Kapitel 2
Aufwand und Komplexität

Die Rechenleistung moderner Computer steigt Jahr für Jahr. In der Allgemeinheit wächst der Glaube an die Möglichkeit, mit einem genügend teuren Computer alles und jedes ausrechnen zu können. Dieser Glaube trügt gleich in zweierlei Hinsicht. Erstens kommt es nicht nur auf schnelle Computer an, sondern auch auf schnelle Algorithmen, und zweitens sind manche Probleme derart komplex, daß auch die schnellsten Algorithmen auf den schnellsten Computern Jahrtausende zur Lösung brauchen würden.

Nach Beispielen braucht man nicht lange zu suchen. Schon das Sortieren von hundert ganzen Zahlen kann den teuersten Computer überfordern, wenn nur der zugrunde liegende Algorithmus nichts taugt. Mit etwas Nachdenken kann man Sortieralgorithmen finden, welche das Ergebnis mit deutlich geringerem Aufwand liefern. Aber irgendwann ist Schluß. Die Komplexität eines Problems ist der unvermeidbare Aufwand zur Lösung. Für das Sortierproblems können wir die Komplexität explizit angeben. Für andere wichtige Probleme ist diese Frage noch offen und führt mitten hinein in die Komplexitätstheorie, einem Forschungsgebiet an der Schnittstelle zwischen Mathematik und Informatik.

2.1 Sortieren ganzer Zahlen

Die ganzen Zahlen

$$z_1, z_2, ..., z_n \in \mathbb{Z}$$

sollen ihrer Größe nach sortiert werden. Wir suchen also eine Änderung der Reihenfolge oder kurz eine *Permutation* $\pi_1, ..., \pi_n$ dieser Zahlen mit der Eigenschaft

$$z_{\pi_1} \leq z_{\pi_2} \leq ... \leq z_{\pi_n} \ . \tag{2.1}$$

Der folgende Algorithmus zur Lösung dieses Problems macht keinen besonders durchdachten Eindruck.

Algorithmus 2.1 (TumbSort).
Probiere alle möglichen Permutationen durch, bis die gewünschte Eigenschaft (2.1) vorliegt.

Da es nur endlich viele verschiedene Permutationen von $1, ..., n$ gibt, liefert dieser Algorithmus tatsächlich irgendwann die Lösung. Das kann aber dauern, denn die Anzahl dieser Permutationen ist $n!$. Für $n = 22$ sind das immerhin schon

$$22! = 1124000727777607680000 \approx 10^{21}.$$

Allgemeiner beschreibt das Wachstum von $n!$ die *Stirlingsche Formel* (siehe zum Beispiel [12, §20, Satz 6])

$$\sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n < n! < e^{1/12n}\sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n \ , \tag{2.2}$$

also $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$. Wenn wir Pech haben, müssen alle diese Permutationen durchprobiert werden. Dabei sind dann jedes Mal n Vergleiche nötig, um festzustellen, ob die Zahlen nun richtig sortiert sind oder nicht. Im schlechtesten Fall benötigen wir also

$$n \cdot n! \ \text{ Vergleiche.}$$

Schon im Falle $n = 100$ würde das mehr als 100^{77} Jahre dauern (ungefähr 10^{145} mal solange wie unser Universum existiert), selbst wenn jeder Vergleich nur 10^{-20} Sekunden benötigen würde[1]

Aber das muß nicht sein. Jedes Kind kennt die *transitive Struktur*

$$x \le y \text{ und } y \le z \quad \Longrightarrow \quad x \le z \tag{2.3}$$

der Ordnungsrelation $\le$ auf $\mathbb{Z}$, welche schon in der Schreibweise (2.1) zum Ausdruck kommt. Damit ist offenbar

$$z_{\pi_n} = \max\{z_1, z_2, \ldots, z_n\} \; .$$

Um als nächstes $z_{\pi_{n-1}}$ zu erhalten, brauchen wir nur das Maximum der restlichen $n-1$ Zahlen $\{z_1, z_2, \ldots, z_n\} \setminus \{z_{\pi_n}\}$ zu bestimmen und so weiter. Der resultierende Sortieralgorithmus heisst *BubbleSort* und sieht als Pseudocode wie folgt aus.

Algorithmus 2.2 (BubbleSort).

```
function S=bubblesort(U)

n=length(U);

S=U;  k=n;
while k>1

    SMax=S(1);  iMax=1;
    for i=2:k
        if S(i) > SMax
            SMax=S(i);  iMax=i;
        end
    end

    S(iMax)=S(k);  S(k)=SMax;
    k=k-1;
end
```

Als Eingabe wird der Vektor $U = (z_1, \ldots, z_n)$ erwartet und als Ausgabe der sortierte Vektor $S = (z_{\pi_1}, \ldots, z_{\pi_n})$ geliefert. Wir illustrieren den Ablauf von BubbleSort anhand eines Beispiels.

$$\begin{aligned}
&S = \{4,1,2\}\;,\; k = 3: && SMax = \max\{4,1,2\} = 4, && iMax = 1\\
&S = \{2,1,4\}\;,\; k = 2: && SMax = \max\{2,1\} = 2, && iMax = 1\\
&S = \{1,2,4\}
\end{aligned}$$

Schon auf den ersten Blick wirkt BubbleSort effizienter als tumbes Probieren. Wir wollen es genauer wissen und ermitteln dazu die *Anzahl der benötigten Vergleiche*.

Satz 2.3. *Zum Sortieren von n ganzen Zahlen benötigt Bubblesort genau*

$$\tfrac{1}{2}n(n-1) \;\; \textit{Vergleiche.}$$

Beweis. Für jedes k erfordert die innere `for`-Schleife zur Bestimmung des Maximums $k-1$ Vergleiche. Die äußere `while`-Schleife wird für $k = n, \ldots, 2$, durchlaufen. Summation von $k-1$ über $k = n, \ldots, 2$ liefert

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

□

Das ist immerhin eine dramatische Reduktion verglichen mit dem mehr als exponentiellen Aufwand von Algorithmus 2.1. Die naheliegende Frage, ob es Sortieralgorithmen gibt, die noch weniger Vergleiche benötigen als BubbleSort oder ob das Sortierproblem *prinizipiell* nicht mit geringerem Aufwand zu lösen ist, führt auf die Frage nach der Effizienz von Algorithmen und der Komplexität von Problemen. Darum geht es im nächsten Abschnitt.

[1] Im Jahre 2010 wären für jeden Vergleich mehr als 10^{-10} Sekunden nötig, denn die Taktfrequenz der zu dieser Zeit verfügbaren Prozessoren liegt unter 10 GHz.

2.2 Aufwand von Algorithmen und Komplexität von Problemen

Wir betrachten ein abstraktes Problem $P(n)$, welches von einem Parameter n abhängt. Wir können uns unter $P(n)$ beispielsweise das Problem vorstellen, n ganze Zahlen zu sortieren oder das Problem, ein lineares Gleichungssystem mit n Gleichungen zu lösen. Im allgemeinen ist n dabei ein Maß für die Länge der Eingabedaten. Wir gehen davon aus, daß eine gewisse Menge zulässiger Eingabedaten der Länge n gegeben ist, etwa die Menge aller Teilmengen von $\mathbb{Z}$ mit n Elementen beim Sortierproblem oder die Menge aller regulären $n \times n$-Koeffizientenmatrizen nebst allen Vektoren der Länge n als rechten Seiten beim linearen Gleichungssystem. Mit $\mathscr{A}(P)$ bezeichnen wir nun die Menge aller Algorithmen, die das Problem $P(n)$ für die zulässigen Eingabedaten lösen. Einen Algorithmus identifizieren wir dabei jeweils mit einer Referenzdarstellung im Pseudocode. Beispielsweise wird BubbleSort mit Algorithmus 2.2 identifiziert. Dabei gehen wir stillschweigend davon aus, daß die Referenzdarstellung sinnvoll ist, insbesondere, daß sie keine überflüssigen Befehle enthält. Unter allen Algorithmen $A \in \mathscr{A}(P)$ sind wir vor allem an möglichst *effizienten Algorithmen* interessiert, bei denen der zur Lösung benötigte *Aufwand* mit wachsender Länge n der Eingabedaten möglichst langsam wächst.

Dazu müssen wir klären wie der Aufwand gemessen werden soll. Die scheinbar einfachste Möglichkeit besteht darin, den Algorithmus auf einem jeweils zur Verfügung stehenden Rechner zu implementieren und einfach die *CPU-Zeit* zu messen. Der Nachteil dieses *experimentellen Aufwandsmaßes* ist, daß das Ergebnis von einer Reihe von Einflüssen abhängt, die mit den jeweiligen Algorithmen überhaupt nicht zu tun haben! Insbesondere kann man durch geschickte Verbindung von Implementierung, Betriebssystem, Prozessor, Compilerwahl, Cache-Optimierung, Vektorisierung oder gar Parallelisierung erstaunliche Laufzeitgewinne erzielen, aber umgekehrt durch ungeschickte Implementierung (auf demselben Rechner!) auch verschenken. Dazu kommt noch, daß im allgemeinen verschiedene Eingabedaten auch verschiedenen Aufwand erfordern. Wir müssten also alle zulässigen Eingabedaten mit wachsender Länge n durchprobieren, um sicher zu gehen.

Die *CPU-Zeit* eignet sich also nicht besonders gut als Aufwandsmaß. Statt dessen wollen wir den Aufwand durch eine Formel ausdrücken, welche die Wirklichkeit gut genug beschreibt. Eine solche Formel nennt man

mathematisches Modell.

Das einfachste Modell besteht darin, a) eine problemspezifische

Referenzoperation

zu identifizieren, die in allen Algorithmen $A \in \mathscr{A}(P)$ vorkommt, und die hinsichtlich der Ausführungszeit alle anderen erforderlichen Operationen dominiert sowie b) unter Vernachlässigung aller anderen erforderlichen Operationen den Aufwand als die Anzahl der benötigten Referenzoperationen zu definieren.

Definition 2.4 (Aufwand). Der *Aufwand* $T_A(n)$ eines Algorithmus $A \in \mathscr{A}(P)$ ist das Maximum der benötigen Anzahl an Referenzoperationen über alle zulässigen Eingabedaten der Länge n.

Als Beispiel betrachten das Sortierproblem aus dem vorgigen Abschnitt. Wir wählen die

$$\text{Referenzoperation:} \quad \text{Vergleich zweier ganzer Zahlen.} \tag{2.4}$$

Die Eingabelänge n ist die Anzahl der zu sortierenden Zahlen. Der Aufwand $T_A(n)$ eines Algorithmus zum Sortieren von n ganzen Zahlen ist demnach die Anzahl der im schlechtest möglichen Fall erforderlichen Vergleiche. Der Aufwand von TumbSort und BubbleSort ist also

$$T_{\text{TumbSort}}(n) = n \cdot n! \quad \text{und} \quad T_{\text{BubbleSort}}(n) = \tfrac{1}{2} n(n-1).$$

Offenbar ist BubbleSort um eine Größenordnung effizienter als TumbSort. Kann man die Effizienz beliebig steigern? Das kommt auf das Problem an.

Definition 2.5 (Komplexität). Sei $T_A(n)$ der Aufwand von $A \in \mathscr{A}(P)$. Dann ist die *Komplexität* $\mathscr{K}_P(n)$ des Problems $P(n)$ die größte Zahl mit der Eigenschaft

$$\mathscr{K}_P(n) \leq T_A(n) \qquad \forall A \in \mathscr{A}(P) \tag{2.5}$$

oder gleichbedeutend $\mathscr{K}_P(n) = \inf_{A\in\mathscr{A}(P)} T_A(n)$.

Man sollte Aufwand und Komplexität nicht verwechseln!

Die Komplexität ist eine Eigenschaft des Problems.

Der Aufwand ist eine Eigenschaft des Algorithmus'.

Jeder Algorithmus $A \in \mathscr{A}(P)$, liefert offenbar eine obere Schranke $T_A(n)$ für die Komplexität $\mathscr{K}_P(n)$. Um untere Schranken zu erhalten, muß man zeigen, daß es keinen Algorithmus mit kleinerem Aufwand geben kann. Das ist oft schwieriger. Bevor wir im folgenden Abschnitt die Komplexität des Sortierproblems untersuchen, betrachten wir als erstes Beispiel die Bestimmung des Maximums von n ganzen Zahlen.

Satz 2.6. *Die Komplexität $\mathscr{K}_{\max}(n)$ des Problems*

$$\textit{Bestimme} \quad U_{\max} = \max\{z_1, z_2, \ldots, z_n\}$$

bezüglich der Referenzoperation (2.4) *genügt der Abschätzung*

$$\tfrac{1}{2}n \le \mathscr{K}_{\max}(n) \le n-1.$$

Beweis. Der Algorithmus FindMax, gegeben durch

```
function UMax=findmax(U)

n=length(U);

UMax=U(1);
for i=2:n
    if U(i) > UMax  UMax=U(i);  end
end
```

mit der Eingabe $U = (z_1, \ldots, z_n)$ benötigt $n-1$ Vergleiche zur Bestimmung der Lösung $U_{\max}$. Also ist $\mathscr{K}_{\max}(n) \le n-1$.

Da jedes Element z_k, $k = 1, \ldots, n$, ein Kandidat für das Maximum ist, muß jedes dieser n Elemente in mindestens einem Vergleich vorkommen. Da an jedem Vergleich genau zwei Elemente beteiligt sind, kann deshalb kein Algorithmus mit weniger als $\frac{1}{2}n$ Vergleichen auskommen. Also ist $\frac{1}{2}n \le \mathscr{K}_{\max}(n)$. □

Algorithmen, deren Aufwand bis auf einen multiplikative Konstante mit der Komplexität des jeweiligen Problems übereinstimmen[2], nennt man *optimal*. Der Algorithmus FindMax aus dem Beweis zu Satz 2.6 ist also in diesem Sinne optimal.

2.3 Die Komplexität des Sortierproblems

Aus den beiden vorigen Abschnitten kennen wir schon relativ gute Schranken für die Komplexität $\mathscr{K}_{\text{sort}}(n)$ des Sortierproblems: Der Aufwand von BubbleSort liefert eine obere Schranke und es gilt $\mathscr{K}_{\text{sort}}(n) \ge \mathscr{K}_{\max}(n)$, da man aus einer sortierten Menge direkt das Maximum dieser Menge ablesen kann. Aus Satz 2.6 und Satz 2.3 folgt daher unmittelbar

$$\tfrac{1}{2}n = T_{\text{FindMax}}(n) \le \mathscr{K}_{\text{sort}}(n) \le T_{\text{BubbleSort}}(n) = \tfrac{1}{2}n(n-1)\ .$$

Wir wollen die Lücke zwischen unterer und oberer Schranke verkleinern indem wir einen Algorithmus entwickeln, der weniger Vergleiche als BubbleSort benötigt.

Die Grundidee findet sich in vielen numerischen Verfahren wieder, von der hierarchischen Summation in Abschnitt 7.4 bis hin zu Mehrgitterverfahren für partielle Differentialgleichungen und noch viel weiter. Sie lautet: *Teile und herrsche*.

[2] Damit ist folgendes gemeint: Falls $f(n)$ und $g(n)$ die Aufwände des Algorithmus bzw. die Komplexität des Problems bezeichnen, dann gilt $\lim_{n\to\infty} f(n)/g(n) = K$ mit einer Konstante $0 < K < \infty$.

Konkret haben wir vor, die unsortierte Menge U in zwei kleinere Teilmengen $U_1^{(1)}$ und $U_2^{(1)}$ aufzuteilen, weil kleinere Mengen offenbar leichter zu sortieren sind. Jede dieser beiden Teilmengen wird nun wieder in zwei kleinere Teilmengen aufgeteilt, die noch leichter zu sortieren sind und so weiter. Am Ende sind wir bei Teilmengen angekommen, die nur noch aus einem Element bestehen und damit fertig. Die Vorgehensweise ist in Figur 2.1 illustriert (man beachte die Analogie zu Figur 7.4). Allerdings

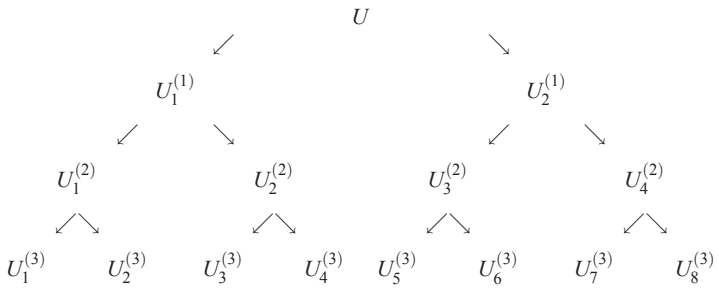

Abb. 2.1 Hierarchische Zerlegung der zu sortierenden Menge U

ist das ganze Prozedere nur dann sinnvoll, wenn es gelingt, die vielen kleinen sortierten Mengen ohne großen Aufwand wieder zu einer großen sortierten Menge zusammenzuführen. Entscheidend ist also das folgendem Lemma.

Lemma 2.7. *Gegeben seien die beiden sortierten Mengen*

$$S_x = \{x_1 \le x_2 \le \cdots \le x_n\}, \qquad S_y = \{y_1 \le y_2 \le \cdots \le y_m\}.$$

Dann läßt sich die Menge $S = S_x \cup S_y$ mit $n+m$ Vergleichen sortieren.

Wir führen einen konstruktiven Beweis, indem wir einen entsprechenden Algorithmus angeben. Dessen Grundidee besteht darin, die beiden Mengen S_x und S_y reissverschlussartig zusammenzuschieben.

Algorithmus 2.8 (Merge).

```
function S=merge(Sx,Sy)

n=length(Sx);
m=length(Sy);

i=1; j=1; k=1;

while k<=n && j<=m

    if Sx(k) < Sy(j)
        S(i)=Sx(k); i=i+1;
        k=k+1;
    else
        S(i)=Sy(j); i=i+1;
        j=j+1;
    end
end

if k>n S(i:n+m)=Sy(j:m);
else   S(i:n+m)=Sx(k:n); end
```

Das folgende Beispiel illustriert die Arbeitsweise von Merge.

$$S_x = \{1,\ 1,\ 3\}, \quad n = 3, \qquad S_y = \{1,\ 2,\ 4,\ 6,\ 8,\ 10\}, \quad m = 6$$

$$
\begin{array}{lll}
k=1,\ j=1: & Sx(1)=1=1=Sy(1): & S(1):=1,\ j:=j+1\\
k=1,\ j=2: & Sx(1)=1<2=Sy(2): & S(2):=1,\ k:=k+1\\
k=2,\ j=2: & Sx(2)=1<2=Sy(2): & S(3):=1,\ k:=k+1\\
k=3,\ j=2: & Sx(3)=3>2=Sy(2): & S(4):=2,\ j:=j+1\\
k=3,\ j=3: & Sx(3)=3>4=Sy(3): & S(5):=3,\ k:=k+1
\end{array}
$$

$k=4>3=n$: Ende der `while`-Schleife: $S(6:9):=Sy(3:6)$

Wir haben noch nachzuweisen, daß Algorithmus 2.8 mit maximal $n+m$ Vergleichen auskommt. Dazu bemerken wir, daß in jedem Durchlauf der `while`-Schleife genau ein Vergleich vorgenommen wird (die zur Kontrolle von `for`-und `while`-Schleifen erforderlichen Vergleiche werden nach wie vor ignoriert). Da in jedem Durchlauf der `while`-Schleife entweder j oder k um eins erhöht wird, sind maximal $n+m$ Durchläufe möglich bis das Abbruchkriterium greift; man sagt auch, dass der Algorithmus nach maximal $n+m$ Schritten terminiert. Somit kommt man auf maximal $n+m$ Vergleiche.

Damit ist Lemma 2.7 bewiesen. Nun führen wir unseren ursprünglichen Plan durch (teile und herrsche). Dabei nehmen wir der Einfachheit halber an, daß n eine Zweierpotenz ist, daß also

$$n=2^m$$

vorliegt. Der folgende Algorithmus lässt sich jedoch auch auf beliebige $n\in\mathbb{N}$ übertragen (Übung).

Algorithmus 2.9 (MergeSort).

```
function Sz = mergesort(Uz)

n = length(Uz);

% Diese Implementierung funktioniert auch, wenn n keine Zweierpotenz ist.
% Warum?

z = Uz;
if n == 1  Sz = z;
else       m = fix(n/2);
           x(1:m) =  z(1:m);
           y(1:n-m) = z(m+1:n);
           Sx = mergesort(x);
           Sy = mergesort(y);
           Sz = merge(Sx,Sy);
end
```

Beachte, daß MergeSort sich selbst aufruft. Man spricht von einem *rekursiven* Algorithmus. Ähnlich wie bei `while`-Schleifen kann es sein, dass rekursive Algorithmen nicht terminieren. Im vorliegenden Fall ist die Sache allerdings klar: Nach m Rekursionsschritten ist man bei der Länge $n=1$ angelangt.

Die Aufwandsanalyse ist diesmal etwas komplizierter. Der Aufwand $T(n)=T(2^m)$ von MergeSort setzt sich zusammen aus dem Aufwand $2\cdot T(n/2)=2\cdot T(2^{m-1})$ zum Sortieren der beiden halbierten Mengen und dem Aufwand für deren Verschmelzen. Auf die gleiche Weise lässt sich der Aufwand $T(2^{m-1})$ von MergeSort zum Sortieren der halbierten Mengen durch den Aufwand zum Sortieren und Verschmelzen der geviertelten Mengen ausdrücken und so weiter. Am Ende sind schließlich Mengen mit einem Element erreicht. Dann gibt es nichts mehr zu sortieren, es liegt also $T(1)=0$ vor. Mit Blick auf Lemma 2.7 genügt daher der Aufwand von MergeSort der rekursiven Abschätzung

$$T(2^k)\le 2\cdot T(2^{k-1})+2^k,\quad k=1,\dots,m,\qquad T(1)=0\,. \tag{2.6}$$

Wir wollen zunächst einen Zusammenhang mit der entsprechenden *inhomogenen Zwei-Term-Rekursion*

$$T_k=2\cdot T_{k-1}+2^k,\quad k=1,\dots,m,\qquad T_0=0\,. \tag{2.7}$$

herstellen.

Lemma 2.10. *Ist T_k, $k=1,\dots,m$, Lösung von* (2.7)*, so gilt*

$$T(2^k) \leq T_k, \quad k = 0, \ldots, m,$$

insbesondere also $T(n) \leq T_m$.

Beweis. Der Beweis erfolgt durch vollständige Induktion über k. Für $k = 0$ ist offenbar $T(1) = T_0$. Es gelte also $T(2^{k-1}) \leq T_{k-1}$ für ein $k > 0$. Dann folgt aus (2.6) und (2.7) die Abschätzung

$$T(2^k) - T_k \leq 2 \cdot (T(2^{k-1}) - T_{k-1}) \leq 0$$

und damit die Behauptung. □

Es lohnt sich also, für die Rekursion (2.7) eine geschlossene Lösung anzugeben.

Lemma 2.11. *Die Lösung von* (2.7) *ist*

$$T_k = k2^k, \quad k = 0, \ldots, m\,. \tag{2.8}$$

Beweis. Anstatt die Gültigkeit der Formel (2.8) einfach durch Einsetzen in die Rekursion (2.7) zu bestätigen (das ginge auch), wollen wir Satz A.39 anwenden. Einsetzen von $a_k = 2$, $b_k = 2^{k+1}$ und $b_{-1} = T_0 = 0$ in A.29 liefert

$$T_k = \sum_{j=1}^{k} 2^j \prod_{i=j}^{k-1} 2 = \sum_{j=1}^{k} 2^j 2^{k-j} = 2^k \sum_{j=1}^{k} 1 = k2^k.$$

□

Wegen $m2^m = n\log_2 n$ folgt die gesuchte Aufwandsabschätzung nun direkt aus Lemma 2.10 und 2.11.

Satz 2.12. *Für den Aufwand von MergeSort gilt die Abschätzung*

$$T_{\text{MergeSort}}(n) \leq n\log_2 n.$$

Da $\log_2 n$ exponentiell langsamer wächst als n haben wir gegenüber BubbleSort eine signifikante Verbesserung erreicht.

n	BubbleSort	MergeSort
8	28	16
64	2016	305
512	130816	3961
4096	8386560	43971
32768	53685452	450212

Tabelle 2.1 Aufwandsvergleich von BubbleSort und MergeSort

Tabelle 2.1 zeigt einen Vergleich dieser beiden Algorithmen 2.2 und 2.9 anhand zufällig gewählter Zahlen z_k, $k = 1, \ldots, n$, und $n = 2^3, 2^6, \ldots, 2^{15}$. Wie erwartet, beobachten wir ein fast lineares Wachstum bei MergeSort gegenüber einem quadratischen Anwachsen bei BubbleSort. Für $n = 2^{15}$ benötigt BubbleSort schon mehr als hundert Mal soviele Vergleiche erforderlich wie MergeSort.

Aus Satz 2.12 folgt für die Komplexität des Sortierproblems die verbesserte obere Schranke

$$\mathscr{K}_{\text{sort}}(n) \leq n\log_2 n\,.$$

Es stellt sich die Frage, ob noch effizientere Algorithmen als MergeSort existieren. Hier kommt die Antwort.

Satz 2.13. *Die Komplexität des Sortierproblems genügt der Abschätzung*

$$\mathscr{K}_{\text{sort}}(n) \geq \tfrac{1}{4} n\log_2 n\,.$$

Beweis. Für $n = 1$ ist $T_A(1) = 0 = \frac{1}{4} \cdot \log 1$. Sei also $n > 1$ und A ein beliebig aber fest gewählter Sortieralgorithmus. Wir ordnen A einen Entscheidungsbaum zu (siehe Figure 2.2). An den Verzweigungen wird entsprechend dem aktuellen Zwischenergebnis entschieden in welche Richtung es weitergeht. Ist man an einem der Endpunkte angekommen (man spricht von Blättern) so hat man eine Lösung gefunden. Da jedes dieser Blätter die gesuchte Lösung sein kann, muss die Anzahl der Blätter mindestens so groß wie die Anzahl der möglichen Lösungen sein. Die Anzahl der möglichen Lösungen stimmt mit der Anzahl der möglichen Permutationen der n Zahlen überein und die ist gerade $n!$. Mit T_A Vergleichen oder,

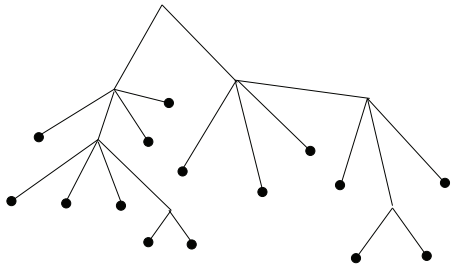

Abb. 2.2 Entscheidungsbaum eines Sortieralgorithmus.

gleichbedeutend, mit T_A Verzweigungen des Entscheidungsbaums lassen sich nun maximal 2^{T_A} Blätter erzeugen, da sich jede Verzweigung sich aus einem binären Vergleich ergibt. Es muss also $2^{T_A(n)} \geq n!$ gelten Daraus folgt mit der Stirlingschen Formel (2.2)

$$\begin{aligned} T_A(n) &\geq \log_2(n!) > \log_2\left(\sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n\right) \\ &= \log_2(\sqrt{2\pi n}) + n\log_2 n - n\log_2 e \geq \tfrac{1}{4}\, n\log_2 n, \end{aligned}$$

und damit die Behauptung. □

MergeSort ist also ein optimaler Sortieralgorithmus. Dennoch ist MergeSort nicht immer unbedingt erste Wahl. Beispielsweise wird in der Praxis oft QuickSort angewandt, obwohl die Anzahl der benötigten Vergleiche im schlechtesten Fall sogar quadratisch wächst. Das liegt daran, daß erstens der schlechteste Fall sehr selten auftritt und zweitens QuickSort im Gegensatz zu MergeSort ohne zusätzlichen Speicherplatz auskommt.

Diesen Sachverhalt kann man mit verfeinerten Aufwandsmodellen quantifizieren, welche anstelle des schlechtest möglichen Falls die durchschnittlich benötigte Anzahl von Referenzoperationen zu Grunde legen und zusätzlich den Speicherbedarf berücksichtigen. Wir verweisen dazu auf einschlägige Lehrbücher zur Algorithmentheorie [6, 33, 39] sowie auf das Standardwerk von D.E. Knuth [21].

2.4 Nichtpolynomielle Komplexität?

Wir betrachten das *Problem des Handlungsreisenden* (Traveling Salesman Problem), kurz TSP–Problem [16, 19]. Gegeben seien

Städte : $1, ..., n$,
Fahrzeiten : c_{ij} zwischen Stadt i und Stadt j, $\quad i, j = 1, \ldots, n$.

Das Problem lautet:

Finde eine Tour oder, gleichbedeutend, eine Permution $\pi_1, ..., \pi_n$, so daß die *Gesamtfahrzeit*

$$\text{cost}\,(\pi) = \sum_{i=1}^{n-1} c_{\pi_i,\pi_{i+1}} + c_{\pi_n,\pi_1}$$

unter allen Permutationen von $\{1, ...n\}$ minimal wird.

Hinter diesem eher spielerisch anmutenden Problem stehen konkrete technologische Anwendungen. Da sich beispielsweise der Bohrkopf bei der Herstellung von Platinen aus mechanischen Gründen möglichst wenig bewegen soll[3], kommt man bei der Bohrkopfsteuerung auf ein TSP-Problem, bei dem Bohrlöcher die Rolle der Städte übernehmen.

Hier ein Beispiel mit $n = 4$ Städten und den Fahrzeiten:

$$C = \begin{pmatrix} 0 & 212 & 171 & 203 \\ 212 & 0 & 186 & 247 \\ 171 & 186 & 0 & 139 \\ 203 & 247 & 139 & 0 \end{pmatrix} .$$

Nun können wir die Gesamtfahrzeit für alle möglichen Touren ausrechnen. Beispielsweise ist

$$\begin{aligned} \text{cost}\,(1,2,3,4) &= 740 \\ \text{cost}\,(1,2,4,3) &= 769 \\ \text{cost}\,(1,3,2,4) &= 807\,. \end{aligned}$$

Es ist nicht schwer, einen Algorithmus anzugeben, der das Problem des Handlungsreisenden löst:

Algorithmus 2.14 (TumbSalesman). Probiere alle möglichen Permutationen durch, bis die minimale Gesamtfahrzeit gefunden ist.

Wir wollen einen Blick auf den Aufwand werfen. Dazu wählen wir das

$$\text{Referenzoperation:} \quad \text{Auswertung des } \textit{Kostenfunktionals } \text{cost}(\pi).$$

Wir wissen schon, daß wir bei Algorithmus 2.14 mit einem Aufwand von $\mathcal{O}(n!)$ rechnen müssen (vgl. A.6). Damit ist dieser Algorithmus nur für sehr kleine n praktisch durchführbar.

Für das Sortierproblem haben wir schnell den Algorithmus 2.2 mit höchstens quadratischem Aufwand gefunden, also mit Aufwand $\mathcal{O}(n^2)$. Gibt es für das TSP-Problem auch einen Algorithmus mit höchstens polynomiellem Aufwand, das heisst mit Aufwand $\mathcal{O}(n^q)$ für ein $q \in \mathbb{N}$, welches nicht von n abhängt? Dies ist ein *offenes Problem*, welches seit langer Zeit im Mittelpunkt der Algorithmentheorie steht! Ein Grund für das große Interesse besteht darin, daß ein polynomieller TSP-Algorithmus sofort auch polynomielle Algorithmen für eine Reihe anderer wichtiger Probleme nach sich ziehen würde (so einen Zusammenhang gibt es auch zwischen Sortieren und Maximieren). Lässt sich, anders als beim Sortieren und Maximieren, dieser Schluß auch umkehren, so gehört das entsprechende Problem zur Klasse der sogenannten *NP-vollständigen* Probleme. NP ist hierbei die Abkürzung für *nicht-deterministisch polynomiell* (nicht zu verwechseln mit nichtpolynomiell) und bezeichnet die Klasse aller Problemen, die auf einer nicht-deterministischen Turingmaschine in polynomieller Zeit lösbar sind [39]. Derzeit herrscht übrigens die Meinung vor, daß es *keinen* polynomiellen TSP-Algorithmus gibt.

Auf zukünftig vielleicht existierende Algorithmen können die praktischen Anwendungen des TSP-Problems nicht warten, und derzeit existierende Algorithmen sind zu langsam. Beispielsweise erfordert Algorithmus 2.14 selbst bei einer kompletten Auswertung des Kostenfunktionals pro Takt und einer Taktrate von 1000 *MHz* schon für nur $n = 100$ Städte immerhin mehr als 100^{141} Jahre zur Lösung. Vor diesem Hintergrund gibt man sich in der Praxis lieber mit suboptimalen Touren zufrieden, die man dafür aber schnell genug ausrechnen kann. Die Entwicklung entsprechender Heuristiken, welche möglichst kostengünstige Touren mit möglichst geringem (polynomiellen!) Aufwand berechnen, ist ein wichtiges Forschungsgebiet der *Diskreten Optimierung*. Näheres dazu findet sich beispielsweise im Buch von Grötschel [17].

[3] Genauer will man in den meisten Anwendungen die Gesamtzeit zur Bohrung aller nötigen Bohrlöcher minimieren, was bei konstanter Bewegungsgeschwindigkeit des Bohrkopfes zwischen den Löchern bedeutet, dass man die zur Bohrung aller Löcher notwendige Wegstrecke minimieren muss.

2.5 Aufgaben

Aufgabe 2.1 *Gegeben sei die sortierte Menge $S = \{y_1 \leq y_2 \leq \cdots \leq y_n\}$. Nun soll ein zusätzliches Element x einsortiere werden. Das Problem besteht also darin, einen Index i_0 zu finden, für den $y_{i_0} \leq x \leq y_{i_0+1}$ vorliegt.*

1. *Bestimmen Sie die Komplexität dieses Problems bezüglich des in Definition 2.4 erklärten Aufwandbegriffs mit der Referenzoperation* (2.4).
2. *Geben Sie einen optimalen Algorithmus zur Lösung an.*

Aufgabe 2.2 *Wir betrachten das folgende Verfahren zum Sortieren ganzer Zahlen.*

```
function S=InsertionSort(U)
S=U;   n=length(S);
for j=2:n
    key = S(j);
    i = j-1;
    while i>0 && S(i)>key
       S(i+1) = S(i);
       S(i) = key;
       i = i-1;
    end
end
```

1. *Wieviele Vergleiche werden benötigt, um die Liste*

$$A = \{4,7,-2,3,8,-5,3,2,0,1,9\}$$

zu sortieren?
2. *Bestimmen Sie den Aufwand $T_{\text{InsertionSort}}(n)$ gemäß Definition 2.4 mit der Referenzoperation* (2.4). *Wie muss die Eingabefolge aussehen, damit das Maximum angenommen wird?*
3. *Wie viele Vergleiche werden mindestens benötigt? Wie muss die entsprechende Eingabefolge aussehen?*