4 Effizienz

Programme und Algorithmen, die man in der Praxis verwendet, sollten möglichst effizient arbeiten, d.h., sie sollten ein vorgegebenes Problem mit möglichst geringem Einsatz an „Betriebsmitteln“ wie Zeit, Speicher, Ein-Ausgabeeinheiten, Hilfsprogrammen, Datenübertragungseinrichtungen usw. lösen. Man sagt auch: Programme und Algorithmen sollten eine möglichst geringe Komplexität besitzen. Man erinnere sich: Bei dem Problem der Eintrittskartenverteilung aus Kapitel 2 („Algorithmen, Daten, Programme I“) war es das primäre Ziel, die benötigte Zeit für diesen Prozeß zu minimieren.

Umgangssprachliche Definition:
Die *Komplexität eines Algorithmus* ist der erforderliche Aufwand an Betriebsmitteln, den eine Implementierung des Algorithmus als Programm auf einem Computersystem benötigt.
Die *Komplexität eines Problems* ist die kleinstmögliche Komplexität eines Algorithmus, der das Problem löst.

Sei P ein Algorithmus, der ein Problem π löst. Dann ist die Komplexität von P offenbar eine *obere Schranke* für die Komplexität von π. Umgekehrt ist die Komplexität von π eine *untere Schranke* für die Komplexität von P.
Die wichtigsten Betriebsmittel, die in eine Komplexitätsberechnung einfließen, sind die *Laufzeit* und der *Speicherplatzbedarf*. Wir werden uns in dieser Vorlesung allerdings vorwiegend mit Laufzeitbetrachtungen beschäftigen. Unter „effizient“ verstehen wir daher im folgenden meist „in möglichst kurzer Zeit“.

In den folgenden Abschnitten wird das Problem der Komplexität genauer betrachtet. Wir werden ein Maß einführen, um die Komplexität von Programmen/Algorithmen zu bestimmen, einige Programme hinsichtlich ihrer Komplexität miteinander vergleichen und Probleme kennenlernen, die bis heute nicht effizient gelöst werden können.

4.1 Laufzeit und Speicherbedarf eines Algorithmus

Betrachten wir ein Problem π gegeben durch die funktionale Spezifikation

S_π: spec f: $X_\pi \to Y_\pi$ with
f(x)=y where
pre P(x)
post Q(x,y).

Beispiele: Die zugehörigen Begriffe erläutern wir im folgenden anhand dreier Beispiele.

1) σ sei das Problem, $n \geq 1$ ganze Zahlen $x_1,x_2,...,x_n$ aufsteigend zu sortieren. Dann ist

$X_\sigma = \{(x_1,...,x_n) \mid n \geq 1, x_k \in \mathbb{Z}, k=1,...,n\}$ und
$Y_\sigma = \{(x_{i_1},...,x_{i_n}) \mid x_j \in \mathbb{Z}, n \geq 1, \text{ mit } x_{i_1} \leq x_{i_2} \leq ... \leq x_{i_n}\}$.

f_σ: $X_\sigma \to Y_\sigma$ liefert zu jeder Zahlenfolge $x \in X_\sigma$ die zugehörige aufsteigend sortierte Folge $f_\sigma(x)=y \in Y_\sigma$.

2) μ sei das Problem, zwei natürliche Zahlen a und b zu multiplizieren. Dann ist
$X_\mu=\{(a,b) \mid a,b \in IN\}$ und $Y_\mu=IN$.
f_μ liefert zu je zwei natürlichen Zahlen a,b das Produkt $f_\mu(a,b)=a*b$.

3) Sei A eine endliche Menge. η sei das Problem, zu einem Element $x \in A$ und einem Wort $w \in A^*$ festzustellen, ob x in w vorkommt oder nicht. Dann ist:
$X_\eta=\{(x,w) \mid x \in A$ und $w \in A^*\}$, $A_\eta=\{$„ja“, „nein“$\}$.
f_η liefert zu jedem Paar (x,w) die Antwort „ja“, falls x in w vorkommt, und sonst „nein“.

Definition A:
Sei P eine Implementierung von S_π. Dann bezeichnen wir für $x \in X_\pi$ mit $\tau_P(x)$ bzw. $\sigma_P(x)$ die **Laufzeit** bzw. den neben der Eingabe zusätzlichen **Speicherplatz**, den P benötigt, um die korrekte Antwort auf die Eingabe x zu ermitteln. τ_P bzw. σ_P sind folglich Abbildungen der Funktionalität $X_\pi \to IN$.

$\tau_P(x)$ setzt sich zusammen aus der Anzahl der elementaren Einzelschritte, die man durchführen muß, um f(x) zu ermitteln, und der für jeden Einzelschritt benötigten Zeit. τ_P kann man auch experimentell für jede beliebige Fragestellung z.B. mit einer Stoppuhr als Anzahl der Sekunden bestimmen. $\sigma_P(x)$ ist die Zahl der Speicherzellen, die P für die Ermittlung des Ergebnisses zusätzlich zur Eingabe benötigt. Da die Eingabe nahezu immer vollständig im Speicher abgelegt werden muß, rechnet sie beim verbrauchten Speicherplatz nicht mit.

Im allgemeinen interessiert man sich aber nicht für die Laufzeit oder den Speicherplatz eines Programms für konkrete Eingaben – die tabellarische Auflistung von Laufzeit und Speicherbedarf für jede Eingabe wäre auch kaum möglich –, vielmehr möchte man wissen, wie sich das Programm qualitativ verhält, insbesondere wie sich die Laufzeit vergrößert, wenn man „schwierigere“ Eingaben vorgibt. Ein geeignetes Maß für die Schwierigkeit einer Eingabe ist offensichtlich ihre „Länge“: Zur Sortierung von 100 Zahlen benötigt man mehr Zeit, als zur Sortierung von 5 Zahlen, und die Multiplikation von 10-stelligen Zahlen dauert länger als die Multiplikation von 2-stelligen. Zugleich faßt man dadurch Eingaben in einer sinnvollen Weise zu Klassen zusammen. Es ist also zweckmäßig, die Laufzeit eines Programms in Beziehung zur „Länge der Eingabe“ zu setzen.

Was verstehen wir genau unter der Länge einer Eingabe?
Oft gibt es für jedes Problem eine *natürliche* Definition für die Länge einer Eingabe; z.B.: Die Länge des Multiplikationsproblems ist die Summe der Ziffernanzahl der beiden Faktoren. Für das Sortierproblem ist dies aber nicht mehr unmittelbar klar: Man könnte unter der Länge einer Eingabe die Anzahl der zu sortierenden Zahlen verstehen, also n, oder

die Summe über die Längen aller zu sortierenden Zahlen, also $|x_1|+...+|x_n|$. Dies ist eine Frage der Problemstellung. Die Länge $|x_i|$ jeder einzelnen Zahl x_i hängt andererseits wieder von der Darstellung der Zahlen ab. Man könnte jede Zahl a zum Beispiel durch a Striche darstellen; in diesem Fall wäre |a|=a. Wählt man dagegen die Darstellung zur Basis 2, dann ist $|a| \approx \log_2(a)$.

Formal assoziiert man mit jedem Problem π eine **Längenfunktion**

$$L_\pi: X_\pi \rightarrow IN,$$

die zu jeder Eingabe $x \in X$ ihre Länge $L_\pi(x)$ festlegt.

Beispiele:

1) Für das Sortierproblem σ setzt man:

   $L_\sigma: X_\sigma \rightarrow IN$ mit

   $L_\sigma(x_1,...,x_n)=n,$

   falls man nur an der Anzahl der Zahlen in der Folge, bzw.

   $L'_\sigma: X_\sigma \rightarrow IN$ mit

   $L'_\sigma(x_1,...,x_n)=|x_1|+...+|x_n|,$

   falls man an dem tatsächlich aufzuwendenden Platz für die Zahlenfolge interessiert ist. L' ist offenbar nur von geringem Interesse, da die Komplexität des Sortierens von n abhängt. n ist aber in L' nicht mehr zu erkennen; denn man erhält ja die gleiche Länge für eine einzige sehr lange Zahl und für viele kurze Zahlen.

2) Für das Multiplikationsproblem μ setzt man:

   $L_\mu: X_\mu \rightarrow IN$ mit

   L_μ(a,b)="Anzahl der Ziffern von a"+"Anzahl der Ziffern von b"
   +"zwei Vorzeichen".

   Mit Hilfe des Logarithmus zur Basis 10 kann man die Anzahl der Ziffern einer Zahl im Dezimalsystem exakt bestimmen. Es gilt nämlich:

   $[\log_{10} x]+2$="Anzahl der Ziffern von x"+"Vorzeichen",

   wobei der Ausdruck [u] die größte ganze Zahl kleiner oder gleich u bezeichnet. Für L_μ erhalten wir dann also:

   $L_\mu(a,b)=[\log_{10} a]+[\log_{10} b]+4.$

Wir haben nun vereinbart, die Laufzeit und den Speicherplatz eines Programms für ein Problem π in Abhängigkeit von der Länge der Eingabe zu bestimmen. Allerdings gibt es meist eine ganze Reihe von verschiedenen Eingaben, die alle die gleiche Länge besitzen, z.B. haben beim Sortierproblem alle Zahlenfolgen mit n Zahlen die gleiche Länge n. Für jede dieser Eingaben gleicher Länge wird der Algorithmus eine unterschiedliche Laufzeit besitzen. Um die Abhängigkeit zwischen Länge der Eingabe und Laufzeit bzw. Speicherplatz weiterhin *funktional* beschreiben zu können (d.h. jeder Längenangabe n wird *genau ein* Laufzeitwert T(n) zugeordnet), muß man sich zu jedem n für eine Laufzeit und einen Speicherplatzbedarf entscheiden. Zur Lösung dieses Dilemmas zieht man sich auf den

Begriff der Laufzeit im schlimmsten Fall (engl. *worst case*) zurück: Für jedes $n \in IN$ greift man sich die Eingab der Länge n heraus, für die das Programm die *größte* Laufzeit besitzt. Dies definiert die Laufzeitfunktion $T_P(n)$ und die Speicherplatzfunktion $S_P(n)$.

Definition B:
Sei π ein Problem mit Spezifikation S_π und P ein Programm, das S_π implementiert.
Die **Laufzeit** T_P **im schlimmsten Fall** (**worst case**) des Programms P ist eine Abbildung

T_P: $IN \to IN$ mit

$T_P(n)=\max\{\tau_P(x) \mid x \in X_\pi \text{ und } L_\pi(x)=n\}$.

Der **Speicherbedarf** S_P **im schlimmsten Fall** des Programms P ist eine Abbildung

S_P: $IN \to IN$ mit

$S_P(n)=\max\{\sigma_P(x) \mid x \in X_\pi \text{ und } L_\pi(x)=n\}$.

Wenn wir im folgenden von „Laufzeit“ oder „Speicherplatz“ sprechen, ist stets die Laufzeit und der Speicherbedarf im schlimmsten Fall gemeint.

Beispiel: Betrachten Sie folgendes Programm P. Das Programm liest eine ganze Zahl x ein und stellt fest, ob in der nachfolgenden Zahlenfolge die Zahl x vorkommt oder nicht. Das Programm löst also das Problem η aus Beispiel 3 für $A=\mathbb{Z}$.

```
program P(input,output);
var   x,y: integer;
      ende: boolean;
begin
      read(x);  {Zeit C}
      ende:=false;  {Zeit C'}
      while not (eof or ende) do  {Zeit C"}
      begin
            read(y);  {Zeit C}
            ende:=x=y  {Zeit C'"}
      end;
      if ende then writeln (‚Zahl ist vorhanden‘)
         else writeln (‚Zahl ist nicht vorhanden‘)  {Zeit C""}
end.
```

Wir berechnen die Laufzeit des Programms. Angenommen, für die Ausführung der einzelnen elementaren Anweisungen, den Vergleich in der while-Schleife und für die if-Anweisung werden die angegebenen Zeiten C, C', C", C'" und C"" benötigt. Die Länge der Eingabe sei die Anzahl der Zahlen in der zu durchsuchenden Folge plus Eins (für die gesuchte Zahl), also gilt für die Längenfunktion

L_η: $X_\eta \to IN$ mit $X_\eta=\{(x, y_1,...,y_n) \mid x,y_1,...,y_n \in \mathbb{Z}, n \geq 0\}$ und

$L_\eta(x, y_1,...,y_n)=n+1$.

Für eine Eingabe der Länge 1, d.h., es wird nur x eingelesen, und die Zahlenfolge ist leer, benötigt das Programm die Zeit C+C'+C"+C"", also

$T_P(1)=C+C'+C"+C""$.

Für eine Eingabe der Länge 2 (d.h., die zu durchsuchende Folge besteht nur aus einem Element) errechnet sich die Laufzeit zu

C+C'+C“+C+C'“+C“+C““=2C+C'+2C“+C'“+C““, also

$T_P(2)$=2C+C'+2C“+C'“+C““.

Für eine Eingabe der Länge 3 (d.h., die zu durchsuchende Folge besteht aus zwei Elementen) beträgt die Laufzeit

2C+C'+2C“+C'“+C““,

falls die gesuchte Zahl x als erste der beiden Zahlen auftritt. Tritt x als zweite Zahl auf oder gar nicht, so ist ein weiterer Schleifendurchlauf erforderlich, und es kommt die Zeit C+C“+C'“ hinzu. In diesem Falle beträgt also die Gesamtzeit

3C+C'+3C“+2C'“+C““.

Nach Definition B ist die Laufzeit im schlimmsten Fall definiert als das Maximum der Laufzeiten für eine vorgegebene Eingabelänge, daher gilt

$T_P(3)$=3C+C'+3C“+2C'“+C““.

Allgemein benötigt das Programm für eine Eingabe der Länge $n\geq 2$ je nachdem, an welcher Stelle die gesuchte Zahl x vorkommt, entweder die Zeit

2C+C'+2C“+C'“+C““, falls x an der 1. Stelle vorkommt, oder

3C+C'+3C“+2C'“+C““, falls x an der 2. Stelle vorkommt, oder

4C+C'+4C“+3C'“+C““, falls x an der 3. Stelle vorkommt, oder

...

nC+C'+nC“+(n-1)C'“+C““, falls x an der letzten Stelle oder gar nicht vorkommt.

Im schlimmsten Falle wird das Maximum aller dieser Zeiten benötigt. Daher gilt für die Laufzeit:

$T_P(n)$=nC+C'+nC“+(n-1)C'“+C““.

Diese Formel gilt auch für n=1 und beschreibt somit die Laufzeit-Komplexität des Programms P. Der Speicherbedarf ist konstant 3, also $S_P(n)=3$ für alle n.

Recht problematisch und unübersichtlich in obigem Beispiel sind die vielen Zeitkonstanten C, C' usw., die wir für jeden Vergleich und jede Elementaranweisung vergeben müssen. Bei größeren Programmen wird dadurch die Formel für $T_P(n)$ ziemlich unübersichtlich.

Man geht daher in der Praxis meist von einem **Einheitskostenmodell** aus. Das Einheitskostenmodell kann man sich als genormten Rechner vorstellen, auf dem Laufzeit und Speicherbedarf aller Programme ermittelt werden. In diesem Modell setzt man für die Ausführung jeder Elementaroperation **eine** Zeiteinheit an. Formeln für die Laufzeit vereinfachen sich dadurch erheblich. Was *ein* Rechenschritt ist, müßten wir eigentlich genauer präzisieren. Man entwirft hierzu meist ein sehr einfaches Maschinenmodell, das aber die wesentlichen Eigenschaften eines realen Computers widerspiegelt, legt den Befehlsvorrat und die Elementaroperationen fest und untersucht die Laufzeit von Programmen an-

hand dieses Modells. Wir rechnen im Einheitskostenmodell einfach für die üblichen elementaren Operationen, also Arithmetik, Vergleich, Zugriff zu Feldern, Zuweisung usw. jeweils einen Rechenschritt, also eine Zeiteinheit.
Die analoge Annahme trifft man für den Speicher: Jedes Datum kann in einer Speicherzelle untergebracht werden.
Das Einheitskostenmodell ist nur insoweit realistisch, als die auftretenden Operanden (z.B. in Zwischenrechnungen) nicht beliebig groß werden, sondern sich immer in der Größenordnung der Länge der Eingabe bewegen, was aber bei den meisten praktischen Problemen der Fall ist. Ausnahmen bilden arithmetische Algorithmen, wie z.B. Multiplikationsalgorithmen. Hierbei setzt man die Zeit für einen Rechenschritt in Beziehung zur Größe der beteiligten Operanden, also der Anzahl ihrer Ziffern. Diese Kostenrechnung führt auf das **logarithmische Kostenmodell**, logarithmisch deswegen, weil die Länge L(n) einer Zahl n in logarithmischem Verhältnis zu ihrem Wert steht, also

$$L(n)=[\log_2 n]+1.$$

Im logarithmischen Kostenmodell setzt man daher für eine elementare arithmetische Operation wie die Addition mit den Operanden a und b nicht einen, sondern

$$L(a)+L(b)=[\log_2 a]+[\log_2 b]+2$$

Rechenschritte als Zeiteinheiten an.
Wir verwenden in der Folge nur das Einheitskostenmodell.

Beispiel: In obigem Beispiel (Durchsuchen einer Zahlenfolge nach einer vorgegebenen Zahl) gilt im Einheitskostenmodell also (in C“ sind zwei Ausdrücke auszuwerten)

$$C=C'=C'''=1 \text{ und } C''=C''''=2.$$

Für die Laufzeit im schlimmsten Falle folgt dann

$$T_P(n)=4n+2.$$

Im allgemeinen werden wir daran interessiert sein, zu jedem Problem immer den schnellsten und bezgl. Speicherplatz anspruchslosesten Algorithmus zu verwenden. Die folgenden Überlegungen zeigen, daß es nicht immer ganz einfach ist, Algorithmen bzgl. ihrer Effizienz miteinander zu vergleichen. Angenommen, uns liegen vier Algorithmen A, B, C und D zu einem Problem π vor, die folgende Laufzeiten besitzen:

$$T_A(n)=100n+30,\ T_B(n)=100n\cdot\log_2 n,$$
$$T_C(n)=10n^2,\ T_D(n)=2^n.$$

Für $2\le n\le 9$ ist D am schnellsten, für $n=10$ ist C am schnellsten, und für $n>10$ ist A am schnellsten. B ist niemals der schnellste Algorithmus. Bis auf wenige (endlich viele) Ausnahmen empfiehlt es sich also stets, den Algorithmus A zu verwenden.
Um diese Verhältnisse zu beschreiben, definiert man zwei unterschiedliche Begriffe, um Algorithmen hinsichtlich ihrer Laufzeit zu vergleichen: Der erste formalisiert die natürliche Vorstellung von „schneller“, der zweite beschreibt die Situation, in der ein Algorithmus für Eingaben kurzer Länge zwar schneller ist als ein anderer, von einer gewissen Eingabe-

größe an jedoch der zweite stets den ersten übertrifft.

Definition C:
Gegeben seien zwei Algorithmen/Programme A und B. Dann gilt:

a) A ist **schneller** als B, falls

$$T_A(n) \leq T_B(n) \text{ für alle } n \in \mathbb{N} \text{ ist.}$$

b) A ist **asymptotisch schneller** als B, falls

$$\lim_{n\to\infty} \frac{T_A(n)}{T_B(n)} = 0.$$

Analog definiert man für den Speicherplatz.

Beispiel: Von den obigen Algorithmen A, B, C und D ist keiner schneller als einer der anderen, jedoch ist A asymptotisch schneller als B, B asymptotisch schneller als C und C asymptotisch schneller als D, denn es gilt z.B.:

$$\lim_{n\to\infty} \frac{T_A(n)}{T_B(n)} = \lim_{n\to\infty} \frac{100n+30}{100n\cdot\log_2 n} = 0.$$

Von den beiden obigen Algorithmen A' und B' mit den Laufzeiten

$$T_{A'}(n)=10n+3 \text{ und } T_{B'}(n)=7n+5$$

ist B' zwar schneller als A', aber nicht asymptotisch schneller, denn

$$\lim_{n\to\infty} \frac{T_{B'}(n)}{T_{A'}(n)} = \lim_{n\to\infty} \frac{7n+5}{10n+3} = \frac{7}{10} \neq 0.$$

Teil b) in Definition C ist offensichtlich eine schärfere Forderung als Teil a). Ist ein Programm asymptotisch schneller als ein anderes, so ist es bis auf endlich viele Ausnahmen „um eine Größenordnung“ schneller. In der Praxis verfolgt man daher meist das Ziel, ein asymptotisch schnellstes Programm zu finden. Man nimmt dann in Kauf, daß dieses Programm in wenigen Fällen möglicherweise nicht optimal ist.

4.2 Die Ordnung einer Funktion

Bei der Suche nach dem asymptotisch schnellsten Programm P interessiert nun nicht mehr für jedes Argument n der genaue Funktionswert $T_P(n)$, wichtig ist nur noch der qualitative Verlauf der Funktion, ihre *Größenordnung* oder kurz *Ordnung*, d.h., man wünscht sich Aussagen der Art: $T_P(n)$ oder auch $S_P(n)$ verhalten sich wie eine quadratische Funktion (also $T_P(n) \approx n^2$), oder $T_P(n)$ verhält sich wie der Logarithmus der Länge der Eingabe, also $T_P(n) \approx \log(L_P(n))$. Den Begriff der (Größen-) Ordnung wollen wir nun genauer untersuchen.

Definition A:
Sei f: IN→IN eine Funktion. Dann ist

$$O(f)=\{g: IN \to IN \mid \text{es gibt Zahlen } c, n_0 \in IN, \text{ so daß } g(n) \leq c \cdot f(n) \text{ für alle } n \geq n_0\}.$$

Falls $g \in O(f)$ ist, so hat g die **Ordnung** f. O heißt auch **Landausches Symbol**.
Man spricht O(f) als „groß O von f“ oder kurz „O von f“.

O(f) ist also die Menge aller Funktionen g: IN→IN, die von f für genügend große Argumente bis auf einen konstanten Faktor majorisiert werden.
Graphisch kann man sich die Verhältnisse folgendermaßen veranschaulichen. Seien f und g Funktionen der vorgeschriebenen Form und $g \in O(f)$. Abb. 1 zeigt Beispiele für f und g. Dann kann man f durch Multiplikation mit einer Konstanten c so nach oben verschieben, daß der Graph von f ab der Stelle n_0 stets oberhalb des Graphen von g liegt (Abb. 2). f soll im allgemeinen von möglichst einfacher Form sein, gleichzeitig aber eine vorgegebene Menge von Funktionen möglichst „glatt“ beschränken.

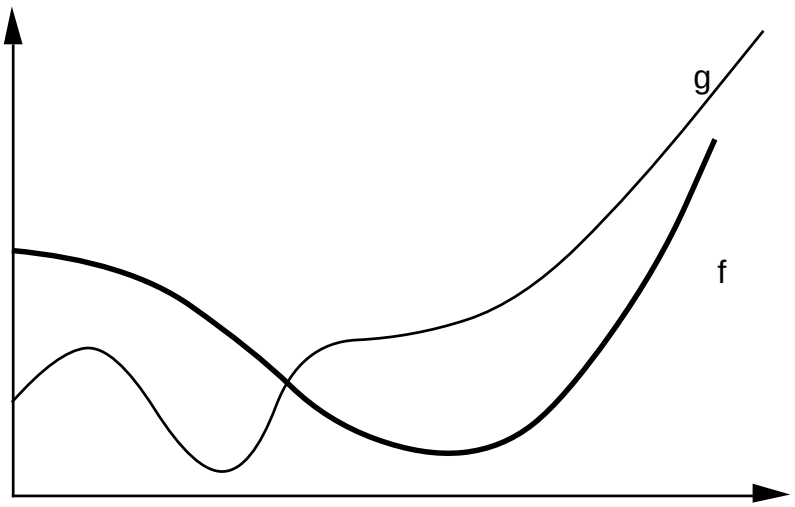

Abb. 1: Zwei Graphen f und g

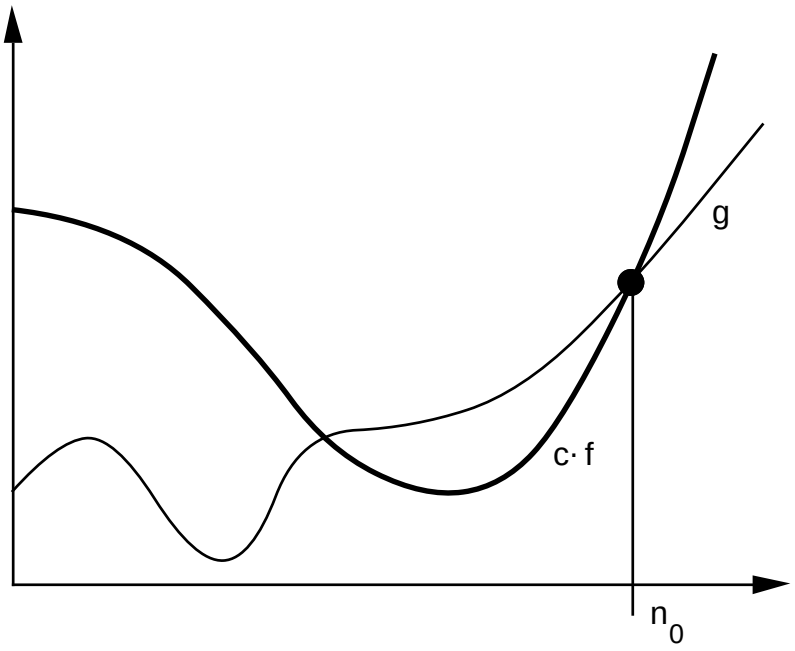

Abb. 2: Veranschaulichung von $g \in O(f)$

Meist unterscheidet man bei der O-Notation nicht zwischen einer Funktion und ihrer Darstellung durch Terme. Statt O(f) schreibt man direkt O(n), wenn f(n)=n ist, statt O(h) auch $O(n^2)$, wenn $h(n)=n^2$ ist, statt O(e) einfach O(1), wenn e die konstante Funktion 1 ist, also e(n)=1 für alle $n \in IN$, usw. Diese Konvention werden auch wir in den folgenden Beispielen einhalten.

Beispiele:

1) g_1 sei eine konstante Funktion, also für festes $a \in IN$:

   $g_1(n)=a$ für alle $n \in IN$.

   Offensichtlich gilt $g_1 \in O(id)$ mit id(n)=n für alle $n \in IN$, wenn man c=1 und $n_0=a$ in Definition A setzt; denn für $n \geq n_0=a$ ist $g_1(n)=a \leq n=id(n)$. Es gilt aber auch $g_1 \in O(1)$. Hierfür wähle man in Definition A die Werte $n_0=1$ und c=a. Dann gilt für alle $n \geq n_0$ die Beziehung

   $g_1(n)=a \leq c \cdot 1=c$.

2) g_2 sei ein Polynom vom Grad m, also

   $g_2(n)=c_m n^m+c_{m-1} n^{m-1}+...+c_2 n^2+c_1 n^1+c_0, \ c_m \geq 0$.

   Dann gilt $g_2 \in O(n^m)$. Beweis dieser Behauptung: Es gibt stets einen Wert n_0, der von m und den Koeffizienten $c_m, c_{m-1}, ... ,c_0$ abhängt, so daß für alle $n \geq n_0$ gilt:

   $c_m n^m \geq c_{m-1} n^{m-1}+...+c_2 n^2+c_1 n^1+c_0$,

   da n^m schneller wächst als die Summe der übrigen Potenzen. Dann ist insbesondere für $n \geq n_0$

   $g_2(n) \leq 2c_m n^m$,

also gilt die Behauptung nach Definition A mit diesem n_0 und $c=2c_m$.

Gegeben sei z.B. die Funktion $h(n)=3n^2+n$. Dann gilt $h\in O(n^2)$, aber auch $h\in O(n^k)$ für beliebiges $k>2$, oder auch $h\in O(n^2\cdot\log_2 n)$.

3) g_3 sei die Fakultätsfunktion, also

$$g_3(n)=n!=1\cdot 2\cdot 3\cdot\ldots\cdot n.$$

Jeder einzelne Faktor ist höchstens gleich n, daher folgt

$$g_3(n)\leq n^n \text{ und somit } g_3\in O(n^n).$$

4) g_4 sei der Logarithmus zur Basis $b\in IR$, $b>1$ beliebig, also

$$g_4(n)=\log_b n.$$

Nach elementaren Logarithmus-Rechenregeln gilt:

$$\log_b n=\log_b 2\cdot \log_2 n.$$

Offensichtlich gilt dann $g_4\in O(\log_2 n)$. In Definition A wählt man als Konstante $c=\log_b 2$.

Es hat sich eingebürgert, die O-Notation auch mit dem Gleichheitszeichen (bzw. Ungleichheitszeichen) statt nur mit den Mengensymbolen $\in$ und $\subseteq$ (bzw. $\notin$) zu verwenden. Anstelle von z.B.

$$5n^3+2n^2\in O(n^3+2n^2)\subseteq O(n^3)\subseteq O(n^7\cdot\log_2 n)$$

schreibt man kurz

$$5n^3+2n^2=O(n^3+2n^2)=O(n^3)=O(n^7\cdot\log_2 n).$$

Da es sich in dieser „Gleichung“ eben **nicht** um Gleichheitszeichen handelt, ist Vorsicht geboten: Man darf eine „Gleichung“, in der O-Ausdrücke vorkommen, nur von links nach rechts lesen. In umgekehrter Richtung gelesen ist die „Gleichung“ falsch, denn $O(n^7\cdot\log_2 n)$ ist nicht Teilmenge von $O(n^3)$. In einer „Gleichung“ der Form

$$f_1=O(f_2)=O(f_3)=\ldots=O(f_m)$$

repräsentieren die O-Ausdrücke von links nach rechts gelesen immer größere Mengen von Funktionen; die zugehörigen Schranken f_i werden bezüglich f_1 also gröber und gröber. Sprechweise für $f_1=O(f_2)$: f_1 ist von höchstens der Größenordnung f_2.

Aufgrund ihrer Ordnung teilt man Probleme und Algorithmen bzw. Programme in *Komplexitätsklassen* ein. Gilt für die Laufzeit T

$$T(n) = O(n),$$

so spricht man von (höchstens) *linearer Laufzeit* (kurz: *Linearzeit*). Entsprechend spricht man von *quadratischer Laufzeit*, wenn

$$T(n) = O(n^2),$$

und von *kubischer Laufzeit*, wenn

$$T(n) = O(n^3).$$

Zusammenfassend spricht man bei Problemen und Algorithmen bzw. Programmen von *polynomieller Laufzeit* (kurz: *Polynomialzeit*), wenn es ein $k\in IN$ gibt mit

$$T(n) = O(n^k).$$

Probleme/Algorithmen/Programme besitzen eine *exponentielle Laufzeit* (kurz: *Exponentialzeit)*, wenn es keine bessere Abschätzung gibt als

$$T(n) = O(2^{p(n)}),$$

wobei p eine Polynom von der Form ist:

$$p(n) = c_m n^m + c_{m-1} n^{m-1} + \ldots + c_2 n^2 + c_1 n^1 + c_0, \quad c_m > 0, \quad m \geq 1.$$

Die Laufzeit ist *superexponentiell*, wenn mindestens

$$T(n) = O(2^{2^{p(n)}}) \text{ gilt.}$$

Für die Praxis sind meist nur Probleme mit polynomieller Laufzeit akzeptabel. Solche Probleme nennt man manchmal auch *leicht*, alle übrigen Probleme heißen *hart* (oder *unzugänglich*). Harte Probleme sind praktisch nicht mehr algorithmisch zu lösen, denn selbst für relativ einfache Fragestellungen benötigt ein Algorithmus eine Rechenzeit, die nicht mehr zumutbar ist und leicht ein Menschenalter überschreiten kann. Auch schnellere Rechner helfen hierbei nicht mehr weiter. Tabelle 1 illustriert die unterschiedlichen Wachstumsraten und Laufzeiten einer polynomiellen und einer exponentiellen Funktion T. Gegeben sei dabei ein Computer, der 100.000 Rechenschritte in der Sekunde ausführen kann. Auf einem solchen Computer benötigt beispielsweise ein Programm mit der Laufzeitfunktion $T(n)=n^5$ für eine Eingabe x, die die Länge L(x)=50 besitzt, insgesamt 3125 Sekunden, also fast eine Stunde.

T(n) \ n	20	30	40	50	100	
n	0,0002	0.0003	0.0004	0.0005	0.001	Sekunden
n^2	0.004	0.009	0.016	0.025	0.1	Sekunden
n^5	32	243	1024	3125	100000	Sekunden
2^n	10 Sek.	3 Std.	4 Mon.	360 Ja.	$4 \cdot 10^{17}$	Jahre

Tab. 1: Polynomial- und Exponentialzeit

Schnellere Rechner machen den Unterschied zwischen Polynomial- und Nicht-Polynomialzeit noch deutlicher. Ein 100-fach schnellerer Rechner benötigt für den Exponentialzeit-Algorithmus in Tabelle 1 für n=40 immer noch eine hohe und für n=50 bzw. n=100 eine viel zu lange Rechenzeit. Man erkennt an diesem Beispiel, daß die Rechengeschwindigkeit eines Computers kaum dazu beiträgt, die Hürde zwischen Polynomialzeit und Exponentialzeit zu überspringen.

Die Erfahrung hat gezeigt, daß man für viele praktische Probleme einen Algorithmus finden kann, der das Problem in $O(n^k)$, $k \leq 3$, Schritten löst. Leider gibt es aber eine große Klasse von wichtigen Problemen, die sog. **NP-vollständigen Probleme**, für die man bisher keinen Polynomialzeitalgorithmus kennt. Es besteht sogar der berechtigte Verdacht, daß es zu diesen Problemen überhaupt keine Polynomialzeitalgorithmen gibt.

4.3 Beispiele: Suchen und Sortieren

Beispiel 1: **Suchen in einem Feld.**

Gegeben sei ein lineares Feld a mit n≥1 Elementen vom Typ integer und eine ganze Zahl x; a ist aufsteigend sortiert. Gesucht ist ein Programm, das ausgibt, ob x unter den Elementen von a vorkommt oder nicht. Dies entspricht dem Problem η aus Beispiel 3, Abschnitt 4.1.

1. Lösung: Wir vergleichen nacheinander alle Elemente von a mit x und stoppen, falls x gefunden bzw. das Ende des Feldes erreicht ist, ohne daß x gefunden wurde. Dieser Algorithmus heißt **sequentielles Suchen**:

```
program seqsuche(input,output);
const n=...;
var   a: array [1..n] of integer;
      i,x: integer;
      gefunden: boolean;
begin
      {Das Feld a sei gegeben}
      read(x); i:=1; gefunden:=false;
      while (not gefunden) and (i≤n) do
            if a[i]=x then gefunden:=true else i:=i+1;
      write(gefunden)
end.
```

Man erhält man als Laufzeit:

T(n)=4n+6, d.h. T(n)=O(n).

Speicherbedarf: S(n)=O(1).

Das sequentielle Suchverfahren ist also ein Linearzeitalgorithmus.

2. Lösung: Wir nutzen nun aus, daß das Feld bereits sortiert ist. Wir nehmen zunächst an, daß n eine gerade Zahl ist und somit auch n/2 eine natürliche Zahl ist. Falls n ungerade ist, dann ersetze man im folgenden überall n durch (n+1). Zuerst vergleichen wir x mit dem in der Mitte des Feldes stehenden Element, also mit a[n/2]. Entweder ist mit x=a[n/2] die Suche erfolgreich beendet, oder man wendet das Verfahren für x<a[n/2] *rekursiv* auf das linke Teilfeld von a[1] bis a[n/2-1] oder für x>a[n/2] auf das rechte Teilfeld von a[n/2+1] bis a[n] an, bis das gesuchte Element x gefunden oder das Teilfeld leer geworden ist. Abb. 3 verdeutlicht den ersten Schritt des Verfahrens für n=8, wenn man x=12 sucht.

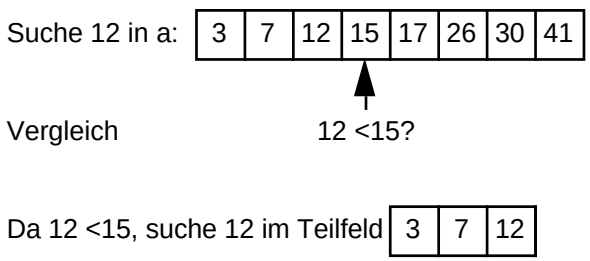

Abb. 3: Prinzip des binären Suchens

Dieses rekursive Suchverfahren nennt man **binäres Suchen**. Folgendes Programm realisiert das beschriebene Verfahren:

```
program binaersuche(input,output);
const n=...;
var   a: array [1..n] of integer;
      x: integer;
function binsuche(i,j,x: integer): boolean;
{binsuche sucht die Zahl x im Feld von a[i] bis a[j]}
var m: integer;
begin
      if j<i then binsuche:=false else
      begin
            m:=(i+j) div 2;
            if x<a[m] then binsuche:=binsuche(i,m-1,x) else
                  if x>a[m] then binsuche:=binsuche(m+1,j,x) else
                        binsuche:=true
      end
end;
begin
      {Das Feld a sei gegeben}
      read(x);
      write(binsuche(1,n,x))
end.
```

Welche Laufzeit T(n) benötigt dieses Programm im schlimmsten Fall? Der schlimmste Fall liegt vor, wenn x nicht in a vorkommt. Die Funktion binsuche teilt dann in jedem Durchlauf den Teil des Feldes von a[i] bis a[j], in dem x vermutet wird, in zwei etwa gleichgroße Hälften und sucht in der entsprechenden Hälfte weiter. Die Halbierung stoppt, wenn j<i geworden ist, das verbleibende Feld also die Länge 0 besitzt.

Jede Halbierung geschieht in konstant vielen Rechenschritten. Wir setzen hierfür C Rechenschritte an. Die rekursive Anwendung der Funktion binsuche auf ein Feld der Länge $^n/_2$ benötigt nach Annahme $T(^n/_2)$ Schritte. Somit erhalten wir für die Laufzeit T(n) die rekursive Gleichung

$$T(n)=C+T(^n/_2),$$

die zu lösen ist. Felder, die nur ein Element enthalten, können in konstanter Zeit C' nach x abgesucht werden, denn hierzu sind stets nur zwei Durchläufe von binsuche erforderlich. Diese Überlegung liefert die Gleichung (sog. *Anfangsbedingung*)

$$T(1)=C'.$$

Auf die genaue Größe der Konstanten C und C' kommt es auch hier wieder nicht an. Wir können daher einfach C=C' setzen. Die gewünschte geschlossene Formel für die Laufzeit T erhalten wir also durch Lösung der beiden Gleichungen

$$T(n)=C+T(^n/_2) \text{ für } n>1,$$

$$T(1)=C.$$

Zur Lösung dieser Rekursionsgleichung setzen wir $n=2^k$:

$T(2^k)=C+T(2^{k-1})$,

$T(2^0)=C$.

Offensichtlich gilt, wenn man $T(2^k)$ durch fortlaufendes Einsetzen der rechten Seite ausrechnet:

$T(2^k)=C(k+1)$.

Macht man die Ersetzung rückgängig, d.h., setzt man $k=\log_2 n$, so folgt

$T(n)=C(\log_2 n+1)$.

Gehen wir nun zur Ordnung über, so erhalten wir

$T(n)=O(\log_2 n)$.

Der Laufzeitgewinn der binären Suche gegenüber der sequentiellen Suche macht sich für große n gewaltig bemerkbar. So benötigt die Suche in einem Feld mit 1 Million Elementen im schlimmsten Fall größenordnungsmäßig

mit sequentieller Suche 1 Million Schritte,

mit binärer Suche 20 Schritte.

Zum Speicherbedarf: Auf den ersten Blick scheint der Speicherbedarf konstant zu sein, da nur ein paar Variablen zusätzlich zur Eingabe benötigt werden. Hinzu kommt jedoch noch die Größe des Stacks für die Implementierung der Rekursion. Offenbar wird bei einem Rekursionsaufruf jeweils eine konstante Zahl von Daten auf den Stack abgelegt, die aktuellen Parameter, die lokale Variable und die Rücksprungadresse. Die Stackgröße selbst hängt von der Zahl der geschachtelten rekursiven Aufrufe ab, die für einen Suchvorgang erforderlich sind. Der schlimmste Fal ist wiederum derjenige, bei dem das gesuchte Element nicht im Array vorkommt. In diesem Falle sind $O(\log_2 n)$ Aufrufe nötig; zugleich enthält der Stack dann maximal $O(\log_2 n)$ Einträge jeweils gleicher Größe. Folglich:

$S(n)=O(\log_2 n)$.

Man kann das binäre Suchen noch effizienter realisieren, indem man einen nicht-rekursiven Algorithmus angibt, der keinen Stack verwendet. Die Laufzeit für diesen Algorithmus (selbst entwickeln und analysieren!) bleibt wie bisher $O(\log_2 n)$ Zeit, der Speicherbedarf reduziert sich auf $O(1)$.

Beispiel 2: **Sortieren eines Feldes.**

Ein gegebenes Feld a mit n Elementen vom Typ integer soll aufsteigend sortiert werden. Der Algorithmus beruht auf der Idee, fortlaufend zwei benachbarte Feldelemente miteinander zu vergleichen und ggf. auszutauschen, wenn sie in der falschen Reihenfolge stehen:

```
program sort;
const n=...;
var   a: array [1..n] of integer;
      i,j,t: integer;
begin
      for i:=n-1 downto 1 do
            for j:=1 to i do
```

```
            if a[j]>a[j+1] then
            begin
                t:=a[j];
                a[j]:=a[j+1];
                a[j+1]:=t
            end
        end.
```

Dieser Algorithmus ist unter der Bezeichnung **Bubblesort** (engl. bubble=Blase) bekannt. Dieser Name rührt von folgender Beobachtung her. Schreibt man das Feld von unten nach oben auf, dann kann man das Verfahren mit dem Aufsteigen von Blasen in einem Sprudelglas vergleichen: größere Blasen (Elemente) steigen solange auf, bis sie durch eine noch größere Blase aufgehalten werden, die ihrerseits weiter aufsteigt.

Beispiel: Die Folge

54,80,11,91,17,23,58,28

soll sortiert werden (Abb. 4). Zuerst steigt die „Blase“ 80 auf, bis sie auf die 91 trifft, die ganz bis oben aufsteigt. Dann wiederholt sich der Vorgang mit den „Blasen“ 80, 58 und 54 usw.

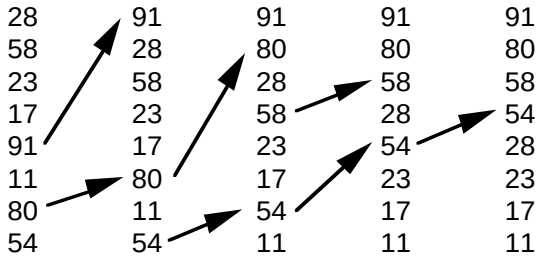

Abb. 4: Arbeitsweise von Bubblesort

Die Laufzeitanalyse von Bubblesort ist einfach. Setzen wir für den Schleifenrumpf (also die if-Anweisung) C Rechenschritte an. Für i=1 wird die innere Schleife einmal durchlaufen, für i=2 zweimal, ..., für i=n-1 entsprechend (n-1)-mal. Also wird der Schleifenrumpf insgesamt

1+2+3+...+(n-2)+(n-1) mal

durchlaufen. Die Ausführung des Algorithmus benötigt daher

$T(n)=C(1+2+3+...+(n-2)+(n-1))={}^{C}/_{2}\cdot n(n-1)=O(n^2)$

Rechenschritte.

Der folgende Algorithmus ist nur geringfügig komplizierter, arbeitet jedoch wesentlich schneller. Wir werden diesen Algorithmus nur skizzieren. Die Skizze reicht jedoch völlig aus, das Verfahren komplexitätsmäßig zu analysieren.

Kern des Sortierverfahrens ist ein Algorithmus zum Mischen zweier Zahlenfolgen f und g

zur Folge h. Zur Erinnerung noch einmal ein Beispiel:

Beispiel: Gegeben seien die beiden Folgen

f: 7 19 2 13
g: 1 24 3 4

Das Verschmelzen der beiden Folgen zur Folge h verläuft in den folgenden Schritten:

1. Schritt:

f: 7 19 2 13
h: 1
g: 24 3 4

2. Schritt:

f: 19 2 13
h: 1 7
g: 24 3 4

3. Schritt:

f: 2 13
h: 1 7 19
g: 24 3 4

usw.

8. Schritt:

h: 1 7 19 2 13 24 3 4

Was wird durch das Mischen erreicht? Bezeichnen wir zur Präzisierung eine maximale aufsteigend sortierte Teilfolge der Folge h als *Lauf*. Offenbar sagt die Anzahl der Läufe einer Folge etwas über den Grad der Sortierung aus: Viele Läufe weisen auf eine kaum sortierte, wenige Läufe auf eine schon gut sortierte Folge hin. Besteht die Folge nur noch aus einem Lauf, so ist sie vollständig sortiert.

Beispiel: In der Ergebnisfolge h des obigen Beispiels gibt es drei Läufe: 1,7,19 und 2,13,24 und 3,4.

Offensichtlich wirkt sich das Mischen auf die Anzahl der Läufe aus. Nach dem Mischen ist die Anzahl der Läufe in der Ergebnisfolge höchstens halb so groß wie die Anzahl der Läufe in den beiden Ausgangsfolgen zusammen. Spalten wir die Folge h nun wieder in zwei Teilfolgen auf, indem wir die Läufe von h abwechselnd auf die beiden Teilfolgen f und g kopieren, so können f und g erneut gemischt werden.

Beispiel: Die Folge h aus obigem Beispiel spaltet man auf in

f: 1 7 19 3 4
g: 2 13 24

und mischt sie zusammen zu

h: 1 2 7 13 19 3 4 24.

Nun gibt es nur noch zwei Läufe in h: 1,2,7,13,19 und 3,4,24. Im nächsten Zerlegungsschritt kopiert man den ersten Lauf auf f und den zweiten auf g. Mischt man dann f und g zusammen, so entsteht die gewünschte sortierte Folge.

In jedem Fall entstehen aus den maximal $n/2$ Läufen der vorangegangenen Phase maximal $n/4$ Läufe in der nächsten Phase. Nun ist klar, wie man sortiert. Man spaltet fortlaufend die Folge in zwei Hälften und mischt die beiden Folgen zu einer einzigen. Besteht die Ergebnisfolge nur noch aus einem einzigen Lauf, dann ist die Sortierung beendet.

Zur Laufzeit: Eine Mischphase erfordert offensichtlich O(n) Schritte; denn man benötigt n Vergleiche, und jeder Vergleich löst eine konstante Anzahl von Operationen aus (Übertragung des kleineren Elements und Löschen). Das einmalige Aufspalten einer Folge kann ebenfalls in O(n) Schritten durchgeführt werden. Die Laufzeit des Gesamtalgorithmus beträgt also in Abhängigkeit von n

$$T(n)=O(n\cdot\text{“Anzahl der Mischphasen“}).$$

Wie oft muß man eine Folge aufspalten und mischen? Nach dem ersten Mischen hat man $\leq n/2$ Läufe in der Ergebnisfolge, nach dem zweiten Mischen $\leq n/4$ Läufe, nach dem dritten Mischen $\leq n/8$ Läufe, nach dem k-ten Mischen $\leq n/2^k$ Läufe usw. Offenbar liegt nach spätestens $\log_2 n$ Phasen eine sortierte Folge vor. Laufzeit des Sortierens durch Mischen also:

$$T(n)=O(n\cdot\log_2 n).$$

Bemerkung: In der Praxis verwendet man nur Sortierverfahren, die in $O(n\cdot\log_2 n)$ Schritten arbeiten. Langsamere Verfahren sind unbrauchbar.

4.4 Untere Schranken für die Laufzeit

Sortieren ist eine der am häufigsten verwendeten Operationen in Computersystemen, und daher ist man bestrebt, möglichst effiziente Algorithmen hierfür zu finden. Mit quadratischer Zeit kommt man auf jeden Fall aus, wie der Algorithmus Bubblesort zeigt.
In diesem Abschnitt behandeln wir die Frage, wie schnell man n Elemente sortieren kann, etwa in $O(n\cdot\log_2\log_2 n)$ oder gar in O(n)? Um dies zu untersuchen, machen wir eine Annahme: Wir betrachten nur Sortieralgorithmen, die auf Vergleichen basieren. In Abhängigkeit vom Ergebnis des Vergleichs $\leq$ oder > vertauscht man gegebenenfalls zwei Elemente und vergleicht anschließend zwei andere Elemente usw. Um die Laufzeit eines solchen Sortierverfahrens nach unten abzuschätzen, genügt es also, die Anzahl der Vergleiche zu zählen, die im schlimmsten Fall notwendig ist, um die sortierte Reihenfolge herzustellen.

Beispiele:

1) Beim Bubblesort wird im schlimmsten Fall jedes Element mit jedem anderen verglichen. Hierzu sind $1/2 n\cdot(n-1)=O(n^2)$ Vergleiche erforderlich.

2) Man betrachte die Sortierung von drei Elementen a_1,a_2,a_3, die der Einfachheit halber alle verschieden sein sollen. Zunächst vergleicht man a_1 mit a_2 und anschließend a_2 mit a_3. Falls jeweils $a_1<a_2$ und $a_2<a_3$ galt, so war die Folge schon sortiert. Falls $a_1<a_2$ und $a_2>a_3$ war, so muß man noch a_1 mit a_3 vergleichen, um die sortierte Reihenfolge zu bestimmen. Falls z.B. $a_1<a_2$ und $a_1>a_3$ galt, so ist a_3,a_1,a_2 die sortierte Folge. Diese aufeinanderfolgenden Vergleiche kann man als binären Baum B (sog. *Entscheidungsbaum*) darstellen (Abb. 5). In den inneren Knoten stehen die jeweils zu vergleichenden Elemente (Die Zeichenfolge i@j bedeutet, daß a_i mit a_j verglichen wird). Je nach Ergebnis des Vergleichs ≤ oder > verzweigt man in den linken oder rechten Teilbaum. In den Blättern steht jeweils die Reihenfolge der Elemente a_1,a_2,a_3, die aufsteigend sortiert ist (statt a_i schreiben wir nur i).

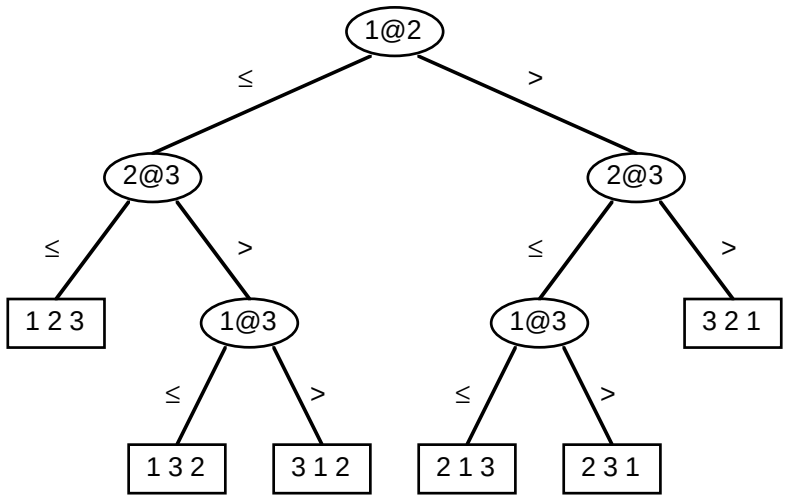

Abb. 5: Entscheidungsbaum B

Definition A:
Ein **Entscheidungsbaum** für eine Folge $a_1,a_2,...,a_n$ ist ein binärer Baum, dessen Knoten mit Ausnahme der Blätter mit Markierungen der Form i@j versehen sind. Die beiden Kanten zu den Söhnen jedes Knotens sind mit ≤ bzw. > markiert. Jedes Blatt ist mit *der* Umordnung der Folge $a_1,...,a_n$ markiert, die alle Vergleiche erfüllt, die auf dem Weg von der Wurzel zu diesem Blatt auftreten.

Jeder Algorithmus, der auf Vergleichen beruht, entspricht solch einem Entscheidungsbaum. Die Laufzeit eines Algorithmus im schlimmsten Fall ist mindestens gleich der *Maximalzahl* von Vergleichen, die notwendig ist, um die Folge zu sortieren. Diese Zahl wiederum ist gleich der Länge des *längsten* Weges von der Wurzel zu einem Blatt (minus Eins), der im Entscheidungsbaum auftritt.
Sei $V_B(n)$ die Maximalzahl von Vergleichen, die zur Sortierung von n Objekten mit dem

Entscheidungsbaum B nötig ist.

Beispiel: Die Länge eines längsten Weges ist in Abb. 1 gleich 4, d.h., die Maximalzahl von Vergleichen beträgt $V_B(3)=3$.

Der Algorithmus mit den wenigsten Vergleichen entspricht nun einem Entscheidungsbaum, in dem die Länge des längsten Weges möglich klein ist. Dieser Wert

$$\min\{V_B(n)|\ B \text{ ist ein Entscheidungsbaum für n Elemente}\}$$

liefert uns eine untere Schranke für die Laufzeit von Sortieralgorithmen. Hierzu überlegen wir uns, wieviele Blätter in Entscheidungsbäumen auftreten müssen. n Elemente können auf n! verschiedene Weisen angeordnet werden ($n!=1\cdot 2\cdot\ldots\cdot n$). Eine der Anordnungen ist die sortierte Reihenfolge. Jede dieser Anordnungen kommt in einem Blatt des Entscheidungsbaumes vor und gibt dort an, wie die Ausgangsfolge $a_1,\ldots,a_n$ umgeordnet werden muß, um eine sortierte Reihenfolge herzustellen. Jeder Entscheidungsbaum muß mindestens so groß sein, daß er alle n! Blätter aufnehmen kann.
Ist $V_B(n)=1$, so liegt ein Baum mit höchstens zwei Blättern vor. Ist $V_B(n)=2$, so hat der Baum höchstens vier Blätter. Allgemein gilt: Ein Baum mit V(n) notwendigen Vergleichen besitzt höchstens $2^{V(n)}$ Blätter.

Um alle n! Blätter aufzunehmen, folgt für einen Entscheidungsbaum also durch Verknüpfung der beiden Ergebnisse die Bedingung

$$2^{V_B(n)} \geq n!$$

bzw. durch Logarithmierung

$$V_B(n) \geq \log_2(n!).$$

Dies ist bereits das gewünschte Ergebnis: Ein optimaler Sortieralgorithmus benötigt im schlimmsten Fall mindestens $\log_2(n!)$ Vergleiche, um n Elemente zu sortieren. Rechnen wir noch die Ordnung aus, so gilt:

$$\log_2(n!) = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=n/2}^{n} \log_2 i \geq {}^{n}/_{2}\cdot\log_2({}^{n}/_{2}) = {}^{n}/_{2}\cdot(\log_2 n-1).$$

Satz B:
Jedes allgemeine auf Vergleichen beruhende Sortierverfahren benötigt im schlimmsten Fall mindestens $O(n\cdot\log_2 n)$ Vergleiche bzw. Rechenschritte.

Der früher eingeführte Algorithmus „Sortieren durch Mischen“ ist in diesem Sinne optimal, da er in $O(n\cdot\log_2 n)$ Schritten abläuft. Bemühungen, die Ordnung dieses Verfahrens zu verbessern, sind also zum Scheitern verurteilt.