2 Sortieralgorithmen

Laut Donald E. Knuth[12] schätzten Computerhersteller in den 60er Jahren, dass mehr als 25 Prozent der Rechenzeit eines durchschnittlichen Computers dazu verwendet wurde zu sortieren. In der Tat gibt es unzählige Anwendungen in denen Datensätze sortiert werden müssen: Unix gibt beispielsweise die Dateien in jedem Verzeichnis alphabetisch sortiert aus; Sortieren ist vor dem Zuteilen von Briefen notwendig (etwa nach Postleitzahl, Bereich usw.); Suchmaschinen sortieren die Treffer nach Relevanz; Internetkaufhäuser sortieren Waren nach den verschiedensten Kriterien, wie Beliebtheit, Preis, usw.; Datenbanken müssen in der Lage sein, Treffer von Suchanfragen nach bestimmten Kriterien zu sortieren.

Wir stellen im Folgenden vier Sortieralgorithmen vor: Insertion Sort, Quicksort, Mergesort und Heapsort. Insertion Sort ist ein sehr einfacher Sortieralgorithmus, den viele der Leser ohne algorithmische Vorbildung – hätten sie die Aufgabe gehabt, eine Sortierroutine zu implementieren – wahrscheinlich programmiert hätten. Die Beschreibung von Quicksort benutzen wir dazu verschiedene Entwurfstechniken und Optimierungsmöglichkeiten zu beschreiben und auch dazu, genau auf die Funktionsweise von sog. Divide-And-Conquer-Algorithmen einzugehen. Im Zuge der Präsentation des Heapsort-Algorithmus gehen wir auch kurz auf die Funktionsweise einer sog. Heapdatenstruktur ein; detailliertere Beschreibungen von Heaps finden sich in einem eigenen Kapitel, dem Kapitel 4 ab Seite 115.

2.1 Insertion Sort

Vermutlich verwenden die meisten Menschen Insertion Sort, wenn sie eine Hand voll Karten sortieren wollen: Dabei nimmt man eine Karte nach der anderen und fügt diese jeweils in die bereits auf der Hand befindlichen Karten an der richtigen Stelle ein; im einfachsten Fall wird die „richtige Stelle“ dabei einfach dadurch bestimmt, dass die Karten auf der Hand sukzessive von links nach rechts durchlaufen werden bis die passende Stelle gefunden ist. Abbildung 2.1 veranschaulicht diese Funktionsweise anhand der Sortierung einer Beispielliste nochmals graphisch.

2.1.1 Implementierung: nicht-destruktiv

Eine mögliche Implementierung besteht aus zwei Funktionen: Der Funktion $insND(l,key)$, die den Wert key in eine schon sortierte Liste l einfügt. Das Kürzel „ND“ am Ende des Funktionsnames steht für „nicht-destruktiv“, d. h. die in Listing 2.1 gezeigte Implementierung verändert die als Parameter übergebene Liste l nicht; sie liefert stattdessen als Rückgabewert eine neue Liste, die eine Kopie der übergebenen Liste, mit dem Wert key

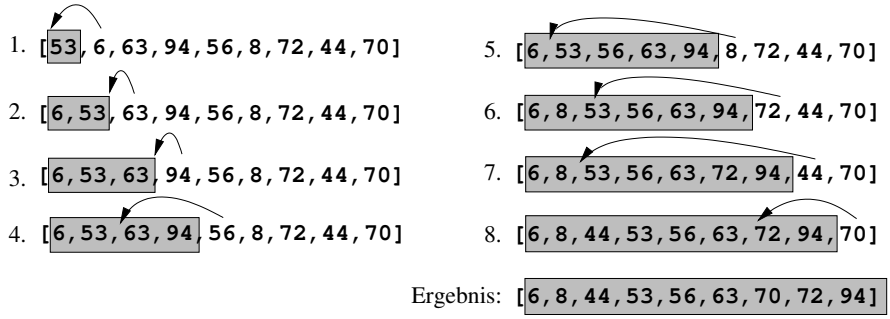


Abb. 2.1: *Veranschaulichung der Funktionsweise von Insertion Sort auf der anfänglich unsortierten Liste* [53, 6, 63, 94, 56, 8, 72, 44, 70]. *Wie man sieht, wird immer das jeweils nächste Element in den schon sortierten Teil der Liste (grau markiert) einsortiert.*

an der „richtigen“ Stelle, enthält.

```
1 def insND(l,key):
2   return [x for x in l  if  x <=key]  + [key]  + [x for x in l  if  x>key]
```

Listing 2.1: *Einfügen eines Wertes in eine schon sortierte Liste*

Die Ergebnisliste besteht zunächst aus allen Werten aus l, die kleiner oder gleich *key* sind – diese werden in der linken Listenkomprehension gesammelt –, gefolgt von *key*, gefolgt von allen Werten aus l, die größer als *key* sind – diese Werte werden in der rechten Listenkomprehension gesammelt.

Listing 2.2 zeigt, wie nun der eigentliche Insertion-Sort-Algorithmus mit Hilfe von *insND* sehr einfach rekursiv implementiert werden kann.

```
1 def insertionSortRek(l):
2   if  len(l)<=1: return l
3   else: return insND(insertionSortRek(l[1:]), l[0])
```

Listing 2.2: *Rekursive Implementierung von Insertion Sort*

Zeile 2 definiert den Rekursionsabbruch: eine einelementige oder leere Liste ist schon sortiert und kann einfach zurückgeliefert werden. Zeile 3 definiert den Rekursionsabstieg: eine Liste kann dadurch sortiert werden, indem das erste Element entfernt wird, der Rest der Liste durch den rekursiven Aufruf *insertionSortRek*($l[1:]$) sortiert wird und anschließend das entfernte Element $l[0]$ wieder an der richtigen Stelle eingefügt wird. Für Neulinge der rekursiven Programmierung empfiehlt sich für das Verständnis der Funktionsweise von *insertionSortRek* das strikte Befolgen des in Abschnitt 1.2.3 beschriebenen „Kochrezepts“: Man gehe einfach davon aus, dass *insertionSortRek*($l[1:]$) für die kürzere Teilliste $l[1:]$ das Richtige tut – nämlich $l[1:]$ zu sortieren. Unter dieser Annahme sollte man sich überlegen, wie man das fehlende Element $l[0]$ mit dieser sortierten Teilliste kombinieren muss, damit eine sortierte Gesamtliste entsteht.

Aufgabe 2.1

Implementieren Sie – ebenfalls unter Verwendung von *insND* – eine iterative Variante von *insertionSortRek*.

2.1.2 In-place Implementierung

Listing 2.3 zeigt als Alternative eine in-place Implementierung des Insertion-Sort-Algorithmus – ohne die Verwendung von Zwischen-Listen (dies ist auch der Grund dafür, dass die folgende Implementierung etwas schneller ist).

```
1 def insertionSort(l):
2   for j in range(1,len(l)):
3     key = l[j]
4     i = j-1
5     while i ≥ 0 and l[i] > key:
6       l[i+1] = l[i]
7       i = i - 1
8     l[i+1] = key
```

Listing 2.3: *In-Place Implementierung des Insertion-Sort-Algorithmus*

In der **for**-Schleife wird in der Variablen j jede Position der Liste durchlaufen; das j-te Element ($l[j]$) ist dabei immer derjenige Wert, der in den schon sortierten Teil der Liste eingefügt werden soll. Die **while**-Schleife zwischen Zeile 5 und 7 durchläuft dabei den schon sortierten Teil der Liste auf der Suche nach der passenden Stelle i – gleichzeitig werden die durchlaufenen Elemente nach „rechts“ verschoben, um Platz für den einzufügenden Wert zu schaffen.

2.1.3 Laufzeit

Machen wir uns Gedanken über die Laufzeit von Insertion Sort zur Sortierung einer Liste der Länge n:

Worst Case. Im „schlimmsten“ denkbaren Fall muss die bereits sortierte Liste immer jeweils vollständig durchlaufen werden, um die richtige Einfügeposition zu finden. Im ersten Durchlauf hat die bereits sortierte Liste die Länge 1, im zweiten Durchlauf die Länge 2, usw. Im letzten, also $(n-1)$-ten, Durchlauf hat die bereits sortierte Liste die Länge $n-1$. Insgesamt erhalten wir also als Laufzeit $L_{worst}(n)$ bei einer Eingabe der Größe n:

$$L_{worst}(n) = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} = O(n^2)$$

Best Case. Im günstigsten Fall genügt immer nur ein Vergleich, um die Einfügeposition in den schon sortierten Teil der Liste zu bestimmen. Da es insgesamt $n-1$

Schleifendurchläufe gibt, erhalten wir also als Laufzeit $L_{best}(n)$ im besten Fall bei einer Eingabe der Größe n

$$L_{best}(n) = \sum_{k=1}^{n-1} 1 = n - 1 = O(n)$$

Average Case. Wird eine k-elementige schon sortierte Liste linear durchlaufen um die richtige Einfügeposition für ein neues Element zu suchen, so ist es im besten Fall möglich, dass man nur einen Vergleich benötigt; es ist möglich, dass man 2 Vergleiche benötigt, usw. Schließlich ist es auch (im ungünstigsten Fall) möglich, dass man k Vergleiche benötigt. Geht man davon aus, dass all diese Möglichkeiten mit gleicher Wahrscheinlichkeit auftreten, so kann man davon ausgehen, dass die Anzahl der Vergleiche im Durchschnitt

$$\frac{1 + \cdots + k}{k} = \frac{k(k+1)/2}{k} = \frac{k+1}{2}$$

beträgt, d. h. in jedem der insgesamt $n - 1$ Durchläufe werden im Durchschnitt $\frac{k+1}{2}$ Vergleiche benötigt. Summiert über alle Durchläufe erhält man also

$$\sum_{k=1}^{n-1} \frac{k+1}{2} = \frac{1}{2} \sum_{i=1}^{n-1} (k+1) = \frac{1}{2} \sum_{i=2}^{n} k = \frac{1}{2} \left(\frac{n(n+1)}{2} - 1 \right) = \frac{n^2 + n - 2}{4}$$

Somit gilt für die Laufzeit $L_{av}(n)$ im Durchschnittsfall bei einer Eingabe der Größe n

$$L_{av}(n) = O(n^2)$$

Insertion Sort vs. Pythons *sort*-Funktion. Tabelle 2.1 zeigt die Laufzeiten des im vorigen Abschnitt implementierten Insertion-Sort-Algorithmus *insertionSort* im Vergleich zur Laufzeit von Pythons mitgelieferter Suchfunktion *list . sort*() – Pythons Standard- Sortierfunktion – für die Sortierung einer Liste mit 50 000 zufällig gewählten *long int*-Zahlen. Wie kann Pythons Standard- Sortierfunktion so viel schneller sein?

Implementierung	Laufzeit (in sek)
insertionSort	244.65
list.sort	0.01

Tabelle 2.1: *Laufzeiten des im letzten Abschnitt implementierten Insertion Sort Algorithmus im Vergleich zu Pythons Standard-Sortierfunktion sort*() *für ein Eingabe-Liste mit* 50 000 *long int-Zahlen*

Im nächsten Abschnitt machen wir uns Gedanken darüber, wie schnell ein Sortieralgorithmus eine Liste von n Zahlen maximal sortieren kann.

Aufgabe 2.2

Die Funktion *insertionSort* durchsucht die bereits sortierte Liste *linear* nach der Position, an die das nächste Element eingefügt werden kann. Kann man die Laufzeit von *insertionSort* dadurch verbessern, dass man eine binäre Suche zur Bestimmung der Einfügeposition verwendet, die Suche also in der Mitte der sortierten Teilliste beginnen lässt und dann, abhängig davon, ob der Wert dort größer oder kleiner als der einzufügende Wert ist, in der linken bzw. rechten Hälfte weitersucht, usw.?

Falls ja: Was hätte solch ein Insertion-Sort-Algorithmus für eine Laufzeit? Implementieren Sie Insertion Sort mit binärer Suche.

2.2 Mindestlaufzeit von Sortieralgorithmen

Will man eine Liste $[a_0, \ldots, a_{n-1}]$ von n Elementen sortieren, so können alle Sortieralgorithmen, die vorab keine besonderen Informationen über die zu sortierenden Elemente besitzen, nur aus Vergleichen zwischen Elementpaaren Informationen über deren sortierte Anordnung gewinnen. Der Durchlauf eines jeden Sortieralgorithmus kann als Entscheidungsbaum modelliert werden; jeder Durchlauf durch den Entscheidungsbaum repräsentiert dabei die Vergleiche, die durch einen Sortieralgorithmus ausgeführt werden, während eine konkrete Liste sortiert wird. Für solch einen Entscheidungsbaum, wie etwa der in Abbildung 2.2 gezeigt, gilt: Jeder innere Knoten repräsentiert einen Vergleich; der linke Teilbaum behandelt den Fall, dass der Vergleich positiv ausfällt, und der rechte Teilbaum behandelt den Fall, dass der Vergleich negativ ausfällt. So sind im linken Teilbaum des mit $a_0 < a_1$ beschrifteten Wurzelknotens des Entscheidungsbaums aus Abbildung 2.2 nur noch Sortierungen denkbar, in denen das 0-te Element links vom 1-ten Element steht.

Jeder Vergleich halbiert die Anzahl der bis zu diesem Zeitpunkt noch denkbaren möglichen Sortierungen. Ist ein Blatt erreicht, so hat der Sortieralgorithmus die passende Sortierung gefunden. Jedes Blatt ist mit einer Permutation der Indizes $i = 0, \ldots n-1$ markiert, die der gefundenen Sortierung entspricht.

Aufgabe 2.3

Erstellen Sie einen Entscheidungsbaum, der die Funktionsweise von Insertion Sort beschreibt, zum Sortieren einer 3-elementigen Liste.

Aufgabe 2.4

Würde Insertion Sort, was die getätigten Vergleiche betrifft, so vorgehen, wie durch den in Abbildung 2.2 gezeigten Entscheidungsbaum beschrieben?

Die Worst-Case-Komplexität eines Sortieralgorithmus entspricht genau dem längsten Pfad von der Wurzel zu einem Blatt im entsprechenden Entscheidungsbaum, in ande-

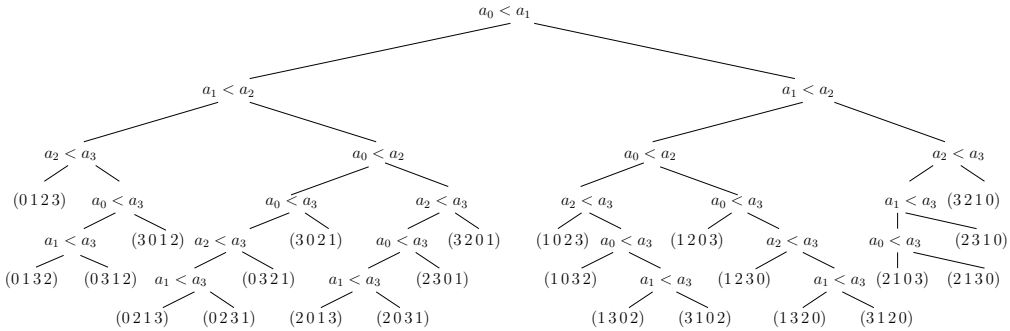


Abb. 2.2: *Ein möglicher Entscheidungsbaum, der modelliert, welche Vergleiche notwendig sind, um eine Liste $[a_0, a_1, a_2, a_3]$ der Länge $n = 4$ zu sortieren. An den Blättern befinden sich alle $n!$ möglichen Permutation. Ein Entscheidungsbaum zum Sortieren einer 4-elementigen Liste muss eine Tiefe von mindestens $\lceil \log_2 4! \rceil = 5$ haben. Der gezeigte Entscheidungsbaum hat eine Tiefe von 6, ist also in diesem Sinne nicht optimal.*

ren Worten: die Worst-Case-Komplexität entspricht der Tiefe des Entscheidungsbaums. Ein Entscheidungsbaum, der die Sortierung einer n-elementigen Liste modelliert, besitzt $n!$ Blätter, d. h. er besitzt mindestens eine Tiefe von $\lceil \log_2 n! \rceil$. Die berühmte *Stirling-Formel* zeigt uns, welches Wachstumsverhalten $\log_2 n!$ besitzt. Die Stirling-Formel besagt, dass $n!$ für große n genauso schnell wächst wie $\sqrt{2\pi n} \cdot (n/e)^n$, und zwar in dem Sinne, dass gilt:

$$\lim_{n \to \infty} \frac{n!}{\sqrt{2\pi n} \cdot (n/e)^n} = 1$$

Somit ergibt sich als untere Schranke für die Worst-Case-Komplexität $L_{worst}(n)$ eines beliebigen Sortieralgorithmus

$$\begin{aligned} L_{worst}(n) \geq \lceil \log_2 n! \rceil &= O(\log_2 (\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n)) \\ &= O(\frac{1}{2} \log_2 2\pi n + n \log_2 \frac{n}{e}) \\ &= O(\log n) + O(n \log n) = O(n \log n) \end{aligned}$$

2.3 Quicksort

Quicksort gehört zur Klasse der sog Divide-And-Conquer-Algorithmen. Bevor wir die Funktionsweise von Quicksort beschreiben, gehen wir in folgendem Abschnitt kurz auf die Besonderheiten dieser Algorithmen ein.

2.3.1 Divide-And-Conquer-Algorithmen

„Divide et Impera“ (deutsch: Teile und Herrsche; englisch: Divide and Conquer) war Julius Cäsars erfolgreiche Strategie ein großes unüberschaubares Reich zu beherrschen. Ein

Divide-And-Conquer-Algorithmus teilt ein Problem p der Größe n in mehrere kleinere Teilprobleme $tp_0, \ldots tp_{k-1}$ auf (häufig ist, wie auch im Falle des Quicksort-Algorithmus, $k = 2$); diese Teilprobleme werden rekursiv gelöst und die so entstandenen Teillösungen $tl_0, \ldots tl_{k-1}$ werden schließlich zu einer Gesamtlösung zusammengefügt. Folgendes Listing formuliert dies nochmals in Python:

def *divideAndConquer*(p):
    $(tp_0, \ldots, tp_{k-1}) = divide_k(p)$

    $tl_0 \quad = divideAndConquer(tp_0)$
    $\ldots \quad = \quad \ldots$
    $tl_{k-1} \; = divideAndConquer(tp_{k-1})$

    return $combine_k(tl_0, \ldots, tl_{k-1})$

Die Laufzeit $L(n)$ eines Divide-And-Conquer-Algorithmus kann am natürlichsten durch eine sog. Rekurrenzgleichung ausgedrückt werden – wir nehmen hierbei der Einfachheit halber an, dass der *divide*-Schritt das Problem in k gleichgroße Teile der Größe n/k zerlegt; L_{div} sei die Laufzeit der *divide*-Funktion, L_{comb} sei die Laufzeit des *combine*-Schritts.

$$L(n) = L_{div}(n) + k \cdot L(\frac{n}{k}) + L_{comb}(n)$$

2.3.2 Funktionsweise von Quicksort

Das Vorgehen von Quicksort bei der Sortierung einer Liste $lst = [a_0, a_1, \ldots, a_{n-1}]$ der Länge n kann folgendermaßen beschrieben werden:

1. Quicksort wählt zunächst ein beliebiges Element $lst[j]$ mit $0 \leq j \leq n-1$ aus der zu sortierenden Liste lst aus. Dieses Element wird *Pivot-Element* genannt.

2. Der *divide*-Schritt: Quicksort zerteilt nun die Liste *lst* in zwei Teil-Listen *lst_l* und *lst_r*. Die „linke“ Teil-Liste *lst_l* enthält alle Elemente aus *lst*, deren Werte kleiner (oder gleich) dem Pivotelement *lst_j* sind; die „rechte“ Teil-Liste enthält alle Elemente aus *lst*, deren Werte größer dem Pivotelement *lst_j* sind.

3. Die Listen *lst_l* und *lst_r* werden rekursiv mit Quicksort sortiert.

4. Der *combine*-Schritt: Die rekursiv sortierten Teil-Listen werden einfach zusammengehängt; das Pivotelement kommt dabei in die Mitte.

Diese Beschreibung der rekursiven Vorgehensweise lässt sich mittels zweier Listenkomprehensionen und entsprechender rekursiver Aufrufe direkt in Python implementieren: Die Listenkomprehension [*x* **for** *x* **in** *lst*[1:] **if** *x*≤*pivot*] berechnet hierbei die linke Teilliste und die Listenkomprehension [*x* **for** *x* **in** *lst*[1:] **if** *x*>*pivot*] berechnet die rechte Teilliste. Listing 2.4 zeigt die Implementierung.

```
1 def quicksort( lst ):
2   if len( lst )≤1: return lst # Rekursionsabbruch
3   pivot = lst[0]
4    lst_l = [a for a in lst[1:] if a ≤ pivot]
5    lst_r = [a for a in lst[1:] if a > pivot]
6   return quicksort( lst_l ) + [pivot] + quicksort( lst_r )
```

Listing 2.4: *Implementierung von Quicksort*

Abbildung 2.3 zeigt als Beispiel die Ausführung von Quicksort veranschaulicht durch zwei zusammengesetzte Binärbäume; der obere Binärbaum modelliert den Rekursionsabstieg, der untere Binärbaum den Rekursionsaufstieg. Eine alternative aber ganz ähnliche graphische Veranschaulichung der Ausführung von Quicksort angewandt auf dieselbe Liste ist in Abbildung 2.4 gezeigt.

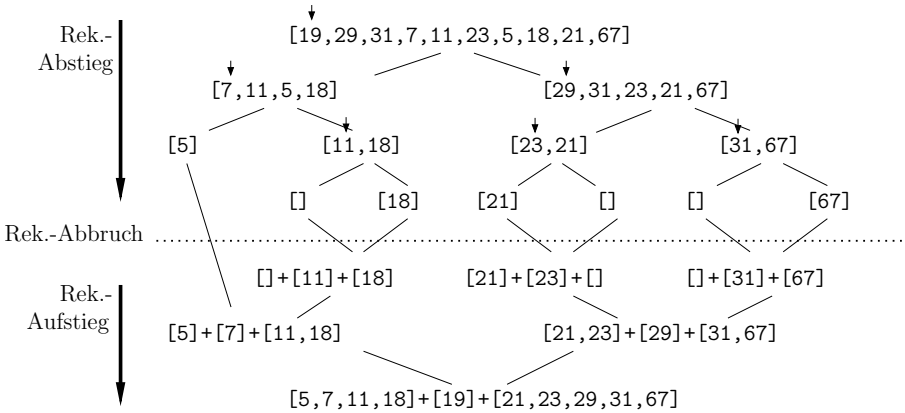


Abb. 2.3: *Darstellung der Funktionsweise von Quicksort am Beispiel der Sortierung der Liste* [19,29,31,7,11,23,5,18,21,67] *dargestellt durch zwei zusammengesetzte Binärbäume, getrennt mit einer gestrichelten Linie, die den Rekursionsabbruch markiert. Für jeden mit einer Liste lst markierten Knoten im oberen Binärbaum gilt: Der linke Teilbaum modelliert den rekursiven Aufruf quicksort(lst_l) und der rechte Teilbaum modelliert den rekursiven Aufruf quicksort(lst_r). Der Weg von der Wurzel des oberen Binärbaums zu den Blättern markiert also den rekursiven Abstieg. Die Pivot-Elemente der Listen sind jeweils mit einem kleinen Pfeil markiert. Eine Verzweigung im unteren Binärbaum entspricht einem combine-Schritt, der zwei sortierte Listen samt dem Pivot-Element zu einer Gesamtlösung zusammensetzt.*

Aufgabe 2.5

Um *quicksort* noch kompakter zu implementieren, verwenden Sie die Hilfsfunktion:

def $o(x,s)$: **return** $[i$ **for** i **in** x **if** $cmp(i,x[0])==s]$

(ein Funktions-Body mit nur einer Zeile ist möglich!)

Abb. 2.4: *Darstellung der Funktionsweise von Quicksort am Beispiel der Sortierung der Liste* [19,29,31,7,11,23,5,18,21,2,67] *dargestellt durch die ausgeführten Kommandos und die Hierarchie der rekursiven Aufrufe. Die Ausdrücke, die den rekursiven Abstieg einleiten (also die rekursiven Aufrufe initiieren), sind schwarz umrandet; die berechneten Werte dieser Ausdrücke, nachdem sie im rekursiven Aufstieg bestimmt wurden, sind grau umrandet.*

2.3.3 Laufzeit

Der günstigste Fall entsteht dann, wenn die gewählten Pivotelemente die Listen immer in jeweils zwei gleichgroße Teillisten aufteilen. In diesem Fall ist die Laufzeit $L_{\text{best}}(n)$ von Quicksort:

$$L_{\text{best}}(n) = 2 \cdot L_{\text{best}}(n/2) + \underbrace{L_{\text{div}}(n) + L_{\text{comb}}(n)}_{O(n)}$$

wobei $L_{\text{div}}(n)$ die Laufzeit der Aufteilung in die beiden Teillisten darstellt und $L_{\text{comb}}(n)$ die Laufzeit der Kombination der rekursiv sortieren Teillisten darstellt. Die Lösung dieser Rekurrenz-Gleichung ist $O(n \log n)$ und damit ist die Laufzeit im bestmöglichen Fall in $O(n \log n)$.

Interessanter ist jedoch der Average-Case-Fall:

Average Case. Wir gehen davon aus, dass die Wahrscheinlichkeit, dass das Pivot-Element das i-kleinste Element von insgesamt n-Elementen ist, $1/n$ beträgt; d. h. wir gehen hier von einer Gleichverteilung aus. Wird das i-kleinste Element als Pivot-Element gewählt, so hat die linke Teilliste eine Größe von $i-1$ und die rechte Teilliste eine Größe von $n-i$; für die Average-Case-Laufzeit $L_{\text{av}}(n)$ zur Sortierung einer n-elementigen Liste durch die in Listing 2.4 gezeigte Funktion *quicksort* ergibt sich für die Average-Case-Laufzeit $L_{\text{av}}(n)$ also folgende Rekurrenz-Gleichung:

$$L_{\text{av}}(n) = \underbrace{(n-1)}_{\text{Partition}} + \frac{1}{n} \cdot \sum_{i=1}^{n} \left(L_{\text{av}}(i-1) + L_{\text{av}}(n-i)\right) + \underbrace{2}_{+\text{-Ops}} \tag{2.1}$$

Da

$$\sum_{i=1}^{n} \left(L_{\text{av}}(i-1) + L_{\text{av}}(n-i)\right) = L_{\text{av}}(0) + \ldots + L_{\text{av}}(n-1) \;+\; L_{\text{av}}(n-1) + \ldots + L_{\text{av}}(0)$$

– also jeder Term $L_{\text{av}}(i)$ in der Summe zweimal vorkommt – kann man die Rekurrenz-Gleichung (2.1) folgendermaßen vereinfachen:

$$L_{\text{av}}(n) = (n+1) + \frac{2}{n} \cdot \sum_{i=0}^{n-1} L_{\text{av}}(i) \tag{2.2}$$

Auf den ersten Blick scheint die Rekurrenz-Gleichung (2.2) schwer aufzulösen; mit einigen „Tricks“ ist sie aber einfacher in den Griff zu bekommen, als so manch andere Rekurrenz-Gleichung. Wir multiplizieren $L_{\text{av}}(n+1)$ mit $n+1$ und $L_{\text{av}}(n)$ mit n:

$$(n+1)L_{\text{av}}(n+1) \;=\; (n+1)(n+2) + \frac{2(n+1)}{n+1} \sum_{i=0}^{n} L_{\text{av}}(i) \qquad \text{und}$$

$$nL_{\text{av}}(n) \;=\; n(n+1) + \frac{2n}{n} \sum_{i=0}^{n-1} L_{\text{av}}(i)$$

Zieht man nun vom $(n+1)$-fachen von $L_{\text{av}}(n+1)$ das n-fache von $L_{\text{av}}(n)$ ab, so erhält man eine einfachere Rekurrenz-Gleichung:

$$\begin{aligned}(n+1)L_{\text{av}}(n+1) - nL_{\text{av}}(n) &= 2(n+1) + 2L_{\text{av}}(n) \qquad \Longleftrightarrow \\ (n+1)L_{\text{av}}(n+1) &= 2(n+1) + (n+2)L_{\text{av}}(n)\end{aligned}$$

Der Trick besteht nun darin, auf beiden Seiten $1/(n+1)(n+2)$ zu multiplizieren; wir erhalten dann:

$$L_{\text{av}}(n+1)/(n+2) = 2/(n+2) + L_{\text{av}}(n)/(n+1)$$

und solch eine Rekurrenz kann man einfach durch eine entsprechende Summation ersetzen:

$$\frac{L_{\text{av}}(n)}{(n+1)} = \frac{2}{n+1} + \frac{2}{n} + \ldots + \underbrace{\frac{L_{\text{av}}(0)}{1}}_{=1}$$

Um möglichst unkompliziert einen konkreten Wert aus dieser Formel zu erhalten, kann man diese Summe durch ein entsprechendes Integral approximieren und erhält dann:

$$2\sum_{i=1}^{n+1}\frac{1}{i} \approx 2 \cdot \int_1^n \frac{1}{x}dx = 2 \cdot \ln n = 2 \cdot \frac{\log_2 n}{\log_2 e} \approx 1,386 \log_2 n$$

Insgesamt erhalten wir also konkret im Durchschnitt

$$L_{\text{av}}(n) \approx 1.386n \log_2 n$$

Vergleiche bei Quicksort bei einer zu sortierenden Eingabe der Länge n. Dies ist – zumindest was die Anzahl der Vergleiche betrifft – nur etwa 38.6% über dem theoretisch möglichen Optimum.

2.3.4 In-Place-Implementierung

Wir stellen hier eine Quicksort-Implementierung vor, die keine neuen Listen anlegt, also keinen zusätzlichen Speicher verwendet und entsprechend etwas performanter ist. Der vorgestellte Algorithmus wird deutlich komplexer sein, als die in Listing 2.4 vorgestellte nicht-destruktive Implementierung. Wir teilen daher den Quicksort-Algorithmus in zwei Teile auf: Zum Einen in die Funktion *partitionIP*, die den *divide*-Schritt ausführt; zum Anderen in eine Funktion *quicksort*, die eine Liste durch wiederholten Aufruf von *partitionIP* sortiert.

Die in Listing 2.5 vorgestellte Funktion *partitionIP* bekommt neben der zu partitionierenden Liste *lst* noch den Teil der Liste – mittels Indizes l und r – übergeben, der partitioniert werden soll. Der Grund dafür, dass bei der In-Place-Implementierung zusätzlich Bereiche mit übergeben werden, liegt darin, dass alle Aufrufe immer auf der gesamten zu sortierenden Liste arbeiten; es muss entsprechend immer noch Information mit übergeben werden auf welchem Bereich der Liste im jeweiligen Aufruf gearbeitet wird.

```
1  def partitionIP( lst , l , r ):
2    pivot=lst[ l ]
3    i=l-1
4    j=r+1
5    while True:
6      while True:
7        j=j-1
8        if lst [ j ]≤pivot: break
9      while True:
10       i=i+1
11       if lst [ i ]≥pivot: break
12     if i<j:
13       lst [ i ], lst [ j ]=lst[ j ], lst [ i ]
14     else:
15       return j
```

Listing 2.5: *C.A.R. Hoare's ursprünglich vorgeschlagene Implementierung[10] einer In-Place-Partition zur Verwendung mit Quicksort.*

Auch hier wird zunächst ein Pivot-Element *pivot* gewählt, und zwar (genau wie in der nicht-destruktiven Implementierung) das Element, das sich am linken Rand des zu partitionierenden Bereichs befindet (Zeile 1: *pivot=lst*[*l*]). Der Zähler *j* läuft vom rechten Rand des Bereiches und der Zähler *i* läuft vom linken Rand des Bereiches über *lst*; die beiden inneren „**while** *True*"-Schleifen bewirken Folgendes:

- Nach Durchlaufen der ersten inneren „**while** *True*"-Schleife (Zeilen 6-8) steht *j* auf einem *lst*-Element, das kleiner-gleich dem Pivot-Element ist.
- Nach Durchlaufen der zweiten inneren „**while** *True*"-Schleife (Zeilen 9-11) steht *i* auf einem *lst*-Element, das größer-gleich dem Pivot-Element ist.

Nun müssen *lst* [*i*] und *lst* [*j*] getauscht werden – dies geschieht in Zeile 13. Falls $i \geq j$, so wurde der zu partitionierende Bereich vollständig durchlaufen und die Partitionierung ist beendet. Der Rückgabewert *j* markiert die Grenze zwischen der linken und der rechten Partition.

Die Abbildung 2.5 veranschaulicht die Funktionsweise von *partitionIP* nochmals graphisch.

Der eigentliche Quicksort-Algorithmus kann nun mit Hilfe der Funktion *partitionIP* einfach implementiert werden:

```
1  def quicksortIP( lst , l , r ):
2    if r>l:
3      i = partitionIP( lst , l , r )
4      quicksortIP( lst , l , i )
5      quicksortIP( lst , i + 1,r)
```

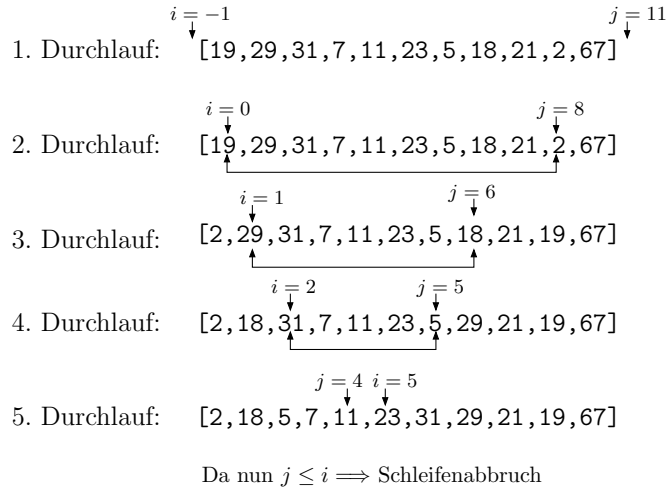


Abb. 2.5: *Darstellung der Funktionsweise von partitionIP am Beispiel der Sortierung der Partitionierung der Liste [19,29,31,7,11,23,5,18,21,2,67]. Die äußere „**while** True“-Schleife wird dabei fünf Mal durchlaufen.*

Aufgabe 2.6

Implementieren Sie eine randomisierte Variante von Quicksort

$$quicksortRandomisiert(lst, l, r)$$

die eine Häufung ungünstiger Fälle dadurch vermeidet, dass das Pivot-Element der Partitionierung von $lst\,[\,l : r{+}1\,]$ zufällig aus den Indizes zwischen (einschließlich) l und r gewählt wird.

Aufgabe 2.7

Implementieren Sie eine weitere randomisierte Variante von Quicksort

$$quicksortMedian(lst, l, r)$$

die das Pivotelement folgendermaßen wählt:

- Es werden zunächst drei zufällige Elemente aus der zu partitionierenden Liste (also aus $lst\,[\,l : r{+}1\,]$) gewählt.
- Als Pivot-Element wird der Median – also das mittlere der zufällig gewählten Elemente – ausgewählt.

Aufgabe 2.8

Vergleichen Sie nun die Algorithmen *quicksortIP*, *quicksortRandomisiert* und *quicksortMedian* folgendermaßen:

- Generieren Sie 100 zufällig erzeugte 10.000-elementige Listen, die Werte aus $\{1, \ldots 100.000\}$ enthalten und lassen sie diese 100 Listen durch die drei Quicksort-Varianten sortieren.
- „Merken“ Sie sich für jeden der Algorithmen jeweils die folgenden Daten:
  1. Die durchschnittliche Zeit, die der jeweilige Algorithmus zum Sortieren einer 10.000-elementigen Liste brauchte.
  2. Die – aus den 100 Sortierdurchläufen – schnellste Zeit, die der jeweilige Algorithmus zum Sortieren einer 10.000-elementigen Liste brauchte.
  3. Die – aus den 100 Sortierdurchläufen – langsamste Zeit, die der jeweilige Algorithmus zum Sortieren einer 10.000-elementigen Liste brauchte.

Bemerkung: Zum Erzeugen einer Liste mit zufällig gewählten Elementen können Sie das Python-Modul *random* verwenden. Der Aufruf *random.randint*(*a*,*b*) liefert eine zufällige *int*-Zahl zwischen einschließlich *a* und *b* zurück.
Zur Zeitmessung können Sie das Python-Modul *time* verwenden. Der Aufruf *time.time*() (unter Windows besser: *time. clock*()) liefert die aktuelle CPU-Zeit zurück.

2.3.5 Eliminierung der Rekursion

Ein Performance-Gewinn kann durch die Überführung der rekursiven Quicksort-Implementierung in eine iterative Implementierung erzielt werden. Warum aber ist ein iterativer Algorithmus unter Umständen schneller? Das hängt damit zusammen, dass jeder Unterprogrammaufruf mit relativ hohen Rechen„kosten“ verbunden ist; bei jedem Unterprogrammaufruf wird ein neuer *Stackframe* auf dem Rechner-internen Stack erzeugt, der alle notwendigen Informationen über das aufgerufene Unterprogramm enthält; dazu gehören unter Anderem die Rücksprungadresse zur aufrufenden Prozedur, Werte der lokalen Variablen und die Werte der übergebenen Parameter. Da das Anlegen eines Stackframes viele Zugriffe auf den Hauptspeicher erfordert und da Hauptspeicherzugriffe im Verhältnis zu anderen CPU-internen Operationen sehr teuer sind, kann eine Eliminierung der Rekursion die Performance steigern.

Anders als bei einer rekursiven Implementierung von beispielsweise der Fakultätsfunktion, kann die Rekursion bei Quicksort jedoch nicht durch eine einfache Schleife aufgelöst werden. Der Stack, der bei jedem Prozeduraufruf (insbesondere bei rekursiven Prozeduraufrufen) verwendet wird, muss hier explizit modelliert werden, wenn die Rekursion beseitigt werden soll. Auf dem Stack „merkt“ sich der rekursive Quicksort (unter Anderem) welche Arbeiten noch zu erledigen sind. Abbildung 2.6 zeigt nochmals im Detail, wie sich der Programmstack bei jedem rekursiven Aufruf erhöht und wie sich der Stack bei jedem Rücksprung aus einer Quicksort-Prozedur wieder verkleinert. Man sieht, dass

die jeweiligen Stackframes die Informationen enthalten, welche Stackframes zu einem evtl. späteren Zeitpunkt noch anzulegen sind.

Eliminieren wir die Rekursion, indem wir den Stack explizit modellieren, gibt uns das mehr Kontrolle und Optimierungspotential: Zum Einen muss nicht jeder rekursive Aufruf von Quicksort mit dem Speichern von Informationen auf dem Stack verbunden sein: ein rekursiver Quicksort-Aufruf, der am Ende der Quicksort-Prozedur erfolgt, muss nichts „merken“, denn nach diesem Aufruf ist nichts mehr zu tun (denn die aufrufende Prozedur ist danach ja zu Ende). Solche sog. endrekursiven Aufrufe (im Englischen als tail-recursive bezeichnet) kann man einfach Eliminieren und durch Iteration ersetzten. Nur die nicht end-rekursiven Aufrufe müssen sich diejenigen Aufgaben auf einem Stack merken, die nach Ihrer Ausführung noch zu erledigen sind.

Das folgende Listing 2.6 zeigt eine Quicksort-Implementierung ohne Verwendung von Rekursion. Der Stack wird durch eine Liste modelliert; eine *push*-Operation entspricht einfach der *list . append*-Methode, also dem Anfügen ans Ende der Liste; die *pop*-Operation entspricht der *list . pop*-Methode, also dem Entnehmen des letzten Elements.

```
1  def quicksortIter( lst ):
2    l=0
3    r=len( lst ) -1
4    stack = []
5    while True:
6      while r>l:
7        i = partitionIP( lst , l , r)
8        stack . append(i + 1)
9        stack . append(r)
10       r=i
11     if stack==[]: break
12     r = stack.pop()
13     l = stack.pop()
```

Listing 2.6: *Eine nicht-rekursive Implementierung von Quicksort unter Verwendung eines expliziten Stacks*

Die Funktion *quicksortIter* führt im ersten Durchlauf der inneren Schleife das Kommando

$$partitionIP(\ lst\ ,0,\ len(\ lst\ -1))$$

aus. Die *push*-Operationen in den Zeilen 8 und 9 „merken“ sich die Grenzen der rechten Teilliste; so kann die rechte Teilliste zu einem späteren Zeitpunkt bearbeitet werden. Mittels der Zuweisung $r{=}i$ in Zeile 10 sind für den nächsten Schleifendurchlauf die Listengrenzen auf die linke Teilliste gesetzt. Es werden nun solange wie möglich (nämlich bis $r \geq i$, was in Zeile 6 getestet wird) linke Teillisten partitioniert. Anschließend holt sich der Algorithmus die Grenzen der als Nächstes zu partitionierenden Teilliste vom Stack; die geschieht mittels der beiden *stack . pop*()-Operationen in den Zeilen 12 und 13.

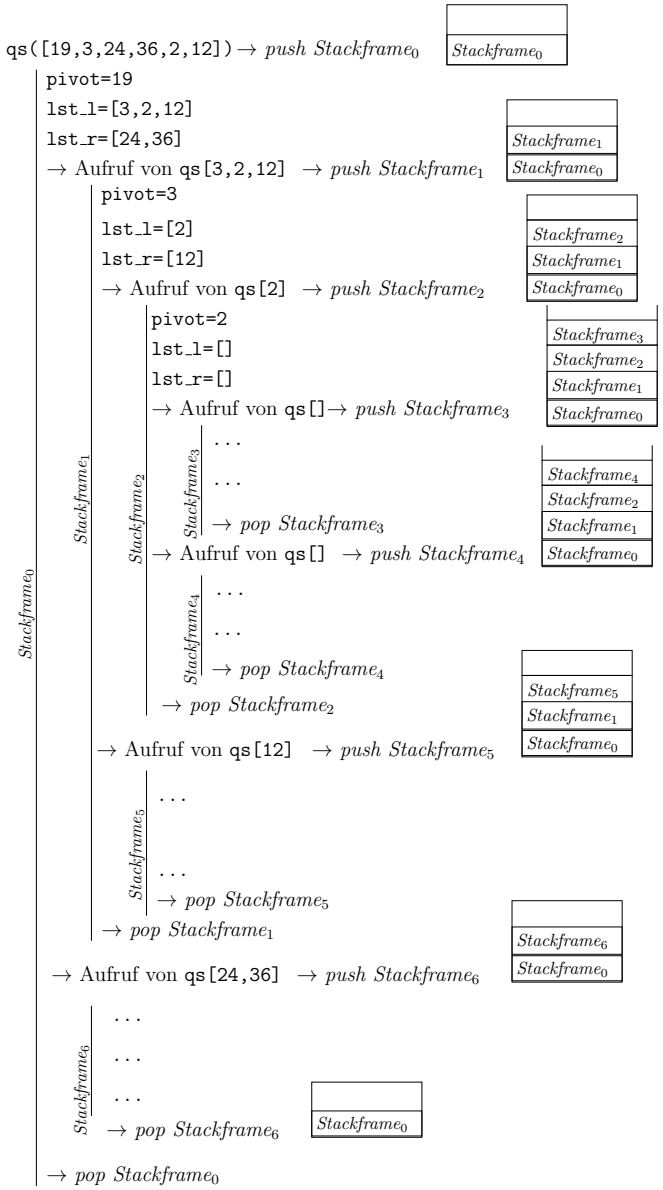


Abb. 2.6: *Darstellung der Aufrufhierarchie der rekursiven Instanzen des Quicksortalgorithmus qs, ähnlich dargestellt wie in Abbildung 2.4 nur diesmal linear in zeitlicher Reihenfolge und mit dem jeweiligen Zustand des Programmstacks bei jedem rekursiven Aufruf von Quicksort. Wie man sieht, „merkt" sich jede Instanz von Quicksort in ihrem jeweiligen Stackframe, welche Arbeit später noch zu erledigen ist. Der Aufruf qs([19,3,24,36,2,12]) beispielsweise ruft rekursiv qs([3,2,12]) auf; der Stackframe*$_0$ *enthält implizit die Information, dass zu einem späteren Zeitpunkt noch der Aufruf qs([24,36]) zu erledigen ist.*

Aufgabe 2.9

Vergleichen Sie die Laufzeiten von *quicksortIter* und *quicksortIP* miteinander. Erklären Sie Ihre Beobachtungen.

Aufgabe 2.10

(a) Wie viele Einträge könnte der Stack im Laufe der Sortierung einer Liste der Länge n mittels der Funktion *quicksortIter* im ungünstigsten Falle haben?

(b) Man kann die Größe des Stacks dadurch optimieren, indem man immer die größere der beiden entstehenden Teillisten auf dem Stack ablegt. Wie groß kann dann der Stack maximal werden?

(c) Schreiben sie die Sortierfunktion *quicksortIterMinStack* so, dass immer nur die größere der beiden Teillisten auf dem Stack abgelegt wird und vergleichen sie anschließend die Laufzeiten vom *quicksortIter* und *quicksortIterMinStack*.

2.4 Mergesort

Mergesort verwendet – wie Quicksort auch – einen klassischen Divide-And-Conquer Ansatz zum Sortieren einer Liste *lst*. Wir erinnern uns, dass im *divide*-Schritt von Quicksort der eigentliche Aufwand steckt; um die Liste zu teilen müssen viele Vergleiche ausgeführt werden. Der *combine*-Schritt dagegen ist bei Quicksort einfach: die beiden rekursiv sortierten Teillisten mussten lediglich aneinander gehängt werden.

Die Situation bei Mergesort ist genau umgekehrt. Bei Mergesort ist der *divide*-Schritt einfach: hier wird die Liste einfach in der Mitte geteilt. Der eigentliche Aufwand steckt hier im *combine*-Schritt, der die beiden rekursiv sortierten Listen zu einer großen sortierten Liste kombinieren muss. Dies geschieht im „Reißverschlussverfahren“: die beiden Listen müssen so ineinander verzahnt werden, dass daraus eine sortierte Liste entsteht. Dies wird in der englischsprachigen Literatur i. A. als *merging* bezeichnet. Das in Listing 2.7 gezeigte Python-Programm implementiert den Mergesort-Algorithmus.

```
1  def mergesort(lst):
2      if len(lst)<=1: return lst
3      l1 = lst[:len(lst)/2]
4      l2 = lst[len(lst)/2:]
5      return merge(mergesort(l1),mergesort(l2))
6
7  def merge(l1,l2):
8      if l1==[]: return l2
9      if l2==[]: return l1
10     if l1[0]<=l2[0]: return [l1[0]]+merge(l1[1:],l2)
11     else: return [l2[0]]+merge(l1,l2[1:])
```

Listing 2.7: *Implementierung von Mergesort*

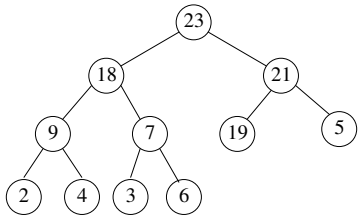
Aufgabe 2.11

Geben Sie eine iterative Variante der Funktion *merge* an.

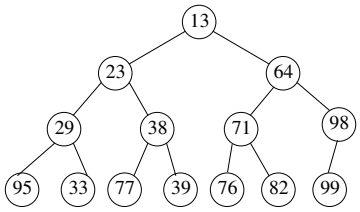
2.5 Heapsort und Priority Search Queues

Ein Heap ist ein fast vollständiger[1] Binärbaum mit der folgenden Eigenschaft: Der Schlüssel eines Knotens ist größer als die Schlüssel seiner beiden Kinder. Einen solchen Binärbaum nennt man auch oft *Max-Heap* und die eben erwähnte Eigenschaft entsprechend die *Max-Heap-Eigenschaft*. Dagegen ist ein *Min-Heap* ein vollständiger Binärbaum, dessen Knoten die Min-Heap-Eigenschaft erfüllen: Der Schlüssel eines Knotens muss also kleiner sein als die Schlüssel seiner beiden Kinder.

Abbildung 2.7 zeigt jeweils ein Beispiel eines Min-Heaps und eines Max-Heaps.



(a) Ein Max-Heap.



(b) Ein Min-Heap.

Abb. 2.7: *Beispiel eines Min-Heaps und eines Max-Heaps: Beides sind binäre Bäume, die der Min-Heap- bzw. Max-Heap-Bedingung genügen. Im Falle des Max-Heaps lautet die Heapbedingung: „Der Schlüssel jedes Knotens ist größer als die Schlüssel seiner beiden Kinder“. Im Falle des Min-Heaps lautet die Heapbedingung: „Der Schlüssel jedes Knotens ist kleiner als die Schlüssel seiner beiden Kinder“.*

2.5.1 Repräsentation von Heaps

Man könnte Heaps explizit als Baumstruktur repräsentieren – ähnlich etwa wie man einen binären Suchbaum repräsentieren würde (siehe Abschnitt 3.1). Heaps sind jedoch per Definition vollständige Binärbäume (d. h. innere Knoten besitzen *genau* zwei Nachfolger), haben also eine statische Struktur und können somit Ressourcen-schonender als „flache“ Liste repräsentiert werden; hierbei schreibt man die Einträge des Heaps von der Wurzel beginnend ebenenweise in die Liste, wobei die Einträge jeder Ebene von links nach rechts durchlaufen werden. Wir werden gleich sehen, dass es bei der Repräsentation von Heaps günstig ist, den ersten Eintrag der repräsentierenden Liste freizuhalten. Zwei Beispiele:

[1] Mit „fast vollständig“ ist die folgende Eigenschaft gemeint: Alle „Ebenen“ des Binärbaums sind vollständig gefüllt, bis auf die unterste Ebene; diese ist evtl. nur teilweise „linksbündig“ gefüllt.

- Der Max-Heap aus Abbildung 2.7(a) wird durch folgende Liste repräsentiert:
  [*None* ,23,18,21,9,7,19,5,2,4,3,6]
- Der Min-Heap aus Abbildung 2.7(b) wird durch folgende Liste repräsentiert:
  [*None* ,13,23,64,29,38,71,98,95,33,77,39,76,82,99]

Repräsentiert man also einen Heap als Liste *lst*, so ist leicht nachvollziehbar, dass das linke Kind von *lst* [*i*] der Eintrag *lst* [2∗*i*] und das rechte Kind der Eintrag *lst* [2∗*i*+1] ist.

Aufgabe 2.12

Welche der folgenden Listen sind Repräsentationen von Min-Heaps bzw. Max-Heaps?

- [*None*,13]
- [*None*,100,99,98, ... ,1]
- [*None* ,100,40,99,1,2,89,45,0,1,85]
- [*None*,40,20,31,21]

Aufgabe 2.13

(a) Implementieren Sie die Funktion *leftChild*, die als Argument eine Liste *lst* und einen Index *i* übergeben bekommt und, falls dieser existiert, den Wert des linken Kindes von *lst* [*i*] zurückgibt; falls *lst* [*i*] kein linkes Kind besitzt, soll *leftChild* den Wert *None* zurückliefern.

(b) Implementieren Sie die Funktion *rightChild*, die als Argument eine Liste *lst* und einen Index *i* übergeben bekommt und, falls dieser existiert, den Wert des rechten Kindes von *lst* [*i*] zurückgibt; falls *lst* [*i*] kein rechtes Kind besitzt, soll *rightChild* den Wert *None* zurückliefern.

(c) Implementieren Sie eine Funktion *father*, die als Argument eine Liste *lst* und einen Index *i* übergeben bekommt und den Wert des Vaters von *lst* [*i*] zurückliefert.

2.5.2 Heaps als Priority Search Queues

Es gibt viele Anwendungen, für die es wichtig ist, effizient das größte Element aus einer Menge von Elementen zu extrahieren. Beispielsweise muss ein Betriebssystem ständig (und natürlich unter Verwendung von möglichst wenig Rechenressourcen) festlegen, welcher Task bzw. welcher Prozess als Nächstes mit der Ausführung fortfahren darf. Dazu muss der Prozess bzw. Task mit der höchsten Priorität ausgewählt werden. Außerdem

kommen ständig neue Prozesse bzw. Tasks hinzu. Man könnte die entsprechende Funktionalität dadurch gewährleisten, dass die Menge von Tasks nach jedem Einfügen eines Elementes immer wieder sortiert wird, um dann das größte Element effizient extrahieren zu können; Heaps bieten jedoch eine effizientere Möglichkeit, dies zu implementieren.

Höhe eines binären Heaps. Für spätere Laufzeitbetrachtungen ist es wichtig zu wissen, welche Höhe ein n-elementiger binärer Heap hat. Auf der 0-ten Ebene hat eine Heap $2^0 = 1$ Elemente, auf der ersten Ebene 2^1 Elemente, usw. Ist also ein Heap der Höhe h vollständig gefüllt, so kann er

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Elemente fassen. Oder andersherum betrachtet: Ein vollständig gefüllter Heap mit n Elementen besitzt eine Höhe von $\log_2 n$. Ist der Heap nicht ganz vollständig gefüllt, so muss man bei der Berechnung der Höhe entsprechend aufrunden. Es gilt also für die Höhe h eines Heaps mit n Elementen die folgende Beziehung:

$$h = \lceil \log_2 n \rceil$$

Zu den wichtigsten Operationen auf Heaps gehören das Einfügen eines neuen Elements in einen Heap und die Extraktion (d. h. das Suchen und anschließende Löschen) des maximalen Elements bei Max-Heaps bzw. die Extraktion des minimalen Elements bei Min-Heaps. Im Folgenden stellen wir die Implementierung dieser zwei Operationen für Min-Heaps vor.

Einfügen. Soll ein neues Element in einen als Liste repräsentierten binären Heap eingefügt werden, so wird es zunächst an das Ende der Liste angefügt. Dadurch wird im Allgemeinen die Heap-Eigenschaft verletzt. Um diese wiederherzustellen, wird das eingefügte Element sukzessive soweit wie nötig nach „oben“ transportiert. Abbildung 2.8 zeigt an einem Beispiel den Ablauf des Einfügens und das anschließenden Hochtransportieren eines Elementes in einem Heap.

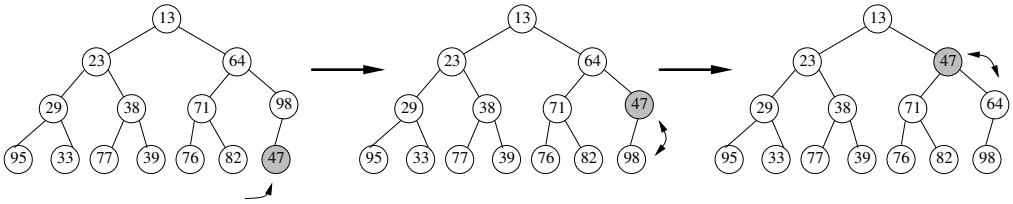


Abb. 2.8: *Das Element 47 wird in einen Heap eingefügt. Anfänglich wird das Element „hinten“ an den Heap angefügt (linkes Bild). Die Heapbedingung ist verletzt; daher wird das Element sukzessive durch Tauschen nach oben transportiert und zwar solange bis die Heapbedingung wieder erfüllt ist.*

Listing 2.8 zeigt eine Implementierung der Einfügeoperation.

```
1 def insert(heap, x):
2   heap.append(x)
3   i = len(heap) -1
4   while heap[i/2]>heap[i]:
5     heap[i/2],heap[i] = heap[i],heap[i/2]
6     i = i/2
```

Listing 2.8: *Einfügen eines Elementes in einen als Liste repräsentierten Min-Heap*

Wir gehen davon aus, dass die als Parameter übergebene Liste *heap* einen Heap repräsentiert. Das einzufügende Element x wird zunächst hinten an den Heap angehängt (*heap.append*(x) in Zeile 2); anschließend wird das eingefügte Element solange durch Tausch mit dem jeweiligen Vaterknoten die Baumstruktur hochtransportiert, bis die Heapbedingung erfüllt ist. Die **while**-Schleife wird hierbei solange ausgeführt, wie der Wert des eingefügten Knotens kleiner ist, als der Wert seines Vaterknotens, d. h. solange die Bedingung *lst* [$i/2$]>*lst*[i] gilt.

Aufgabe 2.14

Die in Listing 2.8 gezeigte Implementierung der Einfüge-Operation ist destruktiv implementiert, d. h. der übergebene Parameter *heap* wird verändert. Geben Sie eine alternative nicht-destruktive Implementierung der Einfügeoperation an, die einen „neuen“ Heap zurückliefert, der das Element x zusätzlich enthält.

Aufgabe 2.15

Wie arbeitet die Funktion *insert*, wenn das einzufügende Element x kleiner ist als die Wurzel des Heaps *lst* [1]? Spielen Sie den Algorithmus für diesen Fall durch und erklären Sie, warum er korrekt funktioniert.

Die Höhe des Heaps begrenzt hierbei die maximal notwendige Anzahl der Vergleichs- und Tauschoperationen. Die Worst-Case-Laufzeit der Einfügeoperation eines Elements in einen Heap mit n Elementen liegt also in $O(\log n)$.

Minimumsextraktion. Entfernt man das minimale Element, also die Wurzel, aus einem Min-Heap, dann geht man am effizientesten wie folgt vor: Das letzte Element aus einer den Heap repräsentierenden Liste *heap*, also *heap*[-1], wird an die Stelle der Wurzel gesetzt. Dies verletzt im Allgemeinen die Heap-Bedingung. Die Heap-Bedingung kann wiederhergestellt werden, indem man dieses Element solange durch Tauschen mit dem kleineren der beiden Kinder im Baum nach unten transportiert, bis die Heap-Bedingung wiederhergestellt ist. Abbildung 2.9 veranschaulicht an einem Beispiel den Ablauf einer solchen Minimumsextraktion.

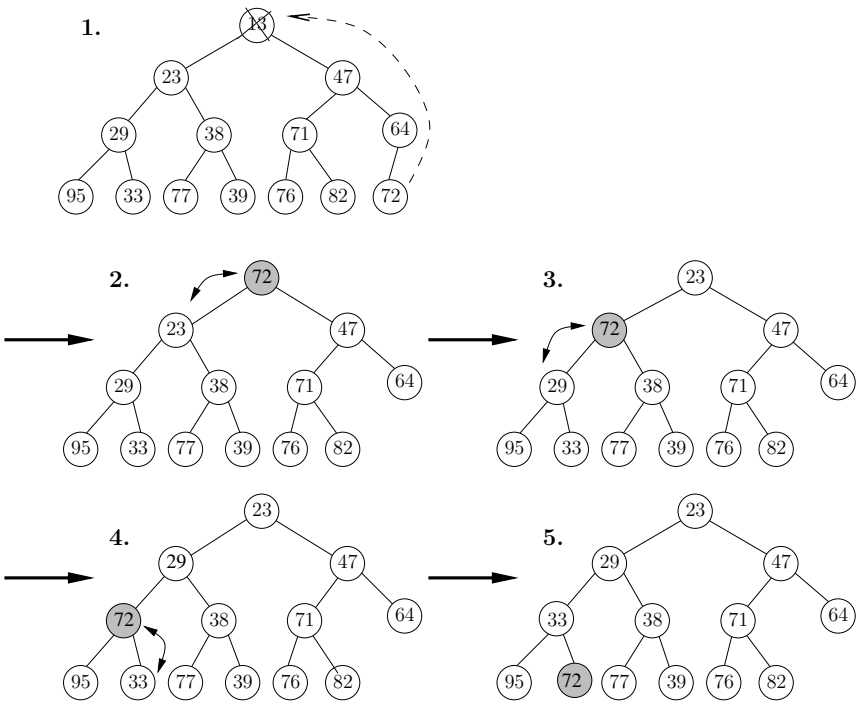


Abb. 2.9: *Ablauf einer Minimumsextraktion.* **1***: Das minimale Element des Heaps, das sich aufgrund der Min-Heap-Bedingung immer an der Wurzel des Heaps befindet, wird gelöscht und an dessen Stelle das „letzte“ Element des Heaps gesetzt, in unserem Falle ist dies der Knoten mit Schlüsselwert „72“.* **2***: In Folge dessen, ist jedoch im unter 2. dargestellten Heap die Heap-Bedingung verletzt.* **3, 4, 5***: Diese kann wiederhergestellt werden, indem man den an der Wurzel befindlichen Knoten durch Tausch-Operationen nach unten transportiert; und zwar wird immer mit dem kleineren der beiden Kinder getauscht. Nach einigen solcher Tausch-Operationen befindet sich der Knoten mit Schlüsselwert „72“ an der „richtigen“ Position, d. h. an einer Position, an der er die Heap-Bedingung nicht mehr verletzt – in diesem Falle wird er zum Wurzelknoten.*

Die in Listing 2.9 gezeigte Funktion *minExtract* implementiert die Minimumsextraktion. In der Variablen n ist während des ganzen Programmablaufs immer der Index des „letzten“ Elements des Heaps gespeichert. In den Zeilen 3 und 4 wird das „letzte“ Element des Heaps an die Wurzel gesetzt. Die Durchläufe der **while**-Schleife transportieren dann das Wurzel-Element solange nach „unten“, bis die Heap-Bedingung wieder erfüllt ist. Am Anfang der **while**-Schleife zeigt die Variable i immer auf das Element des Heaps, das möglicherweise die Heap-Bedingung noch verletzt. In Zeile 9 wird das kleinere seiner beiden Kinder ausgewählt; falls dieses Kind größer ist als das aktuelle Element, d. h. falls $lst\,[i] \leq lst\,[k]$, so ist die Heap-Bedingung erfüllt und die Schleife kann mittels **break** abgebrochen werden. Falls jedoch dieses Kind kleiner ist als der aktuelle Knoten, ist die Heapbedingung verletzt, und Vater und Kind müssen getauscht werden (Zeile 11). Durch die Zuweisung $i{=}j$ fahren wir im nächsten **while**-Schleifendurchlauf damit fort, den getauschten Knoten an die richtige Position zu bringen.

```
1  def minExtract(lst):
2    returnVal=lst[1]
3    lst[1]=lst[-1] # letztes Element an die Wurzel
4    del(lst[-1])
5    n=len(lst)-1   # n zeigt auf das letzte Element
6    i=1
7    while i≤n/2:
8      j=2*i
9      if j<n and lst[j]>lst[j+1]: j+=1 # wähle kleineres der beiden Kinder
10     if lst[i]≤lst[j]: break
11     lst[i], lst[j]=lst[j], lst[i]
12     i=j
13   return returnVal
```

Listing 2.9: *Implementierung der Minimumsextraktion, bei der das Wurzel-Element des Heaps entfernt wird.*

Was die Laufzeit der Minimumsextraktion betrifft, gilt Ähnliches wie für die Einfüge-Operation: Die Höhe des Heaps begrenzt die maximal notwendige Anzahl der Vergleichs- und Tauschoperationen. Damit ist die Worst-Case-Laufzeit des Algorithmus *minExtract* in $O(\log n)$.

Aufgabe 2.16

Implementieren Sie die zwei Heap-Operationen „Einfügen“ und „Maximumsextraktion“ für Max-Heaps.

2.5.3 Konstruktion eines Heaps

Man kann Heaps für den Entwurf eines effizienten Sortieralgorithmus verwenden, der bei der Sortierung einer Liste *lst* folgendermaßen vorgeht: Zunächst wird *lst* in eine Heapdatenstruktur umgewandelt. Anschließend wird mittels der Minimumsextraktion ein Element nach dem anderen aus dem Heap entfernt und sortiert in die Liste hinten eingefügt. Verwendet man Min-Heaps, so kann man eine Liste absteigend sortieren; verwendet man Max-Heaps, so kann man eine Liste aufsteigend sortieren.

Wenden wir uns zunächst dem Aufbau einer Heapdatenstruktur aus einer gegebenen beliebigen Liste *lst* zu. Man kann die hintere Hälfte der Liste (also *lst*[*len*(*lst*)/2:]) als eine Sammlung von *len*(*lst*)/2 Heaps betrachten; nun müssen wir „nur“ noch über den vorderen Teil der Liste laufen und alle verletzten Heap-Bedingungen wiederherstellen.

Wir programmieren zunächst eine Funktion, die für einen gegebenen Knoten die Heapbedingung herstellt; anschließend ist der eigentliche Heapsort-Algorithmus in einer einfachen Schleife leicht zu programmieren. Für die Herstellung der Heap-Bedingung gehen wir so vor, wie schon in der **while**-Schleife aus Listing 2.9 implementiert: Die Knoten,

die die Heap-Bedingung verletzen, werden solange nach „unten“ durchgereicht, bis die Heap-Bedingung wiederhergestellt ist. Wir könnten eigentlich die **while**-Schleife aus Listing 2.9 übernehmen; der besseren Übersicht halber, verwenden wir aber die in Listing 2.10 vorgestellte rekursiv implementierte Funktion *minHeapify*.

```
1  def minHeapify(heap,i):
2      l = 2*i
3      r = l+1
4      n = len(heap)-1
5      nodes = [(heap[v],v) for v in [i,l,r] if v<=n]
6      nodes.sort()
7      smallestIndex = nodes[0][1]
8      if smallestIndex != i:
9          heap[i],heap[smallestIndex] = heap[smallestIndex],heap[i]
10         minHeapify(heap,smallestIndex)
```

Listing 2.10: *Die Funktion minHeapify, die den Knoten an Index i soweit sinken lässt, bis die Heap-Bedingung des Heaps „heap“ wiederhergestellt ist.*

Die Funktion *minHeapify* stellt die Heap-Bedingung, falls diese verletzt ist, für den Knoten an Index i des Heaps *heap* wieder her, und zwar dadurch, dass der Knoten im Heap solange nach „unten“ gereicht wird, bis die Heap-Bedingung wieder erfüllt ist. Die in Zeile 2 und 3 definierten Variablen l und r sind die Indizes der Kinder des Knotens an Index i. In Zeile 5 wird mittels einer Listenkomprehension eine i. A. dreielementige Liste *nodes* aus den Werten des Knotens an Index i und seiner beiden Kinder erstellt; um den Knoten mit kleinstem Wert zu bestimmen, wird *nodes* sortiert; danach befindet sich der *Wert* des kleinsten Knotens in *nodes*[0][0] und der Index des kleinsten Knotens in *nodes*[0][1]. Falls der Wert des Knotens i der kleinste der drei Werte ist, ist die Heap-Bedingung erfüllt und die Funktion *minHeapify* kann verlassen werden; falls andererseits einer der Kinder einen kleineren Wert hat (d. h. falls *smallestIndex*$\neq i$), so ist die Heap-Bedingung verletzt und der Knoten an Index i wird durch Tauschen mit dem kleinsten Kind nach „unten“ gereicht; anschließend wird rekursiv weiterverfahren.

Aufgabe 2.17

Verwenden Sie die in Listing 2.10 vorgestellte Funktion *minHeapify*, um die in Listing 2.9 programmierte **while**-Schleife zu ersetzen und so eine kompaktere Implementierung der Funktion *extraktHeap* zu erhalten.

Aufgabe 2.18

Beantworten Sie folgende Fragen zu der in Listing 2.10 gezeigten Funktion *minHeapify*:

- In welchen Situationen gilt *len*(*nodes*)==3, in welchen Situationen gilt *len*(*nodes*)==2 und in welchen Situationen gilt *len*(*nodes*)==1?
- Können Sie sich eine Situation vorstellen, in der *len*(*nodes*)==0 gilt? Erklären Sie genau!
- Die Funktion *minHeapify* ist rekursiv definiert. Wo befindet sich der Rekursionsabbruch? Und: In welcher Hinsicht ist das Argument des rekursiven Aufrufs „kleiner“ als das entsprechende Argument in der aufrufenden Funktion.
  Denn, wie in Abschnitt 1.2.1 auf Seite 6 besprochen, müssen die rekursiven Aufrufe „kleinere“ (was auch immer „kleiner“ im Einzelnen bedeutet) Argumente besitzen als die aufrufende Funktion, um zu vermeiden, dass die Rekursion in einer Endlosschleife endet.

Aufgabe 2.19

Programmieren Sie eine Funktion *maxHeapify*, die als Argumente einen als Liste repräsentierten Heap *heap* und einen Index i bekommt und die Max-Heap-Bedingung des Knotens an Index i (bei Bedarf) wiederherstellt.

Aufgabe 2.20

Eliminieren Sie die Listenkomprehension in Zeile 5 und deren Sortierung in Zeile 6 und verwenden Sie stattdessen **if**-Anweisungen mit entsprechenden Vergleichen um das kleinste der drei untersuchten Elemente zu bestimmen.

Aufgabe 2.21

Programmieren Sie nun eine iterative Variante der Funktion *minHeapify*; Sie können sich dabei an der **while**-Schleife aus Listing 2.9 orientieren.

Mittels *minHeapify* können wir nun einfach eine Funktion schreiben, die einen Heap aus einer gegebenen Liste erzeugt. Listing 2.10 zeigt die entsprechende Python-Implementierung.

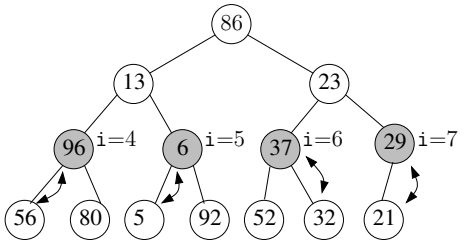
```
1 def buildHeap(lst): # Es muss lst[0]==None gelten
2     for i in range(len(lst)/2,0,-1):
3         minHeapify(lst,i)
```

Listing 2.11: *Konstruktion eines Heaps aus einer gegebenen Liste lst.*

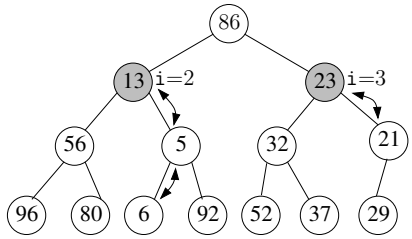
Die Funktion *buildHeap* läuft nun über alle Elemente, die keine Blätter sind (also Elemente mit Index zwischen *len*(*lst*)/2 und einschließlich 1), beginnend mit den „unteren“ Knoten. Der Aufruf *range*(*len*(*lst*)/2,0, -1) erzeugt hierbei die Liste der zu untersuchenden Knoten in der richtigen Reihenfolge. Der Algorithmus arbeitet sich entsprechend sukzessive nach „oben“ vor, bis als letztes die Heap-Bedingung der Wurzel sichergestellt wird. Folgendermaßen könnte die Funktion *buildHeap* verwendet werden:

```
>>> l=[None, 86, 13, 23, 96, 6, 37, 29, 56, 80, 5, 92, 52, 32, 21]
>>> buildHeap(l)
>>> print l
[None, 5, 6, 21, 56, 13, 32, 23, 96, 80, 86, 92, 52, 37, 29]
```

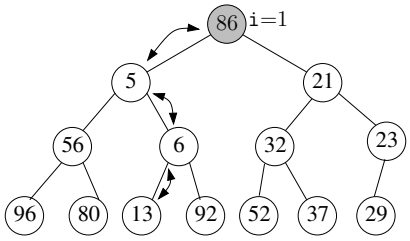
Abbildung 2.10 zeigt die Funktionsweise von *buildHeap* bei der Anwendung auf eben diese Beispiel-Liste.



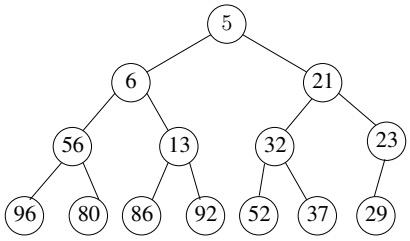
(a) Die Schleifendurchläufe für i = 7, . . . , 4.



(b) Die Schleifendurchläufe für i = 3 und i = 2.



(c) Der Schleifendurchlauf für i = 1.



(d) Der durch Anwendung von *buildHeap* entstandene Heap: Alle Heapbedingungen sind erfüllt.

Abb. 2.10: *Funktionsweise von buildHeap bei Anwendung auf die Liste [None, 86, 13, 23, 96, 6, 37, 29, 56, 80, 5, 92, 52, 32, 21]. Die Blatt-Knoten für sich genommen bilden schon Heaps; für diese trivialen Heaps können keine Heap-Bedingungen verletzt sein. Sei h die Höhe des Heaps; da für die Blätter also nichts zu tun ist, beginnt buildHeap damit, über die Knoten der Ebene $h-1$ zu laufen und verletzte Heap-Bedingungen wieder herzustellen; dies entspricht, wie in Abbildung 2.10(a) zu sehen, den* ***for****-Schleifendurchläufen für $i = 7$ (also len(lst)/2) bis $i = 4$ aus Listing 2.11; Abbildung 2.10(b) zeigt den dadurch entstandenen Baum und das Herstellen der Heap-Bedingungen der Knoten in Ebene* 1*. Abbildung 2.10(c) zeigt den daraus entstandenen Baum und das Herstellen der Heap-Bedingung des Wurzel-Knotens. Abbildung 2.10(d) zeigt den so entstandenen (Min-)Heap.*

Da höchstens $O(n)$ Aufrufe der Funktion *minHeapify* stattfinden, und jeder dieser Aufrufe höchstens $O(\log n)$ Schritte benötigt, gilt: *buildHeap* benötigt $O(n \log n)$ Schritte. Diese Aussage ist zwar korrekt, da die O-Notation immer eine obere Schranke für das Wachstum angibt[2]. Tatsächlich ist es aber so, dass die meisten Aufrufe an *minHeapify* „kleine“ Argumente haben; man kann zeigen, dass *buildHeap* für das Aufbauen eines Heaps aus einer n-elementigen Liste tatsächlich nur $O(n)$ Schritte benötigt.

2.5.4 Heapsort

Das Listing 2.12 zeigt die Implementierung eines effizienten Sortieralgorithmus unter Verwendung von Heaps:

```
1 def heapSort( lst ):
2   buildHeap( lst )
3   for i in range(len( lst ) -1,1, -1):
4     lst [1], lst [i] = lst [i], lst [1]
5     minHeapify3(lst,1, i -1)
```

Listing 2.12: *Implementierung von Heapsort*

Hierbei funktioniert *minHeapify3* eigentlich genauso wie *minHeapify*, außer dass der dritte Parameter zusätzlich angibt, bis zu welchem Index die übergebene Liste als Heap betrachtet werden soll. Das Listing implementiert ein in-place-Sortierverfahren unter Verwendung von Heaps und geht dabei folgendermaßen vor: Zunächst wird aus der übergebenen unsortierten Liste ein Heap generiert. Dann wird, in einer Schleife, immer das kleinste Element vom Heap genommen und an den hinteren Teil von *lst*, in dem die sortierte Liste aufgebaut wird, angehängt.

Oft kann man über die Formulierung von *Schleifeninvarianten* geschickt argumentieren, warum ein bestimmter Algorithmus korrekt ist. Eine Schleifeninvariante ist einfach eine bestimmte Behauptung, die an einer bestimmten Stelle in *jedem* Durchlauf einer Schleife gültig ist. Über automatische Theorembeweiser kann man so sogar die Korrektheit einiger Algorithmen formal beweisen; wir nutzen hier jedoch Schleifeninvarianten nur, um die Korrektheit von Algorithmen informell zu erklären. Im Falle des in Listing 2.12 gezeigten Heapsort-Algorithmus gilt folgende Schleifeninvariante: Zu Beginn *jedes* **for**-Schleifendurchlaufs bildet die Teilliste $lst\,[1:i+1]$ einen Min-Heap, der die i größten Elemente aus *lst* enthält; die Teilliste $lst\,[i+1:]$ enthält die $n-i$ kleinsten Elemente in sortierter Reihenfolge. Da dies insbesondere auch für den letzten Schleifendurchlauf gilt, sieht man leicht, dass die Funktion *heapSort* eine sortierte Liste zurücklässt.

[2] Oder in anderen Worten: die Aussage $f(n) = O(g(n))$ bedeutet, dass die Funktion $f(n)$ *höchstens* so schnell wächst wie $g(n)$, also evtl. auch langsamer wachsen kann; $g(n)$ kann man aus diesem Grund auch als „oberer Schranke“ für das Wachstum von $f(n)$ bezeichnen.

Aufgabe 2.22

Implementieren Sie – indem Sie sich an der Implementierung von *minHeapify* orientieren – die für Heapsort notwendige Funktion *minHeapify3*(i,n), die die übergebene Liste nur bis zu Index n als Heap betrachtet und versucht die Heapbedingung an Knoten i wiederherzustellen.

Aufgabe 2.23

Lassen Sie die Implementierungen von Quicksort und Heapsort um die Wette laufen – wer gewinnt? Versuchen Sie Ihre Beobachtungen zu erklären.

Heaps in Python

Die Standard-Modul *heapq* liefert bereits eine fertige Implementierung von Heaps. Folgende Funktionen sind u. A. implementiert:

- *heapq.heapify*(*lst*): Transformiert die Liste *lst* in-place in einen Min-Heap; entspricht der in Listing 2.11 implementierten Funktion *buildHeap*.
- *heapq.heappop*(*lst*): Enfernt das kleinste Element aus dem Heap *lst*; dies entspricht somit der in Listing 2.9 implementierten Funktion *minExtract*.
- *heapq.heappush*(*lst*, x): Fügt ein neues Element x in den Heap *lst* ein; dies entspricht somit der in Listing 2.8 implementierten Funktion *insert*.

Aufgaben

Aufgabe 2.24

Schreiben Sie eine möglichst performante Python-Funktion

$$smallestn(\, lst\, , n)$$

die die kleinesten n Elemente der Liste n zurückliefert.

Aufgabe 2.25

Schreiben Sie eine Funktion *allInvTupel*, die für eine gegebene Liste von Zahlen $lst = [a_1, a_2, \ldots, a_n]$ alle Paare (x, y) zurückliefert, mit $x \in lst$ und $y \in lst$ und x ist das Einerkomplement von y.

1. Anmerkung: Das Einerkomplement einer Zahl x entsteht dadurch, dass man jedes Bit in der Binärdarstellung invertiert, d. h. eine 0 durch eine 1 und eine 1 durch eine 0 ersetzt.
2. Anmerkung: Verwenden Sie zur Implementierung dieser Funktion die Python-Funktion $sort()$.