Kapitel 2

Sortieren

Das Sortieren ist eines der grundlegenden Probleme in der Informatik. Es wird geschätzt, dass mehr als ein Viertel aller kommerzieller Rechenzeit auf Sortiervorgänge entfällt. Einige Anwendungsbeispiele:

aus [2, p. 71]

- Adressenverwaltung (lexikographisch)
- Trefferlisten bei Suchanfragen (Relevanz)
- Verdeckung (z-Koordinate)
- ...

Wir bezeichnen die Menge der Elemente, die als Eingabe für das Sortierproblem erlaubt sind, mit $\mathcal{U}$ (für Universum). Formal kann das Problem dann folgendermaßen beschrieben werden:

2.1 Problem (Sortieren)
gegeben: *Folge* $(a_1, \ldots, a_n) \in \mathcal{U}^n$ *mit Vergleichsoperator* „$\leq$“ $\subseteq \mathcal{U} \times \mathcal{U}$

gesucht: *Permutation (bijektive Abbildung* $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$*)*
mit $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$

Wir nehmen in diesem Kapitel an, dass die Eingabe in einem Array $M[1, \ldots, n]$ steht und darin sortiert zurück gegeben werden soll. Insbesondere wird daher auch von Interesse sein, welchen zusätzlichen Platzbedarf (Hilfsvariablen, Zwischenspeicher) ein Algorithmus hat.

Eingabe: $M[1, \ldots, n]$

Neben der Zeit- und Speicherkomplexität werden beim Sortieren weitere Gütekriterien betrachtet. Zum Beispiel kann es wichtig sein, dass die Reihenfolge von zwei Elemente mit gleichem Schlüssel nicht umgedreht wird. Verfahren, in denen dies garantiert ist, heißen stabil. Unter Umständen werden auch

- die Anzahl Vergleiche $C(n)$ und „comparisons“
- die Anzahl Umspeicherungen $M(n)$ „moves“

getrennt betrachtet (in Abhängigkeit von der Anzahl n der zu sortierenden Elemente), da Schlüsselvergleiche in der Regel billiger sind als Umspeicherungen ganzer Datenblöcke.

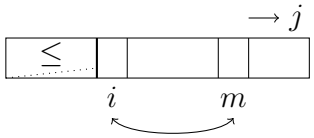
2.1 SelectionSort

„Sortieren durch Auswählen“; hier speziell: MinSort

Algorithmus 3 zur Lösung des Auswahlproblems hat das jeweils kleinste Element der Restfolge mit dem ersten vertauscht. Wird der Algorithmus fortgesetzt, bis die Restfolge leer ist, so ist am Ende die gesamte Folge nichtabsteigend sortiert.

Algorithmus 5: SelectionSort

```
for i = 1, ..., n − 1 do
    m ← i
    for j = i + 1, ..., n do
        if M[j] < M[m] then m ← j
    vertausche M[i] und M[m]
```



Die Anzahlen der Vergleiche und Vertauschungen sind für SelectionSort also

$$\begin{aligned} C(n) &= \quad n-1 \;+\; n-2+\cdots+1 = \textstyle\sum_{i=1}^{n-1} i = \frac{(n-1)\cdot n}{2} \\ M(n) &= \quad 3\cdot(n-1) \end{aligned}$$

und die Laufzeit des Algorithmus damit in $\Theta(n^2)$ im besten wie im schlechtesten Fall. Dadurch, dass nur vertauscht wird, wenn ein echt kleineres Element gefunden wird, ist der Algorithmus außerdem stabil.

Aus der Vorlesung „Methoden der Praktischen Informatik“ ist bereits bekannt, dass es Algorithmen gibt, die eine Folge der Länge n in Zeit $\mathcal{O}(n \log n)$

sortieren. Es stellt sich also die Frage, wie man Vergleiche einsparen kann. Man sieht leicht, dass die hinteren Elemente sehr oft zum Vergleich herangezogen werden. Kann man z.B. aus den früheren Vergleichen etwas lernen, um auf spätere zu verzichten?

2.2 Divide & Conquer

Die nächsten beiden Algorithmen beruhen auf der gleichen Idee:

> Sortiere zwei kleinere Teilfolgen getrennt
> und füge die Ergebnisse zusammen.

Dies dient vor allem der Reduktion von Vergleichen zwischen Elementen in verschiedenen Teilfolgen. Benötigt werden dazu Vorschriften

- zur Aufteilung in zwei Teilfolgen und
- zur Kombination der beiden sortierten Teilfolgen.

Die allgemeine Vorgehensweise, zur Lösung eines komplexen Problems dieses auf kleinere Teilprobleme der gleichen Art aufzuteilen, diese rekursiv zu lösen und ihre Lösungen jeweils zu Lösungen des größeren Problems zusammen zu setzen, ist das divide & conquer-Prinzip.

2.2.1 QuickSort

Wähle ein Element p („Pivot“, z.B. das erste) und teile die anderen Elemente der Eingabe M auf in

M_1: die höchstens kleineren Elemente

M_2: die größeren Elemente

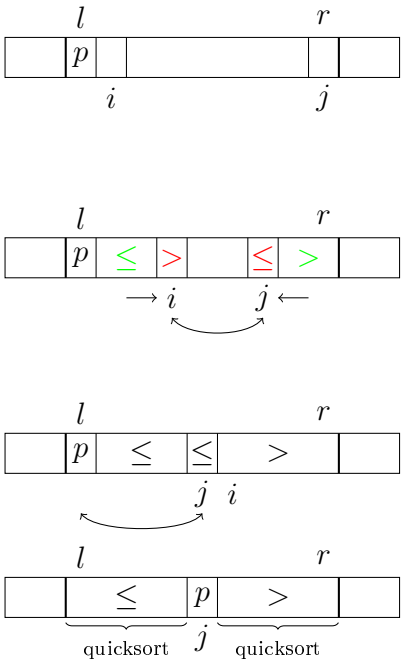
hard split,
easy join

Sind M_1 und M_2 sortiert, so erhält man eine Sortierung von M durch Hintereinanderschreibung von M_1, p, M_2. Algorithmus 6 ist eine mögliche Implementation von QuickSort.

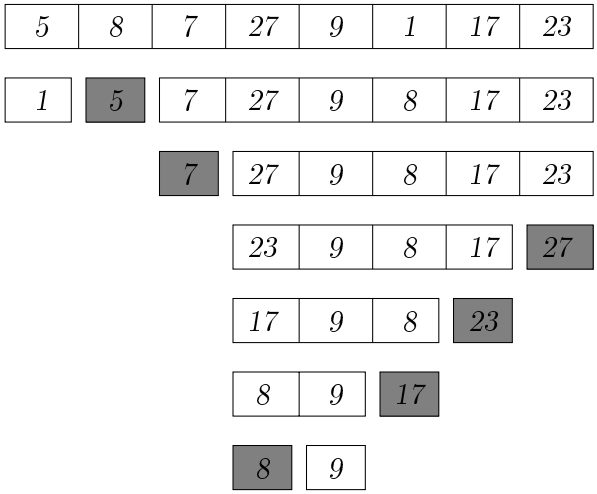
Algorithmus 6: QuickSort

Aufruf: `quicksort`$(M, 1, n)$

```
quicksort(M, l, r) begin
    if l < r then
        i ← l + 1;      j ← r
        p ← M[l]
        while i ≤ j do
            while i ≤ j and M[i] ≤ p do
                i ← i + 1
            while i ≤ j and M[j] > p do
                j ← j − 1
            if i < j then
                vertausche M[i] und M[j]
        if l < j then
            vertausche M[l] und M[j]
            quicksort(M, l, j − 1)
        if j < r then
            quicksort(M, j + 1, r)
end
```



2.2 Beispiel (QuickSort)



2.3 Satz
Die Laufzeit von QuickSort ist

(i) im besten Fall in $\Theta(n \log n)$ — best case

(ii) im schlechtesten Fall in $\Theta(n^2)$ — worst case

Beweis. Die Laufzeit für einen Aufruf von `quicksort` setzt sich zusammen aus

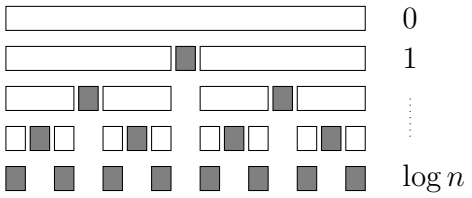
- linearem Aufwand für die Aufteilung und
- dem Aufwand für die Sortierung der Teilfolgen.

Der Gesamtaufwand innerhalb einer festen Rekursionsebene ist damit linear in der Anzahl der Elemente, die bis dahin noch nicht Pivot waren oder sind. Da bei jedem Aufruf ein Pivot hinzu kommt, ist die Anzahl der neuen Pivotelemente in einer Rekursionsebene

- mindestens 1 und
- höchstens doppelt so groß wie in der vorigen Ebene.

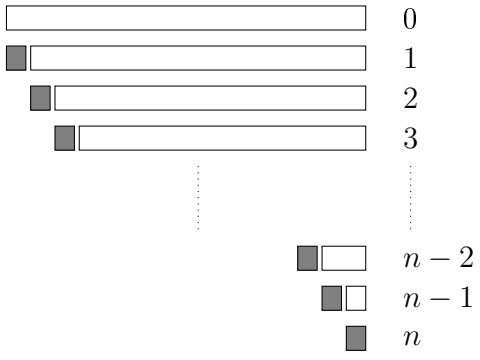
Die folgenden Beispiele zeigen, dass es Eingaben gibt, bei denen diese beiden Extremfälle in jeder Rekursionsebene auftreten. Sie sind damit auch Beispiele für den besten und schlechtesten Fall.

Pivot immer der Median



$\Theta(n \log n)$

vorsortiert



$\Theta(n^2)$

□

Welcher Fall ist typisch? In der folgenden Aussage wird angenommen, dass alle möglichen Sortierungen der Eingabefolge gleich wahrscheinlich sind.

2.4 Satz
Die mittlere Laufzeit von QuickSort ist in $\Theta(n \log n)$. average case

Beweis. Jedes Element von M wird genau einmal zum Pivot-Element. Ist die Eingabereihenfolge in M zufällig, dann auch die Reihenfolge, in der die Elemente zu Pivot-Elementen werden.

Da die Anzahl der Schritte von der Anzahl der Vergleiche bei der Aufteilung in Teilfolgen dominiert wird, bestimmen wir den Erwartungswert der Anzahl von Paaren, die im Verlauf des Algorithmus verglichen werden. Die Elemente von M seien entsprechend ihrer korrekten Sortierung mit $a_1 < \ldots < a_n$ bezeichnet. Werden Elemente a_i, a_j, $i < j$, verglichen, dann ist eines von beiden zu diesem Zeitpunkt Pivot-Element, und keins der Elemente $a_{i+1} < \ldots < a_{j-1}$ war bis dahin Pivot (sonst wären a_i und a_j in verschiedenen Teilarrays). Wegen der zufälligen Reihenfolge der Pivot-Wahlen ist die Wahrscheinlichkeit, dass von den Elementen $a_i < \ldots < a_j$ gerade a_i oder a_j zuerst gewählt werden, gerade $\frac{1}{j-i+1} + \frac{1}{j-i+1}$. Dies gilt für jedes Paar, sodass sich als erwartete Anzahl von Vergleichen und damit mittlere Laufzeit ergibt

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} = 2n \cdot H_n \underset{\text{Satz 1.10}}{\in} \Theta(n \log n) \ .$$

□

Nach Satz 2.3 ist die Laufzeit also immer irgendwo in $\Omega(n \log n) \cap \mathcal{O}(n^2)$, wegen Satz 2.4 jedoch meistens nahe der unteren Schranke.

2.5 Bemerkung (randomisiertes QuickSort)
Die average-case *Analyse zeigt auch, dass eine Variante von Quicksort, in der das Pivot-Element zufällig (statt immer von der ersten Position) gewählt wird, im Mittel auch auf Eingaben schnell ist, die lange vorsortierte Teilfolgen enthalten. Die gleiche Wirkung erhält man durch zufälliges Permutieren der Eingabe vor dem Aufruf von QuickSort.*

2.2.2 MergeSort

Bei QuickSort ist die Aufteilung zwar aufwändig und kann ungünstig erfolgen, garantiert dafür aber eine triviale Kombination der Teilergebnisse. Im

Gegensatz dazu gilt bei MergeSort:

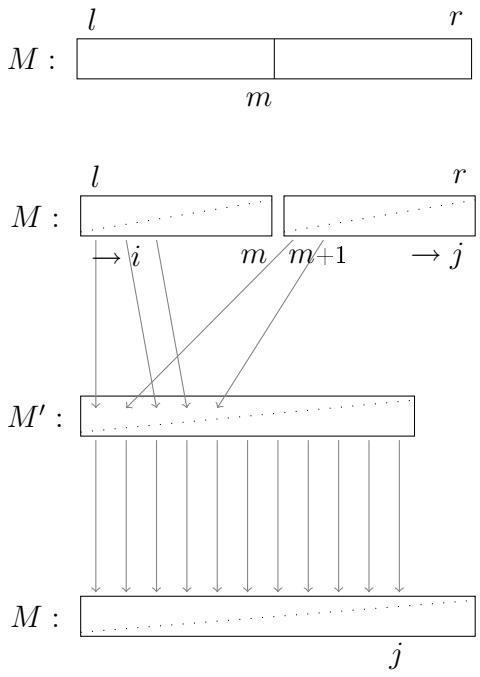
easy split, hard join

- triviale Aufteilung in günstige Teilfolgengrößen
- linearer Aufwand für Kombination (und zusätzlicher Speicherbedarf)

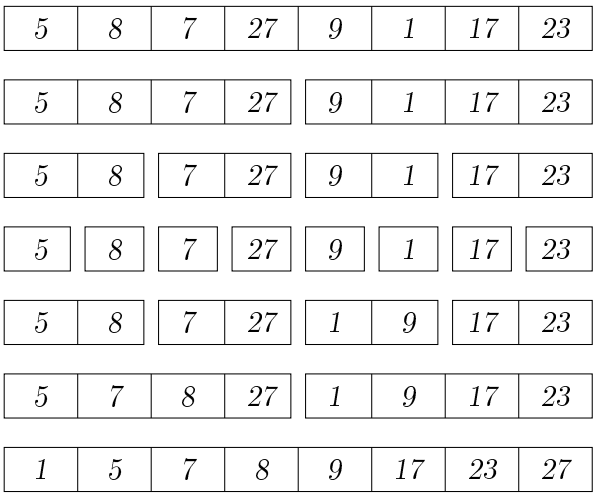
Algorithmus 7: MergeSort

Aufruf: `MergeSort`$(M, 1, n)$

```
MergeSort(M, l, r) begin
    if l < r then
        m ← ⌊(l+r)/2⌋
        MergeSort(M, l, m)
        MergeSort(M, m+1, r)
        i ← l; j ← m+1; k ← l
        while i ≤ m and j ≤ r do
            if M[i] ≤ M[j] then
                M'[k] ← M[i];
                i ← i+1
            else
                M'[k] ← M[j];
                j ← j+1
            k ← k+1
        for h = i, ..., m do
            M[k+(h−i)] ← M[h]
        for h = l, ..., k−1 do
            M[h] ← M'[h]
end
```



2.6 Beispiel (MergeSort)



2.7 Satz

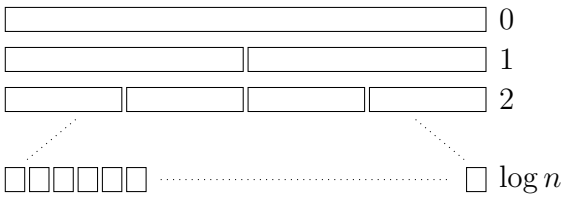
Die Laufzeit von MergeSort ist in $\Theta(n \log n)$.

best, average *und* worst case

Beweis. Wie bei QuickSort setzt sich die Laufzeit aus dem Aufwand für Aufteilung und Kombination zusammen. Für MergeSort gilt:

- konstanter Aufwand für Aufteilung und
- linearer Aufwand für Kombination (Mischen).

In jeder Rekursionsebene ist der Aufwand damit linear in der Gesamtzahl der Elemente



und wegen der rekursiven Halbierung der Teilfolgenlänge ist die Rekursionstiefe immer $\log n$. Die Gesamtlaufzeit ist daher immer in $\Theta(n \log n)$.

□

2.3 HeapSort

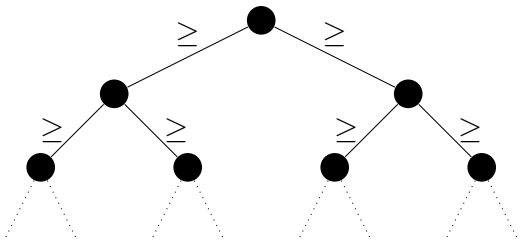
SelectionSort (Abschnitt 2.1) verbringt die meiste Zeit mit der Auswahl des Extremums und kann durch eine besondere Datenstruktur zur Verwaltung der Elemente beschleunigt werden. Wir werden hier immer das Maximum ans Ende der Restfolge setzen und wollen daher einen Datentyp, der schnell das Maximum einer veränderlichen Menge von Werten zurück gibt. Die Prioritätswartenschlange ist ein abstrakter Datentyp für genau diesen Zweck.

MaxPriorityQueque	
	insert(`item` x)
`item`	extractMax()

Falls beide Operationen mit o(n) Laufzeit realisiert sind, ergibt sich eine Verbesserung gegenüber SelectionSort.

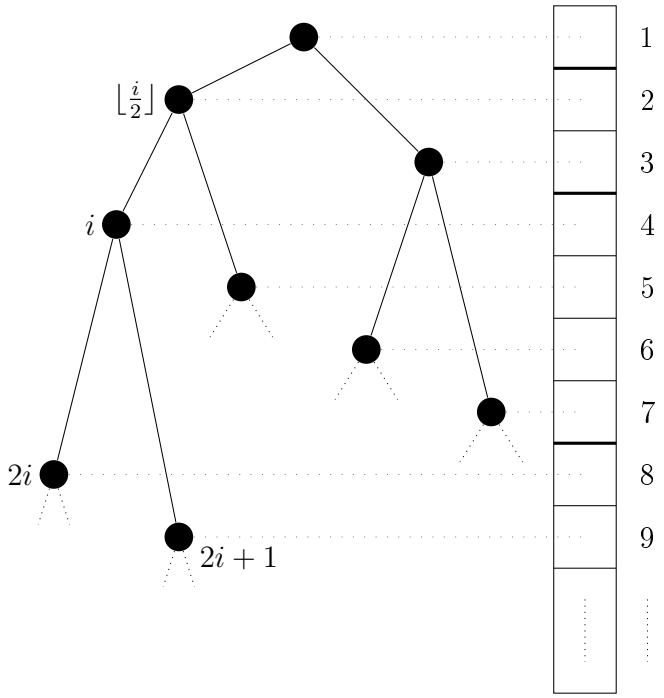
Eine mögliche solche Implementation ist ein binärer Heap: darin werden die Elemente in einem vollständigen binären Baum gespeichert, der die folgende Bedingung erfüllen muss.

Heap-Bedingung: Für jeden Knoten gilt, dass der darin gespeicherte Wert nicht kleiner ist als die beiden Werte in seinen Kinder.



Eine unmittelbare Folgerung aus der Heap-Bedingung ist, dass im ganzen Teilbaum eines Knotens kein größerer Wert vorkommt.

Ein binärer Heap kann in einem Array realisiert werden, d.h. ohne Zeiger etc. zur Implementation der Baumstruktur:



Die beiden Operationen der Prioritätswarteschlange werden dann wie folgt umgesetzt. Bei `insert` $x \rightarrow H$ wird das neue Element hinter allen Elementen im Array eingefügt (also als rechtestes Blatt im Binärbaum) und solange mit seinem Elternknoten vertauscht, bis die Heap-Bedingung wieder hergestellt ist.

Algorithmus 8: insert $x \rightarrow H$ (H enthält aktuell n Elemente)

```
begin
    i ← n + 1
    while (i > 2) and (H[⌊i/2⌋] < x) do      // nicht-striktes and
        H[i] ← H[⌊i/2⌋]
        i ← ⌊i/2⌋
    H[i] ← x
end
```

Da der Binärbaum vom Blatt bis höchstens zur Wurzel durchlaufen wird und jeweils konstanter Aufwand anfällt, ist die Laufzeit in $\mathcal{O}(\log n)$.

Analog kann für `extractMax` $x \leftarrow H$ das erste Arrayelement – die Wurzel des Baumes, und damit das größte Element im Heap – entfernt und das so entstandene „Loch“ jeweils mit einem größten Kind gefüllt werden. Da das schließlich zu löschende Blatt in der Regel nicht an der gewünschten Stelle steht (nämlich am Ende des Arrays), bietet sich jedoch eine andere Vorgehensweise an: Wir schreiben das letzte Element des Arrays in die Wurzel, und vertauschen von dort absteigend solange mit einem größeren Kind, bis die Heap-Bedingung wieder hergestellt ist.

Dieses Methode wird auch `heapify`, der zu Grunde liegende Prozess versickern genannt. Die Implementation erfolgt in der Regel etwas allgemeiner, um die Wiederherstellung der Heap-Bedingung auch an anderer Stelle $i \neq 1$ als der Wurzel veranlassen zu können und die rechte Grenze (den rechtesten tiefsten Knoten) vorgeben zu können, an dem die Vertauschungen stoppen sollen. Um Zuweisungen einzusparen, werden die Vertauschungen außerdem nicht explizit durchgeführt, sondern das Element x in der Wurzel (des Teilbaums) zwischengespeichert, und das entstandene Loch absteigend von unten gefüllt, bis das Element selbst einzutragen ist.

Algorithmus 9: Wiederherstellung der Heap-Bedingung

```
heapify(i.r) begin
    x ← H[i];   j ← 2i
    while j ≤ r do
        if (j < r) and (H[j+1] > H[j]) then
            j ← j + 1
        if x < H[j] then
            H[i] ← H[j]
            i ← j;   j ← 2i
        j ← r + 1
    H[i] ← x
end
```

Unter Verwendung von `heapify` kann die Extraktion des Maximums jetzt wie folgt durchgeführt werden.

Algorithmus 10: extractMax $x \leftarrow H$

```
begin
    if n > 0 then
        x ← H[1]
        if n > 0 then
            H[1] ← H[n]
            heapify(1, n)
        n ← n − 1
    else
        error „Heap leer“
end
```

Wieder wird der Binärbaum maximal von der Wurzel zu einem Blatt durchlaufen, sodass die Opertationen Einfügen und Maximumssuche in $\mathcal{O}(\log n) \subset o(n)$ Zeit augeführt werden.

Durch Einfügen aller Elemente in einen Heap und wiederholter Extraktion des Maximums kann SelectionSort also schneller implementiert werden. Wir vermeiden nun noch die Erzeugung einer Instanz des Datentyps und führen die notwendigen Operationen direkt im zu sortierenden Array aus. Der sich daraus ergebende Sortieralgorithmu heißt HeapSort und besteht aus zwei Phasen:

1. Aufbau des Heaps:
   Herstellen der Heap-Bedingung von unten nach oben,

2. Abbau des Heaps:
   Maximumssuche und Vertauschen nach hinten

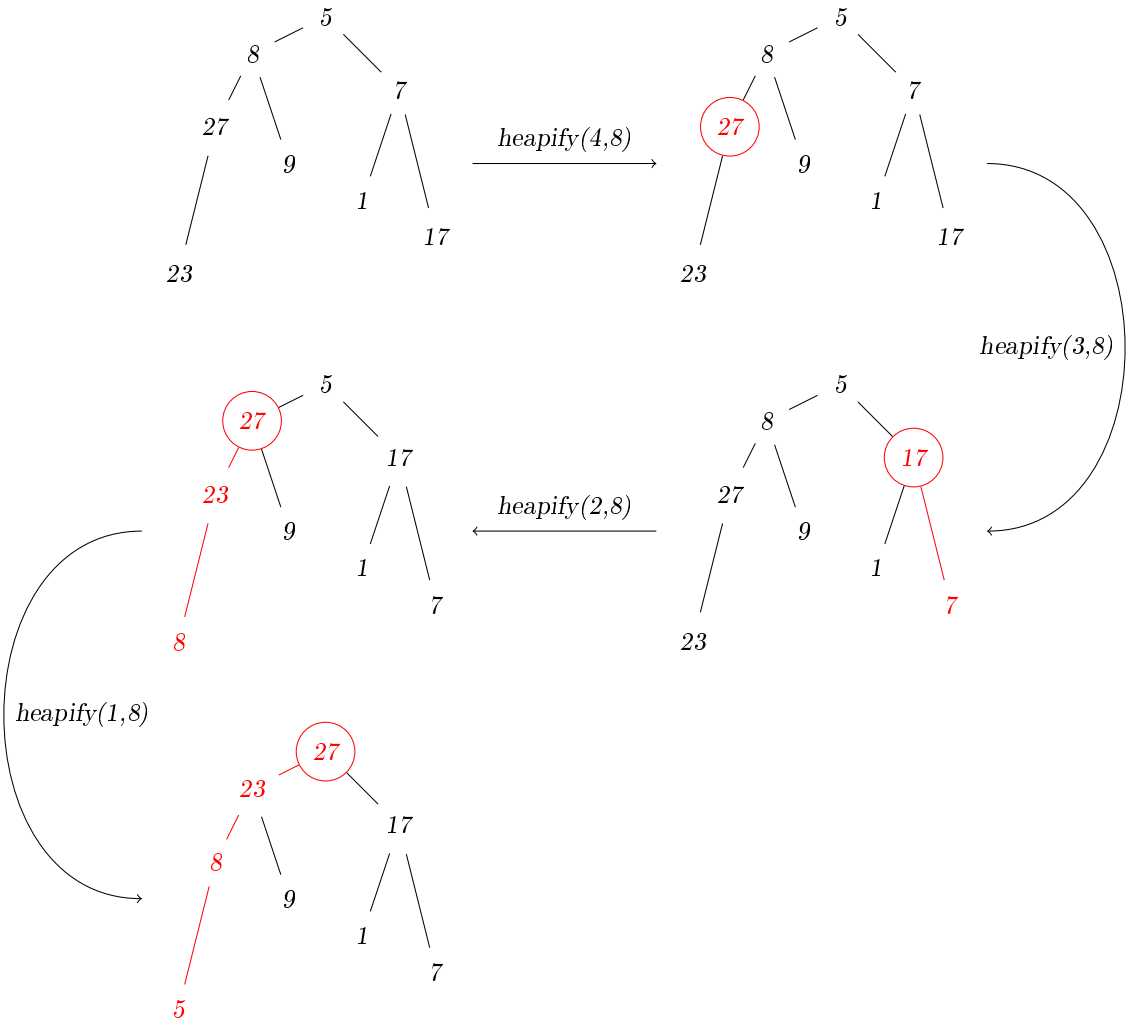
Unter Verwendung von `heapify` ist die Implemention denkbar einfach.

Algorithmus 11: HeapSort

```
begin
    for i = ⌊n/2⌋, ..., 1 do                          // Aufbau
        heapify(i, n)
    for i = n, ..., 2 do                               // Abbau
        vertausche M[1], M[i]
        heapify(1, i − 1)
end
```

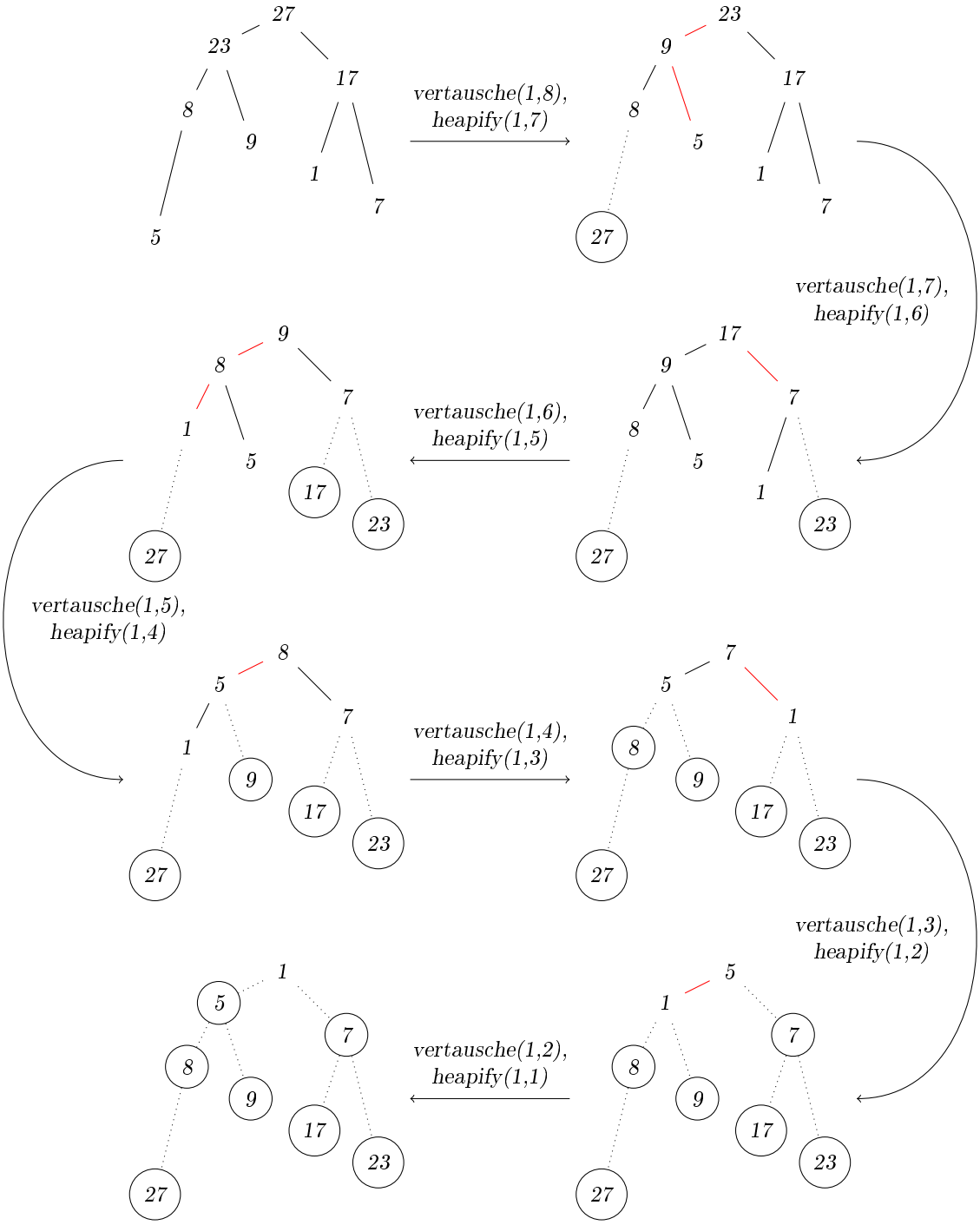
2.8 Beispiel

Aufbau des Heaps:



2.9 Beispiel

Abbau des Heaps:



2.10 Satz
Die Laufzeit von HeapSort ist in $\mathcal{O}(n \log n)$.

Beweis. Wir nehmen zuächst an, dass $n = 2^k - 1$. Beim Aufbau des Heaps werden $\lfloor n/2 \rfloor$ Elemente in ihren Teilbäumen versickert. Für die ersten $n/4$ Elemente haben diese Höhe 1, für die nächsten $n/8$ Elemente die Höhe 2 und allgemein gibt es $n/2^i$ Elemente der Höhe i, $i = 2, \ldots, \log n$. Damit ist der Aufwand für den Aufbau

$$\begin{aligned}
\sum_{i=2}^{\lceil \log n \rceil} \frac{n}{2^i} \cdot i &= n \sum_{i=2}^{\log n} \frac{i}{2^i} \\
&= n \left(2 \sum_{i=2}^{\log n} \frac{i}{2^i} - \sum_{i=2}^{\log n} \frac{i}{2^i} \right) \quad \text{(konstruktive Null)} \\
&= n \left(\sum_{i=1}^{\log n - 1} \frac{i+1}{2^i} - \sum_{i=2}^{\log n} \frac{i}{2^i} \right) \\
&= n \left(\frac{2}{2} + \sum_{i=2}^{\log n - 1} \left(\frac{i+1}{2^i} - \frac{i}{2^i} \right) - \frac{\log n}{n} \right) \\
&= n \left(1 + \underbrace{\sum_{i=2}^{\log n - 1} \frac{1}{2^i}}_{<1} - \frac{\log n}{n} \right) \\
&< 2n \in \Theta(n) \; .
\end{aligned}$$

Ist nun allgemein $2^k - 1 < n < 2^{k+1} - 1$, dann bleiben die Überlegungen richtig und der Aufwand verdoppelt sich höchstens.

Beim Abbau des Heaps ist die Höhe der Wurzel zu jedem Zeitpunkt höchstens $\log n$, sodass der Abbau $\mathcal{O}(n \log n)$ Zeit benötigt. □

Wie MergeSort kommt HeapSort also in jedem Fall mit Laufzeit $\mathcal{O}(n \log n)$ aus, braucht aber nur $\mathcal{O}(1)$ zusätzlichen Speicher. Außerdem ist die Laufzeit im besten Fall linear, z.B. wenn die Eingabe bereits sortiert ist (da die Elemente dann nie versickert werden müssen).

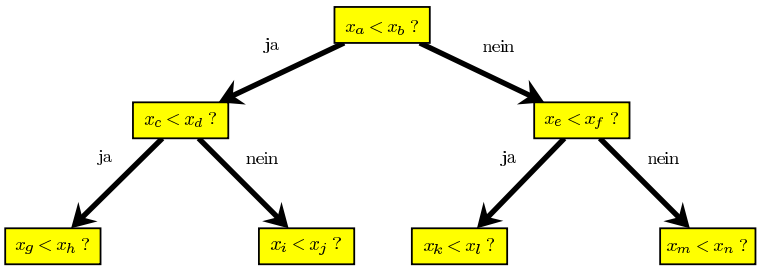
Es bleibt die Frage, ob man nicht auch noch schneller sortieren kann?

2.4 Untere Laufzeitschranke

Eine triviale untere Schranke für die Laufzeit von Sortieralgorithmen ist $\Omega(n)$, da natürlich alle Elemente der Eingabefolge berücksichtigt werden müssen. Wir zeigen in diesem Abschnitt aber, dass die bisher erreichte Laufzeit von $\mathcal{O}(n \log n)$ für die in Problem 2.1 formulierte Aufgabenstellung bestmöglich ist.

Wenn keine Einschränkungen der zu sortierenden Elemente vorliegen, muss ein Sortieralgorithmus Paare von Elementen vergleichen, um ihre Reihenfolge zu bestimmen. Ein solcher vergleichsbasierter Algorithmus heißt auch allgemeines Sortierverfahren (andere Beispiele betrachten wir im nächsten Abschnitt). Wieviele Vergleiche muss der beste Sortieralgorithmus (im schlechtesten Fall) mindestens machen?

Abhängig vom Ergebnis des ersten Vergleichs wird der Algorithmus irgendwelche weiteren Schritte ausführen, insbesondere weitere Vergleiche. Wir können den Ablauf des Algorithmus daher mit einem Abstieg im *Entscheidungsbaum* gleich setzen: An der Wurzel steht der erste Vergleich, an den beiden Kindern stehen die Vergleiche, die im Ja- bzw. Nein-Fall als nächste ausgeführt werden, usw.



Bei n verschiedenen zu sortierenden Elementen (wir schätzen den schlechtesten Fall ab!) hat der Entscheidungsbaum $n!$ Blätter, denn es gibt $n!$ mögliche Eingabereihenfolgen, und wenn zwei verschiedene Eingaben denselben Ablauf (insbesondere die gleichen Umsortierungen) zur Folge haben, wird eine von beiden nicht richtig sortiert. Die Anzahl der Vergleiche entspricht aber gerade der Höhe des Entscheidungsbaumes. In einem Binärbaum haben $n!$ Blätter mindestens $\frac{n!}{2}$ Vorgänger, die wiederum mindestens $\frac{n!}{2^2}$ Vorgänger ha-

ben, usw. Der längste Weg zur Wurzel hat damit mindestens

$$\log n! = \log[n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1] \geq \log \left[\left(\frac{n}{2} \right)^{\frac{n}{2}} \right] = \frac{n}{2} \cdot \log \frac{n}{2} \in \Omega(n \log n)$$

Knoten. Es kann daher keinen vergleichsbasierten Sortieralgorithmus geben, der bei jeder Eingabe eine Laufzeit in $o(n \log n)$ hat.

Wir haben damit den folgenden Satz bewiesen.

2.11 Satz
Jedes allgemeine Sortierverfahren benötigt im schlechtesten Fall $\Omega(n \log n)$ Vergleiche.

2.5 Sortierverfahren für spezielle Universen

Die untere Laufzeitschranke für allgemeine Sortierverfahren kann nur unterboten werden, wenn zusätzliche Annahmen über die zu sortierenden Elemente gemacht werden dürfen. In diesem Abschnitt betrachten wir Spezialfälle, bei denen die Universen aus Zahlen bestehen. Die Sortierverfahren können dann die Größe dieser Zahlen zu Hilfe nehmen; tatsächlich kommen die Algorithmen ganz ohne paarweise Vergleiche aus.

2.5.1 BucketSort

bucket: „Eimer", „Kübel"

Voraussetzung für BucketSort ist, dass die Elemente der Eingabe reelle Zahlen sind. Wir nehmen an, dass $\mathcal{U} = (0, 1]$. Dies ist keine Einschränkung, da andere Eingabewerte immer entsprechend verschoben und skaliert werden können.

Ausgehend von der Hoffnung, dass die Eingabezahlen $a_1, \ldots, a_n$ im Intervall $(0, 1]$ einigermaßen gleichmäßig verteilt sind, werden *buckets*

$$B_i = \left\{ a_j \,:\, \frac{i}{n} < a_j \leq \frac{i+1}{n} \right\} \quad \text{für } i = 0, \ldots, n-1,\ j = 1, \ldots, n$$

erzeugt und mit irgendeinem anderen Verfahren separat sortiert. Die sortierten Teilfolgen können dann einfach aneinander gehängt werden. Dahinter steht die Überlegung, dass es besser ist, k Teilprobleme der Größe n/k zu

bearbeiten, als eines der Größe n, wenn die Laufzeitfunktion schneller als linear wächst (was für allgemeine Sortierverfahren ja der Fall sein muss). Im Idealfall landet jeder Eingabewert in einem eigenen *bucket* und das Sortieren entfällt.

Algorithmus 12: BucketSort

```
begin
    for i = 0,...,n-1 do B[i] ← leere Liste
    for i = 0,...,n-1 do append B[⌈n · M[i]⌉ - 1] ← M[i]
    j ← 0
    for i = 0,...,n-1 do
        sort(B[i])                    // Sortierverfahren frei wählbar
        while B[i] nicht leer do
            M[j] ← detachFirst(B[i])
            j ← j + 1
end
```

Wegen des Rückgriffs auf einen anderen Algorithmus für die Teilprobleme ist BucketSort ein so genanntes Hüllensortierverfahren.

2.12 Satz
Die mittlere Laufzeit von BucketSort ist in $\mathcal{O}(n)$.

Beweis. Annahme: Die zu sortierenden Elemente sind gleichverteilt aus $(0, 1]$. Die Laufzeit wird dominiert von der Sortierung der Buckets (alle übrigen Schritte benötigen Laufzeit $\mathcal{O}(n)$). Wird ein Sortierverfahren verwendet, das Buckets der Größe $n_i = |B[i]|$ in $\mathcal{O}(n_i^2)$ sortiert, ist die Laufzeit $\mathcal{O}(\sum_{i=0}^{n-1} n_i^2)$. Dies asymptotisch gleich mit

$$\mathcal{O}\left(\sum_{0<j_1<j_2\leq n} b_{i_{j_1}} \cdot b_{i_{j_2}}\right) \quad \text{für } b_{i_j} = \begin{cases} 1 & \text{falls } a_j \in B[i] \\ 0 & \text{sonst} \end{cases}$$

Im Mittel ist $b_{i_{j_1}} \cdot b_{i_{j_2}}$ gerade $1/n$, weil es n^2 Kombinationen von Buckets gibt, in denen a_{j_1} und a_{j_2} liegen können, von denen n gerade gleiche Buckets darstellen. Also $\mathcal{O}(\sum n_i^2) = \mathcal{O}(\sum b_{i_{j_1}} \cdot b_{i_{j_2}}) = \mathcal{O}(n)$. □

2.5.2 CountingSort

Sind nur ganzzahlige Werte zu sortieren, d.h. ist $\mathcal{U} \subseteq \mathbb{N}$, kann BucketSort so vereinfacht werden, dass jedem möglichen Wert ein Eimer entspricht und für jeden davon nur die Elemente mit diesem Wert gezählt werden. Die Sortierung ergibt sich dannn einfach dadurch, dass für jeden möglichen Wert in der Reihenfolge vom kleinsten zum größten die Anzahl seiner Vorkommen wieder in das Array M geschrieben wird.

Wir nehmen der Einfachheit halber an, dass es sich um die Zahlen $\{0, \ldots, k-1\}$ handelt (andere endliche Intervalle können wieder entsprechend verschoben werden).

Algorithmus 13:

```
begin
    for j = 0, ..., k − 1 do C[j] ← 0
    for i = 1, ..., n do C[M[i]] ← C[M[i]] + 1
    i ← 1
    for j = 0, ..., k do
        while C[j] > 0 do
            M[i] ← j;   i ← i + 1
            C[j] ← C[j] − 1
end
```

Diese Version ist allerdings von geringer praktischer Bedeutung, da sie voraussetzt, dass tatsächlich nur Zahlen sortiert werden. Sind die Zahlen mit anderen Daten assoziiert, dann geht die Zuordnung verloren, weil die Laufvariable j in der letzte Schleife nur die möglichen Werte annimmt, aber die zu diesen Wert gehörigen Daten nicht in konstanter Zeit ermittelt werden können.

„Sortieren durch Abzählen“

Um die Zuordnung zu erhalten, wird ein Zwischenspeicher eingeführt, damit die Elemente der Eingabe nicht überschrieben werden. Während bei BucketSort vom Wert eines Elements auf die Position geschlossen wurde, geschieht dies beim folgenden CountingSort durch Bestimmung der Anzahl kleinerer Elemente. Durch den eingeschränkten Wertebereich können diese Anzahlen effizient bestimmt werden, indem in einer dritten Vorverarbeitungsschleife die den Positionen entsprechenden Präfixsummen der gezählten Vorkommen gebildet werden. Um das Verfahren darüber hinaus stabil zu machen, werden

die Intervalle gleicher Werte von hinten nach vorne ausgelesen.

Algorithmus 14: CountingSort

```
begin
    for j = 0, ..., k − 1 do C[j] ← 0
    for i = 1, ..., n do C[M[i]] ← C[M[i]] + 1
    for j = 1, ..., k − 1 do C[j] ← C[j − 1] + C[j]
    for i = n, ..., 1 do
        M'[C[M[i]]] ← M[i]
        C[M[i]] ← C[M[i]] − 1
    M ← M'
end
```

Die Laufzeit lässt sich unmittelbar aus den Schleifen ablesen.

2.13 Satz
Die Laufzeit von CountingSort ist in $\mathcal{O}(n + k)$.

2.5.3 RadixSort

Selbst bei ganzen Zahlen kann das Intervall möglicher Werte im Allgemeinen nicht genügend stark eingeschränkt werden (man erhält also zu große k). Um die Idee von CountingSort trotzdem verwenden zu können, nutzen wir nun aus, dass bei Darstellung aller Zahlen bezüglich einer festen Basis zwar die Länge variiert, nicht aber die Anzahl der vorkommenden Ziffern.

Hat man ein stabiles Sortierverfahren für das Universum $\mathcal{U} = \{0, \ldots, d-1\}$ aus Ziffern in der d-ären Darstellung, dann können die Eingabewerte stellenweise von hinten nach vorne sortiert werden. Durch die Stabilität bleibt die Reihenfolge bezüglich niederer Stellen erhalten, wenn bezüglich einer höherwertigen Stelle soriert wird.

Algorithmus 15: RadixSort(M)

```
begin
    s ← ⌊log_d(max_{i=1,...,n} M[i])⌋
    for i = 0, ..., s do
        sortiere M bzgl. Stelle i (stabil) (∗)
end
```

Als Stellen-Sortierverfahren für $(*)$ bietet sich natürlich CountingSort an.

2.14 Satz
Die Laufzeit von RadixSort ist in $\Theta(s \cdot (n + d))$.

Auch wenn dadurch zusätzlicher Speicher benötigt wird, lohnt sich ein großes d (z.B. $d = 256$ für byteweise Aufteilung), da s dann exponentiell kleiner wird.

Die Zähler und sogar der für CountingSort benötigte Zusatzspeicher lassen sich aber auch ganz vermeiden, denn für den Spezialfall von Binärzahlen (oder wenn man die Ziffern eines d-ären System in ihrer Binärdarstellung notiert), kann die folgende Variante verwendet werden. Darin wird wie bei QuickSort partitioniert, allerdings nicht aufgrund eines Pivots, sondern jeweils aufgrund der b-ten Binärziffer. In der Binärdarstllung seien die Bits dabei vom höchstwertigen zum niedrigstwertigen nummeriert und $bit(B_{s-1}B_{s-2}\cdots B_1B_0, b) = B_b$ für $b \in \{0, \ldots, s-1\}$.

Algorithmus 16: RadixExchangeSort (für s-stellige Binärzahlen)

Aufruf: `RadixExchangeSort(`$M, 1, n, s-1$`)`

```
RadixExchangeSort(M, l, r, b) begin
    if l < r then
        i ← l; j ← r
        while i ≤ j do
            while i ≤ j and bit(M[i], b) = 0 do i ← i + 1
            while i ≤ j and bit(M[j], b) = 1 do j ← j − 1
            if i < j then vertausche M[i], M[j]
        if b > 0 then RadixExchangeSort(M, l, i − 1, b − 1)
        RadixExchangeSort(M, i, r, b − 1)
end
```

2.15 Satz
Die Laufzeit von RadixExchangeSort für s-stellige Binärzahlen ist in $\Theta(s \cdot n)$.

Das Verfahren ist besonders geeignet, wenn s klein ist im Verhältnis zu n, insbesondere falls s konstant oder zumindest $s \in \mathcal{O}(\log n)$. Es ist insbesondere dann nicht geeignet, wenn wenige Zahlen mit großer Bitlänge zu sortieren sind. Ein weiterer Nachteil ist, dass die für die Partitionierung benötigten

Austauschoperationen wie bei QuickSort verhindern, dass das Verfahren stabil ist.

2.6 Gegenüberstellung

Zum Abschluss des Kapitels sind einige der wesentlichen Vor- und Nachteile der verschiedenen Sortieralgorithmen in Tabelle 2.6 einander gegenüber gestellt.

Algorithmus	**Laufzeitklasse**			**Zusatz-**	**stabil**	**Einschränkung**
	worst	average	best	**Speicher**	?	
SelectionSort	n^2	n^2	n^2	$\mathcal{O}(1)$	✓	keine
QuickSort	n^2	$n \log n$	$n \log n$	$\mathcal{O}(1)$	✗	keine
MergeSort	$n \log n$	$n \log n$	$n \log n$	$\mathcal{O}(n)$	✓	keine
HeapSort	$n \log n$	$n \log n$	n	$\mathcal{O}(1)$	✓	keine
untere Schranke	$n \log n$	$n \log n$	n	$\mathcal{O}(1)$	k.A.	keine
BucketSort	$n \log n$	n	n	$\mathcal{O}(n)$	✓	reelle Zahlen aus $(0, 1]$
CountingSort	$n + k$	$n + k$	$n + k$	$\mathcal{O}(n + k)$	✓	ganze Zahlen aus $\{0, k - 1\}$
RadixSort	$s \cdot (n + d)$	$s \cdot (n + d)$	$s \cdot (n + d)$	$\mathcal{O}(n + d)$	✓	d-äre Zahlen, Wortlänge s
RadixExchangeSort	$s \cdot n$	$s \cdot n$	$s \cdot n$	$\mathcal{O}(1)$	✗	Binärzahlen, Bitlänge s

Tabelle 2.1: Zusammenstellung der behandelten Sortierverfahren