Algorithmische Mathematik I: Präsentation

Gennadiy Averkov
Email: averkov@ovgu.de

Fakultät für Mathematik
Otto-von-Guericke-Universität Magdeburg
25. April 2016

Inhalt I

Einleitung
- Rechenaufgaben

Sortierproblem
- Sortieren durch Einfügen
- Sortieren durch Mischen
- Heap-Sort
- Prioritätsschlangen auf der Basis von Heaps
- Quicksort
- Untere Schranken für die Laufzeit von Sortieralgorithmen
- Countingsort
- RadixSort

Section 1

Einleitung

Rechenaufgabe

eine Aufgabe anhand einer gegebenen (endlichen) Eingabe eine gewünschte (endliche) Rückgabe berechnen.

Beispiel 1.1 (Index des maximalen Elements)

Anhand einer nichtleere Liste $[a_1, \ldots, a_n]$ von n ganzen Zahlen ein Index einer maximalen Zahl in der Liste bestimmen.

Rechenaufgabe algorithmisch lösen

einen Algorithmus beschreiben, welcher für jede Eingabe die gewünschte Rückgabe erzeugt

Rechenmodell

ein theoretisches Modell eines Rechners

Random Access Machine = RAM (unser Modell)

- Speicher aus unendlich vielen Speicherzellen, die durch ganzzahlige Indizes indexiert sind
- Jede Zelle kann einen beliebigen ganzzahligen Wert speichern
- Zuweisungen möglich
- Elementare Arithmetik (Addition einer Konstante, Multiplikation mit einer Konstante, Ganzzahlige Division durch eine Konstante)
- Standard-Kontrollstrukturen wie *if-then-else*, *while*, *for* und *goto*
- Vergleichsoperationen $=, <, \leq, >, \geq$

Programmiermittel

- Pseudocode mit einer selbsterklärenden Syntax
- Prozeduren mit und ohne Rückgabe
- Arrays
- Records

Beispiel 1.2 (Maximum von zwei Zahlen in Pseudocode)

1.1 $m = \textsc{Maximum}(x, y)$

require: $x, y \in \mathbb{Z}$
ensure: m ist Maximum von x und y

```
1: if x > y :
2:     m := x
3: else:
4:     m := y
5: end
```

Vereinbarung zur Parameterübergabe

Standardmäßig werden Arrays an die Prozeduren durch Referenz und alle anderen Datentypen durch Kopie übergeben.

Analyse von Algorithmen

- Endlichkeit
- Korrektheit
- Effizienz (Laufzeit und Speicheraufwand in Abhängigkeit von der Größe der Eingabe)

Ziel

- für eine gegebene Rechenaufgabe eine möglichst effiziente algorithmische Lösung finden
- Die Korrektheit und Effizienz mathematisch exakt begründen

Mathematische Hilfsmittel

- Mengen (Inklusion, Durchschnitt, Kreuzprodukt, leere Menge), Abbildungen, Logische Verknüpfungen, Quantoren
- Summen, Produkte
- Integration, Differentiation
- Vollständige Induktion
- Die Mengen der natürlichen Zahlen $\mathbb{N}$, der nichtnegativen ganzen Zahlen $\mathbb{N}_0$, der ganzen Zahlen $\mathbb{Z}$, der rationalen Zahlen $\mathbb{Q}$ und der rellen Zahlen $\mathbb{R}$.
- Abrunden $\lceil x \rceil$ und Aufrunden $\lfloor x \rfloor$
- Kombinatorik (endliche Mengen zählen)

Asymptotische Notation

Seien $f, g : \mathbb{N} \to \mathbb{R}$.

- Wir schreiben $f(n) = O(g(n))$, wenn $C > 0$ und $n_0 \in \mathbb{N}$ existieren, für die

$$|f(n)| \leq C|g(n)|$$

  für alle $n \geq n_0$ gilt.
- Wir schreiben $f(n) = \Omega(g(n))$, wenn $C > 0$ und $n_0 \in \mathbb{N}$ existieren, für die

$$|f(n)| \geq C|g(n)|$$

  für alle $n \geq n_0$ gilt.
- Wir schreiben $f(n) = \Theta(g(n))$, wenn $C_1, C_2 > 0$ und $n_0 \in \mathbb{N}$ existieren, für die

$$C_1|g(n)| \leq |f(n)| \leq C_2|g(n)|$$

  für alle $n \geq n_0$ gilt.

Section 2

Sortierproblem

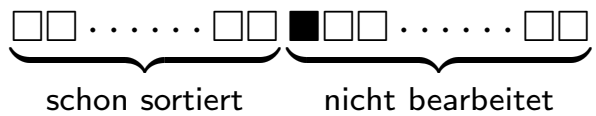
Sortierproblem

Eine Liste von $n \in \mathbb{N}_0$ Zahlen $a_1, \ldots, a_n$ (ganz, rational usw.) aufsteigend anordnen.

- Leere Liste: $[\,] \mapsto [\,]$
- Einelementige Liste: $[2] \mapsto [2]$
- $[2, 1] \mapsto [1, 2]$
- $[2, 1, 2] \mapsto [1, 2, 2]$
- $[2, -1, 3, 3, 2, 3] \mapsto [-1, 2, 2, 3, 3, 3]$.

Sortieren durch Einfügen

Ein sortiertes Teilarray iterativ wachsen lassen:



Das ■ wird in das bereits sortierte Teil solange aufgenommen bis das nicht bearbeitete Teil leer wird.

Gegeben

Array A mit Komponenten $A[i]$, mit $i \in 1, \ldots,$ Länge$[A]$.

2.1 Sortieren-durch-Einfügen(A)

```
1: for i = 2, ..., Länge[A] :
2:     ▷ A[1 .. i − 1] schon sortiert
3:     ▷ TODO: A[1 .. i] sortieren
4: end
```

2.2 SORTIEREN-DURCH-EINFÜGEN(A)

```
1:  for i := 2, ..., LÄNGE[A] :
2:      ▷ A[1 .. i − 1] ist sortiert
3:      x := A[i]  ▷ wird eingefügt
4:      j := i  ▷ A[j] 'frei'
5:      ▷ Steht links vor freien Position was größeres als x?
6:      while j > 1 und A[j − 1] > x :
7:          ▷ Wir ändern die freie Position (wie bei 15-Puzzle)
8:          A[j] := A[j − 1]
9:          j := j − 1
10:     end
11:     A[j] := x  ▷ Position für x gefunden
12: end
```

Bemerkung

Das Und in der *While*-Schleife wird als *träge Operation* implementiert: ist $j > 1$ nicht erfüllt, kennt man das Resultat, sodass $A[j-1] > x$ gar nicht ausgewertet wird (gut so).

Zur Endlichkeit:

Sortieren durch Einfügen terminiert. Denn:

- In der inneren Schleife wird j in jeder Iteration verkleinert. Eine unendliche Anzahl der Iteration ist wegen der Bedingung $j > 1$ nicht möglich.
- Die äußere Schleife macht offensichtlich nicht mehr las $\textsc{Länge}[A]$ Iterationen.

Theorem 2.1 (Korrektheit des Sortierens durch Einfügen)

Sortieren durch Einfügen löst das Sortierproblem.

Beweis.

- Hat A die Länge höchstens 1, so wird A nicht verändert. Korrekt!
- Hat A eine größere Länge, so gilt in der ersten Iteration (mit $i = 2$) offensichtlich die Bedingung, dass $A[1\,..\,i-1]$ sortiert ist.
- Das heißt, wenn am Ende der äußeren Schleife das Teilarray $A[1\,..\,i]$ sortiert wird, so funktioniert das gesamte Verfahren korrekt. Denn so behält man während der gesamten Ausführung die Bedingung, dass $A[1\,..\,i-1]$ zur Beginn der äußeren Schleife sortiert ist, sodass am Ende der Ausführung das gesamte Array sortiert wird.
- Es bleibt zu zeigen, dass jede Iteration der äußeren Schleife das Array $A[1\,..\,i]$ sortiert.
- Dafür reicht es zu beobachten, dass in der inneren Schleife die folgenden Bedingungen beibehalten werden:
  - Die Elemente, die ursprünglich in A waren sind in $A[1\,..\,j-1]$, x und $A[j\,..\,\textsc{Länge}[A]]$ verteilt, sodass $A[j]$ 'frei' ist.
  - $A[1\,..\,j]$ zusammen mit $A[j+1\,..\,i]$ ergeben ein Sortiertes Array, wobei die Elemente von $A[j+1\,..\,i]$ größer als x sind.

□

Invariante:
Aussage, die an einer Stelle des Algorithmus immer erfüllt ist. *Schleifeninvariante* ist eine Invariante, die zur Beginn jeder Iteration einer Schleife gilt.

Laufzeit:
die Anzahl der Elementaroperationen (vgl. RAM), die ein Algorithmus bei der Ausführung auf einer gegebenen Eingabe durchführt.

Worst-Case-Laufzeit:
die maximale Laufzeit auf einer gegebenen Menge der Eingaben.

Speicheraufwand:
Die Anzahl der Speicherzellen, die ein Algorithmus neben der Speicherung der Eingabe zusätzlich benötigt.

Theorem 2.2 (Effizienz des Sortierens durch Einfügen)

Die Worst-Case-Laufzeit des Sortierens durch Einfügen auf Arrays der Länge n ist $\Theta(n^2)$. Der Speicheraufwand ist $\Theta(1)$.

Beweis.

- Der Speicheraufwand ist offensichtlich $\Theta(1)$, da man lediglich drei zusätzliche Variablen i, j und x benutzt.
- Wir zeigen, dass die Laufzeit auf jedem Array der Länge $n \geq 1$ höchstens quadratisch ist.
- Zur Ausführung einer Iteration der äußeren Schleife braucht man höchstens cn Elementaroperationen für eine Konstante $c > 0$ (da die innere Schleife nicht mehr als n Iterationen macht).
- Die äußere Schleife macht nicht mehr als n Iterationen, sodass die Gesamtlaufzeit aller Iteration nicht höher als cn^2 ist.
- Es folgt, dass die Laufzeit des Sortierens durch Einfügen auf eine Array der Länge n gleich $O(n^2)$ ist.
- Es bleibt zu zeigen, dass die Laufzeit auf manchen Arrays der Länge n gleich $\Omega(n^2)$ ist.
- Angenommen, $n \geq 1$ und A ist ein absteigend sortiertes Array mit n unterschiedlichen Elementen, etwa $A = [n, \ldots, 1]$.
- Dann macht für jedes $i \in \{2, \ldots, n\}$ die innere Schleife genau $i - 1$ Iterationen.
- Da man in jeder Iteration mindestens eine Elementaroperation durchführt, ist die Gesamtlaufzeit mindestens

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

Sortieren durch Mischen

Zwei rekursiv sortierte möglichst gleichlange Teilarrays zu einem sortierten Array mischen.

Gegeben

Array A mit Komponenten $A[i]$, mit $i = 1, \ldots,$ Länge$[A]$.

2.3 Sortieren-durch-Mischen(A)

1: **if** Länge$[A] > 1$ **:**
2: $m := \lfloor$Länge$[A]/2 \rfloor$ ▷ *Mitte des Arrays*
3: $A[1 \,..\, m]$ in einem neuen Array L aufheben.
4: $A[m+1 \,..\,$ Länge$[A]]$ in einem neuen Array R aufheben.
5: Sortieren-durch-Mischen(L)
6: Sortieren-durch-Mischen(R)
7: Mischen(A, L, R) ▷ *sortierte L und R wirden in das A gemischt*
8: **end**

2.4 Mischen(A,L,R)

```
1:  a := 1, l := 1, r := 1  ▷ Laufindizes für jeden der drei Arrays
2:  ▷ Los!
3:  while l < Länge[L] und r < Länge[R] :
4:     if L[l] ≤ R[r] :
5:        A[a] := L[l]
6:        a := a + 1, l := l + 1
7:     else:
8:        A[a] := R[r]
9:        a := a + 1, r := r + 1
10:    end
11: end
12: ▷ Nun einen der beiden Reste aus L bzw. R in A kopieren
13: while l < Länge[L] :
14:    A[a] := L[l]
15:    a := a + 1, l := l + 1
16: end
17: while r < Länge[R] :
18:    A[a] := R[r]
19:    a := a + 1, r := r + 1
20: end
```

Beispiel zum Debugging

$[5, 2, 4, 7, 1, 3, 2, 6]$.

Endlichkeit:

- Sortieren durch Mischen terminiert auf Arrays der Länge höchstens 1.
- Ist die Länge des Arrays $n \geq 2$, so macht das Verfahren zwei rekursive Aufrufe auf Arrays der Länge höchstens $n - 1$ und einen Aufruf von Mischen.
- Das heißt, unter der Voraussetzung, dass das Mischen terminiert, kann die Endlichkeit des Sortierens durch Mischen per Induktion nach n gezeigt werden.
- Mischen terminiert, da sich in jeder Iteration jeder Schleife der Index l oder der Index r erhöht wird und die Abbruchbedingungen von der Größe von l und r abhängen.

Lemma 2.3 (Die Laufzeit, die Korrektheit und der Speicheraufwand von Mischen)

Mischen erfüllt die folgenden Bedingungen:

- *Das Verfahren ist korrekt: Komponenten aufsteigend sortierter Arrays L und R werden in das Array A mit* $\textsc{Länge}[A] = \textsc{Länge}[L] + \textsc{Länge}[R]$ *in einer aufsteigend sortierten Reihenfolge gemischt.*
- *Die Laufzeit für jedes Array A der Länge n ist $\Theta(n)$.*
- *Der Speicheraufwand ist $\Theta(1)$.*

Beweis.

- Die Aussagen über den Speicheraufwand sind klar.
- Zur Korrektheit: das Element aus L oder R, welches in das R aktuell kopiert wird ist nicht größer als die Elemente die zu einem späteren Zeitpunkt kopiert werden, denn unter den beiden aktuellen Elementen von L und R wählt man das Kleinere.
- Jedes Element von L und R wird in einem Zeitpunkt der Ausführung in A kopiert.
- Zur Laufzeit: In jeder Iteration wird entweder ein Element von L oder ein Element von R abgearbeitet.
- Das heißt, die Gesamtanzahl der Iteration von allen drei Schleifen ist genau $\textsc{Länge}[L] + \textsc{Länge}[R] = n$.
- Pro Iteration macht man mindestens eine Elementoperation und höchstens konstant viele Elementaroperationen.
- Die Laufzeit ist daher $\Theta(n)$.

□

Theorem 2.4 (Über Sortieren durch Mischen)

Für das Sortieren durch Mischen gelten die folgenden Behauptungen:

- *das Verfahren löst das Sortierproblem.*
- *Die Laufzeit auf jedem Array der Länge n ist $\Theta(n \log n)$.*
- *Der Speicheraufwand auf jedem Array der Länge n ist $\Theta(n)$.*

Beweis der Korrektheit.

- Korrektheit: Induktion über $n \in \mathbb{N}_0$.
- Für Arrays der Größe höchstens eins ändert das Verfahren das Array A nicht und ist somit korrekt.
- Sei $n \in N$, $n \geq 2$ und sei das Verfahren korrekt auf Arrays der Länge höchstens $n - 1$.
- Auf Arrays der Länge n macht Sortieren durch Mischen zwei rekursive Aufrufe des Sortierens durch Mischen mit Arrays der Länge höchstens $\lceil n/2 \rceil \leq n - 1$.
- Nach der Induktionsvoraussetzung Sortieren die rekursiven Aufrufe die beiden Teilarrays.
- Die Korrektheit für Arrays der Länge n folgt nun aus der Korrektheit des Mischens.

□

Beweis (Laufzeit):

- Wir zeigen, dass die Laufzeit $O(n \log n)$ ist.
- Wir wählen eine Konstante $c > 0$, für welche die folgende Aussage durch Induktion gezeigt werden kann: Die Laufzeit vom Sortieren durch Mischen auf Arrays der Länge $n \geq 2$ ist höchstens $cn \log n$.
- Damit diese Aussage für $n \leq 3$ wahr wird, reicht es c eine obere Schranke an die Laufzeit für Arrays der Länge 2 und 3 zu wählen.
- Sei $n \geq 4$ und sei die Laufzeit des Sortierens durch Mischen auf Arrays der Länge $k \in \{2, \ldots, n-1\}$ höchstens $ck \log k$.
- Wir zeigen nun, dass die Laufzeit vom Sortieren durch Mischen auf Arrays der Länge n höchstens $cn \log n$ ist (wenn die Konstante $c > 0$ passend fixiert ist).
- Da das Sortieren durch Mischen zwei rekursive Aufrufe auf Arrays der Länge $\lfloor n/2 \rfloor$ und $\lceil n/2 \rceil$ macht, einen Aufruf des Mischens mit der linearen Laufzeit in n und die Belegung der Arrays L und R mit den Werten aus A der linearen Laufzeit, ergibt sich mit der Berücksichtigung der Induktionsvoraussetzung die obere Schranke

$$T := \underbrace{c \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor}_{\text{Sortieren von } L} + \underbrace{c \lceil n/2 \rceil \log \lceil n/2 \rceil}_{\text{Sortieren von } R} + \underbrace{an}_{\text{Kopieren und Mischen}}$$

an die Gesamtlaufzeit, wobei $a > 0$ eine Konstante ist.

□

Beweis (Laufzeit).

- Die Terme in Logarithmen werden durch $(n+1)/2$ abgeschätzt:

$$T \leq c(\lfloor n/2 \rfloor + \lceil n/2 \rceil) \log((n+1)/2) + an.$$

- Der Ausdruck in Klammern ist n:

$$T \leq cn \log((n+1)/2) + an.$$

- Wir schätzen $(n+1)/2$ durch $3n/4$ ab und erhalten

$$T \leq cn(\log n - \log(4/3)) + an.$$

- Man hat also $T \leq n \log n$ (wie gewünscht), wenn die Konstante c so gewählt wird, dass $c \log(4/3) \geq a$ erfüllt ist.
- Auf eine ähnliche Weise kann man auch zeigen, dass die Laufzeit auf jedem Array der Länge n gleich $\Omega(n \log n)$ ist.
- Es folgt, dass die Laufzeit $\Theta(n \log n)$ ist.

□

Beweis (Speicheraufwand).

- Wir zeigen, dass für eine Konstante $c > 0$ der Speicheraufwand auf Arrays der Länge $n \geq 2$ höchstens cn ist (das heißt, wir zeigen, dass der Speicheraufwand $O(n)$ ist).
- Induktion über n: Für $n = 2$ reicht es, c genüngend groß zu wählen.
- Sei $n \geq 3$ und sei der Speicheraufwand für Arrays der Länge $k \in \{2, \ldots, n-1\}$ höchstens ck.
- Für ein beliebiges Array der Länge n ergibt sich die Abschätzung

$$S \leq \underbrace{c\lceil n/2 \rceil}_{\text{rekursive Aufrufe}} + \underbrace{an}_{\text{Kopieren in } L \text{ und } R},$$

  an den Speicheraufwand S mit einer Konstante $a > 0$
- Wir schätzen $\lceil n/2 \rceil$ durch $\lceil n/2 \rceil \leq (n+1)/2 \leq 3n/4$ ab und erhalten

$$S \leq (3c/4 + a)n.$$

- Somit hat man $S \leq cn$, wenn $c \geq 4a$ gilt.
- Die untere Schranke $\Omega(n)$ an den Speicheraufwand kann analog gezeigt werden.

□

Der Unterschied der Worst-Case-Laufzeiten

$\Theta(n^2)$ und $\Theta(n \log n)$ macht sich bei der Vergrößerung der Array-Länge n sehr schnell bemerkbar. (Ausprobieren!)

Wozu sortieren?

- Um besser zu suchen:
  - $\Theta(\log n)$ mit der binären Suche vs.
  - $\Theta(n)$ mit der sequentiellen Suche
- Nützlich als Hilfsmittel bei vielen weiteren Rechenaufgaben.

Heap-Sort

- Aus Array einen Heap[1] machen.
- Aus Heap alle Elemente entfernen.
- Fertig!

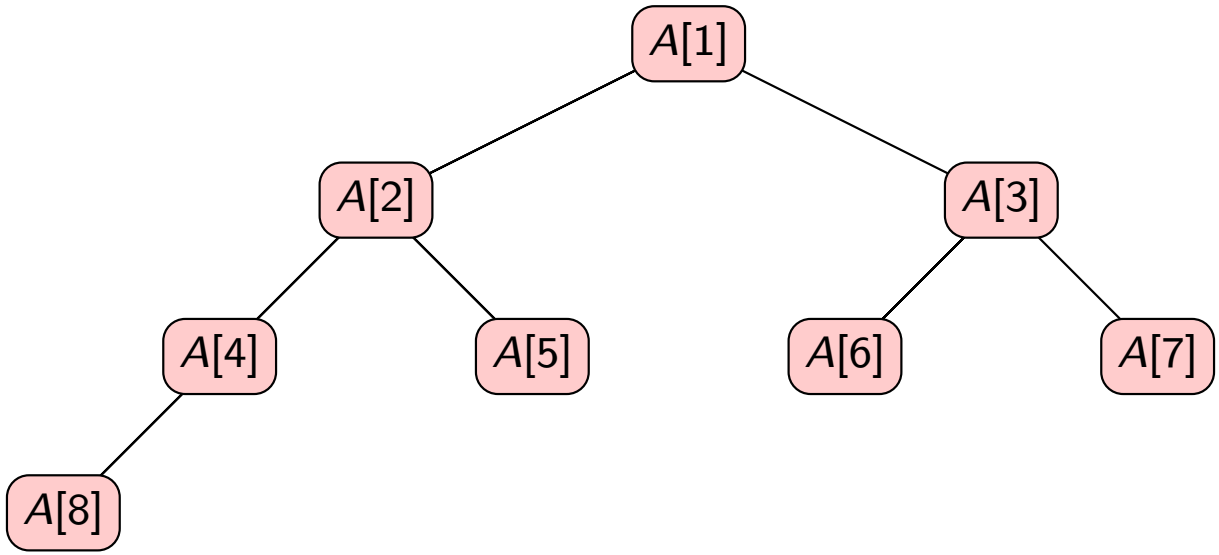
Vorteile

- Asymptotisch so schnell wie Sortieren durch Einfügen.
- Speicheraufwand besser als beim Sortieren durch Einfügen.
- (Nebenbei lernen wir Heaps kennen!)

[1] Heap ist eine Datenstruktur

Geordente Bäume

Wir führen einen zusätzlichen Parameter $\textsc{Grösse}[A]$ ein mit $0 \leq \textsc{Grösse}[A] \leq \textsc{Länge}[A]$ und interpretieren $A[i]$ mit $1 \leq i \leq \textsc{Grösse}[A]$ als Knoten eines sogenannte geordneten Binärbaums:



- Sind $A[i]$ und $A[2i]$ Knoten des Baums, so heißt $A[2i]$ linkes Kind von $A[i]$.
- Sind $A[i]$ und $A[2i+1]$ Knoten des Baums, so heißt $A[2i+1]$ rechtes Kind von $A[i]$.
- Der Knoten $A[1]$ heißt die Wurzel des Baums.
- Ist $A[i]$ ein Konten und keine Wurzel, so ist $A[\lfloor i/2 \rfloor]$ der Vater von $A[i]$.

2.5 $v = \textsc{Vater}(i)$

1: $v := \lfloor i/2 \rfloor$

2.6 $k = \textsc{Linkes-Kind}(i)$

1: $k := 2 \cdot i$

2.7 $k = \textsc{Rechtes-Kind}(i)$

1: $k := 2 \cdot i + 1$

Nachfahre

eines Knotens ist ein Kind oder ein Nachfahre eines Kinds.

Vorfahre

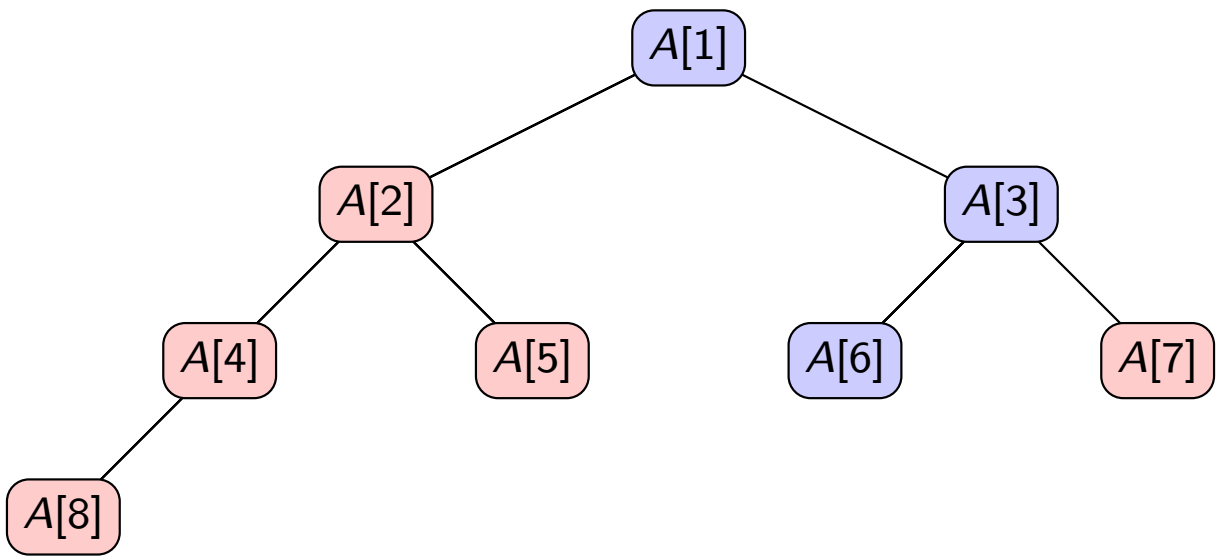
eines Knotens ist der Vater oder ein Vorfahre des Vaters.

Tiefe

Die Konten $A[i]$ mit $2^t \leq i \leq 2^{t+1} - 1$ und $i \leq \textsc{Grösse}[A]$ befinden sich in der Tiefe t.

Pfad

der Länge t ist eine Folge $v_0, \ldots, v_t$, bei der v_i Vater von v_{i+1} für alle $0 \leq i < t$ gilt.

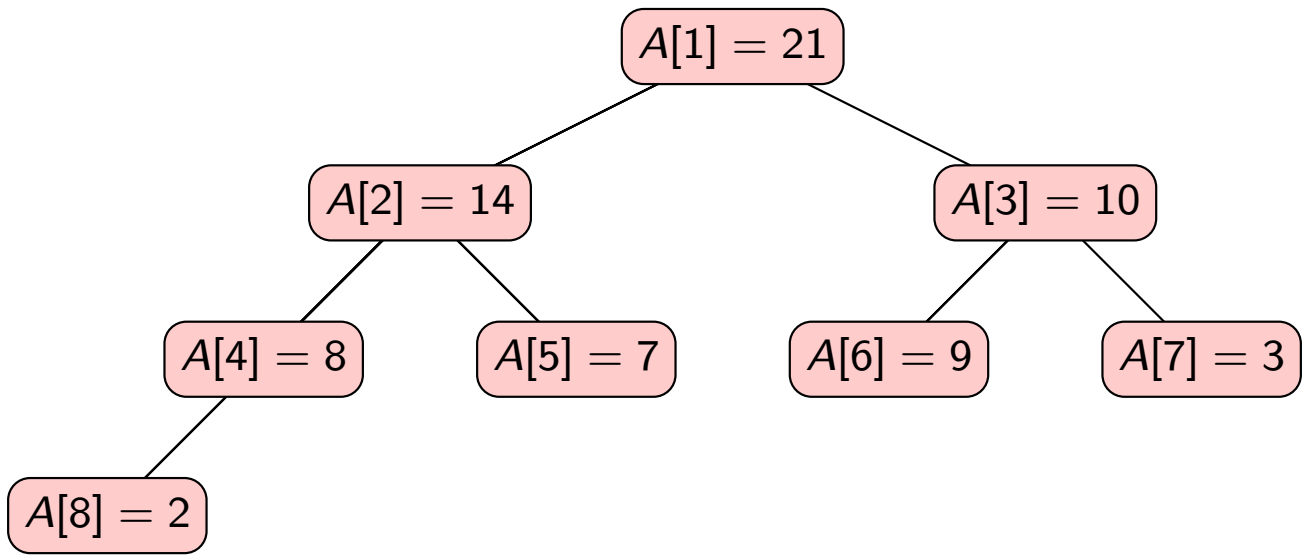


Heap

ist ein Array A mit $\textsc{Grösse}[A] \leq \textsc{Länge}[A]$, für das die sogenannte *Max-Heap-Eigenschaft*

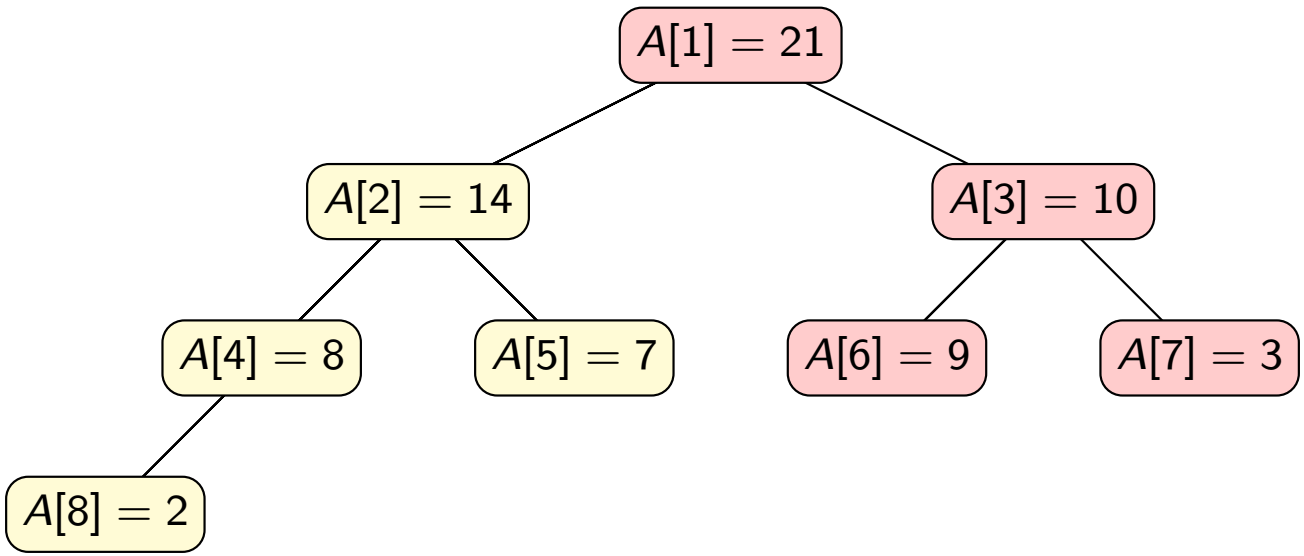
$$A[\textsc{Vater}(i)] \geq A[i]$$

für alle $i = 2, \ldots, \textsc{Grösse}[A]$ erfüllt ist. Analog werden auch *Min-Heap* durch die *Min-Heap-Eigenschaft* eingeführt.



Heaps sind rekursive Datenstrukturen,

denn jeder Knoten des Heaps it Wurzel eines (kleineren) Heaps.



Die Höhe

eines Heaps ist die maximale Tiefe der Knoten des Heaps.

Größter von drei Knoten

2.8 $g :=$ GRÖSSTER-VON-DREI-KNOTEN(A, i)

require: A ist Array mit zusätzlichem Attirbut GRÖSSE$[A]$.
ensure: g ist Index in $\{i, \text{LINKES-KIND}(i), \text{RECHTES-KIND}(i)\} \cap \{1, \ldots, \text{GRÖSSE}[A]\}$ mit dem größten $A[g]$.

```
1: l := LINKES-KIND(i)
2: r := RECHTES-KIND(i)
3: g := i
4: if l ≤ GRÖSSE[A] und A[l] > A[i] :
5:     g := l
6: end
7: if r ≤ GRÖSSE[A] und A[r] > A[g] :
8:     g := r
9: end
```

2.9 MAX-HEAPIFY(A, i)

require: A ist Array mit zusätzlichem Attribut GRÖSSE$[A]$, $i \in \mathbb{N}$ und $1 \leq i \leq$ GRÖSSE$[A]$, die Teilbäume von A mit Wurzeln $A[$LINKES-KIND$(i)]$ und $A[$RECHTES-KIND$(i)]$ sind Heaps (falls LINKES-KIND$(i) \leq$ GRÖSSE$[A]$ bzw. RECHTES-KIND$(i) \leq$ GRÖSSE$[A]$ gilt).

ensure: Der Teilbaum mit Wurzelindex i wird Heap

```
1: g := GRÖSSTER-VON-DREI-KNOTEN(A, i)
2: if g ≠ i :
3:     Vertausche A[i] und A[g]
4:     MAX-HEAPIFY(A, g)
5: end
```

Beispiel 2.5

Sei $A := [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]$ mit GRÖSSE$[A] = 10$. Wir verfolgen die Ausführung von MAX-HEAPIFY$(A, 2)$

- $A[2]$ und $A[4]$ werden vertauscht.
- $A[4]$ und $A[9]$ werden vertauscht.

Korrektheit von Heapify

Theorem 2.6

MAX-HEAPIFY *arbeitet korrekt.*

Beweis.

Die Korrektheit von MAX-HEAPIFY kann per Induktion nach der Anzahl N der Elemente in dem Baum mit Wurzel i bewiesen werden. Ist $N = 1$, so arbeteit MAX-HEAPIFY offensichtlich korrekt. Agenommen, MAX-HEAPIFY arbeitet korrekt für Teilbäume der Größe höchstens N mit $N \in \mathbb{N}$. Sei A und i so, dass der Teilbaum mit Wurzel i Größe $N + 1$ hat. Die Funktion GRÖSSTER-VON-DREI-KNOTEN arbeitet offensichtlich Korrekt, d.h. $A[g]$ das Maximum von $A[i], A[\text{LINKES-KIND}(i)]$ und $A[\text{RECHTES-KIND}(i)]$. Der Umtausch in Zeile 3 von MAX-HEAPIFY erstellt die Heap-Eigenschaft bezüglich i und den Kindern von i. Die Heap-Eigenschaft für Bäume, deren Wurzel die Kinder von i sind, folgt aus der Induktionsvoraussetzung. □

Laufzeit von Heapify

Theorem 2.7

Sei A Array (Baum) mit Attributen $\textsc{Grösse}[A] \leq \textsc{Länge}[A]$. *Sei* $1 \leq i \leq \textsc{Grösse}[A]$ *und seien die Teilbäume von* $A[\textsc{Linkes-Kind}(A, i)]$ *und* $A[\textsc{Rechtes-Kind}(A, i)]$ *Max-Heaps. Dann ist Die Worst-Case-Laufzeit von* $\textsc{Max-Heapify}(A, i)$ *gleich* $\Theta(h)$, *wobei h ist die Höhe von* $A[i]$.

Beweis.

Die Laufzeiten vom vertauschen von Zwei Knoten und dem Aufruf von $\textsc{Grösster-von-drei-Knoten}(A, i)$ sind $\Theta(1)$. Wir zählen nach, wie viel Mal $\textsc{Max-Heapify}$ innerhalb der Ausführung aufgerufen wird. Ist $h = 0$, so wird $\textsc{Max-Heapify}$ genau ein mal aufgerufen. Ist $h > 0$ so wird nach dem Aufruf von $\textsc{Max-Heapify}(A, i)$ entweder kein Aufruf von $\textsc{Max-Heapify}$ erfolgen (in diesem Fall ist die Laufzeit $O(1)$) oder ein Afruf von $\textsc{Max-Heapify}(A, g)$, wobei g ein Kind von i ist. Die Höhe von g ist höchstens $h - 1$. Daher lässt sich per Induktion nachweisen, dass die Laufzeit $O(h)$ ist. Um zu sehen, dass im schlechtesten Fall die Schranke erreicht werden kann, reicht es ein Beispiel zu konstruieren. Ist $A[i]$ kleiner als die restlichen elemente von A, so beträgt die Laufzeit $\Omega(h)$ Zeiteinheiten. □

Heap erzeugen

2.10 MAX-HEAP-ERZEUGEN(A)

require: A ist Array.
ensure: Attribut GRÖSSE[A] wird eingeführt. Durch Vertausch von seinen Elementen wird A in ein Heap konvertiert.

1: Attribut GRÖSSE[A] einführen
2: GRÖSSE[A] := LÄNGE[A]
3: **for** $i := \lfloor \text{GRÖSSE}[A]/2 \rfloor, \ldots, 1$ mit Schritt -1 **:**
4: ▷ *Invariante: Die Bäume mit Wurzeln* $i+1, \ldots,$ GRÖSSE[A] *sind Heaps*
5: MAX-HEAPIFY(A, i)
6: **end**

Beispiel 2.8

Sei $A := [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$ und GRÖSSE[A] := 10. Wir illustrieren die Arbeit von MAX-HEAP-ERZEUGEN(A) an dem gegebenen A.

- $A[5]$ und $A[10]$ werden nicht vertauscht.
- $A[4]$ und $A[8]$ werden vertauscht.
- $A[3]$ und $A[7]$ werden vertauscht.
- $A[2]$ und $A[5]$, $A[5]$ und $A[10]$ werden vertauscht.
- $A[1]$ und $A[2]$, $A[2]$ und $A[4]$, $A[4]$ und $A[9]$ werden vertauscht.

Korrektheit der Erzeugung eines Heaps

Theorem 2.9

MAX-HEAP-ERZEUGEN *arbeitet korrekt.*

Beweis.

Sei $n := \text{GRÖSSE}[A]$. Dann besitzen die Knoten $j = 1, \ldots, n$ mit $2i > n$ keine Kinder (d.h., diese Knoten sind Blätter des Baums A). D.h. beim Antreten der ersten Iteration gilt $i = \lfloor n/2 \rfloor$ und damit sind die Knoten $j = \lfloor n/2 \rfloor + 1, \ldots, n$ Heap-Wurzeln. Wenn die Schleifeninvariante für ein i erfüllt ist, sind die Knoten $i + 1, \ldots, n$ Wurzeln von Heaps. Daher ist die Voraussetzung für den Aufruf von MAX-HEAPIFY(A, i) erfüllt, und nach der Ausführung von MAX-HEAPIFY(A, i) ist auch i Wurzel eines Heaps. Dies zeigt, dass die Schleifeninvariante auch an der nächsten Iteration erfüllt wird. Der Befehl MAX-HEAPIFY$(A, 1)$ ist der letzte Aufruf von MAX-HEAPIFY innerhalb von MAX-HEAP-ERZEUGEN. Dieser Aufruf erstellt die Heap-Eigenschaft für den gesamten Baum A. □

Laufzeit der Erzeugung eines Heaps

Bemerkung

Die Laufzeit von MAX-HEAP-ERZEUGEN ist $O(n \log n)$ mit $n := \text{LÄNGE}[A]$.

Lemma 2.10

Sei A Heap mit $n := \text{GRÖSSE}[A]$ und $h \in \mathbb{Z}$ mit $0 \leq h \leq \lfloor \log_2 n \rfloor$. Dann besitzt A höchstens $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ Elemente der Höhe h.

Laufzeit der Heap-Erzeugung

Theorem 2.11

Die Worst-Case-Laufzeit von MAX-HEAP-ERZEUGEN *auf einem Array der Länge n ist* $\Theta(n)$.

Beweis.

Da MAX-HEAPIFY auf einem Element der Höhe h in Zeit $O(h)$ läuft und auf die Funktion MAX-HEAP-ERZEUGEN auf jedem Element des Heaps höchstens ein Mal MAX-HEAPIFY aufruft, beträgt die Laufzeit von MAX-HEAP-ERZEUGEN

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right) = O(n)$$

Zeiteinheiten. Oben haben wir die Abschätzung

$$\sum_{h=0}^{\infty} \frac{h}{2^h} < \infty$$

benutzt.
Die Laufzeit von MAX-HEAP-ERZEUGEN muss $\Omega(n)$ Operationen betragen, weil jedes Element von A mindestens ein Mal durchgescannt werden muss. □

Prozedur Heapsort

2.11 HEAPSORT(A)

require: A ist Array; Elemente von A sind mit Hilfe von "$\leq$" vergleichbar.
ensure: A wird aufsteigend sortiert.

```
1: MAX-HEAP-ERZEUGEN(A)
2: for i = LÄNGE[A], ..., 2 mit Schritt −1 :
3:     ▷ Invariante: Elemente von A[GRÖSSE[A]+1..LÄNGE[A]] sind aufsteigend sortiert
       und größer als Elemente von A[1..GRÖSSE[A]].
4:     VERTAUSCHEN(A[1], A[i])
5:     GRÖSSE[A] := GRÖSSE[A] − 1
6:     MAX-HEAPIFY(A, 1)
7: end
```

- Die Richtigkeit folgt direkt.
- Die Laufzeit beträgt offensichtlich $O(n \log n)$ Operationen.

Beispiel 2.12

Arbeitsweise von Heapsort auf dem Heap $A := [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ mit GRÖSSE$[A] = 10$...

Prioritätsschlangen

Definition 2.13

Priortitätsschlange ist eine Datenstruktur, die eine Menge S darstellt, in welcher jedem Element x ein Wert zugeordnet ist, der der *Schlüssel* von x genannt wird. Eine *Max-Prioritätswarteschlange unterstützt folgende Operationen*:

- EINFÜGEN(S, x) - ein neues Element einfügen.
- MAXIMUM(S) - maximum zurückgeben.
- MAXIMUM-ENTFERNEN(S) - entfernt ein maximales Element von S und gibt dieses zurück.

Anwendungen von Prioritätsschlangen:

- Zeitplanung (engl.: scheduling) von Jobs auf einem Rechner.
- Ereignisgetriebene Simulatoren.

2.12 $m =$ HEAP-MAXIMUM(A)

```
1: m := A[1]
```

2.13 $m =$ HEAP-MAXIMUM-ENTFERNEN(A)

```
1: m := A[1]
2: A[1] := A[GRÖSSE[A]]
3: GRÖSSE[A] := GRÖSSE[A] − 1
4: MAX-HEAPIFY(A)
```

2.14 HEAP-SCHLÜSSEL-VERGRÖSSERN(A, i, x)

require: A ist Heap; $1 \leq i \leq$ GRÖSSE$[A]$; $x \geq A[i]$.
ensure: Der Wert von $A[i]$ wird gleich x gesetzt; Heap-Eigeschaft wird neu erstellt.

```
1: A[i] := x
2: while i > 1 und A[VATER(i)] < A[i] :
3:     VERTAUSCHEN(A[i], A[VATER(i)])
4:     i := VATER(i)
5: end
```

Analyse der Schlüsselvergrößerung

Theorem 2.14

Pseudocode 2.14 arbeitet korrekt.

Beweis.

Zu Beginn jeder Iteration erfüllt A die Max-Heapeigenschaft $A[j] \geq A[\textsc{Vater}(j)]$ für alle $j = 2, \ldots, \textsc{Grösse}[A]$ mit $j \neq i$. Nach jeder Iteration wird i streng kleiner. Dies zeigt die Korrektheit. □

Bemerkung

Die Laufzeit von Heap-Schlüssel-Vergrössern ist $O(\log n)$ mit $n := \textsc{Grösse}[A]$.

Neues Element einfügen

2.15 MAX-HEAP-EINFÜGEN(A, x)

require: A ist Heap.
ensure: das Element x wird in A eingefügt, sodass A ein Heap bleibt.

1: GRÖSSE$[A]$:= GRÖSSE$[A] + 1$
2: $A[$GRÖSSE$[A]] := -\infty$
3: HEAP-SCHLÜSSEL-VERGRÖSSERN$(A,$ GRÖSSE$[A], x)$

Theorem 2.15

Die Laufzeit von MAX-HEAP-EINFÜGEN(A, x) *ist* $O(\log n)$ *mit* $n :=$ GRÖSSE$[A]$.

Element entfernen

2.16 HEAP-ENTFERNEN(A, i)

require: A ist Heap, der das Element $A[i]$ besitzt.
ensure: Das Element $A[i]$ wird aus A enfernet, sodass A ein Heap bleibt.

```
1: x := A[GRÖSSE[A]]
2: GRÖSSE[A] := GRÖSSE[A] − 1
3: if x > A[i] :
4:     HEAP-SCHLÜSSEL-VERGRÖSSERN(A, x, i)
5: else:
6:     A[i] := x
7:     HEAPIFY(A, i)
8: end
```

Vergleich zu direkter Implementierung

- Direkte Implementierung von Prioritätsschlangen: Prioritätsschlangen auf der Basis von Arrays mit sequentieller Suche.

Worst-Case-Laufzeiten:

	Prioritätsschlangen mit sequentiellen Arrays	Prioritätsschlangen mit Heaps
Maximum	$\Theta(n)$	$\Theta(1)$
Maximum entfernen	$\Theta(n)$	$\Theta(\log n)$
einfügen	$\Theta(1)$	$\Theta(\log n)$
entfernen	$\Theta(n)$	$\Theta(\log n)$
schlüssel verändern	$\Theta(1)$	$\Theta(\log n)$

Quicksort: Generische Beschreibung

2.17 Quicksort (Textbeschreibung)

require: $A[p \,..\, r]$ ist Teilarray des Arrays A; $p, r \in \mathbb{N}$.
ensure: $A[p \,..\, r]$ wird aufsteigend sortiert.

1: Ist $p \geq r$, Prozedur beenden.
2: Komponenten von $A[p \,..\, r]$ vertauschen und gleichzeitig ein q mit $p \leq q \leq r$ bestimmen, sodass $x \leq y \leq z$ für alle x in $A[p \,..\, q-1]$, z in $A[q+1 \,..\, r]$ und $y = A[q]$.
3: Die Arrays $A[p \,..\, q-1]$ und $A[q+1 \,..\, r]$ mit Quicksort sortieren.

2.18 $\textsc{Quicksort}(A, p, r)$

1: **if** $p < r$ **:**
2: $\quad q := \textsc{Partition}(A, p, q)$
3: $\quad \textsc{Quicksort}(A, p, q-1)$
4: $\quad \textsc{Quicksort}(A, q+1, r)$
5: **end**

Richtigkeit von Quicksort

Bemerkung

Solange PARTITION korrekt arbeitet, arbeitet auch QUICKSORT korrekt; der Beweis ist Analog zum Beweis der Richtigkeit von SORTIEREN-DURCH-MISCHEN.

Quicksort: Partitionierung

2.19 $q = \text{PARTITION}(A, p, r)$

require: $A[p\,..\,r]$ ist Teilarray von Array A; $p, r \in \mathbb{N}$, $p < r$.
ensure: Komponenten von $A[p\,..\,r]$ werden vertauscht; Elemente von $A[p\,..\,q-1]$ sind nicht kleiner als $A[q]$; $A[q]$ ist nicht kleiner als die Elemente von $A[q+1\,..\,r]$.

```
1:  i := p − 1
2:  for j := p, . . . , r − 1 :
3:      ▷ Invariante: Elemente von A[p . . i] sind nicht größer als A[r];
4:      ▷ Invariante: Elemente von A[i + 1 . . j − 1] sind nicht kleiner als A[r].
5:      if A[j] ≤ A[r] :
6:          VERTAUSCHEN(A[i + 1], A[j])
7:          i := i + 1
8:      end
9:  end
10: VERTAUSCHEN(A[i + 1], A[r])
11: q := i + 1
```

Richtigkeit der Partitionierung

Theorem 2.16

PARTITION *arbeitet korrekt.*

Beweis.

Die in PARTITION angegebenen Invarianten lässen werden offensichtlich beibehalten (auch wenn $A[p \,..\, i]$ oder $A[i+1 \,..\, j-1]$ leer ist).
Der Fall, wo $A[i+1 \,..\, j-1]$ leer ist, tritt zu Beginn der Ausführung der Schleife auf; denn es gilt $i = p-1$ und $j = p$. In diesem Fall ist $j - i = 1$. Auf weiteren Iterationen wird $j - i$ nicht kleiner; denn auf jeder Iteration wird entweder nur j um eins größer oder i und j gleichzeitig um eins größer. Wann immer $j - i = 1$ gilt, sind ist $A[i+1]$ und $A[j]$ das selbe Element von A. In diesem Fall verursacht VERTAUSCHEN$(A[i+1], A[j])$ keine Veränderungen in A. □

Laufzeit der Partitionierung

Theorem 2.17

Die Laufzeit von PARTITION(A, p, r) *ist* $\Theta(n)$*, wobei* $n :=$ LÄNGE$[A[p \,..\, r]]$.

Beweis.

Die Schleife in PARTITION macht $n - 1$ Iterationen; $\Theta(1)$ Operationen pro Iteration. □

Bemerkung

Die Anzahl der Vergleiche ist immer $n - 1$; denn jedes Element wird mit dem Pivotelement vergliechen. Die Anzahl der Zuweisungen kann sich variieren, je nach dem ob $A[p \,..\, r]$ schon geordnet ist oder nicht.

Laufzeit von Quicksort bei Ungleichmässiger Partitionierung

Ungleichmässige Partitionierung vergrößert die Laufzeit von Quicksort:

Theorem 2.18

Sei $T(n)$ die Laufzeit von QUICKSORT *auf einem aufsteigend sortierten Array der Länge n. Dann $T(n) = T(n-1) + \Theta(n)$ und $T(n) = \Theta(n^2)$.*

Beweis.

Ist A ein aufsteigend sortiertes Array, dann wird bei jedem Aufruf von PARTITION(A, p, r) mit $1 \leq p \leq r \leq$ LÄNGE$[A]$ das Teilarray $A[p\,..\,r]$ durch das Pivotelement $A[q]$ in Teilarrays der Länge $m-1$ und 0 aufgeteilt, wobei $m := r - p + 1$. Weil PARTITION(A, p, r) Laufzeit $\Theta(m)$ hat, ergibt sich die Beziehung $T(n) = T(n-1) + \Theta(n)$. D.h. es existiert eine Funktion $f(n)$ auf mit $C_1 n \leq f(n) \leq C_2 n$ für geeigente $C_1, C_2 > 0$ und alle $n \in \mathbb{N}$, sodass $T(n) = T(n-1) + f(n)$ für alle $n \in \mathbb{N}$. Es gilt:

$$T(n) = T(n-1)+f(n) = T(n-2)+f(n-1)+f(n) = \cdots = T(0)+f(1)+\cdots+f(n).$$

Die Behauptung folgt aus den Abschätzungen

$$C_1 \frac{n(n+1)}{2} \leq f(1) + \cdots + f(n) \leq C_2 \frac{n(n+1)}{2}.$$

□

Laufzeit von Quicksort bei gleichmässiger Partitionierung

Gleichmässige Partitionierung führt zu schneller Laufzeit:

Theorem 2.19

Sei $T(n)$ die Laufzeit von Quicksort *auf Arrays A, bei welchen während der Ausführung von* Quicksort *jeder Aufruf von* Partition(A, p, r) *das Array $A[p \,..\, r]$ in Teile der länge $\lfloor \frac{n-1}{2} \rfloor$ und $\lceil \frac{n-1}{2} \rceil$ aufteilt (wobei n ist die Länge von $A[p \,..\, r]$). Dann erfüllt $T(n)$ die Gleichung*

$$T(n) = T(\left\lfloor \frac{n-1}{2} \right\rfloor) + T(\left\lceil \frac{n-1}{2} \right\rceil) + O(n),$$

und es gilt $T(n) = O(n \log n)$.

Worst-Case-Laufzeit von Quicksort

Theorem 2.20

Die Worst-Case-Laufzeit $T(n)$ von Quicksort auf einem Array der Länge erfüllt $T(n) = \Theta(n^2)$.

Beweis.

Die Gleichheit $T(n) = \Omega(n^2)$ war oben gezeigt. Es bleibt $T(n) = O(n^2)$ zu zeigen. Falls $n > 1$, macht jeder Aufruf von $\textsc{Quicksort}(A, 1, n)$ einen Aufruf von $\textsc{Partition}$ mit der Laufzeit $O(n)$ und zwei rekursive Aufrufe für Teilarrays der Länge j und $n - 1 - j$, mit $j = 0, \ldots, n - 1$. Daher gilt

$$T(n) \leq \max_{j=0,\ldots,n-1} (T(j) + T(n - 1 - j)) + Cn$$

für alle $n \in \mathbb{N}$, wobei $C > 0$ ist eine Konstante. Wir zeigen per Induktion nach n, dass es eine Konstante $C' > 0$ existiert, sodass $T(n) \leq C'n^2$ für alle $n \in \mathbb{N}$ gilt. Für $n = 1$ gilt $T(n) \leq 2T(0) + C \leq C'$, falls C' genügend groß ist. Angenommen, die Aussage gelte für alle n mit $1 \leq n \leq N - 1$ mit $N > 1$, dann gilt

$$\begin{aligned} T(N) &\leq C' \max_{j=0,\ldots,N-1} (j^2 + (N - 1 - j)^2) + CN \\ &\leq C'(N - 1)^2 + CN = C'N^2 - 2C'N + C' + CN \leq C'N^2, \end{aligned}$$

falls $C' \geq C$ gilt. □

Mittlere Laufzeit von Quicksort

Definition 2.21

Man fixiere $x_1 < x_2 < ... < x_n$ und betrachte alle Permutationen der Elemente $x_1, .., x_n$, die in der Form eines Arrays a der Länge n mit $\{A[i] : i = 1, ..., n\} = \{x_1, ..., x_n\}$ dargestellt werden. Die Laufzeit von QuickSort auf A ist von der Auswahl der Werte $x_1, ..., x_n$ unabhängig. Die mittlere Laufzeit von QuickSort auf Permutationen von $x_1, ..., x_n$ nennen wir die *mittlere Laufzeit* von QuickSort auf einem n-elementigen Array.

Theorem 2.22

Angenommen QUICKSORT *benutzt eine Partitionierung mit der linearen Laufzeit, welche die relative Reihenfolge der Elemente* $<$ *Pivotelement und sowie der Elemente* $>$ *Pivotelement nicht verändert. Dann ist die mittlere Laufzeit von* QUICKSORT *auf einem Array der Länge n gleich* $O(n \log n)$.

Bemerkung

Partitionierung mit den o.g. Voraussetzungen existiert.

Bemerkung

Bei einer beliebigen Partiotionierung ist die Voraussetzung nicht unbedingt erfüllt.

Ausrechnen der mittleren Laufzeit, I

Sei A ein Array der Länge n mit

$$\{A[i] : i = 1, \ldots, n\} = \{x_1, \ldots, x_n\}. \tag{2.1}$$

Man fixiere k mit $A[n] = x_k$. Während der Ausführung $\textsc{Quicksort}(A, 1, n)$ wird $\textsc{Quicksort}$ rekursiv auf Teilarrays der Länge $k-1$ und $n-1-k$ aufgerufen. Für $1/n$-tel aller Arrays A mit (2.1) gilt $A[n] = x_k$. Daher gilt:

$$\begin{aligned}
T(n) &= \frac{1}{n} \sum_{k=1}^{n} (T(k-1) + T(n-k)) + \Theta(n) \\
&= \frac{2}{n} \sum_{k=1}^{n} T(k-1) + \Theta(n) \\
&= \frac{2}{n} \sum_{k=0}^{n-1} T(k) + \Theta(n) \\
&= \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n).
\end{aligned}$$

Aussrechnen der mittleren Laufzeit, II

Wir zeigen per Induktion nach n, dass es $\alpha > 0$ und $\beta > 0$ existieren mit $T(n) \leq \alpha n \log_2 n + \beta$ für alle $n \in \mathbb{N}$. Die Behauptung gilt offensichtlich für $n = 1$, falls $\beta > 0$ groß genug ist. Angenommen, es gelte $T(k) \leq \alpha k \log_2 k + \beta$ für alle $k = 1, ..., n-1$ und ein $n \geq 2$. Dann

$$\begin{aligned} T(n) &\leq \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n) \\ &\leq \frac{2}{n} \sum_{k=1}^{n-1} (\alpha k \log_2 k + \beta) + \Theta(n) \\ &= \frac{2\alpha}{n} \sum_{k=1}^{n} k \log_2 k + \frac{2\beta(n-1)}{n} + \Theta(n) \end{aligned}$$

Ausrechnen der mittleren Laufzeit, III

Wir benutzen die Ungleichung:

$$\sum_{k=1}^{n-1} k \log_2 k \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \tag{2.2}$$

Aus (2.2) folgt:

$$\begin{aligned} T(n) &\leq \frac{2\alpha}{n} \left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \frac{2\beta(n-1)}{n} + \Theta(n) \\ &= \alpha n \log_2 n - \frac{\alpha}{4} n + 2\beta + \Theta(n) \\ &\leq \alpha n \log_2 n + \beta, \end{aligned}$$

falls α groß genug ist.
Q.E.D.

Abschätzung der $k \log k$-Summe, Idee

In dem nachfolgenden Lemma wird (2.2) bewiesen.

Lemma 2.23

Für alle $n \in \mathbb{N}, n \geq 2$ gilt:

$$\sum_{k=1}^{n-1} k \log_2 k \leq \frac{1}{2} n^2 \log_2 n - \frac{1}{8} n^2$$

Die Beweisidee: Die Summe in zwei möglichst gleichlange Hälften zerlegen und in beiden Hälften logarithmische Terme abschätzten.

Abschätzung der $k \log k$-Summe

$$\begin{aligned}
\sum_{k=1}^{n-1} k \log_2 k &\leq \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k \log_2 k + \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k \log_2 k \\
&\leq \log_2 \frac{n}{2} \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k + \log_2 n \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k \\
&= (\log_2 n - 1) \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k + \log_2 n \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k \\
&= \log_2 n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k \\
&= \frac{1}{2} n(n-1) \log_2 n - \frac{1}{2} \left(\left\lceil \frac{n}{2} \right\rceil - 1 \right) \left\lceil \frac{n}{2} \right\rceil \\
&\leq \frac{1}{2} n(n-1) \log_2 n - \frac{1}{2} \left(\frac{n}{2} - 1 \right) \frac{n}{2} \\
&= \frac{1}{2} n^2 \log_2 n - \frac{1}{2} n \log_2 n - \frac{1}{8} n^2 + \frac{n}{4} \\
&\leq \frac{1}{2} n^2 \log_2 n - \frac{1}{8} n^2.
\end{aligned}$$

Randomisierte Version von Quicksort

Um die Situation zu vermeiden, dass Quicksort auf gewissen entarteten Arrays langsam wird, kann man die randomisierte Version von *Quicksort* benutzen.

2.20 $q =$ RANDOMISIERTE-PARTITION(A, p, r)

require: $A[p \,..\, r]$ ist Teilarray von Array A; $p, r \in \mathbb{N}$; $p < r$.
ensure: vgl. PARTITION.

```
1: i := ZUFALLSZAHL(p, r)
2: VERTAUSCHEN(A[r], A[i])
3: q := PARTITION(A, p, r)
```

ZUFALLSZAHL(p, r) liefert eine Zahl aus $\{p, \ldots, r\}$, wobei jede Auswahl ist gleichwahrscheinlich (gleichmässige Verteilung).

2.21 RANDOMISIERTER-QUICKSORT(A, p, r)

```
1: if p < r :
2:     q := RANDOMISIERTE-PARTITION(A, p, q)
3:     RANDOMISIERTER-QUICKSORT(A, p, q − 1)
4:     RANDOMISIERTER-QUICKSORT(A, q + 1, r)
5: end
```

Erwartete Laufzeit vom randomisierten Quicksort

Theorem 2.24

Sei A beliebiges Array der Länge $n \in \mathbb{N}$. Dann ist der Erwartungswert der Laufzeit von RANDOMISIERTER-QUICKSORT$(A, 1, n)$ *gleich* $O(n \log n)$.

Bemerkung

Die Abweichung von dem Erwartungswert ist nicht groß.

Endrekursion

Definition 2.25

Endrekursion (engl.: *Tail recursion*) ist, wenn ein rekursiver Aufruf der letzte Aufruf der rekursiven Funktion ist. Dieser Aufruf kann durch eine iterative Struktur ersetzt werden.

Quicksort mit der Umwandlung der Endrekursion in iterative Form

2.22 QUICKSORT-II(A, p, r)

```
1: while p < r :
2:     q := PARTITION(A, p, r)
3:     QUICKSORT-II(A, p, q − 1)
4:     p := q + 1
5: end
```

- Die Gesamtanzahl der rekursive Aufrufe wird kleiner (und dadurch auch das Programmstackbelastung)
- Die Version mit zwei Aufrufen ist "symmetrischer" (leichter zu verstehen).

Untere Schranken für Laufzeit von Sortieralgorithmen

Definition 2.26

Ein Sortieralgorithmus, der die sortierte Reihenfolge nur auf der Basis der Vergleiche der Komponenten (mit Hilfe von "$\leq$") bestimmt, heißt *vergleichender Sortieralgorithmus*.

Theorem 2.27

Die Worst-Case-Laufzeit eines jeden vergleichenden Sortieralgorithmus ist $\Omega(n \log n)$, *wobei* n *ist die Länge der Eingabe.*

Beweis der unteren $n \log n$-Schranke für Sortieren

Wir fixieren einen beliebigen Sortieralgorithmus P, d.h., P bekommt ein Array A als Eingabe, und nach der Ausführung von $P(A)$ ist A sortiert. Sei A ein beliebiges Array der Länge n. Während der Ausführung von P auf A wird die Operation $\leq$ auf den Komponenten von A benutzt; diese Operation liefert den Wert WAHR oder FALSCH. Sei $m := m(A)$ die Anzahl die Vergleiche, die während der Ausführung von P auf A gemacht werden. Für $i = 1, \ldots, k$ sei $b_i = b_i(A)$ das Resultat des i-ten Vergleiches und sei $b(A) := (b_1(A), \ldots, b_m(A))$. Seien zwei A' und A'' verschiedene Arrays mit $m(A') \leq m(A'')$, welche die Elemente $1, \ldots, n$ speichern. Dann existiert $1 \leq i \leq m(A')$ mit $b_i(A') \neq b_i(A'')$. Tatsächlich, gilt $b_i(A') = b_i(A')$ für alle $1 \leq i \leq m(A')$, dann muss $m(A') = m(A'')$ (denn der Algorithmus entscheidet nur auf der Basis der Verlgeiche, ob er terminiert oder nicht). Falls $b_i(A') = b_i(A'')$ für alle $1 \leq i \leq m(A') = m(A'')$, dann permutiert P die Elemente von A' auf die gleiche Weise wie die Elemente von A''. Wenn aber $A' \neq A''$, so werden die Resultate der Permutation verschieden sein. Das heißt A' oder A'' wird nicht sortiert, Widerspruch zur Wahl von P. Sei nun M das maximum von $m(A)$ über allen Arrays A, die die Werte $1, \ldots, n$ speichern. Es existieren höchstens 2^M untershiedliche $b(A)$ mit A wie oben. Anderseits ist die Anzahl von Vektoren $b(A)$ mit A wie oben genau $n!$. Es folgt: $n! \leq 2^M$. Daher $M \geq \log_2(n!) = \Omega(n \log n)$.

Entscheidungsbaum

Ein Sortieralgorithmus auf n-elementigen Arrays kann als Binärbaum mit $n!$ Blättern dargestellt werden.

Countingsort: Elemente zählen

2.23 $A = \textsc{Null-Array}(k)$

require: $k \in \mathbb{N}_0$.
ensure: A ist ein Array der Länge n mit $A[i] = 0$ für alle $1 \leq i \leq k$.

```
1: LÄNGE[A] := k
2: for i = 1, ..., k :
3:     A[i] := 0
4: end
```

2.24 $Z = \textsc{Häufigkeitstabelle}(A, k)$

require: A ist Array mit Elementen aus $\{1, \ldots, k\}$; $k \in \mathbb{N}$.
ensure: Z wird Array mit $\textsc{Länge}[Z] = k$ und $Z[i]$ gleich der Anzahl der Komponenten von A mit dem Wert i.

```
1: Z := Null-Array(k)
2: for j = 1, ..., LÄNGE[A] :
3:     Z[A[j]] := Z[A[j]] + 1
4: end
```

Countingsort: Implementierung

2.25 $Z = \text{HÄUFIGKEITSTABELLE-II}(A, k)$

require: A ist Array mit Elementen aus $\{1, \ldots, k\}$.
ensure: Z wird Array mit $\text{LÄNGE}[Z] = k$ und $Z[i]$ gleich der Anzahl der Komponente von A mit dem Wert $\leq i$.

```
1: Z := HÄUFIGKEITSTABELLE(A, k)
2: for i = 2, ..., k :
3:     Z[i] := Z[i] + Z[i − 1]
4: end
```

2.26 $\text{COUNTING-SORT}(A, B, k)$

require: A ist Array mit Elementen aus $\{1, \ldots, k\}$.
ensure: B ist Sortierung von A.

```
1: Z := HÄUFIGKEITSTABELLE-II(A, k)
2: LÄNGE[B] := LÄNGE[A]
3: for j := LÄNGE[A], ..., 1 mit Schritt −1 :
4:     B[Z[A[j]]] := A[j]
5:     Z[A[j]] := Z[A[j]] − 1
6: end
```

Laufzeit von Countingsort

Theorem 2.28

Die Laufzeit von CountingSort auf Arrays der Länge n mit Elementen aus $\{1, \ldots, k\}$ ist $\Theta(n + k)$. Die Laufzeit ist $\Theta(n)$ für $k = O(n)$.

Beispiel 2.29

Countingsort für $A = [2, 5, 3, 1, 2, 3, 1, 3]$...

Stabilität von Sortieralgorithmen

Definition 2.30

Ein Sortieralgorithmus heißt *stabil*, falls Komponenten mit dem gleichen Wert (oder dem gleichen Schüssel) in dem ausgegebenen sortierten Array in der gleichen Reihenfolge erscheinen wie im Eingabearray.

Bemerkung

Countingsort und Mergesort sind stabile Sortieralgorithmen.

2.27 RadixSort(A, d, k)

require: A ist Array, in welchem jede Komponente eine d-Stellige Zahl im Stellenwertsystem mit der Basis k ist.
ensure: A wird sortiert.

1: **for** $i = 0, \ldots, d-1$ **:**
2: Die Elemente von A nach der i-ten Stelle mit Hilfe von CountingSort sortieren.
3: **end**

Beispiel zu RadixSort

Beispiel 2.31

$[329, 459, 657, 839, 436, 720, 355]$

Richtigkeit von RadixSort

Theorem 2.32

RadixSort arbeitet korrekt.

- Beweis per Induktion nach der Anzahl der Stellen (mit der Verwendung der Stabilität von CountingSort)

Laufziet von RadixSort

Theorem 2.33

Die Laufzeit von RadixSort auf einem n-elementigen Array der d-stelligen Zahlen im Stellenwersystem mit Basis k ist $\Theta(d(n+k))$