Algorithmen

Algorithmen und Datenstrukturen

4., aktualisierte Auflage

Robert Sedgewick
Kevin Wayne

EXTRAS ONLINE

ALWAYS LEARNING

PEARSON

Algorithmen

Algorithmen und Datenstrukturen

4., aktualisierte Auflage

Robert Sedgewick
Kevin Wayne

ALWAYS LEARNING

PEARSON

Satz F: Top-Down-Mergesort benötigt zwischen $\frac{1}{2} N \lg N$ und $N \lg N$ Vergleiche, um ein beliebiges Array der Länge N zu sortieren.

Beweis: $C(N)$ sei die Anzahl der Vergleiche, die benötigt werden, um ein Array der Länge N zu sortieren. Dann ist $C(0) = C(1) = 0$ und wir können für $N > 0$ eine Rekursionsgleichung aufstellen, die die rekursive Methode `sort()` direkt widerspiegelt, um eine obere Schranke festzulegen:

$$C(N) \leq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + N$$

Der erste Term auf der rechten Seite gibt die Anzahl der Vergleiche zum Sortieren der linken Hälfte des Arrays an, der zweite Term die Anzahl der Vergleiche zum Sortieren der rechten Hälfte und der dritte Term die Anzahl der Vergleiche für die Vereinigung. Die untere Schranke

$$C(N) \geq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + \lfloor N/2 \rfloor$$

ergibt sich daraus, dass die Anzahl der Vergleiche für die Vereinigung mindestens $\lfloor N/2 \rfloor$ ist.

Wir erhalten eine exakte Lösung für die Rekursion, wenn die Gleichheit gewahrt bleibt und N einen Zweierpotenz ist (sagen wir $N = 2^n$). Da $\lfloor N/2 \rfloor = \lceil N/2 \rceil = 2^{n-1}$ gilt anfangs:

$$C(2^n) = 2C(2^{n-1}) + 2^n$$

Indem wir beide Seiten durch 2^n teilen, erhalten wir

$$C(2^n)/2^n = C(2^{n-1})/2^{n-1} + 1$$

Wenn wir diese Gleichung auf sich selbst, d.h. auf den ersten Term zur Rechten anwenden, ergibt sich

$$C(2^n)/2^n = C(2^{n-2})/2^{n-2} + 1 + 1$$

Durch $n-1$-maliges Wiederholen dieses Schritts erhalten wir

$$C(2^n)/2^n = C(2^0)/2^0 + n$$

was uns nach der Multiplikation beider Seiten mit 2^n zu folgender Lösung führt:

$$C(N) = C(2^n) = n2^n = N \lg N$$

Genaue Lösungen für allgemeine N sind aufwendiger; aber es ist nicht schwer, die gleiche Beweisführung auf die Ungleichheiten anzuwenden, die die Schranken für die Anzahl der Vergleiche beschreiben, um die Aussage für alle Werte von N zu beweisen. Dieser Beweis ist gültig, unabhängig von den Eingabewerten und der Reihenfolge, in der sie erscheinen.

Listing 2.10: Top-Down-Mergesort (Algorithmus 2.4)

```
public class Merge
{
   private static Comparable[] aux;      // Hilfsarray für Vereinigungen.

   public static void sort(Comparable[] a)
   {
      aux = new Comparable[a.length];    // Weist nur einmal Speicher zu.
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {  // Sortiert a[lo..hi].
      if (hi <= lo) return;
      int mid = lo + (hi - lo)/2;
      sort(a, lo, mid);       // Sortiert linke Hälfte.
      sort(a, mid+1, hi);     // Sortiert rechte Hälfte.
      merge(a, lo, mid, hi);  // Ergebnisse mergen (Code in Listing 2.9).
   }
}
```

Um ein Teilarray `a[lo..hi]` zu sortieren, halbieren wir es in zwei Teilarrays `a[lo..mid]` und `a[mid+1..hi]`, sortieren diese unabhängig voneinander (über rekursive Aufrufe) und führen die beiden geordneten Teilarrays wieder zum endgültigen Ergebnis zusammen.

			a[]													
	lo	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, | 0, | 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, | 2, | 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, | 0, | 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, | 4, | 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, | 6, | 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, | 4, | 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, | 0, | 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, | 8, | 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, | 10, | 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, | 8, | 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, | 12, | 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, | 14, | 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, | 12, | 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, | 8, | 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, | 0, | 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Abbildung 2.11: Ablaufprotokoll der `merge()`-Ergebnisse für Top-Down-Mergesort

Zur Veranschaulichung von *Satz F* kann der Ablauf auch in Form einer Baumstruktur ausgedrückt werden (▶ Abbildung 2.12). In diesem Baum steht jeder Knoten für ein Teilarray, für das `sort()` die Methode `merge()` aufruft. Der Baum besteht aus genau n Ebenen. Für k von 0 bis $n-1$ enthält die k-te Ebene von oben 2^k Teilarrays mit jeweils

der Länge 2^{n-k}, die jeweils höchstens 2^{n-k} Vergleiche für die Vereinigung benötigen. Daraus ergeben sich Gesamtkosten von $2^k \cdot 2^{n-k} = 2^n$ für jede der n Ebenen, was zu einer Gesamtsumme von $n2^n = N \lg N$ führt.

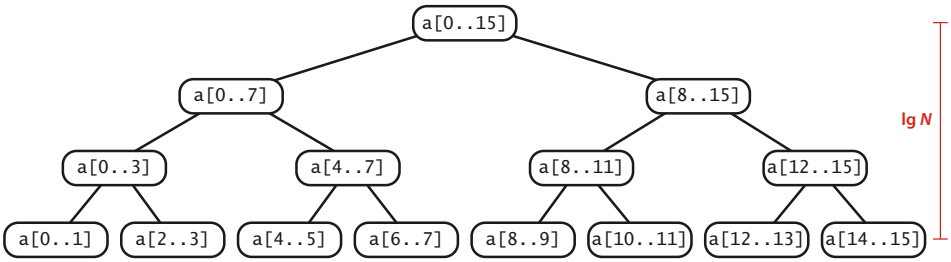


Abbildung 2.12: Baum der Mergesort-Teilarray-Abhängigkeiten für $N = 16$

Satz G: Top-Down-Mergesort benötigt höchstens $6\ N \lg N$ Arrayzugriffe, um ein Array der Länge N zu sortieren.

Beweis: Jede Vereinigung benötigt höchstens $6N$ Arrayzugriffe ($2N$ für die Kopie, $2N$ für das Zurückverschieben und höchstens $2N$ für die Vergleiche). Der Rest ergibt sich aus der Beweisführung zu *Satz F*.

Satz F und *Satz G* lässt sich entnehmen, dass die von Mergesort benötigte Zeit proportional $N \log N$ ist. Verglichen mit den elementaren Sortiermethoden aus Abschnitt 2.1 stoßen wir damit in ganz neue Dimensionen vor, da wir jetzt riesige Arrays in einer Zeit sortieren können, die sich nur um einen logarithmischen Faktor von dem Zeitaufwand unterscheidet, den wir für die Untersuchung aller Elemente benötigen. Sie können mit Mergesort Millionen von Elementen (oder mehr) sortieren, was mit Insertionsort oder Selectionsort nicht möglich wäre. Der größte Nachteil von Mergesort ist, dass beim Vereinigen zusätzlicher Speicherbedarf proportional N für das Hilfsarray anfällt, das heißt, wenn Speicherplatz ein Problem darstellt, müssen wir ein anderes Verfahren in Erwägung ziehen. Andererseits lässt sich die Laufzeit von Mergesort mit einigen wohlüberlegten Änderungen an der Implementierung noch beträchtlich verringern.

Kleine Teilarrays mit Insertionsort sortieren

Da die Technik der Rekursion *garantiert*, dass die rekursive Methode häufig für kleine Fälle aufgerufen wird, lassen sich die meisten rekursiven Algorithmen dadurch verbessern, dass wir für kleine Fälle eine andere Strategie wählen. Was das Sortieren betrifft, so wissen wir, dass Insertionsort (oder Selectionsort) einfach ist und deshalb wahrscheinlich sehr kleine Teilarrays schneller sortiert als Mergesort. Wie immer lässt sich die Funktionsweise von Mergesort am besten anhand einer grafischen Darstellung des Ablaufprotokolls veranschaulichen. Das Ablaufprotokoll in ▶ Abbildung 2.13 zeigt den Ablauf einer Mergesort-Implementierung mit einem Wechsel des Sortierverfahrens (*cutoff*) für kleine Teilarrays. Wenn wir bei kleinen Teilarrays (Länge 15

oder kleiner) zu Insertionsort wechseln, verbessern wir die Laufzeit einer typischen Mergesort-Implementierung um 10 bis 15 Prozent (siehe *Übung 2.2.23*).

Prüfen, ob das Array bereits sortiert ist

Wir können die Laufzeit für bereits geordnete Arrays so weit reduzieren, dass sie linear wird. Dazu müssen wir lediglich einen Test hinzufügen, um den Aufruf von `merge()` zu überspringen, wenn `a[mid]` kleiner gleich `a[mid+1]` ist. Mit dieser Änderung führen wir immer noch alle rekursiven Aufrufe aus, aber die Ausführungszeit für jedes sortierte Teilarray ist linear (siehe *Übung 2.2.8*).

Zeit für das Kopieren in das Hilfsarray eliminieren

Es ist möglich, den Zeitaufwand zum Kopieren der Elemente in das Hilfsarray zu eliminieren (aber nicht den Speicherbedarf!). Der Trick ist, die Sortiermethode zweimal aufzurufen: Ein Aufruf übernimmt die Eingabe aus dem gegebenen Array und legt die sortierte Ausgabe im Hilfsarray ab, der andere Aufruf übernimmt die Eingabe aus dem Hilfsarray und legt die sortierte Ausgabe im gegebenen Array ab. Bei diesem Ansatz können wir mit ein wenig rekursiver Trickserei die Aufrufe so hintereinanderschalten, dass die Berechnung die Rollen von Eingabe- und Hilfsarray auf jeder Ebene tauscht (siehe *Übung 2.2.11*).

Es erscheint uns angebracht, hier noch einmal auf einen Punkt hinzuweisen, der bereits in *Kapitel 1* angesprochen wurde und leicht vergessen wird. Im engeren Kontext behandeln wir jeden Algorithmus in diesem Buch so, als wäre er ein wichtiger Bestandteil irgendeiner konkreten Anwendung. Im weiteren Kontext versuchen wir, allgemeine Aussagen über den zu empfehlenden Ansatz zu machen. Unsere Diskussion bestimmter Verbesserungen ist jedoch nicht notwendigerweise eine Empfehlung, diese auch zu implementieren, sondern vielmehr ein Warnhinweis, die Performance der ersten Implementierungen nicht als absolut und unveränderlich anzusehen. Stehen Sie vor einem neuen Problem, verwenden Sie am besten die einfachste Implementierung, die Ihnen zusagt, und überarbeiten diese, wenn sie zu einem Engpass wird. Verbesserungen, die die Laufzeit lediglich um einen konstanten Faktor verringern, lohnen sich sonst nicht. Und denken Sie daran, die Effektivität von gezielten Verbesserungen, wie in den Übungen immer wieder erwähnt, anhand von Experimenten zu testen.

Im Falle von Mergesort sind die drei gerade genannten Verbesserungen einfach zu implementieren und interessant, wenn Mergesort das Verfahren der Wahl ist – zum Beispiel in Situationen, die am Ende dieses Kapitels besprochen werden.

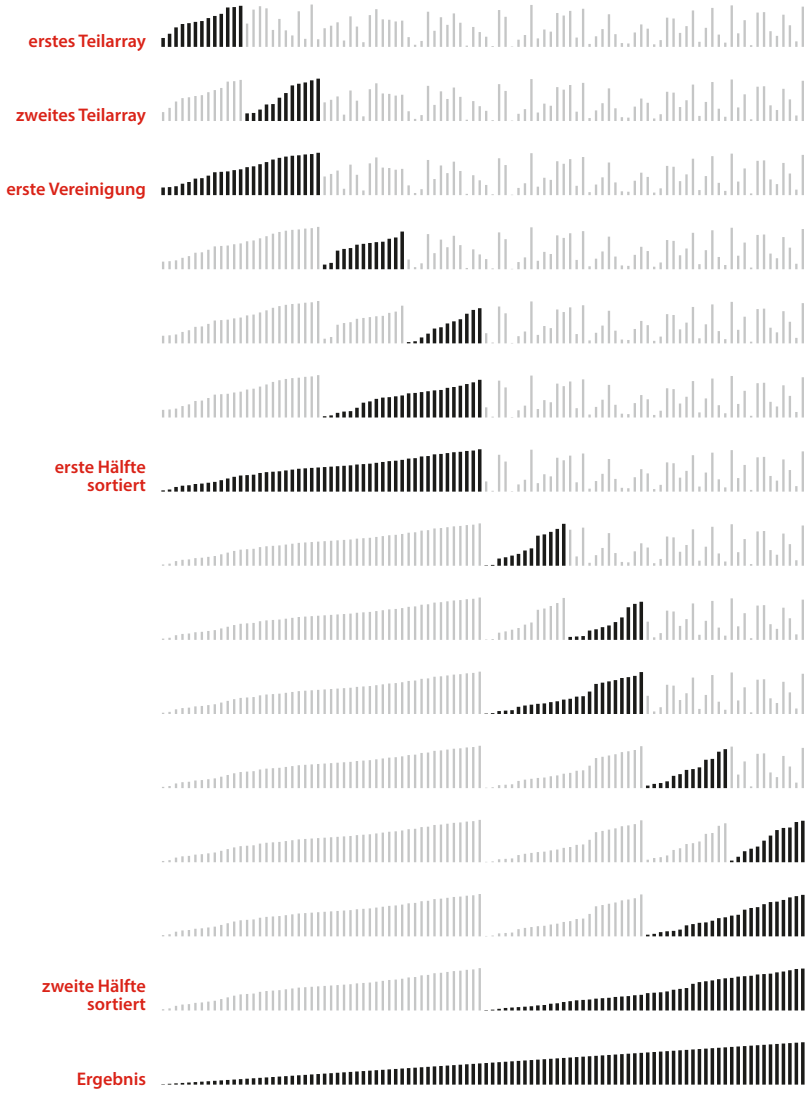


Abbildung 2.13: Grafisches Ablaufprotokoll von Top-Down-Mergesort mit einem Wechsel des Sortierverfahrens für kleine Teilarrays

2.2.3 Bottom-Up-Mergesort

Die rekursive Implementierung von Mergesort ist typisch für das Algorithmendesign-Paradigma *Teile-und-Herrsche*, bei dem wir ein großes Problem dadurch lösen, dass wir es in Teile zerlegen, die Teilprobleme lösen und dann diese Lösungen heranziehen, um das Gesamtproblem zu lösen. Auch wenn wir hier davon sprechen, zwei große Teilarrays zusammenzuführen, sind es in Realität meistens mehrere winzige Teilarrays. Eine andere Möglichkeit, Mergesort zu implementieren, besteht darin, die Operationen so zu organisieren, dass wir im ersten Durchlauf alle winzigen Teilarrays sortiert zusam-

menführen, dann im zweiten Durchlauf die nun doppelt so großen Teilarrays paarweise sortiert zusammenfassen und so fortfahren, bis im letzten Durchlauf das ganze Array sortiert ist. Dieses Verfahren benötigt noch weniger Code als die rekursive Standardimplementierung. Wir beginnen mit einem Durchlauf, bei dem wir einzelne Elemente, d.h. Teilarrays der Größe 1, zu sortierten Teilarrays der Größe 2 zusammenfassen (wir bezeichnen dies als 1-mit-1), dann fassen wir in einem weiteren Durchlauf Teilarrays der Größe 2 zu sortierten Teilarrays der Größe 4 zusammen (2-mit-2), anschließend Teilarrays der Größe 4 zu sortierten Teilarrays der Größe 8 (4-mit-4) und so weiter. In jedem Durchlauf kann bei der letzten Mischoperation das zweite Teilarray kleiner sein als das erste (was für `merge()` kein Problem darstellt). Davon jedoch abgesehen sind die Teilarrays bei allen Operationen gleich groß, sodass sich die Größe der zu sortierenden Teilarrays beim nächsten Durchlauf verdoppelt.

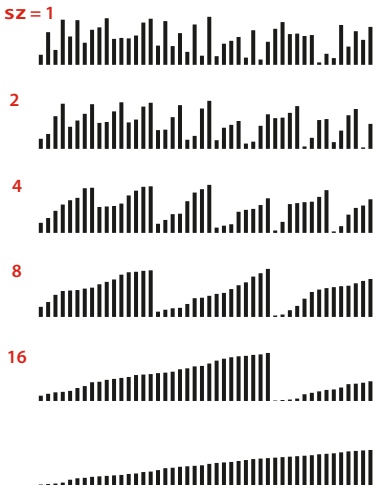


Abbildung 2.14: Grafisches Ablaufprotokoll von Bottom-Up-Mergesort

Satz H: Bottom-Up-Mergesort benötigt zwischen $\frac{1}{2}\, N \lg N$ und $N \lg N$ Vergleiche und höchstens $6N \lg N$ Arrayzugriffe, um ein Array der Länge N zu sortieren.

Beweis: Die Anzahl der Durchläufe durch das Array ist genau $\lceil \lg N \rceil$. Für jeden Durchlauf ist die Anzahl der Arrayzugriff genau $6N$ und die Anzahl der Vergleiche höchstens N und mindestens $N/2$.

Wenn die Arraylänge eine Zweierpotenz ist, werden bei Top-Down- und Bottom-Up-Mergesort genau die gleichen Vergleiche und Arrayzugriffe ausgeführt, nur in anderer Reihenfolge. Wenn die Arraylänge keine Zweierpotenz ist, ist die Folge der Vergleiche und Arrayzugriffe für die beiden Algorithmen unterschiedlich (siehe *Übung 2.2.5*).

Eine Bottom-Up-Mergesort-Version ist das Verfahren der Wahl zum Sortieren von Daten in einer *verketteten Liste*. Stellen Sie sich die Liste als eine Sammlung sortierter Teillisten der Größe 1 vor, die Sie dann durchlaufen, um sortierte Teilarrays der Größe

2 zu erhalten, dann der Größe 4 usw. Dieses Verfahren ordnet die Referenzen um, mit dem Ziel die Liste an *Ort und Stelle* (*in-place*) zu sortieren (ohne neue Listenknoten zu erzeugen).

Listing 2.11: Bottom-Up-Mergesort

```
public class MergeBU
{
   private static Comparable[] aux;      // Hilfsarray für Vereinigungen

   // Den Code für merge() finden Sie in Listing 2.9.

   public static void sort(Comparable[] a)
   {  // lg N Durchläufe von Operationen zur paarweisen Teilarray-
      // Zusammenführung.
      int N = a.length;
      aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)        // sz: Teilarraygröße
         for (int lo = 0; lo < N-sz; lo += sz+sz) // lo: Teilarrayindex
            merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

Bei Bottom-Up-Mergesort wird das ganze Array mehrmals durchlaufen, wobei jeweils Teilarrays der Größe `sz` mit Teilarrays der Größe `sz` (`sz`-mit-`sz`) sortiert zusammengeführt werden. Angefangen wird mit `sz` gleich 1*, wobei `sz` bei jedem Durchlauf verdoppelt wird. Das letzte Teilarray hat die Größe `sz`, jedoch nur, wenn die Arraygröße ein gerades Mehrfaches von `sz` ist (andererseits ist es kleiner als `sz`).*

	a[i]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 8																
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Abbildung 2.15: Ablaufprotokoll der `merge()`-Ergebnisse für Bottom-Up-Mergesort

Beide Ansätze zur Implementierung eines Teile-und-Herrsche-Algorithmus (Top-Down und Bottom-Up) sind intuitiv verständlich. Die Lehre, die Sie aus Mergesort ziehen können, ist, dass Sie, wann immer Sie es mit einem Algorithmus zu tun haben, der auf einem dieser Ansätze basiert, auch den anderen in Erwägung ziehen sollten. Wollen Sie das Problem lösen, indem Sie es wie in `Merge.sort()` in kleinere Probleme zerlegen (und rekursiv lösen) oder indem Sie kleinere Lösungen wie in `MergeBU.sort()` zu größeren zusammenfassen.

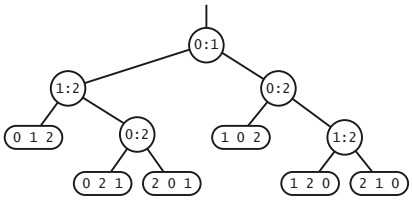
2.2.4 Die Komplexität des Sortierens

Basierend auf Mergesort können wir eine grundlegende Aussage der *Komplexitätstheorie* beweisen, die uns die immanente Schwierigkeit des Sortierens besser verstehen lässt. Die Komplexitätstheorie als solche spielt eine wichtige Rolle für den Entwurf von Algorithmen und da speziell die Aussage, um die es hier geht, für den Entwurf von Sortieralgorithmen von unmittelbarer Bedeutung ist, werden wir hier etwas ausführlicher darauf eingehen.

Der erste Schritt bei der Komplexitätsanalyse ist immer das Aufstellen eines Rechenmodells. Generell versuchen Wissenschaftler, möglichst das einfachste Modell zu verstehen, welches das Problem hinreichend beschreibt. Im Falle des Sortierens untersuchen wir die Klasse der *vergleichsbasierten Algorithmen*, die ihre Entscheidungen auf der Basis von Schlüsselvergleichen treffen. Ein vergleichsbasierter Algorithmus kann zwischen den Vergleichen beliebig viele Berechnungen durchführen; Informationen über einen Schlüssel erhält er aber nur, wenn er ihn mit einem anderen Schlüssel vergleicht. Da wir uns hier auf die `Comparable`-API beschränken, fallen alle Algorithmen dieses Kapitels in diese Klasse (beachten Sie, dass wir die Kosten für die Arrayzugriffe nicht berücksichtigen) – wie auch viele andere Algorithmen. In *Kapitel 5* werden wir Algorithmen betrachten, die nicht auf `Comparable`-Elemente beschränkt sind.

Satz I: Kein vergleichsbasierter Sortieralgorithmus kann garantieren, N Elemente mit weniger als $\lg(N!) \sim N \lg N$ Vergleichen zu sortieren.

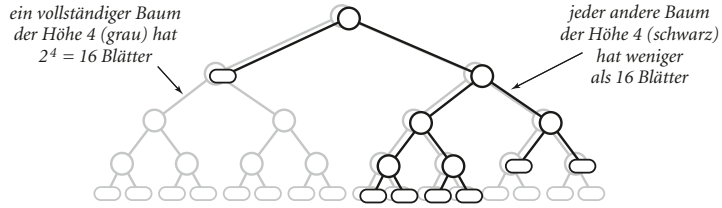
Beweis: Zuerst gehen wir davon aus, dass die Schlüssel alle verschieden sind, da jeder Algorithmus in der Lage sein muss, solche Eingaben zu sortieren. Dann beschreiben wir die Folge der Vergleiche mithilfe eines Binärbaums. Jeder *Knoten* (*node*) im Baum ist entweder ein *Blatt* (*leaf*) (`i0 i1 i2 ... iN-1`), das anzeigt, dass der Sortiervorgang abgeschlossen ist und festgestellt wurde, dass die ursprünglichen Eingaben in der Reihenfolge `a[i0]`, `a[i1]`, ...`a[iN-1]` geordnet sind, oder ein *interner Knoten* (*internal node*) (`i:j`), der einer Vergleichsoperation zwischen `a[i]` und `a[j]` entspricht. Für interne Knoten entspricht der linke Teilbaum der Folge von Vergleichen, die durchlaufen werden, wenn `a[i]` kleiner ist als `a[j]`, und der rechte Teilbaum beschreibt, was passiert, wenn `a[i]` größer ist als `a[j]`. Jeder Pfad von der Wurzel zu einem Blatt entspricht der Folge von Vergleichen, mit der der Algorithmus die im Blatt angegebene Ordnung herstellt. Betrachten Sie beispielsweise den folgenden Vergleichsbaum für $N=3$:



Wir konstruieren einen solchen Baum niemals explizit – er ist vielmehr ein mathematisches Hilfsmittel, das die von einem Algorithmus verwendeten Vergleiche beschreibt.

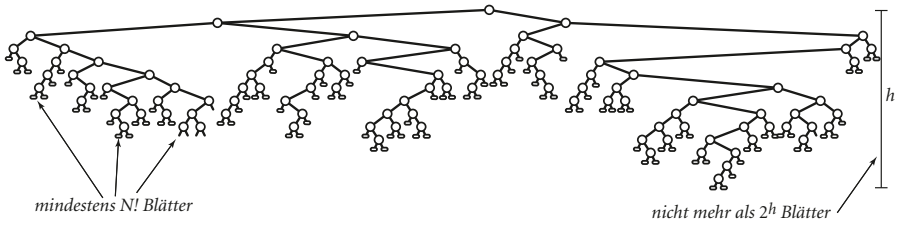
Die erste wichtige Beobachtung in diesem Beweis ist, dass der Baum mindestens $N!$ Blätter haben muss, da es $N!$ verschiedene Permutationen der N verschiedenen Schlüssel gibt. Sind weniger als $N!$ Blätter vorhanden, dann fehlt eine Permutation von den Blättern und der Algorithmus würde für diese Permutation fehlschlagen.

Die Anzahl der internen Knoten auf einem Pfad von der Wurzel zu einem Blatt im Baum entspricht der Anzahl der Vergleiche, die der Algorithmus für eine bestimmte Eingabe durchführt. Unser Interesse gilt vor allem der Länge des längsten Pfades im Baum (der sogenannten *Baumhöhe*), da dieser Wert die maximale Anzahl an Vergleichen von dem Algorithmus angibt (der schlimmste Fall). Nun besagt die Kombinatorik der Binärbäume, dass ein Baum der Höhe h maximal 2^h Blätter aufweist – man sagt auch, dass ein Baum der Höhe h mit der maximalen Anzahl Blättern perfekt balanciert oder *vollständig* ist. Nachfolgend finden Sie ein Beispiel für $h=4$.



Den beiden vorangehenden Absätzen können wir entnehmen, dass jeder Sortieralgorithmus, der auf Vergleichen basiert, einem Vergleichsbaum der Höhe h entspricht mit

$$N! \leq \text{Anzahl der Blätter} \leq 2^h$$



Der Wert von h entspricht exakt der Anzahl der Vergleiche im schlimmsten Fall, sodass wir die beiden Seiten dieser Gleichung (zur Basis 2) logarithmieren und die Schlussfolgerung ziehen können, dass die Anzahl der Vergleiche für jeden Algorithmus mindestens $\lg N!$ sein muss. Die Näherung $\lg N! \sim N \lg N$ ergibt sich direkt aus der Stirlingformel für die Fakultätsberechnung (siehe Tabelle 1.64 in *Kapitel 1*).

Dieses Ergebnis verrät uns, wie gut ein Sortieralgorithmus überhaupt sein kann. Jemand, der dieses Ergebnis nicht kennt, könnte vielleicht versucht sein, einen vergleichsbasierten Sortieralgorithmus zu entwerfen, der für den Worst-Case-Fall nur halb so viele Vergleiche benötigt wie Mergesort. Der Versuch wird allerdings vergeblich sein, denn die untere Schranke in *Satz I* besagt, dass *es einen solchen Algorithmus nicht gibt*. Dies ist eine klare und extrem starke Aussage, die sich auf jeden nur denkbaren vergleichsbasierten Algorithmus anwenden lässt.

Satz H besagt, dass Mergesort im schlimmsten Fall $\sim N \lg N$ Vergleiche benötigt. Dieses Ergebnis ist eine *obere Schranke* für die Komplexität des Sortierproblems insofern, als ein besserer Algorithmus garantieren müsste, mit weniger Vergleichen auszukommen. *Satz I* besagt, dass kein Sortieralgorithmus weniger als $\sim N \lg N$ Vergleiche garantieren kann. Es ist eine *untere Schranke* für die Komplexität des Sortierproblems insofern, als sogar der bestmögliche Algorithmus im schlimmsten Fall mindestens so viele Vergleiche benötigt. Zusammen laufen die beiden Behauptungen auf Folgendes hinaus:

Satz J: Mergesort ist ein asymptotisch optimaler vergleichsbasierter Sortieralgorithmus.

Beweis: Was wir genau mit dieser Aussage meinen, ist, dass *sowohl die Anzahl der Vergleiche von Mergesort im schlimmsten Fall und als auch die minimale Anzahl der Vergleiche, die ein vergleichsbasierter Sortieralgorithmus garantieren kann*, $\sim N \lg N$ beträgt. Dies ist die Kernaussage der *Sätze H* und *I*.

Es ist zu beachten, dass wir – ebenso wie das Rechenmodell – genau definieren müssen, was wir unter einem optimalen Algorithmus verstehen. Wir könnten beispielsweise die Definition von Optimalität einengen und darauf beharren, dass ein optimaler Algorithmus zum Sortieren *genau* $\lg N!$ Vergleiche benötigt. Wir sehen jedoch davon ab, da wir für große N zwischen einem solchen Algorithmus und (beispielsweise) Mergesort keinen Unterschied feststellen würden. Umgekehrt könnten wir die Definition von Optimalität dahingehend erweitern, dass wir jeden Sortieralgorithmus berücksichtigen, dessen Anzahl der Vergleiche im schlimmsten Fall *innerhalb eines konstanten Faktors* von $N \lg N$ liegt. Wir sehen auch hiervon ab, da wir in diesem Fall für große N einen deutlichen Unterschied zwischen einem solchen Algorithmus und Mergesort feststellen würden.

Die Komplexitätstheorie mag ziemlich abstrakt wirken, aber sie ist Teil der Grundlagenforschung zu den inhärenten Schwierigkeiten, Rechenprobleme zu lösen, und bedarf als solche wohl keiner weiteren Rechtfertigung. Außerdem wurde festgestellt, dass die Komplexitätstheorie, wo sie zur Anwendung kommt, Einfluss auf die Entwicklung guter Software hat. Erstens erlauben gute obere Schranken den Informatikern, Laufzeitgarantien zu formulieren; es gibt viele Belege dafür, dass schlechtes Laufzeitverhalten darauf zurückzuführen ist, dass ein quadratischer Sortieralgorithmus und kein leicht überlinearer gewählt wurde. Zweitens ersparen uns gute untere Schranken die Mühe, nach Möglichkeiten zur Laufzeitverbesserung zu suchen, die nicht erreichbar ist.

Copyright