1 lw r9,(r1)
 S+1 add r1,r2,r12
 S+1 lw r8,(r1)
 S+1 j L11
 S+1 nop
L8:
 1 addi r5,r0,#2
 1 sle r1,r5,r14
 1 beqz r1,L40
 1 nop
 1 lhi r9,((_a-4)>>16)&0xffff
 1 addui r9,r9,(_a-4)&0xffff
 1 addi r10,r9,#4
 1 addi r8,r9,#12
 1 addi r7,r9,#8
L42:
 N-1 lw r1,(r7)
 N-1 lw r2,(r8)
 N-1 sgt r1,r1,r2
 N-1 beqz r1,L41
 N-1 nop
 D add r6,r0,r2
 D add r4,r0,r5
 D slli r1,r5,#0x2
 D add r2,r1,r9
 D slli r1,r5,#0x2
 D add r3,r1,r10
L44:
 E lw r15,(r2)
 E sw (r3),r15
 E addi r2,r2,#-4
 E addi r3,r3,#-4
 E addi r4,r4,#-1
 E lw r1,(r2)
 E sgt r1,r1,r6
 E bnez r1,L44
 E nop
 D slli r1,r4,#0x2
 D add r1,r1,r10
 D sw (r1),r6
L41:
 N-1 addi r8,r8,#4
 N-1 addi r7,r7,#4
 N-1 addi r5,r5,#1
 N-1 sle r1,r5,r14
 N-1 bnez r1,L42
 N-1 nop
L40:
 1 lw r2,0(r29)
 1 lw r3,4(r29)
 1 lw r4,8(r29)
 1 lw r5,12(r29)
 1 lw r6,16(r29)
 1 lw r7,20(r29)
 1 lw r8,24(r29)
 1 lw r9,28(r29)
 1 lw r10,32(r29)
 1 lw r11,36(r29)
 1 lw r12,40(r29)
 1 lw r13,44(r29)
 1 lw r14,48(r29)
 1 lw r15,52(r29)
 1 lw r31,-8(r30)
 1 add r29,r0,r30
 1 jr r31
 1 lw r30,-4(r30)
```

Abbildung 1.3: Mit gcc übersetztes Assemblerprogramm für Quicksort. Branch-delay slots sind nicht gefüllt, um das Programm einfacher verstehen zu können. Alle anderen Optimierungen wurden eingeschaltet.

```
quicksort:
sw -4(r29),r30
add r30,r0,r29
sw -8(r29),r31
subui r29,r29,#464
sw 0(r29),r2
sw 4(r29),r3
sw 8(r29),r4
sw 12(r29),r5
sw 16(r29),r6
sw 20(r29),r7
sw 24(r29),r8
sw 28(r29),r9
sw 32(r29),r10
sw 36(r29),r11
sw 40(r29),r12
lw r11,(r30)
lw r9,4(r30)
slli r1,r11,#0x2
lhi r8,((_a+4)>>16)&0xffff
addui r8,r8,(_a+4)&0xffff
lhi r12,((_a)>>16)&0xffff
addui r12,r12,(_a)&0xffff
add r7,r1,r12
sgt r1,r11,r9
beqz r1,L8
addi r4,r30,#-408
add r10,r0,r4
L11:
addi r3,r8,#-4
L51:
add r2,r0,r7
lw r5,(r7)
L15:
addi r3,r3,#4
L49:
lw r1,(r3)
slt r1,r1,r5
bnez r1,L49
addi r3,r3,#4
addi r3,r3,#-4
addi r2,r2,#-4
L50:
lw r31,(r2)
slt r1,r5,r31
bnez r1,L50
addi r2,r2,#-4
addi r2,r2,#4
lw r6,(r3)
sw (r3),r31
sltu r1,r3,r2
bnez r1,L15
sw (r2),r6
lw r12,(r3)
sw (r2),r12
lw r12,(r7)
sw (r3),r12
sub r1,r7,r3
srai r5,r1,#0x2
sub r1,r3,r8
srai r2,r1,#0x2
sge r1,r5,r2
beqz r1,L24
sw (r7),r6
sgt r1,r2,r9
beqz r1,L25
addi r1,r3,#4
sw (r4),r1
addi r4,r4,#4
sw (r4),r7
addi r4,r4,#4
j L11
addi r7,r3,#-4
L25:
sgt r1,r5,r9
beqz r1,L34
addi r8,r3,#4
j L51
addi r3,r8,#-4
L24:
sgt r1,r5,r9
beqz r1,L32
addi r1,r3,#-4
sw (r4),r8
addi r4,r4,#4
sw (r4),r1
addi r4,r4,#4
j L11
addi r8,r3,#4
L32:
sgt r1,r2,r9
bnez r1,L11
addi r7,r3,#-4
L34:
seq r1,r4,r10
bnez r1,L8
addi r4,r4,#-4
lw r7,(r4)
addi r4,r4,#-4
j L11
lw r8,(r4)
L8:
slli r1,r11,#0x2
lhi r2,((_a)>>16)&0xffff
addui r2,r2,(_a)&0xffff
add r7,r1,r2
addi r3,r2,#8
sleu r1,r3,r7
beqz r1,L40
addi r4,r2,#4
L42:
lw r1,(r4)
lw r2,(r3)
sgt r1,r1,r2
beqz r1,L41
add r5,r0,r2
add r2,r0,r3
addi r31,r3,#-4
L44:
lw r12,(r31)
sw (r2),r12
addi r31,r31,#-4
lw r1,(r31)
sgt r1,r1,r5
bnez r1,L44
addi r2,r2,#-4
sw (r2),r5
L41:
addi r3,r3,#4
sleu r1,r3,r7
bnez r1,L42
addi r4,r4,#4
L40:
lw r2,0(r29)
lw r3,4(r29)
lw r4,8(r29)
lw r5,12(r29)
lw r6,16(r29)
lw r7,20(r29)
lw r8,24(r29)
lw r9,28(r29)
lw r10,32(r29)
lw r11,36(r29)
lw r12,40(r29)
lw r31,-8(r30)
add r29,r0,r30
jr r31
lw r30,-4(r30)
```

Abbildung 1.4: Mit gcc übersetztes Assemblerprogramm für eine optimierte Version von Quicksort. Branch-delay slots sind gefüllt. Die innere Schleife besteht jetzt nur noch aus 4 Anweisungen.

1.5 Wir haben festgestellt, daß $I' + I'' = 1$ gilt und damit beide dieser Zufallsvariablen genau 1 oder 0 sind. Wir mussten das Verhalten nicht genauer studieren, da beide Variablen gleich vielen Anweisungen zugeordnet wurden. Bei einer optimierenden Übersetzung kann es aber passieren, daß für I'' eine Maschineninstruktion mehr verwendet wird als für I'. In diesem Fall kommen die Variablen dann explizit in der Gesamtlaufzeit vor, weil sie sich nicht mehr wegkürzen.

Analysieren Sie den Erwartungswert von I'. Was würden Sie erwarten und was kommt tatsächlich heraus?

1.6 In Abbildung 1.3 finden Sie das Assemblerprogramm zu Quicksort. Zu jeder Anweisung ist angegeben, wie oft sie ausgeführt wird. Zählen Sie alles zusammen und stellen Sie eine Formel auf, die besagt *wieviele Assembleranweisungen* im Durchschnitt ausgeführt werden, um N Elemente zu sortieren.

1.7 Eine verbesserte Version von Quicksort, die Zeiger statt Arrayzugriffe über einen Index verwendet, ist noch schneller. In Abbildung 1.4 finden Sie das übersetzte DLX-Programm. Identifizieren Sie, welche Anweisungen C'- und C''-mal ausgeführt werden. Finden Sie anschließend heraus, wieviele Assembleranweisungen insgesamt ausgeführt werden. (Diese Aufgabe ist nicht leicht!)

1.8 Erweitern Sie das ursprüngliche C-Programm um Anweisungen, die explizit die Größen A, B, C, D, E und S zählen.

Lassen Sie dieses Programm genau einmal für jede Permutation von $(1, 2, \ldots, 10)$ laufen und bestimmen Sie so $A_{10}, \ldots, S_{10}$.

Vergleichen Sie die gezählten Werte mit den Werten, welche unsere Formeln liefern.

1.9 Schreiben Sie ein C-Programm für den *Mergesort*-Algorithmus und führen dann eine ebenso gründliche Analyse wie für *Quicksort* durch.

Kapitel 2

Das Kirchhoffsche Gesetz

Bei der Analyse von Quicksort haben wir bereits durch verschiedene Methoden die Anzahl der zu analysierenden Variablen stark eingeschränkt. Eine Technik, die wir im folgenden formal entwickeln, verwendet das einfache *Kirchhoffsche Gesetz* um die Anzahl der Variablen einzuschränken.

Wir gehen von einem gerichteten Graphen aus, dessen Knoten die ausführbaren Anweisungen eines Programms sind. Zwischen zwei Knoten ist genau dann eine Kante, wenn die zweite Anweisung direkt nach der ersten ausgeführt werden kann. Es können mehr als eine Kante aus einem Knoten herausführen, nämlich dann wenn es sich um eine bedingte Verzweigung handelt. Ebenso können mehrere Kanten in einen Knoten führen, da dieser Ziel vieler (Sprung-)Anweisungen sein kann.

Weiter nehmen wir an, es gibt einen Knoten, welcher mit START bezeichnet wird und einen der mit STOP bezeichnet ist. Der Programmablauf beginnt bei Start und endet bei STOP. Wir nehmen an, der Graph habe n Knoten (inklusive START und STOP) und m Kanten. Wir bezeichnen die Kanten mit e_i für $i = 1, \ldots, m$.

Aus Symmetriegründen fügen wir schließlich noch eine Kante e_0 von STOP nach START hinzu. Mit E_i bezeichnen wir die Anzahl, die ein Programmablauf die Kante e_i verwendet. Wir setzen $E_0 = 1$, als ob das Programm nach getaner Arbeit wieder an der Beginn des Programms zurückkehrt (und erst dann anhält). Wir haben nun die m verschiedenen Variablen E_i. Sie sind nicht alle unabhängig, sondern hängen von Gleichungen ab. Diese Gleichungen werden vom Kirchhoffschen Gesetz geliefert:

Theorem 1 (Kirchhoffsches Gesetz)
Sei I die Menge aller i, für welche die Kante E_i in den Knoten X hineinführt und

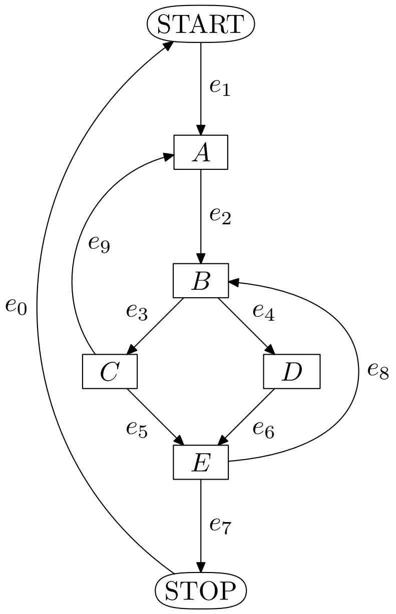

Abbildung 2.1: Beispiel für einen Programmflußgraphen mit und ohne Spannbaum.

O die Menge aller i für welche E_i aus X hinausführt. Dann gilt

$$\sum_{i \in I} E_i = \sum_{i \in O} E_i$$

und diese Zahl sagt gleichzeitig auch wie oft die Anweisung X ausgeführt wird.

Mit folgender Methode können wir eine Menge von unabhängigen Variablen E_i bestimmen, so daß wir die Werte aller anderen Variablen aus diesen berechnen können.

Zunächst bestimmen wir einen *Spannbaum* für den (ungerichteten) Graphen. Wir ignorieren dabei, daß die Kanten eine Richtung haben. Ein Spannbaum ist in Abbildung **??** durch dick gezeichnete Kanten hervorgehoben, er besteht aus den Kanten e_1, e_2, e_3, e_4, e_5 und e_7.

Fügen wir eine weitere Kante zu diesem Spannbaum hinzu, erhalten wir einen eindeutigen Zyklus. Wir bezeichnen diese Zyklen als *fundamentale Zyklen*. In unserem Beispiel erhalten wir für die Kanten e_0, e_6 und e_8 fundamentale Zyklen. Wenn wir die Kanten eines fundamentalen Zyklus aufzählen, geben wir ihnen dabei ein Vorzeichen: "+", wenn die Richtung dieser Kante dieselbe ist wie die Richtung der (einzigen) Kante im Zyklus, welche nicht zum Spannbaum gehört; ansonsten "−".

In unserem Beispiel haben wir also folgende fundamentale Zyklen:

$$\begin{aligned}
C_0 &= e_0 + e_1 + e_2 + e_3 + e_5 + e_7 \\
C_6 &= e_6 - e_5 - e_3 + e_4 \\
C_8 &= e_8 + e_3 + e_5 \\
C_9 &= e_9 + e_2 + e_3
\end{aligned}$$

Eine Tatsache, welche diese Definition interessant für die Analyse von Algorithmen macht, ist es, daß jeder fundamentale Zyklus eine Lösung der Kirchhoffschen Gleichungen liefert: Wir setzen alle $E_i = 0$, für welche e_i nicht Teil des fundamentalen Zyklus ist. Falls aber e_i zum fundamentalen Zyklus gehört, dann setzen wir $E_i = 1$ oder $E_i = -1$, je nach dem Vorzeichen von e_i im Zyklus.

In unserem Beispiel sind diese drei Lösungen folgende:

1. $E_0 = 1$, $E_1 = 1$, $E_2 = 1$, $E_3 = 1$, $E_4 = 0$, $E_5 = 1$, $E_6 = 0$, $E_7 = 1$, $E_8 = 0$, $E_9 = 0$

2. $E_0 = 0$, $E_1 = 0$, $E_2 = 0$, $E_3 = -1$, $E_4 = 1$, $E_5 = -1$, $E_6 = 1$, $E_7 = 0$, $E_8 = 0$, $E_9 = 0$

3. $E_0 = 0$, $E_1 = 0$, $E_2 = 0$, $E_3 = 1$, $E_4 = 0$, $E_5 = 1$, $E_6 = 0$, $E_7 = 0$, $E_8 = 1$, $E_9 = 0$

4. $E_0 = 0$, $E_1 = 0$, $E_2 = 1$, $E_3 = 1$, $E_4 = 0$, $E_5 = 0$, $E_6 = 0$, $E_7 = 0$, $E_8 = 0$, $E_9 = 1$

Da alle Gleichungen linear sind, sind auch alle Linearkombinationen dieser Lösungen wieder Lösungen der Kirchhoffschen Gleichungen. Der Übersichtlichkeit halber schreiben wir die Linearkombination in Vektorschreibweise:

$$\vec{E} = \begin{pmatrix} E_0 \\ E_1 \\ E_2 \\ E_3 \\ E_4 \\ E_5 \\ E_6 \\ E_7 \\ E_8 \\ E_9 \end{pmatrix} = \lambda_1 \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \lambda_2 \begin{pmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 1 \\ -1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \lambda_3 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \lambda_4 \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} =$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{pmatrix} \tag{2.1}$$

Für alle $\lambda_1, \lambda_2, \lambda_3, \lambda_4 \in \mathbf{R}$ erhalten wir eine Lösung der Kirchhoffschen Gleichungen und *jede Lösung läßt sich so darstellen*.

Wir sehen jetzt auch auf den ersten Blick, daß $E_0 = E_1 = E_7$ und $E_4 = E_6$ gilt, denn die Zeilen der Matrix sind dort jeweils gleich.

Wir können uns nun drei linear unabhängige E_i aussuchen und diese Größen analysieren (oder gar drei unabhängige Linearkombinationen verschiedenster E_i). Dann können wir alle weiteren E_i durch diese ausdrücken. Im Beispiel wählen wir natürlich erst einmal E_0, denn $E_0 = 1$ ist sowieso bekannt. Nun brauchen wir noch drei weitere; nehmen wir an, wir wählen $A = E_2$, $C = E_3$ und $D = E_4$. Wir erhalten in diesem Fall:

$$\begin{pmatrix} 1 \\ A \\ C \\ D \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{pmatrix}$$

Dies ist ein lineares Gleichungssystem, welches sich mit den üblichen Methoden lösen läßt. Hier erhalten wir als Ergebnis $\lambda_1 = 1$, $\lambda_2 = D$,

$\lambda_3 = -A + C + D$ und $\lambda_4 = A - 1$. Setzen wir dies in (2.1) ein, erhalten wir die Lösung der anderen E_i:

$$\vec{E} = \begin{pmatrix} E_0 \\ E_1 \\ E_2 \\ E_3 \\ E_4 \\ E_5 \\ E_6 \\ E_7 \\ E_8 \\ E_9 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ D \\ C + D - A \\ A - 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ A \\ C \\ D \\ 1 + C - A \\ D \\ 1 \\ C + D - A \\ A - 1 \end{pmatrix}$$

All diese Rechnungen lassen sich selbstverständlich durch Computeralgebra (Mathematica, Maple oder Macsyma) durchführen. Im allgemeinen sind die Matrizen natürlich groß.

Bei unserer Wahl von A, C und D haben wir uns darauf festgelegt diese Größen zu analysieren. Welche man hier am besten nimmt hängt natürlich vom speziellen Problem ab. Es verbleibt nun noch C und E: Wir erhalten $B = E_3 + E_4 = C + D$ und $E = E_5 + E_6 = 1 + C + D - A$.

Eine ausführlichere Darstellung dieser Technik findet sich bei Knuth [1, Section 2.3.4.1].

Aufgaben

2.1 Falls ein Flußdiagramm genau n Knoten und m Kanten enthält, wieviele fundamentale Zyklen werden wir dann erhalten?

2.2 Beweisen oder widerlegen Sie: In jedem Flußdiagramm läßt sich ein Spannbaum so bestimmen, daß alle fundamentalen Zyklen nur positiv orientierte Kanten enthalten (d.h. man kann den Zyklus durchlaufen und dabei nie eine Kante in Gegenrichtung. . .

2.3 In dieser Aufgabe betrachten wir den Algorithmus von Prim, mit dessen Hilfe minimale Spannbäume berechnet werden können. Eingabe sind ein ungerichteter Graph $G = (V, E)$, eine Kantengewichtsfunktion $w : E \to \mathbf{R}$ und ein Ausgangsknoten r.

```
for each u ∈ V do
    key[u] ← ∞
    π[u] ← NIL
key[r] ← 0
M ← V
while (M ≠ ∅) do
    u ← min-from(M)
    for each v ∈ neighbors(u) do
        if (v ∈ M) ∧ (w(u, v) < key[v]) then
            π[v] ← u
            key[v] ← w(u, v)
```

Malen Sie ein Flußdiagramm für diesen Algorithmen, der alle grundlegenden Blöcke enthält. Bestimmen Sie einen Spannbaum und die dazugehörigen fundamentalen Zyklen. Wählen Sie eine minimale Menge von Blöcken aus, deren Laufzeit analysiert werden muß, und geben Sie an, wie sich dann die anderen Größen daraus ergeben.

Kapitel 3

Rekursionsgleichungen

Bei der Analyse eines Algorithmus versucht man eine mathematische Formel zu finden, welche die Größe, die man analysiert beschreibt. Oft kann man nicht direkt eine geschlossene Formel für diese Größe angeben, sondern nur eine Formel, welche die Größe implizit beschreibt. Bei der Analyse von Quicksort, zum Beispiel, konnten wir die uns interessierenden Größen X_N stets ausdrücken durch

$$X_N = \frac{2}{N} \sum_{k=0}^{N-1} X_k + f_N.$$

Eine Gleichung, in welcher neben X_N auch X_k mit $k < N$ vorkommt, nennen wir eine *Rekursionsgleichung*.

Eine Rekursionsgleichung *lösen* bedeutet eine geschlossene Formel für X_N zu finden.

Im allgemeinen muß es keine geschlossene Formel für die Lösung geben und für viele Rekursionsgleichung trifft dies auch zu—allerdings lassen sich jene Rekursionsgleichungen, welche bei der Analyse von Algorithmen auftreten, oft lösen oder es läßt sich wenigstens eine sehr gute Näherung für die Lösung finden. Es gibt auch einige Klassen von Rekursionsgleichungen, welche sich stets durch ein festes, einfaches Verfahren lösen lassen.

3.1 Klassifikation von Rekursionsgleichungen

Die allgemeinste Rekursionsgleichung, welche wir betrachten wollen, hat die Form

$$a_n = f(a_{n-1}, a_{n-2}, \ldots, a_0) \text{ für } n \geq t. \tag{3.1}$$

Wir betrachten a_n nur für $n \geq 0$ und vereinbaren $a_{-1} = a_{-2} = a_{-3} = \cdots = 0$.

Da (3.1) erst für $n \geq t$ gilt, können wir a_n berechnen, wenn $a_0, a_1, \ldots, a_{t-1}$ vorgegeben sind. Je nach Vorgabe wird die Lösung der Rekursionsgleichung im allgemeinen verschieden sein. Stammt die Rekursionsgleichung aus der Analyse eines Algorithmus, dann sind die Werte von $a_0, a_1, \ldots, a_{t-1}$ normalerweise durch den Algorithmus vorgegeben.

Die Rekursionsgleichung für die Anzahl der Vergleiche während des Partitionierens im Quicksort-Algorithmus lautete

$$C_N = N + 1 + \frac{2}{N}\sum_{k=0}^{N-1} C_k \text{ für } N > M$$

mit den Anfangsbedingungen $C_0 = C_1 = C_2 = \ldots = C_M = 0$. Diese Anfangsbedingungen sind leicht dem Algorithmus zu entnehmen: Wenn $N \leq M$ ist, wird ja gar nicht partitioniert.

Wir klassifizieren Rekursionsgleichungen folgendermaßen:

$$\begin{aligned}
a_n &= f(a_{n-1}, a_{n-2}, \ldots, a_{n-t}) \text{ RG } t\text{er Ordnung} \\
a_n &= \sum_{k=0}^{n-1} x(k, n) \text{ homogene, lineare RG} \\
a_n &= \sum_{k=0}^{n-1} x(k, n) + f(n) \text{ lineare RG} \\
a_n &= x_1 a_{n-1} + x_2 a_{n-2} + \cdots + x_t a_{n-t} \text{ linear mit konstanten Koeffizienten}
\end{aligned}$$

In diesem Kapitel folgen nun verschiedenste Verfahren, um Rekursionsgleichungen, wie sie auftreten, zu lösen.

3.2 Eine Tabelle erstellen

Ein meist nützlicher erster Schritt besteht darin, einige Werte zu berechnen und so eine kurze Tabelle anzulegen. Betrachten wir zum Beispiel die Rekursionsgleichung

$$a_n = a_{n-1} + 2a_{n-2} \text{ für } n > 1 \text{ und } a_0 = a_1 = 1.$$

Wir rechnen $a_2 = a_1 + 2a_0 = 3$, $a_3 = a_2 + 2a_1 = 5$, $a_4 = a_3 + 2a_2 = 11$, $a_5 = 21$, $a_6 = 43$. Die Tabelle sieht nun so aus:

n	0	1	2	3	4	5	6
a_n	1	1	3	4	11	21	43

Durch die Tabelle erhalten wir einen ersten Eindruck von der Lösung und wir können sie später verwenden, um unsere geschlossene Lösung an einigen Werten zu überprüfen. Stimmen sie mit der Tabelle überein können wir zuversichtlicher sein, bei der Aufstellung der geschlossenen Formel keinen Fehler gemacht zu haben.

3.3 Die Lösung raten und mit Induktion beweisen

Mit Hilfe einiger kleiner Werte können wir versuchen, eine geschlossene Formel zu erraten. Aus obiger Tabelle für a_n beispielsweise erkennen wir schnell, daß sich die Zahlen ungefähr verdoppeln; allerdings sind sie niemals genau doppelt so groß. Sie scheinen manchmal um eins größer zu sein als das Doppelte und manchmal um eins kleiner. Bei genauerem Hinsehen erkennen wir, daß sie abwechseln zu groß und zu klein sind. Betrachten wir also die Summen aufeinanderfolgender Zahlen; dies könnte eine gute Idee sein, da sich ja dann die Differenzen gerade ausgleichen. Wir erhalten: 2, 4, 8, 16, 32, 64.

Die Lösung sollte also nah bei $2^{n+1}/3$ liegen. Betrachten wir also einmal $2^{n+1}/3$:

n	0	1	2	3	4	5	6
$\frac{1}{3}2^{n+1}$	$\frac{2}{3}$	$1\frac{1}{3}$	$2\frac{2}{3}$	$5\frac{1}{3}$	$10\frac{2}{3}$	$21\frac{1}{3}$	$42\frac{2}{3}$

Tatsächlich, es scheint zu stimmen, aber die Werte sind abwechselnd um $\frac{1}{3}$ zu klein und zu groß. Unsere Vermutung für eine geschlossene Form ist daher:

$$a_n = \frac{1}{3}2^{n+1} + \frac{1}{3}(-1)^n$$

Überprüfen wir sie für einige Werte. Für $n = 0$ erhalten wir $\frac{1}{3}2 + \frac{1}{3}(-1)^0 = 1$, für $n = 1$ erhalten wir $\frac{1}{3}4 + \frac{1}{3}(-1)^1 = 1$ und schließlich für $n = 5$ ergibt sich $\frac{1}{3}64 + \frac{1}{3}(-1)^5 = \frac{63}{3} = 21$.

Wir können mit unserer Vermutung zufrieden sein, aber wir müssen doch noch beweisen, daß sie stimmt. Dafür bietet sich im allgemeinen vollständige Induktion an. Wir haben bereits gezeigt, daß unsere geschlossene Form für $n = 0$ und $n = 1$ stimmt. Nun sei $n > 1$.

Wir erhalten aus der Induktionsannahme

$$\begin{aligned} a_n = a_{n-1} + 2a_{n-2} &= \frac{1}{3}2^n + \frac{1}{3}(-1)^{n-1} + \frac{2}{3}2^{n-1} + \frac{2}{3}(-1)^{n-2} \\ &= \frac{1}{3}2^{n+1} + \frac{1}{3}(-1)^{n-2} \end{aligned}$$

und dies stimmt wieder mit unserer geschlossenen Form für a_n überein, denn $(-1)^{n-2} = (-1)^n$. Wir haben also ohne Zweifel bewiesen, daß wirklich $a_n = \frac{1}{3}2^{n+1} + \frac{1}{3}(-1)^n$ gilt.

3.4 Nachschlagen

Es gibt ein kurioses Buch, welches die meisten bekannten Folgen in lexikographischer Reihenfolge enthält [2]. Natürlich ist nur der Anfang jeder Folge aufgeführt zusammen mit einer kurzen Beschreibung und Angaben, wo über diese Folge etwas geschrieben wurde. Unsere Folge $1, 1, 3, 5, 11, 21, 43, \ldots$ findet sich tatsächlich in diesem Buch. Sie hat dort die Bezeichnung “$A(N) = A(N-1) + A(N-2)$” und es sind zwei Papiere in den Zeitschriften *Eureka, the Journal of the Archimedeans (Cambridge University Mathematical Society* und *Nouvelles Correspondance Mathématique* enthalten. Dort läßt sich dann näheres über die Folge finden.

Inzwischen gibt es auch eine modernere und vor allem bequemere Methode über das WWW unter der URL

`http://www.research.att.com/njas/sequences/index.html`

zu finden. Dort läßt sich der Anfang einer Folge eingeben und die Antwort auf die Eingabe $1, 1, 3, 5, 11, 21, 43$ sieht so aus wie in Abbildung 3.1. Unsere Folge hat also einen Namen: *Jacobstahl sequence*. Außerdem erhalten wir dort drei weitere Literaturangaben und vor allem eine geschlossene Form für den Term a_n.

Greetings from the On-Line Encyclopedia of Integer Sequences!

Abbreviations used | To identify a reference | To identify a sequence from its A number | Return to previous page

Please wait a moment while I search the data-base.

```
Matches (up to a limit of 10) found for  1 1 3 5 11 21 43  :

----------------------

%I A001045 M2482 N0983

%S A001045 0,1,1,3,5,11,21,43,85,171,341,683,1365,2731,5461,10923,21845,43691,

%T A001045 87381,174763,349525,699051,1398101,2796203,5592405,11184811,22369621

%N A001045 Jacobstahl sequence: a(n) = a(n-1) + 2a(n-2).

%R A001045 FQ 10 499 72; 26 306 88. JCT A26 149 79. AMM 102 705 95. TYCM 28 76 9

%O A001045 0,4

%A A001045 njas

%K A001045 nonn

%F A001045 a(n) = 2^n - a(n-1). a(n) = 2*a(n-1) - (-1)^n = {2^n - (-1)^n}/3.
```

Abbreviations used | To identify a reference | To identify a sequence from its A number | Return to previous page

See also: My home page | Integer Sequences home page | The book: The Encyclopedia of Integer Sequences | The full bibliography

[Home | People | Library | Demos | History | News | TOC]

Tabelle 3.1: Antwort der *On-Line Encyclopedia of Integer Sequences* auf die Eingabe 1,1,3,5,11,21,43.

3.5 Mathematica, Maple & Co

Viele algebraische Formelmanipulationssystem können einfache Rekursionsgleichungen direkt lösen. Bei Mathematica heißt die entsprechende Funktion `RSolve`. Unser Beispiel ist für Mathematica kein Problem:

```
RSolve[{a[n] == a[n - 1] + 2 a[n - 2],
  a[0] == a[1] == 1}, a[n],
 n]
```

$$\left\{\left\{a(n) \to \frac{(-1)^n}{3} + \frac{2^{n+1}}{3}\right\}\right\}$$

Mathematica kommt also auf dieselbe Lösung wie wir. Maple hat ebenfalls keine Probleme:

```
> rsolve({a(n) = a(n-1)+2*a(n-2), a(0..1)=1}, a(n));
                                            n        n
                                 1/3 (-1)   + 2/3 2

>
```

3.6 Versteckte Produkte und Summen

Die einfachsten Rekursionsgleichungen sind von der Form

$$a_n = x_n a_{n-1} \text{ und } b_n = b_{n-1} + y_n.$$

Beide Formen sind eng verwandt. Substituiert man $\bar{a}_n = \log(a_n)$ dann erhält man aus der linken Rekursionsgleichung eine Rekursionsgleichung des rechten Typs. Wir können die beiden Typen *verstecktes Produkt* und *versteckte Summe* nennen.

Der linke Seite läßt sich durch fortgesetztes Einsetzen lösen, ein Verfahren welches zu einem Produkt führt:

$$a_n = x_n a_{n-1} = x_n x_{n-1} a_{n-2} = \cdots = x_n x_{n-1} x_{n-2} x_{n-3} \cdots x_2 x_1 a_0 = a_0 \prod_{k=1}^{n} x_n$$

Ebenso führt fortgesetztes Einsetzen in die Rekursionsgleichung auf der rechten Seite zu einer Summe:

Theorem 2 *Die Lösung der Rekursionsgleichungen*

$$a_n = x_n a_{n-1} \qquad \text{und} \qquad b_n = b_{n-1} + y_n$$

sind

$$a_n = a_0 \prod_{k=1}^{n} x_n \qquad \text{und} \qquad b_n = b_0 + \sum_{k=1}^{n} y_n.$$

3.7 Lineare Rekursionsgleichungen mit konstanten Faktoren

Eine besonders einfache Form ist die homogene, lineare Rekursionsgleichung mit konstanten Faktoren. Ihre allgemeine Form ist

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_t a_{n-t} \text{ für } n \geq t \tag{3.2}$$

Es handelt sich hier um eine Rekursionsgleichung $t$ter Ordnung. Die $c_i \in \mathbf{R}$ sind die Koeffizienten der Gleichung und sie hängen nicht von n ab (daher *konstante* Koeffizienten).

Lineare Rekursionsgleichungen mit konstanten Koeffizienten lassen sich immer lösen und, mehr noch, mit Hilfe eines festen Schemas. Im folgenden wollen wir eine allgemeine Lösungsmethode entwickeln.

Nehmen wir zunächst einmal an, daß es eine Lösung gibt, welche die Form $a_n = \alpha^n$ hat, wobei $\alpha \in \mathbf{C}$ eine komplexe Zahl sein darf. Setzen wir dies in die RG ein und setzen $n = t$, erhalten wir

$$\alpha^t = c_1 \alpha^{t-1} + c_2 \alpha^{t-2} + \cdots + c_{t-1}\alpha + c_t$$

und das bedeutet natürlich, daß α eine Nullstelle des *charakteristischen Polynoms*

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1} z - c_t$$

sein muß. Umgekehrt ist aber auch $a_n = \alpha^n$ eine Lösung von (3.2), wenn α eine Nullstelle obigen Polynoms ist.

Wenn α eine k-fache Nullstelle von χ ist, dann ist auch $a_n = n^j \alpha^n$ für $0 \leq j < k$ eine Lösung der Rekursionsgleichung. Wir überprüfen dies durch Einsetzen:

$$n^j \alpha^n = \sum_{r=1}^{t} c_r (n-r)^j \alpha^{n-r},$$

was gleichbedeutend mit

$$n^j \alpha^t - \sum_{r=1}^{t} c_r (n-r)^j \alpha^{t-r} = 0.$$

Die linke Seite dieser Gleichung ist aber eine Linearkombination von $\chi(\alpha)$, $\chi'(\alpha)$, $\chi''(\alpha)$, $\ldots$, $\chi^{(j)}(\alpha)$. Die ersten k Ableitungen von χ sind an der Stelle α natürlich 0, da ja α eine k-fache Nullstelle von χ ist.

Theorem 3 *Die homogene, lineare Rekursionsgleichung mit konstanten Koeffizienten*

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_t a_{n-t} \text{ für } n \geq t$$

hat die Lösungen $a_n = n^j \alpha^n$, für alle Nullstellen α des charakteristischen Polynoms

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1} z - c_t,$$

und für alle $j = 0, 1, \ldots, k-1$, wobei k die Vielfachheit der Nullstelle α ist. Diese Lösungen sind alle linear unabhängig. Sie bilden eine Basis des Vektorraums aller Lösungen.

Natürlich sind auch Vielfache einer Lösung und Summen von Lösungen wieder Lösungen der RG, denn sie ist ja linear. Wir erhalten so insgesamt genau t linear unabhängige Lösungen, welche eine Basis des Vektorraums aller Lösungen bilden.

Haben wir nun t Anfangsbedingungen, beispielsweise die Werte $a_0, a_1, \ldots, a_{t-1}$, erhalten wir durch Linearkombination genau eine Lösung. Um die richtige Linearkombination zu finden, muß lediglich ein lineares Gleichungssystem gelöst werden.

Nehmen wir als Beispiel wieder

$$a_n = a_{n-1} + 2a_{n-2} \text{ für } n > 1 \text{ und } a_0 = a_1 = 1.$$

Das charakteristische Polynom ist $q(z) = z^2 - z - 2$. Man sieht sofort, daß -1 eine Nullstelle ist. Dividieren wir $q(z)$ durch $z + 1$ erhalten wir $z - 2$ und die zweite Nullstelle ist 2. Die Lösungen sind also von der Form

$$a_n = \lambda 2^n + \mu(-1)^n.$$

Um die Konstanten zu bestimmen, verwenden wir jetzt die Anfangsbedingungen und setzen ein. Wir erhalten $1 = \lambda + \mu$ und $1 = 2\lambda - \mu$. Daraus ergibt sich $\lambda = \frac{1}{3}$ und $\mu = \frac{2}{3}$.

3.8 Summationsfaktor

Eine lineare Rekursionsgleichung erster Ordnung läßt sich stets auf eine Summe zurückführen. Wir haben diese Technik bei der Lösung der Quicksort-Rekursionsgleichung verwendet. Nachdem wir die Ordnung auf 1 brachten, hatte die RG folgende Form:

$$NX_N = (N+1)X_{N-1} + Nf_N - (N-1)f_{N-1}, \text{ für } N > M+1$$

Wir multiplizierten diese Gleichung mit $1/N(N+1)$ und erhielten nach Substitution eine Gleichung der Form

$$Y_N = Y_{N-1} + g_N.$$

Im folgenden werden wir sehen, wie wir dies mit allen linearen Rekursionsgleichungen anstellen können.

Theorem 4 (Summationsfaktor)
Die lineare Rekursionsgleichung erster Ordnung

$$a_n = x_n a_{n-1} + y_n \textit{für } n > 0$$

und $a_0 = 0$ *hat die Lösung*

$$a_n = y_n + \sum_{j=1}^{n-1} y_j x_{j+1} x_{j+2} \dots x_n.$$

Wir beweisen diesen Satz, indem wir die Rekursionsgleichung zunächst durch $x_n x_{n-1} x_{n-2} \dots x_1$ teilen, was zu

$$\frac{a_n}{x_n x_{n-1} x_{n-2} \dots x_1} = \frac{a_{n-1}}{x_{n-1} x_{n-2} x_{n-3} \dots x_1} + \frac{y_n}{x_n x_{n-1} x_{n-2} \dots x_1}$$

führt. Substituieren wir $b_n = a_n/(x_n x_{n-1} x_{n-2} \dots x_1)$ erhalten wir die einfache Rekursionsgleichung

$$b_n = b_{n-1} + \frac{y_n}{x_n x_{n-1} x_{n-2} \dots x_1},$$

welche wir leicht durch Summieren lösen können. Dabei nennen wir das Produkt $x_n x_{n-1} x_{n-2} \dots x_1$ einen *Summationsfaktor.* Er gibt der Methode den Namen. Oft ist der Summationsfaktor sehr einfach, da sich im Produkt Terme kürzen.

Versuchen wir als Beispiel die Rekursionsgleichung

$$a_n = 2a_{n-1} + n \text{ für } n > 0$$

und $a_0 = 0$ zu lösen. Hier ist also $x_n = 2$ und $y_n = n$. Die Lösung ist also

$$a_n = n + \sum_{j=1}^{n-1} j \cdot 2^{n-j} = 2^{n+1} - 2 - n.$$

Mathematica schafft diese einfache Rekursionsgleichung aber auch noch:

```
In[4]:= RSolve[{a[0]==0, a[n]==2a[n-1]+n}, a[n], n]
                              1 + n
Out[4]= {{a[n] -> -2 + 2           - n}}
```

3.9 Repertoire

Die Repertoire-Methode ist hauptsächlich für lineare Rekursionsgleichungen geeignet. Sie zeigt wieder einmal, daß Lösen von Rekursionsgleichungen mehr eine Kunst als eine Wissenschaft ist. Die Repertoire-Methode verlangt viel Intuition. Bei der Analyse von Algorithmen wissen wir glücklicherweise meist, wie sich die Lösung in etwa verhalten muß. Im allgemeinen gehen wir bei der Repertoire-Methode von einer Rekursionsgleichung der Form

$$a_n = x_{1,n}a_{n-1} + x_{2,n}a_{n-2} + x_{3,n}a_{n-3} + \ldots + x_{t,n}a_{n-t} + f_n$$

aus. Wir stellen uns nun vor, die Lösung der Gleichung hat eine bestimmte Form und berechnen für welches f_n diese Lösung herauskommen würde. Dies machen wir für viele verschiedene potentielle Lösungen und erhalten jedesmal ein anderes f_n. Da Linearkombinationen der Lösungen wieder Lösungen einer RG sind, für welche das f_n die entsprechende Linearkombination der entsprechenden f_n's ist, haben wir eine Lösung der ursprünglichen Gleichung, wenn wir das ursprüngliche f_n als Linearkombination der f_n's unseres Repertoires darstellen können. Als Beispiel soll die Rekursionsgleichung von Quicksort dienen (mit $M = 0$):

$$a_n = f_n + \frac{2}{n} \sum_{k=0}^{n-1} a_k$$

Wählen wir als Lösung $a_n = 1$ erhalten wir

$$f_n = a_n - \frac{2}{n}\sum_{k=0}^{n-1} a_k = 1 - \frac{2}{n}\sum_{k=0}^{n-1} 1 = -1.$$

Die Kunst besteht nun darin, geeignete potentielle Lösungen auszuwählen, um anschließend das gewünschte f_n als Linearkombination darstellen zu können.

Für die Quicksort-Gleichung können wir folgendes Repertoire wählen:

a_n	$f_n = a_n - \frac{2}{n}\sum_{k=0}^{n-1} a_k$	a_0
1	-1	1
H_n	$2 - H_n$	0
nH_n	$\frac{1}{2}(n-1) + H_n$	0

Nehmen wir an, wir wollten die Anzahl der Vergleiche analysieren. Dort war $f_n = n + 1$. Für dieses f_n haben wir keine Lösung in unserem Repertoire. Allerdings können wir $n + 1$ als Linearkombination der f_n's unseres Repertoires darstellen:

$$n + 1 = 2\left(\frac{1}{2}(n-1) + H_n\right) + 2\,(2 - H_n) + 2\,(-1)$$

Eine Lösung der Rekursionsgleichung mit $f_n = n + 1$ ist daher

$$a_n = 2\,(nH_n) + 2\,(H_n) + 2\,(1) = 2(n+1)H_n + 2.$$

Die Anfangsbedingung $a_0 = 0$ ist hier aber nicht erfüllt, sondern hier ist $a_0 = 2$. Wir stocken das Repertoire also noch auf:

a_n	$f_n = a_n - \frac{2}{n}\sum_{k=0}^{n-1} a_k$	a_0
1	-1	1
H_n	$2 - H_n$	0
nH_n	$\frac{1}{2}(n-1) + H_n$	0
n	1	0

Wir können jetzt $n+1$ als Linearkombination von $2 - H_n$, $\frac{1}{2}(n-1) + H_n$ und 1 darstellen und erhalten dann eine Lösung mit der richtigen Anfangsbedingung, da in allen Fällen $a_0 = 0$ gilt:

$$n + 1 = 2\left(\frac{1}{2}(n-1) + H_n\right) + 2\,(2 - H_n) - 2\,(1)$$

und die Lösung ist

$$2(nH_n) + 2(H_n) - 2(n) = 2nH_n + 2H_n - 2n$$

Das allgemeine Vorgehen bei der Repertoire-Methode ist also folgendes: Gegeben ist eine Rekursionsgleichung der Form

$$a_n = \sum_{i=1}^{t} x_{i,n} a_{n-t} + f(n)$$

Die Koeffizienten $x_{i,n}$ können von n abhängen.

1. Schritt: Man wählt ein Repertoire $b_n, c_n, d_n, \ldots$ von Folgen und berechnet jeweils $f_b(n), f_c(n), \ldots$, wobei $f_b(n) = b_n - \sum_{i=1}^{t} x_{i,n} b_{n-t}$ ist. Damit ist b_n eine geschlossene Lösung der Rekursionsgleichung

$$b_n = \sum_{i=1}^{t} x_{i,n} b_{n-1} + f_b(n) \text{ für } n \geq t$$

2. Schritt: Kann man nun $f(n)$ als Linearkombination von $f_b(n), f_c(n), \ldots$ darstellen, also als

$$f(n) = \beta f_b(n) + \gamma f_c(n) + \delta f_d(n) + \cdots$$

dann gilt

$$a_n = \beta b_n + \gamma c_n + \delta d_n + \cdots$$

und man hat eine Lösung der Rekursionsgleichung mit bestimmten Anfangsbedingungen.

3. Schritt: Wünscht man eine Lösung für a_n für andere Anfangsbedingungen, dann muß man eine andere Linearkombination versuchen. Das Repertoire muß groß genug sein, um soviele linear unabhängige Lösungen für a_n zu liefern, daß die richtigen Anfangsbedingungen durch eine Linearkombination der Lösungen dargestellt werden kann.

3.10 Reduktion der Ordnung

In manchen Fällen läßt sich eine Gleichung höherer Ordnung auf eine Gleichung niedrigerer Ordnung reduzieren. Wir definieren hierzu den sogenannten *shift operator E*, welcher Folgen auf Folgen abbildet. Er ist definiert vermöge

$$Ef_n = f_{n+1},$$

verschiebt also alle Folgenglieder um 1 nach oben. Der Ausdruck Ef_n läßt sich aber auf zwei verschiedene Weisen interpretieren: Fassen wir f_n als Folge auf, dann ist $Ef_n = f_{n+1}$ einfach die verschobene Folge. Wir können aber f_n auch als einen Operator auffassen, welcher angewandt auf eine Folge g_n die Folge $f_n g_n$ liefert. Dann ist $Ef_n = f_{n+1}E$.

Das Rechnen mit linearen Operatoren ist anfangs ungewohnt. Es gilt zwar das Assoziativgesetz (z.B. ist $E(f_n g_n) = (Ef_n)g_n$ und man muß sich nicht um Klammerungen kümmern, aber es gibt leider kein Kommutativgesetz, z.B. ist $En = (n+1)E$ (n wird hier als Folge aufgefaßt, deren $n$tes Glied n ist). Ähnlich ist $En^2 = (n^2 - 2n + 1)E$.

Eine lineare Rekursionsgleichung $t$ter Ordnung läßt sich stets schreiben als

$$p(E)a_n = f(n),$$

wobei p ein Polynom $t$ten Grades ist (dessen Koeffizienten selbst Folgen sind), denn

$$a_n = x_{1,n}a_{n-1} + x_{2,n}a_{n-2} + \cdots + x_{t,n}a_{n-t} + f(n)$$

ist äquivalent zu

$$(E^t - x_{1,n}E^{t-1} - x_{2,n}E^{t-2} - x_{3,n}E^{t-3} - \cdots - x_{t,n}E^0)a_{n-t} = f(n).$$

Es geht nicht immer, aber manchmal kann man $p(E) = q(E)r(E)$ schreiben, das heißt, $p(E)$ *faktorisieren*. Dann müssen wir $q(E)r(E)a_n = f(n)$ lösen. Wir können jetzt die Substitution $b_n = r(E)a_n$ verwenden und lösen zunächst

$$q(E)b_n = f(n),$$

eine Rekursionsgleichung niedrigerer Ordnung. Auf diese Weise erhalten wir eine geschlossene Form für b_n. Anschliessend lösen wir noch

$$r(E)a_n = b_n,$$

welches wieder eine Rekursionsgleichung ist und erhalten so schließlich eine Lösung für a_n.

Allgemein geht man nach folgendem Kochrezept vor:

Man schreibt eine lineare Rekursionsgleichung in Operatorform auf als

$$p(E)a_n = f(n)$$

und findet eine Faktorisierung $p(E) = q(E)r(E)$. Man löst jetzt nacheinander

$$q(E)b_n = f(n) \text{ und } r(E)a_n = b_n.$$

Die RG wird so auf zwei RG mit kleinerer Ordnung zurückgeführt.

Betrachten wir als Beispiel die Rekursionsgleichung

$$a_{n+2} - (n+2)a_{n+1} + na_n = n. \tag{3.3}$$

In Operator-Schreibweise sieht sie so aus:

$$(E^2 - (n+2)E + n)a_n = n$$

Wir können dieses Polynom faktorisieren, denn $(E-1)(E-n) = E^2-(n+2)E+n$. Hierbei ist zu beachten, daß $En = (n+1)E$ ist. Die Rekursionsgleichung hat jetzt die Gestalt

$$(E-1)(E-n)a_n = n$$

und wir lösen zunächst $(E-1)b_n = n$ oder, äquivalent, $b_{n+1} = b_n + n$. Dies ist eine versteckte Summe; wir erhalten aus Theorem 2 sogleich die Lösung

$$b_n = \sum_{k=0}^{n}(n-1) = \frac{n(n-1)}{2} - 1.$$

Nun müssen wir noch $(E-n)a_n = b_n = n(n-1)/2 - 1$ lösen, eine Rekursionsgleichung, welche sich auch als

$$a_{n+1} = na_n + \frac{n(n-1)}{2} - 1 \tag{3.4}$$

schreiben läßt.

3.11 Gewinnung von Rekursionsgleichungen aus Algorithmen

Betrachten wir folgende Schleife:

while $i \leq j$ **do**
  $i \leftarrow i+1;\ j \leftarrow j-i$
od

Wie oft wird diese Schleife durchlaufen? Diese Anzahl hängt offensichtlich davon ab, wie die Variablen i und j zu Beginn belegt sind. Nennen wir diese Belegung i_0, j_0 und bezeichnen wir weiter mit i_n, j_n die Belegung der Variablen i und j *nach dem nten Durchlauf* der Schleife.

Machen wir zunächst wieder eine kleine Tabelle für eine Beispiel. Wählen wir dazu $i_0 = -3$ und $j_0 = 10$.

n	0	1	2	3	4	5	6	7
i_n	−3	−2	−1	0	1	2	3	4
j_n	10	12	13	13	12	10	7	3

Da $i_7 > j_7$ ist, wird die Schleife kein 8tes Mal durchlaufen.

Es ist leicht Rekursionsgleichungen für i_n und j_n anzugeben:

$$i_n = i_{n-1} + 1 \text{ und } j_n = j_{n-1} - i_{n-1} - 1$$

Diese Gleichungen sind verschränkt, aber nur die zweite mit der ersten. Die Rekursionsgleichung für i_n läßt sich sofort lösen, das Ergebnis ist $i_n = i_0 + n$, und in die zweite einsetzen:

$$j_n = j_{n-1} - (i_0 + n - 1) - 1 = j_{n-1} - i_0 - n$$

Dies ist wieder eine versteckte Summe und die Lösung ist daher

$$j_n = j_0 - \sum_{k=1}^{n} (i_0 + k) = j_0 - ni_0 - \frac{n(n+1)}{2}.$$

Die Schleife wird durchlaufen, solange $i \leq j$ gilt. Der $(n+1)$te Durchlauf findet also statt, wenn der $n$te Durchlauf stattfindet und $j_n - i_n \geq 0$ gilt. Sollte $i_0 > j_0$ sein, dann wird die Schleife natürlich überhaupt nicht durchlaufen.

Sehen wir uns die Bedingung $j_n - i_n \geq 0$ näher an:

$$j_n - i_n = -\frac{n(n+1)}{2} - ni_0 + j_0 - i_0 - n = -\frac{1}{2}n^2 - \frac{1}{2}(3 + 2i_0)n + j_0 - i_0 \geq 0$$

Multiplizieren wir die Ungleichung mit −2 erhalten wir die ein wenig schönere Ungleichung

$$n^2 + (3 + 2i_0)n - 2(j_0 - i_0) \leq 0. \tag{3.5}$$

Die Gleichung $x^2 + (3 + 2i_0)x - 2(j_0 - i_0) = y$ beschreibt eine nach oben offene Parabel. Da wir annehmen können, daß $n = 0$ eine Lösung von (3.5) ist (sonst wird die Schleife nie durchlaufen) hat diese Parabel reelle Nullstellen. Die Ungleichung (3.5) ist also für alle n von 0 bis zur rechten Nullstelle erfüllt. Diese Nullstelle ist

$$\zeta = -i_0 - \frac{3}{2} + \frac{1}{2}\sqrt{4i_0(i_0 + 1) + 8j_0 + 9}.$$

Das heißt, (3.5) ist erfüllt für $0 \le n \le \zeta$, was gleichbedeutend mit $0 \le n \le \lfloor\zeta\rfloor$ ist. Die Schleife wird also insgesamt $\lfloor\zeta\rfloor + 1$ mal durchlaufen. Diese Zahl ist explizit

$$\left\lfloor \frac{1}{2}\sqrt{4i_0(i_0+1)+8j_0+9} - i_0 - \frac{1}{2} \right\rfloor. \tag{3.6}$$

Es kann nichts schaden, diese explizite Formel für die Anzahl der Schleifendurchläufe an einem Beispiel zu testen. Nehmen wir wieder $i_0 = -3$ und $j_0 = 10$. Wenn wir diese Werte in (3.6) einsetzen erhalten wir

$$\left\lfloor \frac{1}{2}\sqrt{4(-3)(-2)+8\cdot 10+9} + 3 - \frac{1}{2} \right\rfloor = \left\lfloor \frac{1}{2}\sqrt{117} + \frac{5}{2} \right\rfloor = \lfloor 7.91 \rfloor = 7.$$

Es stimmt.

Versuchen wir ein schwierigeres Problem:

for $k = 1$ **to** m
  $i \leftarrow k;\ j \leftarrow k^2;$
  while $i \le j$ **do**
    $i \leftarrow i+1;\ j \leftarrow j-i$
  od
od

Wie oft wird jetzt die innere Schleife durchlaufen?

Bezeichnen wir mit $W(i_0, j_0)$ die Anzahl der Schleifendurchläufe, wenn anfangs $i = i_0$ und $j = j_0$ gilt. Wir haben $W(i_0, j_0)$ oben bestimmt: $W(i_0, j_0) = \lfloor \frac{1}{2}\sqrt{4i_0(i_0+1)+8j_0+9} - i_0 - \frac{1}{2} \rfloor$.

Die Gesamtzahl ist jetzt einfach

$$\begin{aligned}
\sum_{k=1}^{m} W(k,k^2) &= \sum_{k=1}^{m} \left\lfloor \frac{1}{2}\sqrt{4k(k+1)+8k^2+9} - k - \frac{1}{2} \right\rfloor \\
&= \sum_{k=1}^{m} \left\lfloor \frac{1}{2}\sqrt{12k^2\left(1+\frac{1}{3k}+\frac{4}{4k^2}\right)} - k - \frac{1}{2} \right\rfloor \\
&= \sum_{k=1}^{m} \left\lfloor \sqrt{3}\cdot k\left(1+\frac{1}{6k}+O\left(\frac{1}{k^2}\right)\right) - k - \frac{1}{2} \right\rfloor \\
&= \sum_{k=1}^{m} \left\lfloor k(\sqrt{3}-1) + \frac{\sqrt{3}}{6} - \frac{1}{2} + O(k^{-1}) \right\rfloor \\
&= \sum_{k=1}^{m} \Big(k(\sqrt{3}-1) + O(1)\Big) = \frac{\sqrt{3}-1}{2}m^2 + O(m) \approx 0.366\cdot m^2
\end{aligned} \tag{3.7}$$

Statt die Summe genau zu berechnen, begnügen wir uns mit einer groben Abschätzung. Wir haben hierzu den Satz von Taylor verwendet:

$$\sqrt{1+x} = 1 + \frac{1}{2}x + O(x^2)$$

Setzen wir $m = 1000$ in die Näherungsformel ein erhalten wir etwa 366025. Der exakte Wert ist 365682.

3.12 Suchen in einem ungeordneten Feld

Betrachten wir ein weiteres Beispiel. Nehmen wir an, wir haben ein Array $a[1] \ldots a[n]$ mit paarweise verschiedenen Zahlen. Wir wollen ein Programm schreiben, welches nachsieht, ob eine bestimmte Zahl im Array enthalten ist. Ein naheliegendes Programm in C könnte so aussehen:

```
int search1(int v)
{
    int i;
    for(i=1; i<=n; i++)
        if(a[i]==v) return 1;
    return 0;
}
```

Wie groß ist die Laufzeit des Programms?

Offensichtlich hängt die Antwort diesmal von verschiedenen Faktoren ab, zunächst davon, ob v im Array vorkommt oder nicht.

Betrachten wir zunächst die *erfolglose* Suche: Die `for`-Schleife wird $n$mal durchlaufen und anschließend wird 0 als Ergebnis geliefert.

Bei einer erfolgreichen Suche dagegen gibt es ein i mit $a[i] = v$. Um eine Analyse machen zu können, müssen wir etwas darüber wissen, welches i dies sein wird. Im allgemeinen können wir eine statistische Aussage über die Eingabe machen. Hier nehmen wir an, daß die Elemente im Array in einer zufälligen Reihenfolge vorliegen. Für jedes i zwischen 1 und n ist dann die Wahrscheinlichkeit für $a[i] = v$ gerade $1/n$.

Definieren wir $L(i)$ als die Laufzeit des Programms, wenn $a[i] = v$ ist. Die durchschnittliche Laufzeit ist dann

$$\frac{1}{n}\sum_{i=1}^{n} L(i).$$

```
1 sw -4(r29),r30
1 add r30,r0,r29
1 sw -8(r29),r31
1 subui r29,r29,#24
1 sw 0(r29),r2
1 sw 4(r29),r3
1 sw 8(r29),r4
1 lhi r1,((_n)>>16)&0xffff
1 addui r1,r1,(_n)&0xffff
1 lw r3,(r1)
1 sgti r1,r3,#0
1 beqz r1,L3
1 lw r4,(r30)
1 lhi r1,((_a)>>16)&0xffff
1 addui r1,r1,(_a)&0xffff
1 lw r2,(r1)
1 addi r31,r2,#4
1 slli r1,r3,#0x2
1 add r2,r1,r2
L5:
Z lw r1,(r31)
Z seq r1,r1,r4
Z bnez r1,L8
Z addi r1,r0,#1
Z-1 addi r31,r31,#4
Z-1 sle r1,r31,r2
Z-1 bnez r1,L5
Z-1 nop
L3:
1 addi r1,r0,#0
L8:
1 lw r2,0(r29)
1 lw r3,4(r29)
1 lw r4,8(r29)
1 lw r31,-8(r30)
1 add r29,r0,r30
1 jr r31
1 lw r30,-4(r30)
```

Abbildung 3.1: Suchprogramm

Wir müssen also nur noch $L(i)$ bestimmen. Die `for`-Schleife und die `if`-Anweisung werden gerade i mal ausgeführt. Bezeichnen wir aber genauer die Anzahl der Maschinenanweisungen mit $L(i)$ benötigen wir das Assemblerprogramm (Abbildung 3.1). Sei Z die durchschnittliche Anzahl von Durchläufen der `for`-Schleife. Wir erhalten

$$Z = \frac{1}{n}\sum_{k=1}^{n} k = \frac{n+1}{2}.$$

Die Laufzeit des Programms ist $24 + 8Z$ Maschineninstruktionen gemäß Abbildung 3.1. Das ergibt $24 + 4(n + 1) = 4n + 28$ Instruktionen im Durchschnitt, wenn das Array Größe n hat.

Wir benötigten zur Lösung dieses Problems gar keine Rekursionsgleichung, scheint es. Das ist aber nicht ganz richtig. Hier ist die Situation so einfach, daß man sofort die Lösung sieht. In Wirklichkeit kann man es auch systematisch mit Rekursionsgleichungen versuchen. Wir müssen hierzu den Fall mit n Elementen auf den Fall mit $n - 1$ Elementen zurückführen. Das ist einfach: $Z_1 = 1$, denn bei einem Element sucht (und findet) man natürlich einmal. Für $n > 1$ aber gilt $Z_n = 1 \cdot \frac{1}{n} + (1 - \frac{1}{n})(1 + Z_{n-1})$, denn mit Wahrscheinlichkeit $\frac{1}{n}$ finden wir v schon im ersten Platz des Feldes und mit Wahrscheinlichkeit $1 - \frac{1}{n}$ müssen wir es in den verbleibenden $n - 1$ Plätzen suchen, was nocheinmal Z_{n-1} Vergleiche im Durchschnitt benötigt. Die Rekursionsgleichung lautet also

$$Z_n = 1 + (1 - \frac{1}{n})Z_{n-1}$$

mit $Z_1 = 1$. Dies ist eine lineare Rekursionsgleichung erster Ordnung, welche wir mit einem Summationsfaktor lösen können. Die Lösungsformel

```
1 sw -4(r29),r30
1 add r30,r0,r29
1 sw -8(r29),r31
1 subui r29,r29,#16
1 sw 0(r29),r2
1 sw 4(r29),r3
1 lhi r1,((_a)>>16)&0xffff
1 addui r1,r1,(_a)&0xffff
1 lw r2,(r1)
1 lhi r1,((_n)>>16)&0xffff
1 addui r1,r1,(_n)&0xffff
1 lw r31,(r1)
1 sgti r1,r31,#0
1 beqz r1,L3
1 lw r3,(r30)
1 addi r2,r2,#4
L9:
Z lw r1,(r2)
Z seq r1,r1,r3
Z bnez r1,L8
Z addi r1,r0,#1
Z-1 addi r31,r31,#1
Z-1 sgti r1,r31,#0
Z-1 bnez r1,L9
Z-1 addi r2,r2,#4
1 addi r2,r2,#-4
L3:
1 addi r1,r0,#0
L8:
1 lw r2,0(r29)
1 lw r3,4(r29)
1 lw r31,-8(r30)
1 add r29,r0,r30
1 jr r31
1 lw r30,-4(r30)
```

Abbildung 3.2: Noch ein Suchprogramm

auf Seite 31 sagt, die Lösung sei

$$
\begin{aligned}
Z_n &= 1 + \sum_{j=1}^{n-1} \left(1 - \frac{1}{j+1}\right)\left(1 - \frac{1}{j+2}\right)\left(1 - \frac{1}{j+3}\right)\cdots\left(1 - \frac{1}{n}\right) \\
&= 1 + \sum_{j=1}^{n-1} \frac{j}{j+1}\frac{j+1}{j+2}\frac{j+2}{j+3}\cdots\frac{n-1}{n} = 1 + \sum_{j=1}^{n-1}\frac{j}{n} = 1 + \frac{n-1}{2} = \frac{n+1}{2}.
\end{aligned}
$$

Das Ergebnis stimmt. Durften wir die Lösungsformel überhaupt verwenden? Ja, denn die Bedingungen sind alle erfüllt, vor allem $Z_0 = 0$.

Verbessern wir den Suchalgorithmus jetzt, um zu sehen wie stark sich die Verbesserungen bemerkbar machen. Zunächst ersetzen wir den Arrayzugriff über einen Index durch einen Zugriff über einen Pointer. Außerdem zählen wir rückwärts, um einen effizienteren Vergleich auf Null durchzuführen. Das zugehörige Programm wäre

```
int search2(int v)
{
    int i; int *p=&a[0];
    for(i=n;i>0;i++)
        if(*++p==v) return 1;
    return 0;
}
```

Das zugehörige DLX-Assemblerprogramm findet sich in Abbildung 3.2.

Diesmal erhalten wir 20+8Z Maschinenanweisungen, im Durchschnitt also $4n + 24$. Diese “Verbesserung” ist also nichts wert; sie spart nur 4 Anweisungen. (Allerdings hat der Compiler hier nicht besonders gut optimiert.)

Es ist also gar nicht so einfach dieses Programm wesentlich schneller zu machen. Ein Trick hilft aber doch gewaltig: Wir ersparen uns das Zählen.

```
1 sw -4(r29),r30                1 lw r1,(r1)                     L9:                      1 lw r2,0(r29)
1 add r30,r0,r29                1 lhi r2,((_a+4)>>16)&0xffff   P−1 lw r2,(r31)            1 lw r3,4(r29)
1 sw -8(r29),r31                1 addui r2,r2,(_a+4)&0xffff    P−1 sgt r1,r2,r3           1 lw r4,8(r29)
1 subui r29,r29,#24             1 slli r1,r1,#0x2              P−1 bnez r1,L9             1 lw r31,-8(r30)
1 sw 0(r29),r2                  1 add r1,r1,r2                 P−1 addi r31,r31,#4        1 add r29,r0,r30
1 sw 4(r29),r3                  1 lhi r4,#1                      1 addi r31,r31,#-4       1 jr r31
1 sw 8(r29),r4                  1 addui r4,r4,#34464             1 seq r2,r2,r3           1 lw r30,-4(r30)
1 lw r3,(r30)                   1 sw (r1),r4                     1 bnez r2,L6
1 lhi r1,((_n)>>16)&0xffff      1 addi r31,r2,#-4                1 addi r1,r0,#0
1 addui r1,r1,(_n)&0xffff       1 addi r31,r31,#4                L6:
```

Abbildung 3.3: Schlauere Suche. Der Sprung zum Label L6 wird bei einer erfolgreichen Suche nie durchgeführt.

Wir müssen nur das gesuchte Element an das Ende des Arrays speichern und wir brauchen nicht dafür zu sorgen, über das Arrayende hinaus zu suchen:

```
int search3(int v)
{
    int i=0, *p;
    a[n+1]=v;
    p = a;
    do { p++; } while(*p != v);
    if(p==a+n+1) return 0;
    else return 1;
}
```

Diesmal sind es $35 + 4Z$ Maschinenbefehle. Im Durchschnitt also genau $2n + 37$ Anweisungen. Dies ist für große n schneller als die bisherigen Lösungen, aber andererseits für kleine n schlechter. Interessiert man sich nur für die erfolgreiche Suche, dann ist die schlauere Suche ab $n = 5$ schneller.

3.13 Suchen in einem geordneten Feld und binäre Suchbäume

Betrachten wir als nächstes die Suche in einem geordneten Feld. Wieder nehmen wir an im Array $a[1], \ldots, a[n]$ befinden sich n verschiedene Zahlen, diesmal sind sie aber geordnet. Wie lange dauert es im Durchschnitt, eine dieser Zahlen zu finden, wenn wir jede Zahl mit gleicher Wahrscheinlichkeit wählen? (Das ist wieder die *erfolgreiche* Suche.) Weiter können wir fragen, wie lange es dauert, festzustellen, daß eine Zahl *nicht* im Feld enthalten ist. Welche Wahrscheinlichkeitsverteilung sollen wir jetzt nehmen? Wenn der Algorithmus nur auf Vergleichen beruht, dann hängt die Laufzeit nur davon ab, wo die gesuchte Zahl in das Feld eingeordnet werden müßte. Dafür gibt es $n+1$ Möglichkeiten, nämlich vor dem ersten Element, hinter dem letzten Element oder zwischen zwei Elemente, also in einen der $n-1$ Zwischenräume.

Der erste Algorithmus, den wir betrachten, sucht wieder von links nach rechts. Sobald ein betrachtetes Arrayelement größer als das zu suchende Element ist, können wir die Suche abbrechen. Wir nehmen an, wir können am Ende des Arrays eine Pseudozahl H eintragen, welche größer als alle vorkommenden Zahlen ist.

```
int search4(int v)
{
    int *p;
    a[n+1]=H;
    p=a;
    do { p++; } while( *p>v);
    if( *p==v) return 1;
    else return 0;
}
```

Die Laufzeit hängt sehr davon ab, wie oft die Anweisung `p++` ausgeführt wird. Ist die Suche erfolglos, wird p solange erhöht, bis es auf den Wert $a[K]$ zeigt, wobei K der kleinste Index mit $a[K] \geq v$ ist. Am Anfang zeigt p auf $a[0]$. Es wird also bei einer erfolglosen Suche K mal erhöht. Der Wert K ist eine Zufallsvariable mit der Verteilung

$$\Pr(K = k) = \frac{1}{n+1} \text{ für } 1 \leq k \leq n+1$$

Nennen wir die durchschnittliche Zahl, wie oft `p++` ausgeführt wird P_n, wenn das Array n Elemente hat. Die Zahl P_n ist also einfach der Erwartungswert von K:

$$P_n = E[K] = \sum_{k=1}^{n+1} k \cdot \Pr(K = k) = \frac{(n+2)(n+1)}{2(n+1)} = 1 + \frac{n}{2}$$

Wie groß ist P_n für eine erfolgreiche Suche? Wenn die Suche erfolgreich ist, dann gilt $a[K] = v$ für genau ein $1 \leq K \leq n$. Die Wahrscheinlichkeit ist

```
1 sw -4(r29),r30
1 add r30,r0,r29
1 sw -8(r29),r31
1 subui r29,r29,#24
1 sw 0(r29),r2
1 sw 4(r29),r3
1 sw 8(r29),r4
1 lw r3,(r30)
1 lhi r1,((_n)>>16)&0xffff
1 addui r1,r1,(_n)&0xffff
1 lw r1,(r1)
1 lhi r2,((_a+4)>>16)&0xffff
1 addui r2,r2,(_a+4)&0xffff
1 slli r1,r1,#0x2
1 add r1,r1,r2
1 lhi r4,#1
1 addui r4,r4,#34464
1 sw (r1),r4
1 addi r31,r2,#-4
1 addi r31,r31,#4
  L9:
P-1 lw r2,(r31)
P-1 sgt r1,r2,r3
P-1 bnez r1,L9
P-1 addi r31,r31,#4
  1 addi r31,r31,#-4
  1 seq r2,r2,r3
  1 bnez r2,L6
  1 addi r1,r0,#0
  L6:
1 lw r2,0(r29)
1 lw r3,4(r29)
1 lw r4,8(r29)
1 lw r31,-8(r30)
1 add r29,r0,r30
1 jr r31
1 lw r30,-4(r30)
```

Abbildung 3.4: Lineare Suche in geordnetem Feld.

diesmal $\Pr(K = k) = 1/n$. Wir erhalten daher

$$P_n = E[K] = \sum_{k=1}^{n+1} k \cdot \Pr(K = k) = \frac{(n+1)n}{2n} = \frac{1}{2} + \frac{n}{2}.$$

Der erfolgreiche und der erfolglose Fall werden sich nicht sehr unterscheiden.

Wir können jetzt die Anzahl der ausgeführten Maschineninstruktionen des entsprechenden Assemblerprogramms in Abbildung 3.4 zählen.

Die Laufzeit ist $27 + 4P_n$ im Mittel, also $2n + 31$ für eine erfolglose Suche und $2n + 29$ für eine erfolgreiche Suche.

Im folgenden werden wir Algorithmen, welche auf Vergleichen basieren, durch die Theorie der *binären Suchbäume* analysieren. Diese Theorie macht es leicht, die durchschnittliche Anzahl von Vergleichen, welche ein Algorithmus ausführt, auszurechnen.

Ein binärer Suchbaum ist ein natürlich ein binärer Baum. Entweder er besteht nur aus einem Knoten, der Wurzel, oder die Wurzel hat zwei Kinder, welche selbst wieder binäre Bäume sind. Wir unterscheiden zwischen *internen* und *externen* Knoten: Ein interner Knoten ist ein Knoten, welcher selbst Kinder hat, ein externer Knoten dagegen hat keine Kinder (oft *Blatt* genannt).

Die Vergleiche, welche ein Algorithmus ausführt, führen in natürlicher Weise zu einem binären Suchbaum: Die Wurzel des Baums wird mit dem ersten Vergleich beschriftet. Das linke Kind ist der Binärbaum des Vergleiche des Algorithmus, wenn der Vergleich *kleiner* ergab, das rechte Kind der entsprechende Binärbaum für das Ergebnis *größer*. Wir nehmen im Augenblick an, es handelt sich um einen Suchalgorithmus und wenn das Ergebnis *gleich* heißt, dann hört der Algorithmus auf zu suchen, da er das gewünschte Element eben fand.

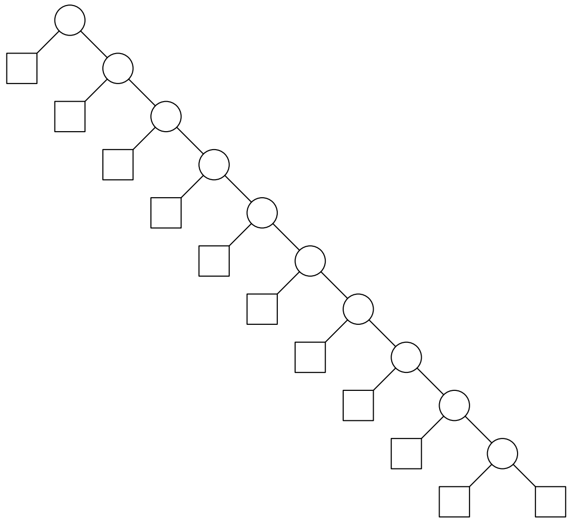

Abbildung 3.5: Ein Suchbaum für die lineare Suche in einem geordnetem Feld der Länge 10.

Wir zeichnen binäre Suchbäume wie üblich, aber interne Knoten werden als Kreise, externe Knoten als Quadrate dargestellt. Ein Suchbaum des Algorithmus für lineare Suche in einem geordnetem Feld findet sich in Abbildung 3.5 für ein Feld der Größe 10.

Wir definieren nun rekursiv einige wichtige Größen für binäre Suchbäume T: die *Größe* $|T|$, welche gerade die Anzahl der internen Knoten ist, die *interne Pfadlänge* $\pi(T)$ und die *externe Pfadlänge* $\xi(T)$.

Wenn T nur aus der Wurzel besteht, dann definieren wir $|T| = 0$, $\pi(T) = 0$, $\xi(T) = 0$. Besteht T aus der Wurzel, welche die binären Bäume T_1 und T_2 als Kinder besitzt, dann definieren wir $|T| = |T_1| + |T_2| + 1$, $\pi(T) = \pi(T_1) + \pi(T_2) + |T| - 1$ und $\xi(T) = \xi(T_1) + \xi(T_2) + |T| + 1$.

Anschaulich ist die interne Pfadlänge die Summe der *Level* aller internen Knoten und die externe Pfadlänge die Summe der *Level* aller externen Knoten. Der *Level* eines Knotens ist seine Entfernung zur Wurzel, die Wurzel selbst ist daher auf Level 0.

Das Schöne an diesen Definition ist, daß wir mit Hilfe von $\pi(T)$ und $\xi(T)$ einfache Formel für die durchschnittlich Anzahl von Vergleichen finden können.

Theorem 5 *Wenn T der Suchbaum eines Algorithmus ist und jedes Element wird bei einer erfolgreichen Suche, beziehungsweise jede Position zwischen Elementen bei einer erfolglosen Suche, mit gleicher Wahrscheinlichkeit gesucht, dann ist die*

Anzahl der Vergleiche

$$\begin{aligned} C^+ &= \frac{\pi(T)}{|T|} + 1 \text{ im erfolgreichen Fall,} \\ C^- &= \frac{\xi(T)}{|T|+1} \text{ im erfolglosen Fall.} \end{aligned}$$

Außerdem gilt immer folgende Beziehung zwischen der internen und externen Pfadlänge:

$$\xi(T) = \pi(T) + 2|T|$$

Zwischen C^+ und C^- besteht folgende Beziehung:

$$C^- = (C^+ + 1)\left(1 - \frac{1}{|T|+1})\right)$$

Die Anzahl externer Knoten ist immer $|T| + 1$.

Wenden wir diese Formeln für die lineare Suche in einem Array mit n Elementen an. Wir erhalten als Suchbäume "Fischgräten" L_n, wie in Abbildung 3.5. Die interne Pfadlänge ist offensichtlich

$$\pi(L_n) = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2},$$

denn auf jedem Level zwischen 0 und $n-1$ gibt es gerade einen internen Knoten. Die externe Pfadlänge ist dann

$$\xi(L_n) = \pi(L_n) + 2n = \frac{n(n+3)}{2}.$$

Die mittlere Zahl von Vergleichen bei einer erfolglosen Suche ist

$$\frac{\pi(L_n)}{|L_n|} + 1 = \frac{n-1}{2} + 1 = \frac{n+1}{2}$$

und die mittlere Zahl von Vergleichen bei einer erfolglosen Suche wäre nach der Formel eigentlich

$$\frac{\xi(L_n)}{|L_n|+1} = \frac{n(n+3)}{2(n+1)} = \frac{n}{2} + \frac{n}{n+1}.$$

Dies ist aber nicht richtig.

Wo liegt der Fehler? Es liegt an der Art und Weise, wie das Programm verfährt, wenn das zu suchende Element größer als $a[n]$ ist. Wenn das Programm $a[n] < v$ verifiziert hat, wäre kein weiterer Vergleich mehr nötig. Dennoch wird ein weiterer Vergleich mit $a[n+1] = v$ durchgeführt. Dies ist aber kein richtiger Vergleich in unserem Sinn, denn *es gibt nur eine Antwort, der Vergleich ist redundant.* Die Theorie mit den Suchbäumen funktioniert nur, wenn die Wahrscheinlichkeit des Erreichens jeden externen Knotens gleich ist. Das kann aber nicht mehr der Fall sein, wenn es redundante Vergleiche gibt. Dieser letzte Vergleich kommt in dem von uns verwendeten Suchbaum auch nicht vor: Er enthält nur n interne Knoten, beschriftet mit den Vergleichen $a[1] : v, \ldots, a[n] : v$. Wird aber der allerletzte externe Knoten im Baum besucht, führt der Algorithmus einen weiteren redundanten Vergleich aus, was mit Wahrscheinlichkeit $1/(n+1)$ bei einer erfolglosen Suche passiert. Die tatsächliche Anzahl von Vergleichen, welche der Algorithmus im Durchschnitt bei einer erfolglosen Suche vollführt ist daher

$$\frac{\xi(L_n)}{|L_n| + 1} + \frac{1}{n+1} = \frac{n}{2} + 1,$$

wie wir bereits vorher "zu Fuß" bewiesen haben.

Keine Probleme hätte das Programm in der Form

```
int search5(int v)
{
    int i=0;
    do { i++; } while( i<=n && a[i]>v);
    if( i<=n) return 1;
    else return 0;
}
```

gemacht. Hier werden tatsächlich genau $n/2 + n/(n+1)$ Vergleiche durchschnittlich bei einer erfolglosen Suche durchgeführt. Dieses Programm ist aber viel langsamer. Die Moral von dieser Rechnung ist: Die Formeln für die durchschnittliche Anzahl von Vergleichen sind wirklich nur korrekt, wenn alle Vorraussetzungen erfüllt sind. Dies muß genau kontrolliert werden und Abweichungen in die Rechnung miteinbezogen werden.

Binäre Suche

Wenn das Feld geordnet ist, bietet sich natürlich eine *binäre* Suche an: Man vergleicht v mit dem Element, welches in der Mitte des Feldes liegt.

Dann hat man das Problem auf eine Suche in einem halb so großen Feld zurückgeführt. Folgender Algorithmus leistet das Gewünschte:

```
int binsearch(int v)
{
    int l,r,m;
    l=1; r=n;
    while(l<=r) {
        m=(r+l)/2;
        if(v==a[m]) return 1;
        if(v<a[m]) r=m-1; else  l=m+1;
    }
    return 0;
}
```

Dieser Algorithmus verwendet die Variablen l und r, um sich die gerade gültigen Grenzen des Subarrays zu merken, in welchem noch gesucht werden muß. Es wird mit dem Element in der Mitte verglichen. Ist dieses Element das richtige, terminiert der Algorithmus. Andernfalls wird die rechte oder linke Grenze des noch zu durchsuchenden Bereichs angeglichen und die Suche geht weiter.

Bezeichnen wir mit B_n, wie oft die Anweisung `m=(r+l)/2` ausgeführt wird, was gerade die Anzahl der Vergleiche $v : a[i]$ ist. Wir betrachten zuerst die erfolglose Suche, da sie einfacher zu analysieren ist und außerdem zunächst nur den worst-case.

Sei $N = r - l + 1$ die Größe des zu durchsuchenden Subarrays. Sei C_N die Anzahl, wie oft `m=(r+l)/2` noch ausgeführt wird, wenn $r - l + 1 = N$ ist. Dann ist $C_1 = 1$, denn $N = 1$ bedeutet $r = l$ und nachdem `m=(r+l)/2` ausgeführt wird terminiert der Algorithmus oder r wird vergrößert oder l wird erniedrigt. Dann terminiert die `while`-Schleife sofort.

Wenn $N > 1$, dann wird `m=(r+l)/2` einmal ausgeführt und anschließend $r := \lceil (r + l)/2 \rceil - 1$ oder $l := \lceil (r + l)/2 \rceil + 1$ ausgeführt. In beiden Fällen heißt das aber $N := \lfloor N/2 \rfloor$; wie erwartet halbiert sich der Suchraum in jedem Schritt.

Wie sieht der binäre Suchbaum dieses Algorithmus aus? Abbildung 3.6 zeigt den Suchbaum für $n = 10$. Im allgemeinen ist es ein fast vollständiger Binärbaum, in welchem nur auf dem letzten Level Knoten fehlen können.

Hat der Suchbaum 2^k externe Knoten, dann liegen alle auf Level k und die externe Pfadlänge ist genau $k2^k$. Nehmen wir jetzt den allgemeinen Fall.

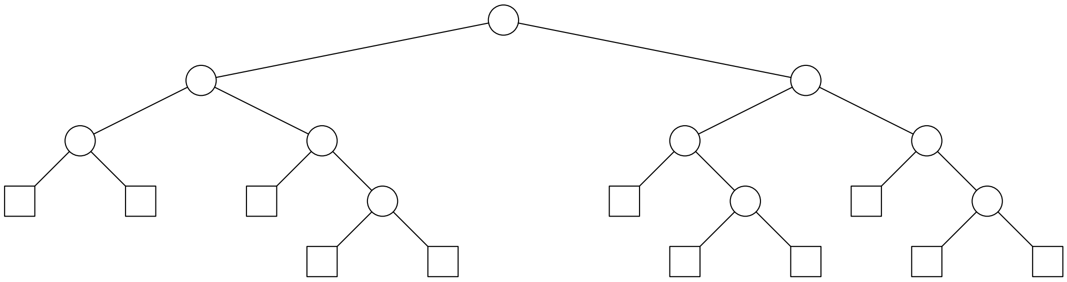

Abbildung 3.6: Ein Suchbaum für die binäre Suche in einem geordnetem Feld der Länge 10.

Wenn wir n interne Knoten haben, haben wir $n+1$ externe Knoten und $2(n+1-2^{\lfloor \log(n+1) \rfloor})$ von ihnen liegen auf Level $\lfloor \log(n+1) \rfloor + 1$. Der Rest, also $2^{\lfloor \log(n+1) \rfloor + 1} - n - 1$ aber liegen auf Level $\lfloor \log(n+1) \rfloor$. Bezeichnen wir den Suchbaum für binäre Suche in einem Array mit n Elementen mit B_n, dann erhalten wir also

$$\xi(B_n) = (n+1)(\lfloor \log(n+1) \rfloor + 2) - 2^{\lfloor \log(n+1) \rfloor + 1}.$$

Daraus können wir nun leicht die interne Pfadlänge berechnen:

$$\pi(B_n) = \xi(B_n) - 2n = (n+1)\lfloor \log(n+1) \rfloor - 2^{\lfloor \log(n+1) \rfloor + 1} + 2.$$

Es kann nichts schaden, diese Formel an einem kleinen Beispiel nachzuvollziehen. Wählen wir hierzu wieder $n = 10$, da wir den dazugehörigen Suchbaum in Abbildung 3.6 haben. Wir erhalten $\pi(B_{10}) = 11 \cdot 5 - 16 = 39$ und $\pi(B_{10}) = 11 \cdot 3 - 16 + 2 = 19$. Beide Ergebnisse lassen sich mit Hilfe des Baumes leicht nachzählen.

Aufgaben

3.1 Vergleichen Sie diese Lösung mit der allgemeinen Lösung aus dem ersten Kapitel, indem Sie dort $M = 0$ setzen.

3.2 Wenden Sie einen Summationsfaktor auf (3.4) an und finden Sie schließlich die Lösung der ursprünglichen Rekursionsgleichung (3.3) zwar nicht in geschlossenerer Form, aber als eine Summe.

3.3 Analysieren Sie die Laufzeit der erfolgreichen Suche, wenn jedes Element genau *zweimal* im Array vorkommt und wieder jede Permutation gleich wahrscheinlich ist.

3.4 Vergleichen Sie alle drei Suchalgorithmen bezüglich der erfolglosen Suche.

3.5 Verifizieren Sie, daß $N := \lfloor N/2 \rfloor$ wirklich stimmt.

Kapitel 4

Erzeugende Funktionen

Wir haben uns oft mit Folgen $(g_0, g_1, g_2, \ldots)$ beschäftigt, wobei die einzelnen Glieder g_n den Verbrauch einer Ressource oder eine andere kombinatorische Größe angeben. Oft ist die Folge implizit, z.B. durch eine Rekursionsgleichung gegeben. Wir nennen

$$G(z) = \sum_{n=0}^{\infty} g_n z^n$$

die *Erzeugende Funktion (Generating Funktion, GF)* der Folge $(g_n)_{n=0}^{\infty}$. Die fundamentale Aufgabe in der Analyse von Algorithmen ist es, einen expliziten Ausdruck für g_n oder eine gute Näherung dafür zu finden, wenn die Folge nur implizit gegeben ist. Erzeugende Funktionen sind das wichtigste Werkzeug hierfür, welches wir kennenlernen werden.

Ein wichtiger Teilschritt in vielen Analysen ist es, aus $G(z)$ das $n$te Folgenglied zu extrahieren. Theoretisch kann man $G(z)$ in eine Taylorreihe entwickeln,

$$G(z) = G(0) + zG'(0) + \frac{z^2}{2}G''(0) + \frac{z^3}{3!}G'''(0) + \ldots,$$

woraus sich

$$[z^n]G(z) = \frac{d^n}{dz^n}\frac{G(z)}{n!}\bigg|_{z=0}$$

ablesen läßt. Mit $[z^n]G(z)$ bezeichnen wir den Koeffizienten vor z^n in der Potenzreihe $G(z)$. Dieser Weg ist gewöhnlich zu steinig und es gibt in der Regel bessere Methoden den $n$ten Koeffizienten zu extrahieren. Eine Tabelle mit wichtigen, bekannten erzeugenden Funktionen ist dabei sehr nützlich. Tabelle 4.1 enthält die wichtigsten.

Folge	a_n	OGF
$1, 0, 0, 0, 0, 0, 0, \ldots$	$(n = 0)$	1
$0, 1, 0, 0, 0, 0, 0, \ldots$	$(n = 1)$	z
$1, 1, 1, 1, 1, 1, 1, \ldots$	1	$\frac{1}{1-z}$
$0, a, a^2, a^3, a^4, a^5, \ldots$	a^n	$\frac{1}{1-az}$
$2, 3, 4, 5, 6, 7, 8, \ldots$	$n+1$	$\frac{1}{(1-z)^2}$
$\binom{r}{0}, \binom{r}{1}, \binom{r}{2}, \binom{r}{3}, \ldots$	$\binom{r}{n}$	$(1+z)^r$
$\binom{k}{k}, \binom{k+1}{k}, \binom{k+2}{k}, \binom{k+3}{k}, \ldots$	$\binom{k+n}{k}$	$\frac{1}{(1-z)^{n+1}}$
$1, 1, \frac{1}{2!}, \frac{1}{3!}, \frac{1}{4!}, \ldots$	$\frac{1}{n!}$	e^z

Tabelle 4.1: Wichtige erzeugende Funktionen und ihre Folgen.

$$A(z) = \sum_{n=0}^{\infty} a_n z^n \qquad B(z) = \sum_{n=0}^{\infty} b_n z^n$$

$zA(z) = \sum_{n=0}^{\infty} a_{n-1} z^n$	(Rechts shift)
$\frac{A(z) - a_0}{z} = \sum_{n=0}^{\infty} a_{n+1} z^n$	(Links shift)
$A'(z) = \sum_{n=0}^{\infty} (n+1) a_{n+1} z^n$	(Differenzieren)
$\int_0^z A(t)\,dt = \sum_{n=1}^{\infty} \frac{a_{n-1}}{n} z^n$	(Integrieren)
$A(\lambda z) = \sum_{n=0}^{\infty} \lambda^n a_n z^n$	(Skalieren)
$A(z) + B(z) = \sum_{n=0}^{\infty} (a_n + b_n) z^n$	(Addition)
$(1-z)A(z) = a_0 + \sum_{n=0}^{\infty} (a_n - a_{n-1}) z^n$	(Differenz)
$A(z)B(z) = \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} a_k b_{n-k} \right) z^n$	(Faltung)
$\frac{A(z)}{1-z} = \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} a_k \right) z^n$	(Partialsumme)

Tabelle 4.2: Einige Operationen mit erzeugenden Funktionen. Wir definieren $a_n = b_n = 0$ für $n < 0$.

Mit Hilfe von Tabelle 4.1 und 4.2 und anderen Formelsammlungen lassen sich viele Funktionen in Reihen entwickeln. Man muß die erzeugenden Funktionen nur erst entsprechend umformen, damit sie zu einem Eintrag in den Tabellen passen.

Wenden wir unsere bisherigen Kenntnisse an einer einfachen Rekursionsgleichung an, der Rekursionsgleichung für Fibonacchi-Zahlen:

$$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2} \text{ für } n > 1$$

Der erste Schritt, um eine solche Rekursionsgleichung anzugehen, ist es eine einzige Formel zu finden, welche F_n für alle n richtig definiert. Wir wollen also die Fallunterscheidung loswerden. Wir verwenden dazu folgende Konvention:

$$(\mathit{Bedingung}) = \begin{cases} 1 & \text{falls } \mathit{Bedingung} \text{ wahr} \\ 0 & \text{sonst} \end{cases}$$

Die Rekursionsgleichung $F_n = F_{n-1} + F_{n-2}$ ist nur für $n > 1$ gültig, für $n = 1$ ist sie aber falsch, denn $F_1 = 1$, aber $F_0 + F_{-1} = 0$ (wir nehmen $F_n = 0$ an, für $n < 0$). Zufällig stimmt aber $F_n = F_{n-1} + F_{n-2}$ auch für $n = 0$. Folgende Formel ist daher für alle n richtig:

$$F_n = F_{n-1} + F_{n-2} + (n = 1) \tag{4.1}$$

Wir interessieren uns nun für die erzeugende Funktion

$$G(z) = \sum_{n=0}^{\infty} F_n z^n.$$

Um $G(z)$ zu erhalten, multiplizieren wir beide Seiten von (4.1) mit z^n und summieren über n von 0 bis ∞:

$$\sum_{n=0}^{\infty} F_n z^n = \sum_{n=0}^{\infty} F_{n-1} z^n + \sum_{n=0}^{\infty} F_{n-2} z^n + \sum_{n=0}^{\infty} (n = 1) z^n \tag{4.2}$$

Die letzte Summe ist einfach z und die anderen Summen lassen sich leicht umformen und wir erhalten

$$G(z) = zG(z) + z^2 G(z) + z. \tag{4.3}$$

Wir erhalten diese Gleichung auch schneller direkt aus (4.1) indem wir die Umformungsregeln aus Tabelle 4.2 verwenden. Die ersten beiden Terme der rechten Seite F_{n-1} und F_{n-2} sind die Folge F_n um 1 bzw. 2 nach rechts verschoben und daher sind ihre erzeugenden Funktionen einfach $zG(z)$ und $z^2G(z)$. Die erzeugende Funktion von $(0, 1, 0, 0, 0, 0, \ldots)$ ist z, denn diese Folge ist die Folge $(1, 0, 0, 0, 0, 0, \ldots)$ um 1 nach rechts verschoben. Die letzte Folge findet sich wieder in Tabelle 4.1. Die erzeugende Funktion der rechten Seite ist jetzt einfach die Summe der drei Funktionen (Additionsregel). Auf diese Weise erhalten wir eine algebraische Gleichung für $G(z)$. Unter Umständen können wir auch eine Differentialgleichung oder auch eine Integral- oder Integro-Differentialgleichung.

Lösen wir (4.3) nach $G(z)$ auf, dann erhalten wir die Lösung der algebraischen Gleichung und damit eine explizite Formel für $G(z)$:

$$G(z) = \frac{z}{1 - z - z^2}$$

Jetzt müssen wir $G(z)$ in eine Potenzreihe entwickeln. Dazu formen wir $G(z)$ so um, daß wir es in Tabelle 4.1 finden. Wir machen eine Partialbruchzerlegung der rationalen Funktion $1/(1 - z - z^2)$. Hierzu brauchen wir die Nullstellen von $z^2 + z - 1$, welche

$$\frac{1}{\phi} = \frac{\sqrt{5} - 1}{2} \text{ und } \frac{1}{\hat{\phi}} = \frac{-\sqrt{5} - 1}{2}$$

sind. Wir können also

$$\frac{1}{1 - z - z^2} = \frac{A}{1 - \phi z} + \frac{B}{1 - \hat{\phi} z}$$

schreiben, wobei wir die Parameter A und B noch bestimmen müssen.

Setzen wir $z = 0$ erhalten wir $1 = A + B$, also ist $B = 1 - A$. Setzen wir $z = 1$ erhalten wir $-1 = A/(1 - \phi) + (1 - A)/(1 - \hat{\phi})$. Nun vereinfachen wir. Aus $1/(1 - \phi) = -\phi$ und $1/(1 - \hat{\phi}) = -\hat{\phi}$ erhalten wir $1 = \phi A + \hat{\phi} - \hat{\phi} A = \hat{\phi} + \sqrt{5} A$, denn $\phi - \hat{\phi} = \sqrt{5}$. Schließlich erhalten wir $A = \phi/\sqrt{5}$.

Aus Tabelle 4.1 lesen wir ab, daß

$$\frac{1}{1 - \phi z} = \sum_{n=0}^{\infty} \phi^n z^n \text{ und } \frac{1}{1 - \hat{\phi} z} = \sum_{n=0}^{\infty} \hat{\phi}^n z^n$$

ist. Insgesamt erhalten wir

$$G(z) = \frac{z}{1 - z - z^2} = Az \sum_{n=0}^{\infty} \phi^n z^n + Bz \sum_{n=0}^{\infty} \hat{\phi}^n z^n,$$

woraus sich sofort der $n$te Koeffizient ablesen läßt:

$$[z^n]G(z) = A\phi^{n-1} + B\hat{\phi}^{n-1} = \frac{1}{\sqrt{5}}\left(\phi^n + \hat{\phi}^n\right)$$

Dieses Ergebnis hätten wir natürlich auch leicht durch konventionelle Methoden gewinnen können, denn die Rekursionsgleichung war linear mit konstanten Koeffizienten. Dieses Beispiel diente aber nur zur Einführung der erzeugenden Funktionen.

Wir werden die Formeln in den Tabellen 4.1 und 4.2 nicht alle beweisen. Begnügen wir uns in Tabelle 4.1 mit einem Beweis der letzten beiden Formeln.

Zunächst haben wir dort die Folge

$$0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \ldots$$

Die GF dieser Folge ist dann nach Definition und der Regel *Integrieren* in Tabelle 4.2

$$\sum_{n=1}^{\infty} \frac{1}{n} z^n = \int_0^z \frac{1}{1-t}\, dt,$$

denn $1/(1-z)$ ist die GF der Folge $(1, 1, 1, 1, \ldots)$. Wir können dieses Integral mit der Formel

$$\int \frac{f'(t)}{f(t)} dt = \ln(f(t)) + C$$

lösen und erhalten als GF

$$\int_0^z \frac{1}{1-t}\, dt = \ln \frac{1}{1-z}.$$

Als nächstes nehmen wir uns die Folge

$$H_0, H_1, H_2, H_3, H_4, \ldots$$

vor. Schreiben wir die GF aus, mit der Definition von H_n eingesetzt erhalten wir

$$\sum_{n=0}^{\infty} \left(\sum_{k=1}^{n} \frac{1}{k} \right) z^n.$$

Dieser Ausdruck ist ein Spezialfall der Regel *Partialsumme* mit $a_k = 1/k$ für $k > 0$ und $a_0 = 0$. Die Folge a_k haben wir eben behandelt und ihre GF ist $ln(1/(1-z))$. Daher erhalten wir wie erwartet

$$\sum_{n=0}^{\infty} \left(\sum_{k=1}^{n} \frac{1}{k} \right) z^n = \frac{1}{1-z} \ln \frac{1}{1-z}.$$

4.1 Zählen mit erzeugenden Funktionen

Mit erzeugenden Funktionen können wir die Anzahl von Gegenständen einer bestimmten Größe in einer Menge zählen. Wir wenden dies am Beispiel binärer Bäume an.

Ein binärer Baum besteht entweder aus nur einem externen Knoten oder aus einem internen Knoten, welcher selbst zwei binäre Bäume als Kinder besitzt. Wir sind in der Anzahl der binären Bäume mit einer bestimmten Anzahl interner Knoten interessiert. Wir definieren also die erzeugende Funktion

$$T(z) = \sum_{n=0}^{\infty} t_n,$$

wobei t_n die Anzahl binärer Bäume mit n interner Knoten sind.

Wir erhalten für t_n die Rekursionsgleichung

$$t_n = \sum_{k=0}^{n-1} t_k t_{n-1-k} + (n = 0).$$

Durch Multiplikation beider Seiten mit z^n und Summieren über n ergibt sich daraus

$$\begin{aligned} T(z) &= \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n-1} t_k t_{n-1-k} \right) z^n + 1 \\ &= z \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} t_k t_{n-k} \right) z^n + 1 \\ &= zT(z)^2 + 1 \end{aligned}$$

Wir kommen aber viel einfacher auf diese Beziehung: Wir denken uns T als die Menge aller binären Bäume. Dann ist $T = E + ITT$. Wir abstrahieren aber und interessieren uns nur für die Anzahl interner Knoten. Das heißt wir ersetzen E durch z^0 und I durch z^1. Statt T schreiben wir $T(z)$. Das ergibt:

$$T(z) = zT(z)^2 + 1$$

Wir müssen nun $T(z)$ in eine Reihe entwickeln, um $t_n = [z^n]T(z)$ ablesen zu können.

Zuerst sollten wir die Gleichung nach $T(z)$ auflösen, was nicht schwer ist, da es eine quadratische Gleichung ist. Wir erhalten

$$T(z) = \frac{1}{2z} \pm \frac{1}{2z}\sqrt{1-4z}.$$

Es gibt natürlich zwei Lösungen, denn es war ja eine quadratische Gleichung, aber unser Problem hat nur eine Lösung. Wie ist das möglich? Ganz einfach: Wir lösen eine funktionale Gleichung für $T(z)$. Die eine Lösung der Gleichung ist eine im Ursprung analytische Funktion, welche sich in eine Potenzreihe entwickeln läßt, während die andere Lösung im Ursprung nicht analytisch ist. Wir sehen schnell, welche Lösung die richtige ist. Entwickeln wir zunächst $\sqrt{1-4z}$ in eine Potenzreihe, was direkt mit der Newtonschen Formel geht.

$$T(z) = \frac{1}{2z} \pm \frac{1}{2z}\sqrt{1-4z} = \frac{1}{2z} \pm \frac{1}{2z}\sum_{n=0}^{\infty}\binom{1/2}{n}(-4)^n z^n$$

Jetzt sehen wir, das wir die Lösung mit dem Minuszeichen wählen müssen, denn dann verschwindet der Pol bei $z = 0$.

$$T(z) = -\frac{1}{2z}\sum_{n=1}^{\infty}\binom{1/2}{n}(-4)^n z^n = -\frac{1}{2}\sum_{n=0}^{\infty}\binom{1/2}{n+1}(-4)^{n+1} z^n$$

Wir lesen das Ergebnis ab:

$$t_n = [z^n]T(z) = -\frac{1}{2}\binom{1/2}{n+1}(-4)^{n+1}$$

Es kann nichts schaden, diese Formel an einem kleinen Beispiel zu überprüfen, zum Beispiel $n = 3$.

$$t_3 = -\frac{1}{2}\binom{1/2}{n+4}(-4)^4 = 5$$

Das Ergebnis läßt sich noch vereinfachen, aber statt dessen gehen wir noch einen anderen Weg, um zu diesem Ergebnis zu kommen.

Unangenehm an der Herleitung war die quadratische Gleichung. Wir lernen nun einen wichtigen Satz kennen, welcher die Koeffizienten in einer Potenzreihe in Beziehung zu den Koeffizienten in der Potenzreihe der inversen Funktion setzt.

Theorem 6 (Lagrange Inversion)
Sei $G(z)$ eine GF, für welche $z = f(G(z))$ gilt mit $f(0) = 0$ und $f'(0) \neq 0$. Dann gilt

$$[z^n]G(z) = \frac{1}{n}[u^{n-1}]\left(\frac{u}{f(u)}\right)^n.$$

Direkt auf $T(z)$ können wir das Lagrange Inversion nicht anwenden, da das resultierende f nicht die nötigen Eigenschaften besitzt. Setzen wir $H(z) = zT(z)$ geht es aber sehr gut. Die entsprechende funktionale Gleichung ist dann

$$H(z) = z + H(z)^2,$$

welche sich sehr leicht nach z auflösen läßt, aber wieder nicht so leicht nach $H(z)$. Wir haben $z = f(H(z))$ mit $f(t) = t - t^2$. Da $f(0) = 0$ und $f(0) = 1$ sind die Bedingungen für die Lagrange Inversion erfüllt:

$$[z^n]H(z) = \frac{1}{n}[u^{n-1}]\left(\frac{1}{1-u}\right)^n$$

Eine Formel aus Tabelle 4.1 paßt zur rechten Seite,

$$\frac{1}{(1-u)^n} = \sum_{k=0}^{\infty}\binom{k+n-1}{k}z^k$$

so daß sich

$$[z^n]H(z) = \frac{1}{n}\binom{2n-2}{n-1}$$

ergibt. Für $T(z)$ heißt das

$$t_n = \frac{1}{n+1}\binom{2n}{n}.$$

Überprüfen wir diese Formel wieder für $n = 3$:

$$t_3 = \frac{1}{4}\binom{6}{3} = 5$$

4.2 Bivariate erzeugende Funktionen

Bisher betrachteten wir GFs mit einer Variable z, welche mit einer Folge korrespondiert. Wir können dies selbstverständlich auf GFs mit mehreren Variablen verallgemeinern. Diese entsprechen dann mehrdimensionalen

Folgen. Im Falle zweier Variablen nennen wir sie *bivariate erzeugende Funktionen (Bivariate Generating Functions, BGFs)*.

Rechnen wir ein einfaches Beispiel mit BGFs, für welches wir das Ergebnis bereits kennen. Wieviele Strings gibt es über dem Alphabet $\{0, 1\}$, welche genau m 1en enthalten und n Zeichen lang sind?

Wir können die Menge aller Bitstrings rekursiv folgendermaßen definieren:

$$B = \{\epsilon\} \cup 0B \cup 1B$$

Definieren wir die BGF

$$B(u, z) = \sum_{n,m \geq 0} b_{mn} u^m z^n,$$

wobei b_{mn} die Anzahl der Bitstrings der Länge n mit genau m 1en bezeichnet, dann erhalten wir

$$B(u, z) = 1 + zB(u, z) + uzB(u, z).$$

Wir lösen nach $B(u, z)$ auf und entwickeln das Ergebnis in eine Reihe:

$$B(u, z) = \frac{1}{1 - z(1 + u)} = \sum_{n=0}^{\infty} z^n (1 + u)^n = \sum_{n=0}^{\infty} \sum_{k=0}^{n} \binom{n}{k} z^n u^k$$

Nun können wir $b_{mn} = [u^m z^n]B(u, z) = \binom{n}{m}$ ablesen.

4.3 Exponentielle erzeugende Funktionen

Wir definieren zu einer Folge $g_0, g_1, g_2, g_3, \ldots$ die *exponentielle erzeugende Funktion (Exponential Generating Function, EGF)*

$$G(z) = g_0 \frac{1}{1!} + g_1 \frac{z}{2!} + g_2 \frac{z^2}{3!} + g_2 \frac{z^3}{3!} + \cdots = \sum_{n=0}^{\infty} g_n \frac{z^n}{n!}.$$

Der $n$te Koeffizient in der EGF $G(z)$ von $(g_n)_{n=0}^{\infty}$ ist dann natürlich

$$g_n = n![z^n]G(z).$$

In manchen Fällen kommt man mit einer EGF leichter zum Ziel als mit einer GF und manchmal ist es umgekehrt. Der Unterschied besteht darin, daß es für verschiedene Folgen verschieden einfache GFs und EGFs gibt und daß sich EGFs verschiedenen transformieren, wenn man die Folge verändert. Die Tabelle 4.3 enthält EGFs für mehrere oft vorkommende Folgen und Tabelle 4.4 enthält Transformationsregeln für EGFs. Diese beiden Tabellen entsprechen den Tabellen 4.1 und 4.2 für GFs.

Folge	a_n	OGF
$1, 0, 0, 0, 0, 0, 0, \dots$	$(n = 0)$	1
$0, 1, 0, 0, 0, 0, 0, \dots$	$(n = 1)$	z
$1, 1, 1, 1, 1, 1, 1, \dots$	1	e^z
$1, c, c^2, c^3, c^4, c^5, c^6, \dots$	c^n	e^{cz}
$0, 1, 2, 3, 4, 5, 6, \dots$	n	ze^z
$\binom{0}{m}, \binom{1}{m}, \binom{2}{m}, \dots$	$\binom{n}{m}$	$\frac{z^m}{m!}e^z$
$1, 0, 1, 0, 1, 0, 1, \dots$	$1 + (-1)^n$	$\cosh(z) = \frac{1}{2}(e^z + e^{-z})$
$0, 1, 0, 1, 0, 1, 0, \dots$	$1 - (-1)^n$	$\sinh(z) = \frac{1}{2}(e^z - e^{-z})$
$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \frac{1}{7}, \dots$	$\frac{1}{n+1}$	$\frac{e^z - 1}{z}$
$0!, 1!, 2!, 3!, 4!, 5!, 6!, \dots$	$n!$	$\frac{1}{1-z}$

Tabelle 4.3: Wichtige EGFs und ihre Folgen.

$$A(z) = \sum_{n=0}^{\infty} a_n \frac{z^n}{n!} \qquad B(z) = \sum_{n=0}^{\infty} b_n \frac{z^n}{n!}$$

$\int_0^z A(t)\,dt = \sum_{n=0}^{\infty} a_{n-1} \frac{z^n}{n!}$	(Rechts shift)
$A'(z) = \sum_{n=0}^{\infty} a_{n+1} \frac{z^n}{n!}$	(Links shift)
$zA(z) = \sum_{n=0}^{\infty} n a_{n-1} \frac{z^n}{n!}$	(Indexmultiplikation)
$\frac{A(z) - A(0)}{z} = \sum_{n=1}^{\infty} \frac{a_{n+1}}{n+1} \frac{z^n}{n!}$	(Indexdivision)
$A(\lambda z) = \sum_{n=0}^{\infty} \lambda^n a_n \frac{z^n}{n!}$	(Skalieren)
$A(z) + B(z) = \sum_{n=0}^{\infty} (a_n + b_n) \frac{z^n}{n!}$	(Addition)
$A'(z) - A(z) = \sum_{n=0}^{\infty} (a_{n+1} - a_n) \frac{z^n}{n!}$	(Differenz)
$A(z)B(z) = \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k} \right) \frac{z^n}{n!}$	(Binomialfaltung)
$e^z A(z) = \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} \binom{n}{k} a_k \right) \frac{z^n}{n!}$	(Binomialsumme)

Tabelle 4.4: Einige Operationen mit EGFs. Wir definieren wieder $a_n = b_n = 0$ für $n < 0$.

4.4 Die symbolische Methode

In diesem Unterkapitel lernen wir etwas systematischer, wie man Objekte mit erzeugenden Funktionen zählen kann, ohne den Umweg über eine Rekursionsgleichung zu machen. Wir konzentrieren uns dabei auf *rekursive* Objekte.

Für die Konstruktion von Mengen von Objekten lassen wir folgende Möglichkeiten zu, wobei $\mathcal{M}$ die ganze Menge bezeichnet und $\mathcal{M}_1$ und $\mathcal{M}_2$ bereits definierte Mengen sind.

1. atomares Objekt: $\mathcal{M} = \{x\}$
2. Paarbildung: $\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2$
3. Vereinigung: $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2$
4. Folgen von Objekten: $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_1 \times \mathcal{M}_1 \cup \mathcal{M}_1 \times \mathcal{M}_1 \times \mathcal{M}_1 \cup \cdots$

Sagen wir, die Größe eines Objekts ist die Anzahl der Atome, die es enthält. Eine genaue Definition der Größe können wir durch den rekursiven Aufbau der Objekte angeben:

$$\begin{aligned}
|x| &= f(x) && \text{(atomares Objekt)}\\
|(x,y)| &= |x| + |y| && \text{(Paarbildung)}\\
|(x_1, x_2, \ldots, x_m)| &= |x_1| + \cdots + |x_m| && \text{(Folgen)}
\end{aligned}$$

Die Größe eines atomaren Objekts können wir dabei beliebig festlegen. Wir sind nun aber interessiert in der Anzahl von Objekten einer bestimmten Größe in einer gegebenen Menge von Objekten.

Sei $M(z)$ die erzeugende Funktion der Folge, deren $n$tes Glied die Anzahl der Objekte der Größe n in der Menge $\mathcal{M}$ angibt und $M_1(z)$ und $M_2(z)$ die entsprechenden erzeugenden Funktionen für die Mengen $\mathcal{M}_1$ und $\mathcal{M}_2$. Dann gilt:

$$\begin{aligned}
\mathcal{M} = \emptyset &\Longrightarrow M(z) = 0\\
\mathcal{M} = \{x\} &\Longrightarrow M(z) = z^{|x|}\\
\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2 &\Longrightarrow \mathcal{M}(z) = \mathcal{M}_1(z) + \mathcal{M}_2(z) \text{ falls } \mathcal{M}_1 \cap \mathcal{M}_2 = \emptyset\\
\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2 &\Longrightarrow \mathcal{M}(z) = \mathcal{M}_1(z)\mathcal{M}_2(z)\\
\mathcal{M} = \bigcup_{k=1}^{\infty} \mathcal{M}_1^k &\Longrightarrow \mathcal{M}(z) = \frac{\mathcal{M}_1(z)}{1 - \mathcal{M}_1(z)}
\end{aligned}$$

Beispiel: Binärbäume lassen sich definieren als

$$\mathcal{B} = \square \cup \bigcirc \times \mathcal{B} \times \mathcal{B},$$

mit $|\square| = 0$ und $|\bigcirc| = 1$. Daraus erhalten wir folgende Funktionalgleichung für die erzeugende Funktion $B(z)$:

$$B(z) = 1 + zB(z)^2$$

Wir haben diese Gleichung schon früher gelöst.

Beispiel: Geordnete Bäume mit beliebigem Grad können wir definieren als

$$\mathcal{T} = \square \cup \bigcirc \times \bigcup_{k=1}^{\infty} \mathcal{T}^k,$$

woraus sich sofort folgende Gleichung ergibt:

$$T(z) = 1 + \frac{zT(z)}{1 - T(z)}$$

Lösen wir die Gleichung nach $T(z)$ auf erhalten wir

$$T(z) = 1 - \frac{z}{2} \pm \frac{1}{2}\sqrt{z(z-4)}.$$

Leider ist diese Funktion im Ursprung nicht analytisch und besitzt keine Entwicklung in eine Potenzreihe. Wie ist das möglich?

Zählen wir einfach einmal selbst nach. Es gibt nach unserer Definition genau einen Baum der Größe 0, nämlich den Baum, der nur aus einem externen Knoten besteht. Wieviele Bäume der Größe 1 gibt es? Aha, offensichtlich gibt es hier unendlich viele, nämlich die Wurzel mit einem, zwei, drei, usw. externen Knoten als Kinder. Mit erzeugenden Funktionen können wir nur Folgen darstellen, deren Glieder alle endlich sind.

Formulieren wir also die Frage um: Wieviele Bäume gibt es mit n Knoten, wobei wir interne und externe Knoten zählen?

Jetzt erhalten wir die Funktionalgleichung

$$T(z) = z + \frac{zT(z)}{1 - T(z)},$$

welche so aussieht, wenn wir sie nach $T(z)$ auflösen:

$$T(z) = \frac{1}{2} \pm \frac{1}{2}\sqrt{1 - 4z}$$

In eine Potenzreihe entwickelt:

$$T(z) = \frac{1}{2} + \frac{1}{2}\sum_{n=0}^{\infty}\binom{1/2}{n}(-4)^n z^n$$

Für $n > 0$ erhalten wir also

$$[z^n]T(z) = -\frac{1}{2}\binom{1/2}{n}(-4)^n =$$

Wir haben bisher angenommen, daß sich alle Atome in einem zusammengesetzten Objekt sich nicht unterscheiden. Zum Beispiel erhalten wir je ein Objekt der Größe n für folgende rekursive Definition:

$$\mathcal{A} = \{x\} \cup \{x\} \times \mathcal{A}$$

Die Menge $\mathcal{A}$ besteht hier einfach aus allen n-Tupeln $(x, x, x, \ldots, x)$ und für jede Größe gibt es genau eines.

Würden sich aber die Atome unterscheiden, hätten wir $n!$ verschiedene n-Tupel.

Wir nennen die bisher untersuchten Objekte *unmarkierte Objekte* und Objekte, in welchen sich die Atome unterscheiden *markierte Objekte*.

Für diese *markierten Objekte* bieten sich EGFs an, denn es gibt um einen Faktor $n!$ mehr markierte Objekte als entsprechende unmarkierte Objekte (wenn das Objekt insgesamt n Atome enthält).

Wir können zur Anschauung beispielsweise annehmen, daß die Atome in einem Objekt gerade $x_1, x_2, \ldots, x_n$ sind. Wenn wir diese Vorstellung haben, müssen beim Zusammensetzen von Objekten diese Bezeichnungen natürlich entsprechend geändert werden.

Sei nun $\hat{M}(z)$ die EGF

$$\hat{M}(z) = \sum_{n=0}^{\infty} m_n \frac{z^n}{n!},$$

wobei m_n die Anzahl verschiedener markierter Objekte in der Menge $\mathcal{M}$ ist.

$$\begin{aligned}
\mathcal{M} = \emptyset &\Longrightarrow \hat{M}(z) = 0 \\
\mathcal{M} = \{x\} &\Longrightarrow \hat{M}(z) = z^{|x|} \\
\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2 &\Longrightarrow \hat{M}(z) = \hat{M}_1(z) + \hat{M}_2(z) \text{ falls } \mathcal{M}_1 \cap \mathcal{M}_2 = \emptyset \\
\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2 &\Longrightarrow \hat{M}(z) = \hat{M}_1(z)\hat{M}_2(z) \\
\mathcal{M} = \bigcup_{k=1}^{\infty} M_1^k &\Longrightarrow \hat{M}(z) = \frac{\hat{M}_1(z)}{1 - \hat{M}_1(z)}
\end{aligned}$$

Die EGFs verhalten sich also bei markierten Objekten gerade so, wie sich OGFs bei unmarkierten Objekten verhalten.

4.5 Mittlere Stackgröße

Rechnen wir nun ein kleines Beispiel. Wir nehmen an, ein Algorithmus löst ein Problem der Größe n auf folgende Weise: Wenn $n = 1$, dann kann er das Problem direkt lösen. Wenn $n > 1$, dann teilt es das Problem in zwei Teile mit den Größen m und $n - m$ mit $1 \leq m \leq n - 1$ und lößt rekursiv zuerst das Problem der Größe m und dann das zweite Problem der Größe $n - m$. Während der Algorithmus das erste Problem lößt, merkt er sich auf dem Keller, welches zweite Problem er noch lösen muß. Während er das zweite Problem löst, merkt er sich die Lösung des ersten Problems auf dem Keller. Sind dann beide Probleme gelöst kann der Algorithmus aus beiden Teillösungen direkt die Lösung des ursprünglichen Problems bilden.

Wir interessieren uns dafür, wieviel Speicher für den Keller benötigt wird. Genauer gesagt, wir interessieren uns für den durchschnittlichen Speicherverbrauch im Stack gemittelt über die gesamte Ausführungszeit des Algorithmus. Zusätzlich nehmen wir an, daß jede mögliche Rekursionsstruktur mit gleicher Wahrscheinlichkeit vorkommt (was nicht damit gleichbedeutend ist, daß wenn ein Problem in Teilprobleme zerteilt wird, stets alle Teilungsmöglichkeiten gleich wahrscheinlich sind).

Den Ablauf des Algorithmus können wir uns als Binärbaum vorstellen: Jeder innere Knoten entspricht der Lösung eines Teilproblems, dessen Größe mindestens 2 ist und die externen Knoten Teilproblemen der Größe 1. Der Baum hat also $n-1$ innere Knoten und n externe Knoten. Die Belegung des Kellers während eines Schrittes ist nun proportional zur Tiefe des Knotens dieses Schrittes im Baum. Die mittlere Belegung des Kellers ist daher

$$\frac{\xi(t) + \pi(t)}{n} = \frac{2\pi(t) + 2n}{n}$$

für einen Baum t mit n internen Knoten.

Es genügt daher folgende Frage zu beantworten: Wir groß ist $\pi(t)$ durchschnittlich für alle Binärbäume t mit n internen Knoten?

Hierzu berechnen wir einfach

$$p_n = \sum_{t \in \mathcal{T}} \pi(t)(|t| = n),$$

wobei $\mathcal{T}$ die Menge aller Binärbäume ist. Um die durchschnittliche externe Pfadlänge aller Binärbäume mit n externen Knoten zu erhalten müssen wir dann nur noch p_n durch die Anzahl solcher Bäume teilen. Diese Anzahl haben wir bereits berechnet — sie ist $b_n = \frac{1}{n+1}\binom{2n}{n}$.

Wir bestimmen nun p_n. Hierzu sei $P(z)$ die zugehörige OGF,

$$P(z) = \sum_{n=0}^{\infty} p_n z^n = \sum_{t \in \mathcal{T}} \pi(t) z^{|t|}.$$

Ferner werden wir die Anzahl b_n von Binärbäumen der Größe n benötigen und die zugehörige OGF $B(z)$,

$$B(z) = \frac{1 - \sqrt{1-4z}}{2z}.$$

Wenn $t \in t_1 \times t_2$, dann $\pi(t) = \pi(t_1) + \pi(t_2) + |t| - 1$. Wir zerlegen $P(z)$ entsprechend:

$$\begin{aligned} P(z) = \sum_{t=t_1 \times t_2 \in \mathcal{T}} \pi(t_1) z^{|t|} + \sum_{t=t_1 \times t_2 \in \mathcal{T}} \pi(t_2) z^{|t|} + \sum_{t=t_1 \times t_2 \in \mathcal{T}} |t| z^{|t|} - \sum_{t=t_1 \times t_2 \in \mathcal{T}} z^{|t|} = \\ = A + B + C - D, \end{aligned}$$

wobei

$$A = \sum_{t=t_1 \times t_2 \in \mathcal{T}} \pi(t_1) z^{|t|} \qquad B = \sum_{t=t_1 \times t_2 \in \mathcal{T}} \pi(t_2) z^{|t|}$$
$$C = \sum_{t=t_1 \times t_2 \in \mathcal{T}} |t| z^{|t|} \qquad D = \sum_{t=t_1 \times t_2 \in \mathcal{T}} z^{|t|}$$

Sehen wir uns nun nacheinander A, B, C und D genauer an. Am einfachsten ist allerdings D, welches sich auf den ersten Blick als das altbekannte $B(z)-1$ herausstellt.

$$\begin{aligned} A = \sum_{t=t_1 \times t_2 \in \mathcal{T}} \pi(t_1) z^{|t|} = \sum_{t=t_1 \times t_2 \in \mathcal{T}} \pi(t_1) z^{|t_1|+|t_2|+1} = \\ = z \sum_{t_1 \in \mathcal{T}} \pi(t_1) z^{|t_1|} \sum_{t_2 \in \mathcal{T}} z^{|t_2|} = z \sum_{t \in \mathcal{T}} \pi(t) z^{|t|} \sum_{t \in \mathcal{T}} z^{|t|} = zP(z)B(z) \end{aligned}$$

Wir können bei B genauso verfahren und erhalten wegen der Symmetrie zwischen t_1 und t_2 dann $A = B$. Die Potenzreihe C sieht nach einer differenzierten Reihe aus. Wir müssen sie nur ein wenig umformen.

$$C = \sum_{t=t_1 \times t_2 \in \mathcal{T}} |t| z^{|t|} = z \sum_{t=t_1 \times t_2 \in \mathcal{T}} |t| z^{|t|-1} = z\Big(\sum_{t=t_1 \times t_2 \in \mathcal{T}} z^{|t|} \Big)' = zB(z)'.$$

Insgesamt erhalten wir

$$P(z) = 1 + 2zP(z)B(z) + zB(z)' - B(z)$$

oder, aufgelöst nach $P(z)$,

$$\begin{aligned}
P(z) &= \frac{1-1/z}{\sqrt{1-4z}} + \frac{1/z-3}{1-4z} \\
&= \left(1-\frac{1}{z}\right)\sum_{n=0}^{\infty}\binom{-1/2}{n}(-4)^n z^n + \left(\frac{1}{z}-3\right)\sum_{n=0}^{\infty} 4^n z^n \\
&= \left(1-\frac{1}{z}\right)\sum_{n=0}^{\infty}\binom{2n}{n} z^n + \left(\frac{1}{z}-3\right)\sum_{n=0}^{\infty} 4^n z^n \\
&= \sum_{n=0}^{\infty}\binom{2n}{n} z^n - \sum_{n=0}^{\infty}\binom{2(n+1)}{n+1} z^n - 3\sum_{n=0}^{\infty} 4^n z^n + \sum_{n=0}^{\infty} 4^{n+1} z^n \\
&= \sum_{n=0}^{\infty} 4^n z^n - \sum_{n=0}^{\infty}\frac{3n+1}{n+1}\binom{2n}{n}
\end{aligned}$$

Aufgaben

4.1 Berechnen Sie aus der geschlossenen Form von p_n nun die durchschnittliche Stackhöhe. Verwenden dabei Sie die Stirlingsche Formel, um eine asymptotische Näherung von $\binom{2n}{n}$ zu erhalten. Geben Sie das Ergebnis möglichst genau an.

Kapitel 5

Asymptotische Abschätzungen

Oft genügt uns eine Näherungslösung, welche nur asymptotisch gültig ist, deren Abweichung von der tatsächlichen Lösung aber auch für kleine Zahlen nur wenig abweicht. Oft ist es gar nicht möglich, die genaue Lösung in geschlossener Form zu erhalten. Manchmal ist dieses wohl möglich, aber die resultierende Formel ist sehr kompliziert. Eine nur asymptotisch gültige, aber einfacher zu durchschauende Formel, kann dann nützlicher sein. Ein Beispiel für diesen Fall war die die mittlere interne Pfadlänge eines Binärbaums mit n internen Knoten. Die exakte und asymptotische Lösung, welche wir früher erarbeiteten, waren

$$4^n - \frac{3n+1}{n+1}\binom{2n}{n} = n\sqrt{\pi n} - 3n + O(\sqrt{n}).$$

Schließlich gibt es noch eine dritte Möglichkeit: Manchmal kann man mit wenig Aufwand eine gute Näherung berechnen, kann die exakte Lösung aber nur mit sehr großem Aufwand herausfinden. Wenn die Näherungslösung genügt, dann lohnt es sich nicht diesen Aufwand zu treiben.

Neben der O-Notation verwenden wir noch die Symbole $\sim$ und $\asymp$. Die Definition von $\asymp$ folgt später und $\sim$ wird auf zwei Arten verwendet.

Wir schreiben

$$f(n) \sim g(n) \iff f(n) = g(n) + o(g(n)).$$

Zum Beispiel ist $\ln(n + 1/n) \sim \ln(n)$ für $n \to \infty$, wobei wir letzteres weglassen, wenn es aus dem Zusammenhang erkennbar ist. Ein weiteres Beispiel ist $e^x \sim 1 + x$ für $x \to 0$.

Die zweite Möglichkeit um $\sim$ zu verwenden sind *asymptotische Expansionen*. In diesem Fall ist die rechte Seite der Relation eine Reihe.

Wir schreiben

$$f(n) \sim \sum_{k=0}^{\infty} g_k(n) \iff f(n) \sim \sum_{k=0}^{m} g_l(n) \text{ fr alle } m \in \mathbf{N}. \tag{5.1}$$

Eine asymptotische Expansion bietet also eine ganze Folge von immer genauer werdenden Näherungen an $f(n)$ an.

Die einfachste Weise, zu asymptotischen Expansionen zu kommen, ist die Darstellung einer Funktion als Potenzreihe. In diesen Fällen können wir statt $\sim$ sogar das Symbol $=$ verwenden. Im allgemeinen können wir in einer asymptotischen Expansion $\sim$ aber *nicht* durch $=$ ersetzen. Die Reihe in (**??**) kann divergent sein und ist es oft auch.

5.1 Die Eulersche Summationsformel

Die Eulersche Summationsformel beruht darauf, eine Summe durch ein Integral zu ersetzen.

Theorem 7 (Eulersche Summationsformel)
Wenn $\int_1^\infty |f^{(i)}(x)|\,dx$ für $1 \le i \le 2m$ existiert, dann gilt

$$\sum_{k=1}^{n} f(k) = \int_1^\infty f(x)\,dx + \frac{1}{2}f(n) + C + \sum_{k=1}^{m} \frac{B_{2k}}{(2k)!} f^{(2k-1)}(n) + R_m,$$

wobei $R_m = O\left(\int_n^\infty |f^{(2m)}(x)|\,dx\right)$ und $B_k = n![z^n]z/(e^z-1)$ die Bernoulli-Zahlen sind:

n	0	1	2	3	4	5	6
B_n	1	$-\frac{1}{2}$	$\frac{1}{6}$	0	$-\frac{1}{30}$	0	$\frac{1}{42}$

Theorem 8

$$\begin{array}{cccccccc} n & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ B_n & 1 & -\frac{1}{2} & \frac{1}{6} & 0 & -\frac{1}{30} & 0 & \frac{1}{42} \end{array} \tag{5.2}$$

Wenn stets $f^{(i)} = o(f^{(i+1)})$, dann liefert die Eulersche Summationsformel eine asymptotische Expansion. Wir rechnen zwei Beispiele.

Die harmonischen Zahlen sind durch die Summe

$$H_n = \sum_{k=1} \frac{1}{k}$$

definiert. Wir setzen also $f(x) = 1/x$ und müssen nun die $i$te Ableitung von $f(x)$ berechnen. Dies ist hier sehr leicht und wir erhalten

$$f^{(i)}(x) = i!(-1)^i x^{-1-i}.$$

Die Bedingungen der Eulerschen Summationsformel sind erfüllt und wir erhalten durch Einsetzen

$$\begin{aligned} \sum_{k=1}^{n} \frac{1}{k} &\sim \int_1^n \frac{dx}{x} + \frac{1}{2n} + C + \sum_{k=1}^{m} \frac{B_{2k}}{(2k)!} \frac{-(2k-1)!}{n^{-2k}} \\ &\sim \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \cdots \end{aligned}$$

Die unbekannte Konstante wird γ genannt und läßt sich nicht auf einfache Weise durch andere bekannte mathematische Konstanten darstellen.

Als nächstes Beispiel wählen wir $n!$. Da $n!$ durch ein Produkt statt durch eine Summe definiert ist, wenden wir die Eulersche Summationsformel statt dessen an $\ln(n!)$ an.

$$\ln(n!) = \sum_{k=1}^{n} \ln(k) \sim \int_1^n \ln(x)\,dx + \tfrac{1}{2}\ln n + \ln\sigma + \textstyle\sum_{k=!}^{\infty} \frac{B_{2k}}{(2k)!} \frac{(2k-2)!}{n^{2k-1}} \tag{5.3}$$

$$\sim (n + \tfrac{1}{2}) \ln n - n + \ln\sigma + \tfrac{1}{12n} - \tfrac{1}{360n^3} + \cdots \tag{5.4}$$

Daraus erhalten wir die Näherung für $n!$. Es stellt sich heraus, daß $\sigma = \sqrt{2\pi}$.

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{5140n^3} + \cdots\right)$$

5.2 Singularitätenanalyse

Die Eulersche Summationsformel war eine Anwendung reeller Analysis. Die Methoden der reellen Analysis sind jedoch zu beschränkt. Wir wenden

$$\begin{aligned}
n! &= \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\frac{1}{12n}+\frac{1}{288n^2}+O(n^{-3})\right) \quad \text{(Stirlingsche Formel)}\\
H_n &= \ln n+\gamma+\frac{1}{2n}-\frac{1}{12n^2}+O(n^{-4}) \quad \text{(Harmonische Zahlen),}\\
&\qquad \text{mit } \gamma \approx 0.5772156649015328606065l209
\end{aligned}$$

Tabelle 5.1: Einige wichtige asymptotische Näherungen

uns nun der komplexen Analysis zu. Im folgenden interessieren wir uns in erster Linie für Näherungen der Koeffizienten einer OGF oder EGF.

Unser erstes Theorem wird am einfachsten anzuwenden sein, aber auch die ungenauesten Näherungen liefern. Wir beschäftigen uns zunächst nur mit dem *exponentiellen Wachstum* dieser Koeffizienten.

Wir definieren hierzu für eine Folge a_n und eine positive reelle Zahl

$$a_n \asymp K^n \iff \limsup_{n\to\infty} |a_n|^{1/n} = K.$$

Wir können $a_n \asymp K^n$ auch so ausdrücken: Für jedes $\epsilon > 0$, sei es noch so klein, gilt $|a_n| > (K-\epsilon)^n$ für unendlich viele n, und $|a_n| < (K+\epsilon)^n$ gilt für alle n bis auf endlich viele Ausnahmen.

Wir können auch sagen, daß $a_n = \theta(n)K^n$ für alle n, wobei $\theta(n)$ eine subexponentielle Funktion ist, also langsamer als jede Exponentialfunktion wächst.

Eine Funktion ist *analytisch* im Punkt z_0 wenn sich durch eine Potenzreihe $\sum_{n=0}^{\infty} f_n z^n$ in einer Umgebung um z_0 darstellen läßt. Hierbei ist z_0 eine komplexe Zahl und die Umgebung liegt in der komplexen Zahlenebene.

Die Funktion $z \mapsto 1/z$ ist in der ganzen Ebene außer im Ursprung analytisch. Ein Punkt, in welchem eine Funktion aufhört analytisch zu sein, heißt *Singularität* dieser Funktion. Die Funktion $z \mapsto 1/z$ hat also eine Singularität und zwar 0.

Wir nennen eine Singularität *dominant*, wenn sie eine Singularität mit minimalem Betrag ist. Hier hilft meist ein Satz von *Pringsheim*, welcher besagt, daß Potenzreihen mit nicht-negativen Koeffizienten stets eine positive, reelle dominante Singularität besitzen. Wir können in diesem Fall also auf der reellen Achse bleiben, wenn wir nach dem Betrag der dominanten Singularitäten suchen.

Theorem 9 *Die dominanten Singularitäten einer GF $f(z)$ bestimmen ihr exponentielles Wachstum: Es sei $f(z)$ eine GF und z_0 eine dominante Singularität mit $R = |z_0|$. Dann gilt*

$$[z^n]f(z) \asymp \left(\frac{1}{R}\right)^n .$$

Betrachten wir als Beispiele die EGF $S(z)$ und die OGF $U(z)$, in geschlossener Form dargestellt als

$$S(z) = \frac{1}{2-e^z} \text{ und } U(z) = \frac{1-z-\sqrt{(1-3z)(1+z)}}{2z}.$$

Die Funktion $S(z)$ hat Singularitäten bei allen z, für welche $e^z = 2$ ist, d.h., $z = \ln(2) + 2k\pi i$ für alle $k \in \mathbf{Z}$. Die dominante Singularität ist $\ln 2$ und wir erhalten

$$n![z^n]S(z) \asymp n!\left(\frac{1}{\ln 2}\right)^n .$$

Bei $U(z)$ finden wir die Singularitäten bei jenen z, für welche der Ausdruck unter der Wurzel 0 wird; dieses sind $\frac{1}{3}$ und -1. Bei 0 hat $U(z)$ keine Singularität, denn $\lim_{z\to 0} U(z) = \frac{1}{2}$. Die dominante Singularität ist $\frac{1}{3}$ und daher

$$[z^n]U(z) \asymp 3^n.$$

5.3 Meromorphe Funktionen

Das exponentielle Wachstum läßt sich leicht für beliebige GFs bestimmen. Die nächste Formel liefert genauere Näherungen, ist aber nur für eine Unterklasse aller Funktionen gültig — für die meromorphen Funktionen, welche wir jetzt definieren.

Eine Funktion $f(z)$ heißt in z_0 meromorph, wenn es Funktionen $g(z)$ und $h(z)$ gibt, so daß in einer Umgebung von

$$f(z) = \frac{g(z)}{h(z)}$$

gilt und $g(z)$ und $h(z)$ in z_0 analytisch sind. Außerdem darf natürlich nicht $h = 0$ sein.

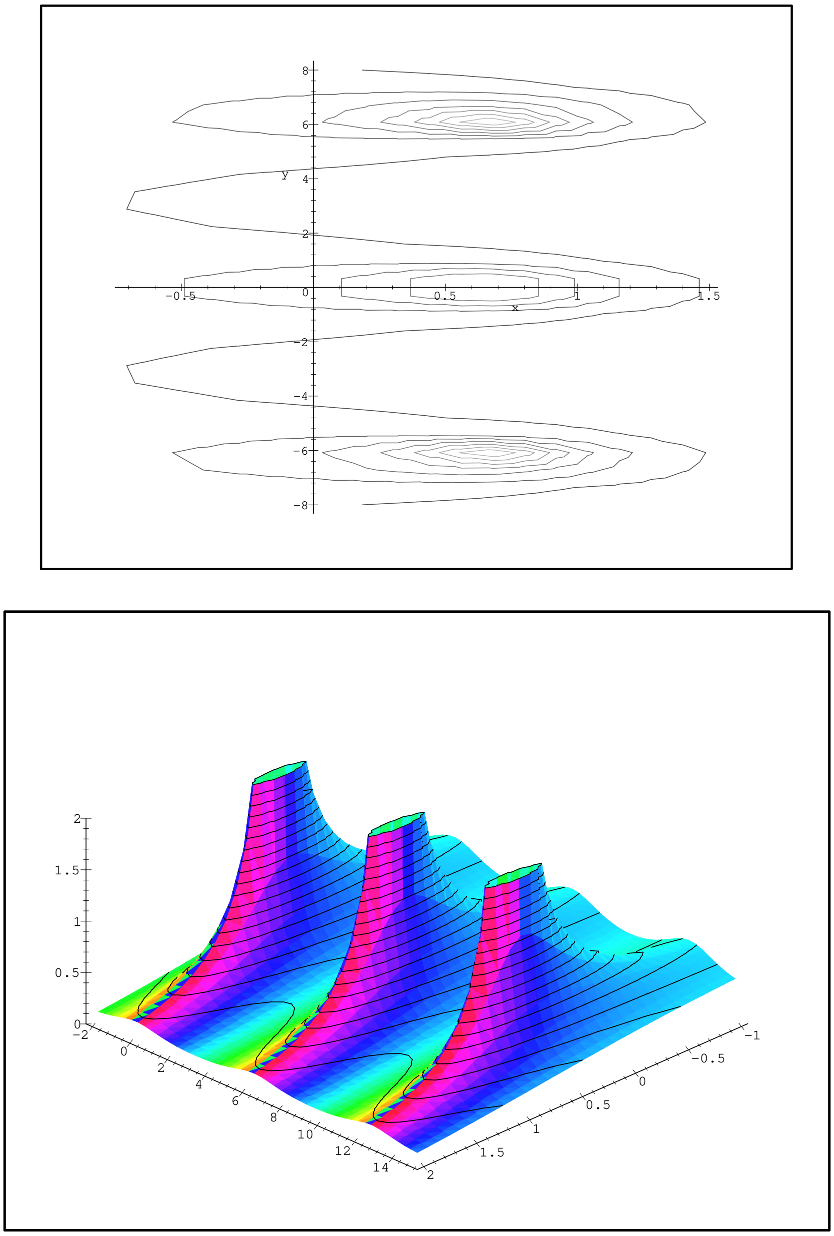

Abbildung 5.1: Die Funktion $S(z) = 1/(2 - e^z)$.

Es ist leicht einzusehen, daß $f(z)$ in z_0 genau dann meromorph ist, wenn es eine Entwicklung

$$f(z) = \sum_{n=-k}^{\infty} a_n(z-z_0)^n =$$
$$= \frac{a_{-k}}{(z-z_0)^k} + \frac{a_{-(k-1)}}{(z-z_0)^{k-1}} + \cdots + \frac{a_{-1}}{z-z_0} + \sum_{n=0}^{\infty} a_n(z-z_0)^n =$$
$$\frac{P(z)}{(z-z_0)^k} + \sum_{n=0}^{\infty} a_n(z-z_0)^n =$$

gibt, die in einer Umgebung von z_0 gültig ist. Dabei ist k eine beliebige natürliche Zahl und wir können annehmen, daß $a_{-k} \neq 0$ ist. Das Polynom $P(z)$ ist dabei

$$P(z) = \sum_{n=0}^{k-1} a_{n-k}(z-z_0)^n.$$

Wir sagen, daß die Funktion $f(z)$ in z_0 einen Pol $r$ter Ordnung hat, wenn $f(z)(z_0 - z)^r$ in z_0 analytisch ist, aber $f(z)$ selbst nicht. Dann hat $f(z)$ hier einen Pol $k$ter Ordnung. Wie zu erwarten, sagen wir daß eine Funktion in einem Gebiet G meromorph ist, wenn sie in jedem Punkt von G meromorph ist. Es ist jetzt aber klar, daß

$$f(z) - \sum_{n=-k}^{-1} a_n(z-z_0)^n = f(z) - \frac{P(z)}{(z-z_0)^k}$$

in z_0 keinen Pol mehr hat. Wenn f in einem Gebiet G meromorph ist, können wir also im Prinzip durch Addition einfacher Funktionen die Pole „verschwinden“ lassen. Es bleibt dann eine analytische Funktion übrig. Das Verhalten der Koeffizienten $[z^n]f(z)$ hängt aber dann im wesentlichen nur von den wegaddierten Polen ab, denn die übrigbleibende analytische Funktion hat dann asymptotisch langsamer wachsende Koeffizienten. Wir können diese Erkenntnis im folgenden Theorem zusammenfassen.

Theorem 10 *Es sei $f(z)$ meromorph für $|z| \leq R$. Innerhalb dieses Kreises habe $f(z)$ die Pole $\alpha_1, \alpha_2, \ldots, \alpha_m$. Außerdem sei $f(z)$ im Ursprung und für $|z| = R$ analytisch. Dann gibt es Polynome $P_1(z), \ldots, P_m(z)$, so daß*

$$[z^n]f(z) = \sum_{j=1}^{m} P_j(n)\alpha_j^{-n} + O(R^{-n}).$$

Der Grad von $P_j(z)$ ist um eins kleiner als die Ordnung des Pols α_j.

Die Polynome lassen sich leicht bestimmen. Wir nehmen wieder $S(z) = 1/(2-e^z)$ als Beispiel. Wählen wir $R = 6$, dann gibt es nur die eine Singularität $\ln 2$ innerhalb des Kreises $|z| = R$. Der Pol $\ln 2$ hat die Ordnung 1. Wir sehen uns nun an, wie sich $S(z)$ für $z \to \ln 2$ verhält.

Es gilt $e^z \sim 2 - 2\ln(2) + 2z$ für $z \to \ln 2$ und daher

$$\frac{1}{2-e^z} \sim \frac{1}{2}\frac{1}{\ln 2 - z} = \frac{1}{2\ln 2}\frac{1}{1-(1/\ln 2)z} = \frac{1}{2\ln 2}\sum_{n=0}^{\infty}\left(\frac{1}{\ln 2}\right)^n z^n$$

Das gesuchte Polynom $P_1(z)$ ist hier einfach $1/(2\ln 2)$. Im allgemeinen lassen sich alle $P_i(z)$ so bestimmen.

Die Lösung unseres Beispielproblems ist hiermit

$$[z^n]S(z) = \frac{1}{2}\left(\frac{1}{\ln 2}\right)^{n+1} + O(6^{-n}).$$

Betrachten wir ein weiteres Beispiel, welches belegt, daß man mit asymptotischen Methoden sehr leicht eine sehr gute Näherung berechnen kann, obwohl eine exakte Lösung unerreichbar scheint.

Ein bekanntes Beispiel aus der Wahrscheinlichkeitsrechnung sind die Matrosen, die ihre Kojen zufällig wählen. In dieser Geschichte gibt es n Matrosen, die sich auf einem Landurlaub betrinken und dann, auf das Schiff zurückgekehrt, betrunken eine beliebige Koje wählen. Es läßt sich nun leicht berechnen, daß die Wahrscheinlichkeit, daß ein bestimmter Matrose in seiner eigenen Koje schläft, gerade $1/n$ ist und damit im Mittel ein Matrose in das eigene Bett findet. Wie groß ist aber die Wahrscheinlichkeit, daß *kein* Matrose in seinem eigenen Bett schläft?

Eine Permutation, die keinen Fixpunkt enthält, wird *derangement* genannt. Die Matrosenfrage ist also verwandt mit dem Problem, die Anzahl D_n der *derangements* einer n-elementigen Menge zu bestimmen. Da ein *derangements* zwischen einem und n Fixpunkte entält und es genau $\binom{n}{k}D_{n-k}$ Permutationen mit *genau* k Fixpunkten gibt, ist es sehr leicht eine Rekursionsgleichung für D_n anzugeben:

$$D_n = n! - \sum_{k=1}^{n}\binom{n}{k}D_{n-k}$$

Leicht umgeschrieben bekommt sie diese einfachere Form:

$$n! = \sum_{k=1}^{n}\binom{n}{k}D_k$$

Da es sich um markierte Objekte handelt, versuchen wir die Gleichung mit einer EGF zu lösen. Die rechte Seite ist eine Binomialsumme und so erhalten wir direkt

$$\frac{1}{1-z} = e^z D(z)$$

und damit die EGF für D_n in geschlossenere Form:

$$D(z) = \frac{e^{-z}}{1-z}$$

Wir können jetzt sofort interessante Eigenschaften von D_n direkt ablesen. Zunächst fällt auf das es nur eine Singularität bei 1 gibt. Das heißt also $n![z^n]D(z) \asymp n!$ und damit $D_n \asymp n!$ oder, noch einmal anders ausgedrückt, daß das Verhältnis von D_n und $n!$ subexponentionell ist. Das ist schon einmal überraschend, denn es bedeutet daß es relativ viele Permutationen *derangements* sind. Die Wahrscheinlichkeit, daß alle Sehmänner in fremden Kojen liegen, muß also relativ hoch sein.

Wie hoch ist sie aber genau? Gehen wir in der Analyse weiter e^{-z} ist in der ganzen Gaußschen Zahlenebene analytisch und hat nur bei ∞ eine Singularität. Daher ist $D(z)$ meromorph und hat nur bei $z = 1$ einen Pol erster Ordnung. Also können wir D_n darstellen als

$$n![z^n]D(z) = P_1(n)\alpha_1^n + O(R^{-n}) = C \cdot 1^n + O(R^{-n})$$

wobei wir R beliebig groß wählen können. Die Konstante C ist das Polynom erster Ordnung und wir können C bestimmen, indem wir $D(z)$ für $z \to 1$ betrachten:

$$\frac{e^{-z}}{1-z} \sim \frac{1}{e}\sum_{n=0}^{\infty} z^n \text{ für } z \to 1$$

Unser Ergebnis ist jetzt schließlich

$$D_n = \frac{n!}{e} + O(n!\,\epsilon^n)$$

für jedes $\epsilon > 0$. Da der Fehlerterm sehr schnell fällt, können wir erwarten, daß die Näherung auch für kleine n sehr gut ist. Dafür gibt es aber keine Garantie, denn die versteckte Konstante im grossen O könnte ja gigantisch sein (und sie hängt natürlich von ϵ ab).

Tabelle 5.2 zeigt, daß die Näherung in der Tat exzellent ist.

Betrachten wir als weiteres Beispiel die erzeugende Funktion

$$L(z) = \sum_{n=0}^{\infty} L_n z^n = \frac{z(1-z)}{e^{z-1} - z},$$

n	D_n	$n!$	$n!/D_n$
2	1	2	2.000000000000000
3	2	6	3.000000000000000
4	9	24	2.666666666666667
5	44	120	2.727272727272727
6	265	720	2.716981132075472
7	1 854	5 040	2.718446601941748
8	14 833	40 320	2.718263331760264
9	133 496	362 880	2.718283693893450
10	1 334 961	3 628 800	2.718281657666404
11	14 684 570	39 916 800	2.718281842777827
12	176 214 841	479 001 600	2.718281827351874
13	2 290 792 932	6 227 020 800	2.718281828538486
14	32 071 101 049	87 178 291 200	2.718281828453728
15	481 066 515 734	1 307 674 368 000	2.718281828459379
16	7 697 064 251 745	20 922 789 888 000	2.718281828459026
17	130 850 092 279 664	355 687 428 096 000	2.718281828459046
18	2 355 301 661 033 953	6 402 373 705 728 000	2.718281828459045
19	44 750 731 559 645 106	121 645 100 408 832 000	2.718281828459045

Tabelle 5.2: Eine sehr gute Näherung für D_n ist $n!/e$. Der Wert von e auf 15 Dezimalstellen ist 2.718281828459045.

wobei L_n die Länge des n-ten Laufs einer zufälligen Folge von Zahlen ist. Es ist leicht zu sehen, daß $L_1 = 1 + \frac{1}{2} + \frac{1}{3!} + \frac{1}{4!} + \cdots = e - 1$ ist.

Was sind nun die Singularitäten von $L(z)$? Wir finden die dominante Singularität $z_0 = 1$ auf der positiven reellen Achse und die nächsten beiden konjugiert komplexen Singularitäten z_1 und $\bar{z}_1$ im rechten oberen bzw. unteren Quadranten. Da wir z mit $e^{z-1} = z$ suchen müssen simultan die beiden Gleichungen $e^{x-1} \cos y = x$ und $e^{x-1} \sin y = y$ für den Real- und Imaginärteil erfüllt sein. Das Diagramm in Abbildung **??** zeigt die Singularitäten als Schnittpunkte der Kurven, welche diese Gleichungen erfüllen.

Wir bestimmen nun die Ordnung des Pols $z_0 = 1$: Es stellt sich heraus, daß

$$\lim_{z\to 1} \frac{1-z}{e^{z-1} - z} = \lim_{z\to 1} \frac{-1}{e^{z-1} - 1}$$

nicht existiert, aber

$$\lim_{z\to 1} \frac{1-z}{e^{z-1} - z}(z-1) = \lim_{z\to 1} \frac{z^2 - 2z + 1}{z - e^{z-1}} = \lim_{z\to 1} \frac{2z-2}{1 - e^{z-1}} = \lim_{z\to 1} \frac{2}{-e^{z-1}} = -2$$

sehr wohl existiert. Daher ist die $z_0 = 1$ ein Pol erster Ordnung.

Außerdem hat $L(z)+2/(z-1)$ jetzt keine Singularität bei 1 mehr. Die nächsten Singularitäten sind weit weg: Es ist $|z_1| = |\bar{z}_1| = 8.07556\ldots > 8$ und daher ist

$$[z^n]L(z) = L_n = [z^n]\frac{2}{z-1} + O(8^{-n}) = 2 + O(8^{-n}).$$

Die Folge der L_n konvergiert also recht schnell gegen 2.

5.4 Algebraische Singularitäten

Eine Funktion $f(z)$ hat eine *algebraische Singularität* im Punkt z_0, wenn wir $f(z)$ schreiben können als

$$f(z) = h(z) + \sum_{j=1}^{m} \left(1 - \frac{z}{\alpha}\right)^{c_j} g_j(z),$$

wobei $h(z)$ und $g_j(z)$ analytisch in z_0 sind und $c_j \neq \{0, 1, 2, \ldots\}$.

Pole sind spezielle algebraische Singularitäten.

Leider können wir bei algebraischen Singularitäten nicht so gute Abschätzungen finden, wie bei meromorphen Funktionen. Wir begnügen uns mit dem folgenden Satz von Darboux, welcher es erlaubt mit wenig Aufwand eine asymptotische Abschätzung zu finden.

Theorem 11 *Es sei $f(z)$ analytisch für $|z| < R$ und habe auf $|z| = R$ nur algebraische Singularitäten. Es gelte*

$$f(z) = h(z) + \sum_{j=1}^{k} \left(1 - z/\alpha_j\right)^{c_j} g_j(z),$$

wobei $h(z)$ und alle $g_j(z)$ analytisch in $|z| \le R$ sind und α_j die Singularitäten auf $|z| = R$. Es sei $c_j \notin \{0, 1, 2, \ldots\}$. Wir definieren $a = \min\{\Re(\alpha_j) \mid 1 \le j \le k\}$. Dann gilt

$$[z^n]f(z) = -\sum_j \frac{g_j(\alpha_j)n^{-c_j-1}}{\Gamma(-c_j)\alpha_j^n} + o(R^{-n}n^{-a-1}),$$

wobei die Summe nur über alle j mit $\Re(c_j) = a$ geht.

Die Idee hinter der Methode von Darboux ist dieselbe, wie hinter der Behandlung von Polen: Falls $G(z)$ bei α eine algebraische Singularität hat, dann finden wir eine Funktion $H(z)$, die einerseits leicht in eine Potenzreihe entwickelt werden kann und andererseits ebenfalls bei α eine algebraische Singularität besitzt. Wenn wir das geschickt anstellen, dann wird $[z^n]G(z) \sim [z^n]H(z)$ gelten, weil $[z^n](G(z) - H(z))$ asymptotisch langsamer wächst. Leider können wir die algebraische Singularität auf diese Weise im allgemeinen nicht ganz verschwinden lassen, wie es bei Polen möglich war.

Nehmen wir uns zur Illustration jetzt die OGF

$$G(z) = \sqrt{(1-z)(1-\alpha z)}$$

vor, wobei $\alpha < 1$ gelten soll. Jetzt hat $G(z)$ die Singularitäten 1 und $1/\alpha$, wobei 1 die dominante ist. Die Singularität 1 ist algebraisch, da ja $G(z) = (1-z)^{1/2}R(z)$ ist, wobei dann $R(z)$ in 1 analytisch ist. Wir wollen jetzt eine Vergleichsfunktion finden, so daß $[z^n](G(z) - H(z))$ klein ist, so daß $H(z)$ eine gute Näherung ist. Hierfür sollte $H(z)$ eine algebraische Singularität bei 1 haben und sich in der Nähe von 1 wie $G(z)$ verhalten. Durch eine Taylorentwicklung an der Stelle 1 von $G(z)$ können wir eine schöne Funktion finden, die in der Nähe von 1 sich wie $G(z)$ verhält:

$$G(z) \sim (z-1)^{1/2}\sqrt{\alpha - 1} + (z-1)^{3/2}\frac{\sqrt{\alpha(\alpha-1)}}{2(\alpha-1)} - (z-1)^{5/2}\frac{\sqrt{\alpha^2(\alpha-1)}}{8(\alpha-1)^2}$$

Wählen wir also zunächst einmal $H(z) = (z-1)^{1/2}\sqrt{\alpha-1}$ und betrachten wir $G(z) - H(z) = (1-z)^{1/2}(\sqrt{1-\alpha z} - \sqrt{1-\alpha})$. Der zweite Faktor $\sqrt{1-\alpha z} -$

$\sqrt{1-\alpha}$ hat jetzt bei 1 keine Singularität mehr. Ganz im Gegenteil: Es gilt $\sqrt{1-\alpha z} - \sqrt{1-\alpha} \sim 0$ für $z \to 1$. Das haben wir auch erwartet und sogar

$$\frac{\sqrt{1-\alpha z} - \sqrt{1-\alpha}}{1-z}$$

hat keine Singularität bei 1. Daher können wir $G(z) - H(z)$ als

$$G(z) - H(z) = (1-z)^{3/2} \frac{\sqrt{1-\alpha z} - \sqrt{1-\alpha}}{1-z}$$

schreiben. Ebenso wie $G(z)$ hat $G(z) - H(z)$ eine algebraische Singularität bei 1, aber die Ordnung ist 3/2 statt 1/2. Wir werden nun sehen, daß die Koeffizienten von $G(z) - H(z)$ asymptotisch wesentlich kleiner sind als die Koeffizienten von $G(z)$. Dadurch wird $[z^n]H(z)$ eine gute Näherung für $[z^n]G(z)$.

Um $[z^n](G(z) - H(z)$ abzuschätzen, schreiben wir $G(z) - H(z) = A(z)B(z)$ mit $A(z) = (1-z)^{3/2}$ und $B(z) = (\sqrt{1-\alpha z} - \sqrt{1-\alpha})/(1-z)$. Die dominante Singularität für $B(z)$ ist $1/\alpha$ und daher gilt $[z^n]B(z) \asymp \alpha^{-n}$ oder $[z^n] = O(r^{-n})$ für ein $r > 1$ (und $r < 1/\alpha$).

Den Koeffizienten für $A(z)B(z)$ erhalten wir als Faltung:

$$[z^n]A(z)B(z) = \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k}$$

Am Anfang der Summe dominieren die a_k und am Ende die b_{n-k}, was eine Zerlegung der Summe in wenigstens zwei Teile empfiehlt. Für b_n haben wir bereits die sehr starke Abschätzung $b_n = O(r^{-n})$ und wir erhalten eine Abschätzung für a_n, indem wir $A(z)$ nach der Newton-Formel entwickeln:

$$a_n = \binom{3/2}{n}(-1)^n = \binom{n-5/2}{n} = O(n^{-5/2})$$

Jetzt können wir die Teilsummen großzügig abschätzen:

$$\sum_{k=0}^{\lceil n/2 \rceil} a_k b_{n-k} = O(r^{-n/2})$$
$$\sum_{\lfloor n/2 \rfloor}^{n} a_k b_{n-k} = O(n^{-5/2})$$

Insgesamt haben wir $[z^n](G(z) - H(z)) = O(r^{-n/2}) + O(n^{-5/2}) = O(n^{-5/2})$ und damit $[z^n]G(z) = [z^n]H(z) + O(n^{-5/2})$. Da $H(z)$ sich direkt expandieren läßt, erhalten wir folgendes Ergebnis:

$$[z^n]G(z) = \sqrt{\alpha - 1}\binom{1/2}{n}(-1)^n + O(n^{-5/2}) = \sqrt{\alpha - 1}\frac{n^{-3/2}}{\Gamma(1/2)} + O(n^{-5/2})$$

Betrachten wir $n = 50$ und $\alpha = 1/2$. Der genaue Wert von $[z^n]G(z)$ ist dann

$$-\frac{998278640117642127792954588191043963044 31}{178405961588244985132285746181186892047843328} = -5.595545\ldots\times 10^{-4}$$

und

$$\sqrt{1 - 1/2}(-1)^{50}\binom{1/2}{50} = -5.684655\ldots \times 10^{-4}.$$

5.5 Die Sattelpunktmethode

Im letzten Abschnitt befassen wir uns mit erzeugenden Funktionen, die keine Singularitäten besitzen. Für solche Funktionen haben wir noch kein allgemeines Verfahren, welches die Koeffizienten bestimmt.

Wir behelfen uns hier mit dem Residuumtheorem, einem Satz der Funktionentheorie, aus welchem folgt:

$$g_n = \frac{1}{2\pi i}\int_\Gamma \frac{G(z)}{z^{n+1}}dz,$$

wobei Γ eine geschlossene Kurve ist, die einmal den Ursprung im mathematisch positivem Sinn umläuft.

Warum gilt dieser Satz? Betrachten wir einmal einen Kreis um den Ursprung:

$$\int_C \frac{1}{z^{n+1}} = \int_0^{2\pi} e^{-(n+1)\phi i} de^{\phi i} = \int_0^{2\pi} e^{-(n+1)\phi i} ie^{\phi i} d\phi =$$
$$i\int_0^{2\pi} e^{n\phi i} d\phi = \begin{cases} 2\pi i & \text{falls } n = 0 \\ \frac{1}{n}e^{n\phi i}\Big|_{\phi=0}^{2\pi} = 0 & \text{sonst} \end{cases}$$

Es stellt sich also heraus, daß nur für $n = 0$ das Integral nicht verschwindet.

Wegen der Linearität des Integrals erhalten wir daraus direkt das Residuumtheorem.