Algorithmen und Datenstrukturen

Prof. Martin Lercher

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf
Algorithmen und Datenstrukturen

Teil 2
Sortieren

Version vom: 7. Dezember 2016

21. Oktober 2016

Sortieren

Gegeben: Eine Folge mit $n > 0$ Elementen $a_0, \ldots, a_{n-1}$; jedes Element a_i hat einen Schlüssel $k_i \in \mathbb{N}$.

Aufgabe: Konstruiere eine bijektive Abbildung

$$\pi : \{0, 1, 2, \cdots, n-1\} \to \{0, 1, 2, \cdots, n-1\}$$

mit

$$k_{\pi(0)} \leq k_{\pi(1)} \leq \cdots \leq k_{\pi(n-1)}.$$

"bijektiv": $\Longleftrightarrow$ umkehrbar eindeutig

Diese Problembeschreibung lässt einige Details offen.

Sortieren

Voraussetzungen und erlaubte Operationen für die nachfolgenden Algorithmen:

1. Die Elemente sind in einem Feld (Array) A auf den Positionen 0 bis $n-1$ ($A[0], \ldots, A[n-1]$) gespeichert; $A[i].k$ ist der Schlüssel und $A[i].s$ der Datensatz auf Position i.
2. Es ist erlaubt zwei Schlüssel zu vergleichen:

   $$A[i].k = A[j].k, \quad A[i].k < A[j].k, \quad A[i].k > A[j].k$$

   Das Ergebnis eines Vergleichs ist die einzige Information, die zur Sortierung herangezogen werden kann.
3. Es ist erlaubt zwei Elemente im Feld zu vertauschen.

Sortieren durch Auswahl

Methode: Bestimme die Position j_1 an der das Element $A[j_1]$ mit einem kleinsten Schlüssel unter $A[0].k, \ldots, A[n-1].k$ auftritt und vertausche $A[0]$ mit $A[j_1]$. Dann bestimme die Position j_2 an der das Element $A[j_2]$ mit einem kleinsten Schlüssel unter $A[1].k, \ldots, A[n-1].k$ auftritt und vertausche $A[1]$ mit $A[j_2]$, usw. bis alle Elemente am richtigen Platz stehen.

Algorithmische Umsetzung:

Für den Algorithmus verwenden wir eine Methode

vertausche (Element A [ ], int i, int j),

die das Element an Position i mit dem Element an Position j im Feld A vertauscht.

Sortieren durch Auswahl

```
(1)  Auswahl_Sort (Element A [ ], int n)
(2)  {
(3)     for (int i := 0; i < n-1; i++)
(4)     {
(5)        int min_i := i;
(6)        for (int j := i+1; j < n; j++)
(7)        {
(8)           if (A[j].k < A[min_i].k)
(9)              min_i := j;
(10)       }
(11)       vertausche (A, min_i, i);
(12)    }
(13) }
```

Sortieren durch Auswahl

Beispiel

Schlüsselfolge					
5,	6,	4,	2,	1,	3
1,	6,	4,	2,	5,	3
1,	2,	4,	6,	5,	3
1,	2,	3,	6,	5,	4
1,	2,	3,	4,	5,	6

Sortieren durch Auswahl

Analyse: Die Anzahl der **Vergleiche** ist unabhängig von der anfänglichen Anordnung der Elemente:

$$n-1+n-2+\cdots 1 \;=\; \frac{n(n-1)}{2} \;\in \Theta(n^2)$$

Die Anzahl der **Vertauschungen** ist unabhängig von der Ausgangsanordnung, da nicht überprüft wird, ob $A[min\_i]$ mit sich selbst vertauscht wird.

$$n-1 \;\in \Theta(n).$$

Kann das Verfahren beschleunigt werden, indem das Minimum schneller gefunden wird? Nein!

Sortieren durch Auswahl

Satz

Jeder Algorithmus zur Bestimmung des Minimums von n unterschiedlichen Schlüsseln benötigt mindestens $n-1$ Schlüsselvergleiche.

Beweisidee.

Sei a_1 das Minimum von $a_1, \ldots, a_n$ unterschiedlichen Zahlen.

Jeder Schlüssel $a_i \in \{a_1, \ldots, a_n\}$ muss an mindestens einem Schlüsselvergleich mit einer weiteren Zahl $a_j \in \{a_1, \ldots, a_n\}$ beteiligt sein, die kleiner als a_i ist. □

Bemerkung: Beweis für untere Schranken sind oft technisch sehr aufwendig, da alle Voraussetzungen und Randbedingungen für alle denkbaren Verfahren vorher genau präzisiert werden müssen.

Hier: nur das Ergebnis eines Vergleichs darf genutzt werden.

Sortieren durch Einfügen

Methode: Die n zu sortierenden Elemente werden nacheinander betrachtet und in die jeweils bereits sortierte, anfangs leere Teilfolge bis zur richtigen Stelle verschoben.

Angenommen wir wollen das i-te Element auf Position $i-1$ in eine bereits sortierte Folge

$$A[0], \ldots, A[i-2]$$

einfügen.

Sei r, $0 \leq r \leq i-2$, die größte Position mit $A[r].k \leq A[i-1].k$. Dann vertauschen wir $A[j]$ mit $A[j+1]$ für $j = i-2, \ldots, r+1$.

Sortieren durch Einfügen

Algorithmische Umsetzung:

```
(1)  Einfüge_Sort (Element A [ ], int n)
(2)  {
(3)     for (int i := 1; i < n; i++) // A[0] ist das 1. Element!
(4)     {
(5)        int j := i;
(6)        while ((j > 0) and (A[j].k < A[j-1].k))
(7)        {
(8)           vertausche (A, j, j-1);
(9)           j−−;
(10)       }
(11)    }
(12) }
```

Sortieren durch Einfügen

Beispiel

Runde	Schlüsselfolge
	$\underline{1}, \quad 7, \quad 3, \quad 2, \quad 4$
	$\underline{1}, \quad 7, \quad 3, \quad 2, \quad 4$
1	$1, \quad \underline{7}, \quad 3, \quad 2, \quad 4$
	$1, \quad \underline{7}, \quad 3, \quad 2, \quad 4$
2	$1, \quad 7, \quad \underline{3}, \quad 2, \quad 4$
	$1, \quad \underline{3}, \quad 7, \quad 2, \quad 4$
3	$1, \quad 3, \quad 7, \quad \underline{2}, \quad 4$
	$1, \quad 3, \quad \underline{2}, \quad 7, \quad 4$
	$1, \quad \underline{2}, \quad 3, \quad 7, \quad 4$
4	$1, \quad 2, \quad 3, \quad 7, \quad \underline{4}$
	$1, \quad 2, \quad 3, \quad \underline{4}, \quad 7$

Sortieren durch Einfügen

Analyse: Im ungünstigsten Fall werden für das Einfügen des i-ten Elementes mit $i = 1, \ldots, n-1$ höchstens $i-1$ Schlüsselvergleiche und $i-1$ Vertauschungen durchgeführt. Daraus folgt, dass insgesamt höchstens

$$1 + 2 + \cdots + n - 1 \;=\; \frac{n(n-1)}{2} \;\in \Theta(n^2)$$

Vergleiche und Vertauschungen durchgeführt werden.

Im günstigsten Fall ist die Folge bereits sortiert. Dann werden $\Theta(n)$ viele Vergleiche und keine Vertauschungen durchgeführt.

Sortieren durch Einfügen

Der Aufwand ist abhängig von der Anzahl der Fehlstellungen (Inversionszahl) in der ursprünglichen Anordnung der Elemente.

Im Mittel kann man erwarten, dass die Hälfte der dem i-ten Element vorangehenden Elemente größer als $A[i-1].k$ sind. Die mittlere Anzahl der Vergleiche und Vertauschungen ist somit

$$\sum_{i=1}^{n} \frac{i-1}{2} = \frac{n(n-1)}{4} \in \Theta(n^2).$$

Bubblesort

Methode: Durchlaufe das Feld $A[0], \ldots, A[n-1]$ von 0 bis $n-2$ und betrachte je zwei benachbarte Elemente $A[i], A[i+1]$, $0 \leq i < n-1$.

Ist $A[i].k > A[i+1].k$, so vertauscht man $A[i]$ und $A[i+1]$.

Dieses Durchlaufen wird solange wiederholt, bis keine Vertauschungen mehr aufgetreten sind. Damit ist das Feld a nach aufsteigenden Schlüsseln sortiert.

Bubblesort

Algorithmische Umsetzung:

```
(1)  Bubble_Sort (Element A [ ], int n)
(2)  {
(3)     bool vertauscht;
(4)     do
(5)     {
(6)        vertauscht := false;
(7)        for (int i := 0; i < n-1; i++)
(8)        {
(9)           if (A[i].k > A[i+1].k)
(10)          {
(11)             vertausche (A, i, i+1);
(12)             vertauscht := true;
(13)          }
(14)       }
(15)    }
(16)    while (vertauscht);
(17) }
```

Bubblesort

Beispiel

	Schlüsselfolge				
Vergleich	$\underline{1}$,	$\underline{7}$,	4,	2,	3
Vergleich	1,	$\underline{7}$,	$\underline{4}$,	2,	3
	1,	$\underline{4}$,	$\underline{7}$,	2,	3
Vergleich	1,	4,	$\underline{2}$,	$\underline{7}$,	3
Vergleich	1,	4,	2,	$\underline{7}$,	$\underline{3}$
	1,	4,	2,	$\underline{3}$,	$\underline{7}$
Vergleich	$\underline{1}$,	$\underline{2}$,	4,	3,	7
Vergleich	1,	$\underline{2}$,	$\underline{4}$,	3,	7
Vergleich	1,	2,	$\underline{4}$,	$\underline{3}$,	7
	1,	2,	$\underline{3}$,	$\underline{4}$,	7
Vergleich	1,	2,	3,	$\underline{4}$,	$\underline{7}$

Bubblesort

Analyse: Spätestens nach dem ersten Durchlauf ist das größte Element an seinem richtigen Platz. Spätestens nach dem i-ten Durchlauf ist das i-t-größte Element auf seinem richtigen Platz. Spätestens nach dem $(n-1)$-ten Durchlauf ist das Feld a sortiert.

Im günstigsten Fall, wenn das Feld bereits sortiert ist, werden $n-1$ Vergleiche und keine Vertauschungen vorgenommen. Im ungünstigsten Fall, wenn das Feld absteigend sortiert ist, sind $n-1$ Durchläufe nötig. In diesem Fall werden beim i-ten Durchlauf $n-1$ Vergleiche und $n-i$ Vertauschungen durchgeführt. Somit sind im ungünstigsten Fall die Anzahl der Vergleiche und Vertauschungen aus $\Theta(n^2)$.

Man kann zeigen, dass die Anzahl der Vergleiche und Vertauschungen auch im Mittel aus $\Theta(n^2)$ sind.

Bemerkung: Bubblesort ist zwar populär, aber im Grunde schlecht.

Wird das Feld abwechselnd von links nach rechts und umgekehrt durchlaufen, erhält man eine Variante namens Shakersort (schöner Name, aber keine weiteren Vorzüge).

25. Oktober 2016

Quicksort

Quicksort wurde 1962 von C.A.R. Hoare veröffentlicht. Es ist eines der schnellsten internen Sortierverfahren. Es benötigt im schlechtesten Fall $\Theta(n^2)$ viele Vergleichsoperationen, aber im Mittel nur $\Theta(n \cdot \log(n))$ viele Vergleiche.

Quicksort ist wie die anderen bereits diskutierten Verfahren ein in-situ-Sortierverfahren. Es wird kein zusätzlicher Speicherplatz zur Speicherung der Datensätze benötigt außer einer konstanten Anzahl von Hilfsspeicherplätzen für Tauschoperationen und die auf dem Stack gespeicherten Zwischenwerte bei jedem rekursiven Aufruf.

Quicksort

Methode: Um eine Folge $F = A[0], \ldots, A[n-1]$ aufsteigend zu sortieren, wird ein beliebiges Element

$$p = A[i], i \in \{0, \ldots, n-1\},$$

zur Aufteilung der Folge F in zwei Teilfolgen F_1 und F_2 gewählt. Dieses Element p nennen wir Pivotelement. Folge F_1 besteht nur aus Elementen von F, dessen Schlüssel kleiner oder gleich $p.k$ sind, F_2 nur aus Elementen von F, dessen Schlüssel größer oder gleich $p.k$ sind. F_1 und F_2 (falls sie mehr als ein Element enthalten) werden anschließend rekursiv mit Quicksort sortiert.

Wesentlich ist der Aufteilungsschritt, d.h., die Wahl des Pivotelementes. Wähle zum Beispiel als Pivotelement das letzte Element der Folge.

Die folgende Implementierung verändert das Feld A so, dass die Elemente in der Reihenfolge F_1, p, F_2 stehen.

Quicksort

Algorithmische Umsetzung:

```
(1)  Quick_Sort (Element A [ ], int l, int r) // sortiere Teilfolge A[l]...A[r]
(2)  {
(3)     if (l < r) { // >1 Element
(4)        int p := A[r].k; // Pivot-Element
(5)        int i := l;
(6)        int j := r-1;
(7)        do
(8)        {
(9)           while ((i ≤ j) and (A[i].k ≤ p)) i++; // 1. El.>p von links
(10)          while ((i ≤ j) and (A[j].k ≥ p)) j−−; // 1. El.<p von rechts
(11)          if (i < j) // sonst fertig
(12)             vertausche (A, i, j);
(13)       }
(14)       while (i < j);
(15)       vertausche (A, i, r); // Pivot-Element in die Mitte
(16)       Quick_Sort (A, l, i-1); // rekursiv linke Teilfolge
(17)       Quick_Sort (A, i+1, r); // rekursiv rechte Teilfolge
(19) } }
```

Quicksort

Beispiel

						Vergleich	Ergebnis
l ↓					r ↓		
8 ↑ i	2	7	3	6 ↑ j	5	$8 \leq 5$	**false**
l ↓					r ↓		
8 ↑ i	2	7	3	6 ↑ j	5	$6 \geq 5$	**true**

Quicksort

Beispiel

						Vergleich	Ergebnis
l					r		
↓					↓		
8	2	7	3	6	5	$3 \geq 5$	**false**
↑			↑				
i			j				
l					r		
↓					↓		
3	2	7	8	6	5	$3 \leq 5$	**true**
↑			↑				
i			j				

Quicksort

Beispiel

						Vergleich	Ergebnis
l					r		
↓					↓		
3	2	7	8	6	5	$2 \leq 5$	**true**
	↑		↑				
	i		j				
l					r		
↓					↓		
3	2	7	8	6	5	$7 \leq 5$	**false**
		↑	↑				
		i	j				

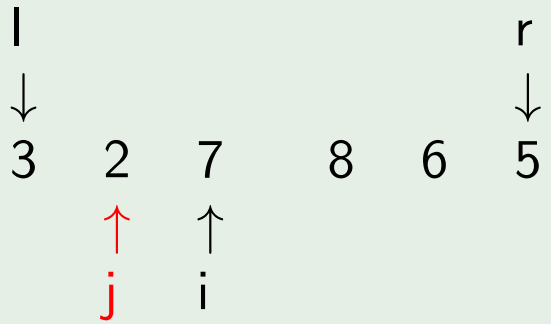
Quicksort

Beispiel

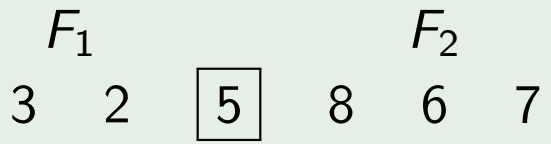
Quicksort

Beispiel

Vergleich Ergebnis



vertausche(A, i, r)



Quicksort

Die beiden Teilfolgen F_1 und F_2 werden anschließend rekursiv mit Quicksort sortiert.

Analyse: Bei der Aufteilung von F in F_1, p, F_2 werden höchstens $\Theta(n)$ Vergleiche mit p durchgeführt. Möglicherweise wechselt dabei jedes Element einmal seinen Platz. Die im ungünstigsten Fall insgesamt auszuführende Anzahl von Vergleichen ist stark von der Anzahl der Aufteilungsschritte abhängig. Ist das Pivotelement das Element mit kleinstem oder größtem Schlüssel, so ist die eine Folge leer und die andere hat nur ein Element (das Pivotelement) weniger als die Ausgangsfolge. Dieser Fall tritt zum Beispiel bei einer bereits aufsteigend sortierten Folge auf. Quicksort benötigt somit im ungünstigsten Fall $\Theta(n^2)$ viele Vergleiche.

Quicksort

Im günstigsten Fall wird die Folge jedes Mal etwa halbiert. Dadurch ist die Anzahl der Vergleiche bei einer Sortierung von n Elementen

$$C_n \leq (n+1) + C_{\lfloor (n-1)/2 \rfloor} + C_{\lceil (n-1)/2 \rceil}.$$

Die Lösung dieser Rekurrenz mit $C_1 = C_0 = 0$ ergibt $O(n \cdot \log(n))$ Vergleiche.

Obwohl die Situation nicht immer so günstig ist, trifft es jedoch zu, dass die Zerlegung im Durchschnitt auf die Mitte fällt.

Die Berücksichtigung der exakten Wahrscheinlichkeit jeder Position der Zerlegung macht die rekurrente Beziehung komplizierter und schwerer auflösbar, doch das Endergebnis ist ähnlich.

Master-Theorem (Hauptsatz der Laufzeitfunktionen)

Satz

Seien $a, b, k \in \mathbb{N}$, mit $a \geq 1$, $b \geq 2$ und $k \geq 0$.
Sei die Laufzeit eines Algorithmus durch folgende Rekurrenz-Relation beschrieben:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

mit

$$f(n) \in \Theta(n^k).$$

Dann gilt:

$$\begin{array}{lll} T(n) \in \Theta(n^k) & \text{wenn} & a < b^k \\ T(n) \in \Theta(n^k log(n)) & \text{wenn} & a = b^k \\ T(n) \in \Theta(n^{log_b(a)}) & \text{wenn} & a > b^k \end{array}$$

Gilt genauso auch für $\mathcal{O}$ (mit $T(n) \leq ...$) und Ω (mit $T(n) \geq ...$).

Beweis.

...mit Analysis, siehe z.B. Jonsonbaugh and Schaefer S. 60-65...

Quicksort: Laufzeit im Mittel

Satz

Quicksort benötigt im Mittel $O(n \cdot \log(n))$ Vergleiche.

Beweis.

Angenommen die Wahrscheinlichkeit, das an Position i, $0 \leq i \leq n-1$, zerlegt wird, sei $1/n$. Alle n Positionen sollen gleich wahrscheinlich sein. (Wichtige Voraussetzung!) Um die Rechnung etwas zu vereinfachen, setzen wir voraus, das die Aufteilung der Folge in zwei Teilfolgen genau $n+1$ Vergleiche benötigt (es könnten auch n sein).

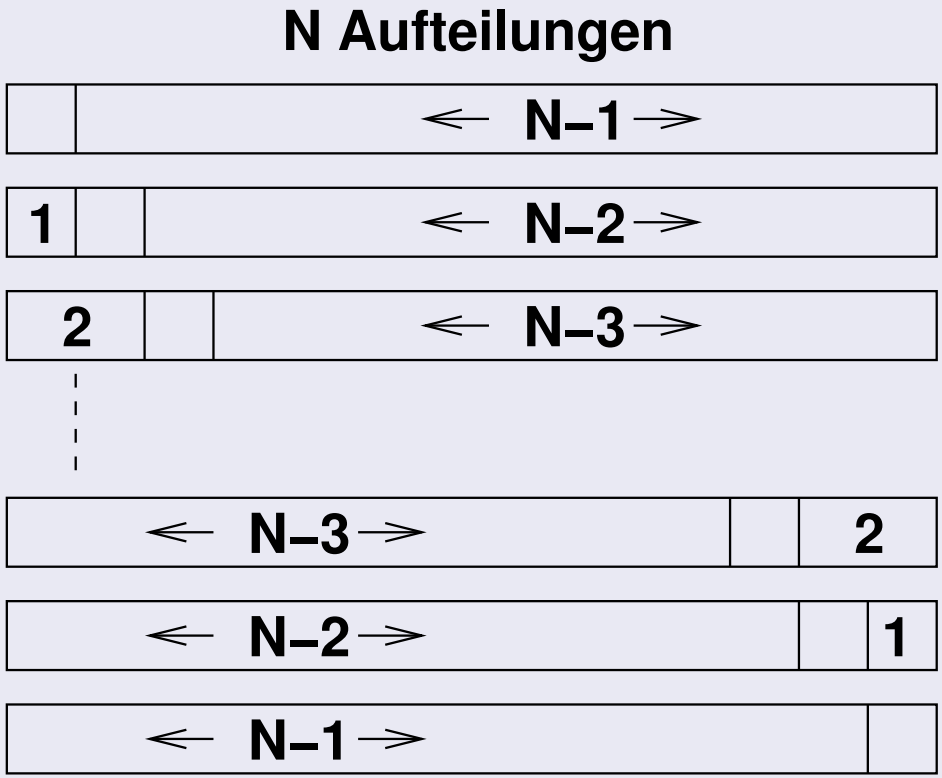
Dann lautet die rekurrente Beziehung für die Anzahl der Vergleiche im Mittel

$$C_n = n+1 + \frac{1}{n}\sum_{i=0}^{n-1}(C_i + C_{n-1-i})$$

für $n \geq 2$ und $C_1 = C_0 = 0$.

Quicksort: Laufzeit im Mittel

Beweis Fortsetzung.



Die Aufteilungen werden als gleichwahrscheinlich angenommen.

Quicksort: Laufzeit im Mittel

Beweis Fortsetzung.

Da $C_0 + C_1 + \cdots + C_{n-2} + C_{n-1} = C_{n-1} + C_{n-2} + \cdots + C_1 + C_0$, *kann die Summe wie folgt umgeschrieben werden:*

$$C_n = n + 1 + \frac{2}{n} \cdot \sum_{i=0}^{n-1} C_i$$

Multiplikation mit $n \geq 2$ *ergibt*

$$n \cdot C_n = n \cdot (n+1) + 2 \cdot \sum_{i=0}^{n-1} C_i$$

Subtraktion der gleichen Gleichung für $n-1$ *ergibt*

$$n \cdot C_n - (n-1) \cdot C_{n-1} = n \cdot (n+1) + 2 \cdot \sum_{i=0}^{n-1} C_i - \left[(n-1) \cdot n + 2 \cdot \sum_{i=0}^{n-2} C_i \right]$$

Quicksort: Laufzeit im Mittel

Beweis Fortsetzung.

vereinfacht:

$$n \cdot C_n \; - \; (n-1) \cdot C_{n-1} \; = \; 2n \; + \; 2 \cdot C_{n-1}$$

Durch Addition von $(n-1) \cdot C_{n-1}$ *erhalten wir*

$$n \cdot C_n \; = \; 2n \; + \; (n+1) \cdot C_{n-1}$$

Division beider Seiten durch $n \cdot (n+1)$ *ergibt eine rekurrente Beziehung, die sich wie folgt fortsetzen lässt:*

$$\begin{aligned} \frac{C_n}{n+1} &= \frac{C_{n-1}}{n} + \frac{2}{n+1} \\ &= \frac{C_{n-2}}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &= \frac{C_{n-3}}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &\vdots \end{aligned}$$

Quicksort: Laufzeit im Mittel

Beweis Fortsetzung.

$$
\begin{aligned}
&= \frac{C_{n-3}}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&= \frac{C_{n-4}}{n-3} + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\vdots \\
&= \frac{C_1}{2} + \frac{2}{3} + \ldots + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}
\end{aligned}
$$

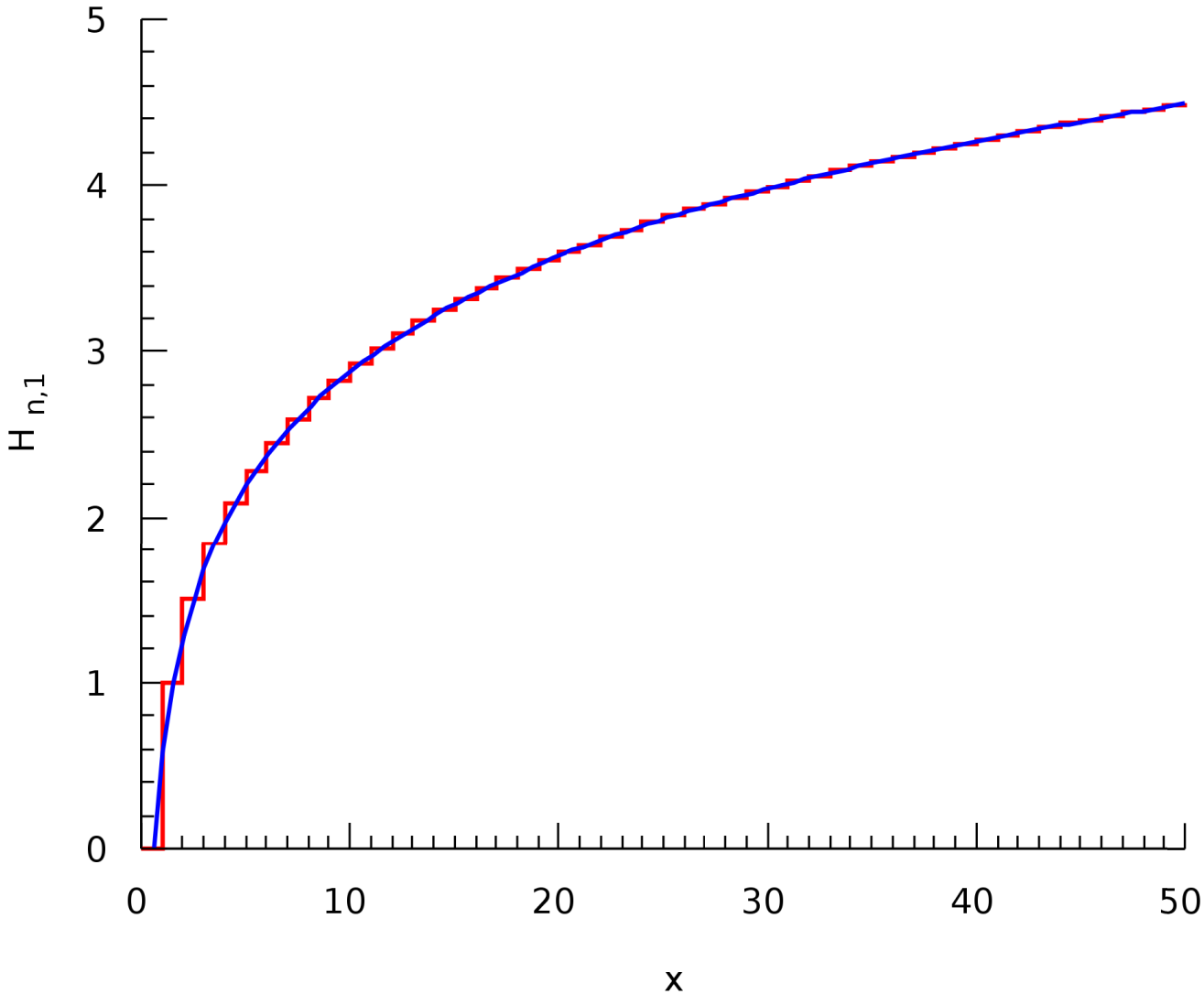
Durch Umformung und $C_1 = 0$ erhalten wir

$$
\begin{aligned}
\frac{C_n}{n+1} &= \frac{2}{n+1} - \frac{2}{1} - \frac{2}{2} + \frac{2}{1} + \frac{2}{2} + \frac{2}{3} + \ldots + \frac{2}{n-1} + \frac{2}{n} \\
&= \frac{2}{n+1} - 3 + 2 \cdot \sum_{p=1}^{n} \frac{1}{p} \\
&\approx \frac{1}{n+1} - 3 + 2 \cdot \int_1^n \frac{1}{x} dx \\
&= \frac{1}{n+1} - 3 + 2 \ln n
\end{aligned}
$$

Daraus folgt $C_n \in \Theta(n \cdot \log(n))$. □

Quicksort: Laufzeit im Mittel

Näherung der harmonischen Summe $\sum_{p=1}^{n} \frac{1}{p}$ durch $\int_1^n \frac{1}{x} dx$:



Quicksort mit beschränkter Rekursionstiefe

Im ungünstigsten Fall ist die Rekursionstiefe $\Theta(n)$.

Der zusätzlich benötigten Platz für die Rekursionsaufrufe kann gering gehalten werden, wenn das kleinere Teilproblem rekursiv und das größere Teilproblem iterativ direkt gelöst wird.

Dadurch verringert sich die Rekursionstiefe auf $O(\log(n))$.

Algorithmische Umsetzung:

```
(1)  Quick_Sort2 (Element A [ ], int l, int r)
(2)  {
(3)     while (l < r) {
(4)        int p := A[r].k;
(5)        int i := l;
(6)        int j := r-1;
(7)        do
(8)        {
(9)           while ((i < j) and (A[i].k ≤ p)) i++;
(10)          while ((i < j) and (A[j].k ≥ p)) j−−;
```

Quicksort mit beschränkter Rekursionstiefe

```
(11)          if (i < j)
(13)             vertausche (A, i, j);
(14)       }
(15)       while (i < j);
(16)       vertausche (A, i, r);
(17)       if ((i-l) < (r-i))
(18)       {
(19)          Quick_Sort2 (A, l, i-1); // rekursive Lösung
(20)          l := i+1; // iterative Loesung
(21)       }
(22)       else
(23)       {
(24)          Quick_Sort2 (A, i+1, r); // rekursive Lösung
(25)          r := i-1; // iterative Lösung
(26)       }
(27)    }
(28) }
```

Quicksort: Verbesserte Pivotwahl

Die 3-Median-Strategie: Wähle als Pivotelement das mittlere Element von drei Elementen (z.B. von $A[l].k$, $A[r].k$ und $A[\lfloor \frac{r+l}{2} \rfloor].k$, indem es mit $A[r]$ vertauscht wird.

Die Zufallsstrategie (randomisiertes Quicksort): Wähle als Pivotelement ein zufälliges Element aus $A[l].k, \ldots, A[r].k$ und vertausche es mit $A[r]$.

Der Erwartungswert für die zum Sortieren einer beliebigen aber festen Eingabefolge bei randomisierten Quicksort ist aus $O(n \cdot \log(n))$.

Man nennt ein Sortierverfahren glatt (smooth), wenn es im Mittel n verschiedene Schlüssel in $O(n \cdot \log(n))$ und n gleiche Schlüssel in $O(n)$ Schritten zu sortieren vermag mit einem weichen Übergang zwischen diesen Werten.

Quicksort: Smooth

Beispiel

Teile die Folge $A[l], \ldots, A[r]$ in drei Folgen F_l, F_m, F_r auf.

1. F_l enthält die Elemente mit Schlüssel $< p$.
2. F_m enthält die Elemente mit Schlüssel $= p$.
3. F_r enthält die Elemente mit Schlüssel $> p$.

Nach der Aufteilung werden F_l und F_r auf dieselbe Weise sortiert.

Gibt es keine zwei gleiche Schlüssel, so bedeutet dies keine Ersparnis.

Sind alle Schlüssel identisch, erfolgt kein rekursiver Aufruf.

Das oben skizzierte, auf Drei-Wege-Split beruhende Quicksort benötigt im Mittel $O(n \cdot \log(n') + n)$ Schritte, wobei n' die Anzahl der verschiedenen Schlüssel unter den n Schlüsseln der Eingabefolge ist.

Quicksort: Smooth

Algorithmische Umsetzung

```
(1)  Quick_Sort3 (Element A [ ], int l, int r)
(2)  {
(3)     if (l < r)
(4)     {
(5)        int p := A[r].k;
(6)        int i := l, x := l;
(7)        int j := r-1, y := r-1;
(8)        do
(9)        {
(10)          while ((i < j) and (A[i].k ≤ p)) i++;
(11)          while ((i < j) and (A[j].k ≥ p)) j−−;
(12)          if (i < j) {
(13)             if ((A[i].k > p) and (A[j].k < p))
(14)                vertausche (A, i, j);
(15)             else if ((A[i].k > p) and (A[j].k = p))
(16)             {
(17)                vertausche (A, i, j); // hänge A[j] links an
```

Quicksort: Smooth

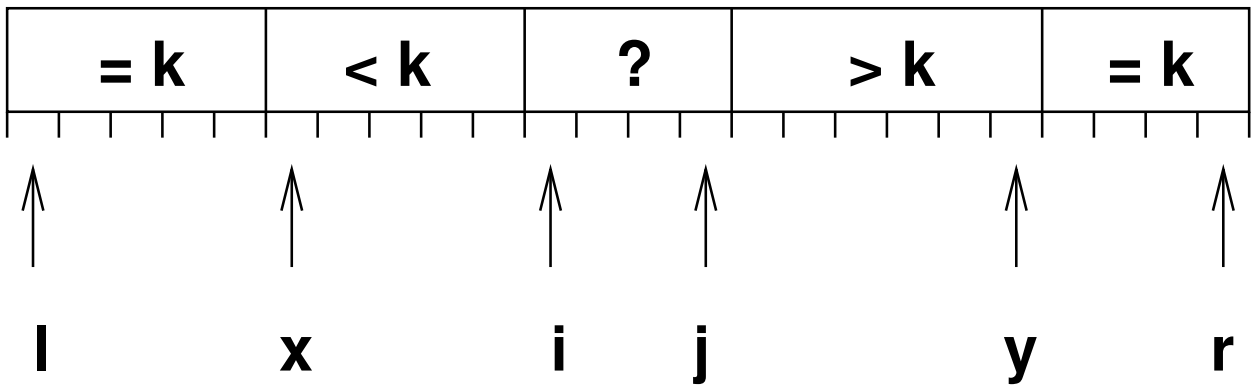
```
(18)              vertausche (A, i, x);
(19)              x++;
(20)           }
(21)           else if ((A[i].k = p) and (A[j].k < p))
(22)           {
(23)              vertausche (A, i, j); // hänge A[i] rechts an
(24)              vertausche (A, j, y);
(25)              y−−;
(26)           }
(27)           else if ((A[i].k = p) and (A[j].k = p))
(28)           {
(29)              vertausche (A, i, x); // hänge A[i] links an
(30)              x++;
(31)              vertausche (A, j, y); // hänge A[j] rechts an
(32)              y−−;
(33)           }
(34)        }
(35)     }
(36)     while (i < j); // falls i ≥ j, so ist i die Pivotposition
```

Quicksort: Smooth

```
(37)       while (x > l) {
(38)         i−−; // schiebe alle Pivotelemente von
(39)         x−−; // links in die Mitte
(40)         vertausche (A, i, x);
(41)       }
(42)       while (y < r) {
(43)         j++; // schiebe alle Pivotelemente von
(44)         y++ // rechts in die Mitte
(45)         vertausche (A, j, y);
(46)       }
(47)       Quick_Sort3 (A, l, i-1);
(48)       Quick_Sort3 (A, j+1, r);
(49)     }
(50) }
```

Quicksort: Smooth

Die Pivotelemente werden zuerst am Rand gesammelt (zwischen l und x sowie zwischen y und r-1) und vor dem rekursiven Aufruf in die Mitte transportiert.



Frage: Gibt es auch Sortierverfahren, die im ungünstigsten Fall, also im worst case, mit $O(n \cdot \log(n))$ Vergleichen auskommen? Ja!

28. Oktober 2016

Heapsort

Sortieren mit einer Halde durch geschickte Auswahl.

Methode: Es wird eine Datenstruktur (Heap, Halde) eingesetzt, mit der das Maximum aus einer Menge von n Schlüsseln in einem Schritt bestimmt werden kann.

Definition

Eine Folge $F = A[0], \ldots, A[n-1]$ von n Elementen nennen wir einen Heap, falls $A[i].k \geq A[2 \cdot i + 1].k$ und $A[i].k \geq A[2 \cdot i + 2].k$ für $i \geq 0$ und $2 \cdot i + 1 \leq n - 1$ bzw. $2 \cdot i + 2 \leq n - 1$.

Die Elemente $A[2 \cdot i + 1]$ und $A[2 \cdot i + 2]$ sind die Nachfolger von $A[i]$.

Beispiel

$F = A[0], \ldots, A[7]$ mit $A[0].k = 8$, $A[1].k = 6$, $A[2].k = 7$, $A[3].k = 3$, $A[4].k = 4$, $A[5].k = 5$, $A[6].k = 2$, $A[7].k = 1$ ist ein Heap, weil $8 \geq 6$, $8 \geq 7$, $6 \geq 3$, $6 \geq 4$, $7 \geq 5$, $7 \geq 2$ und $3 \geq 1$.

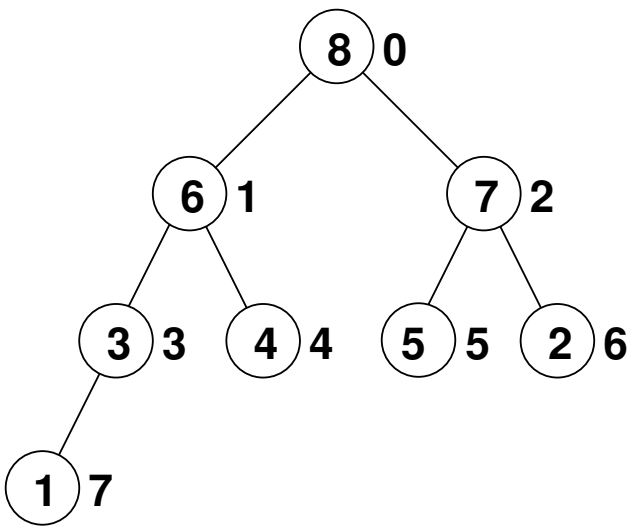
Heapsort

Folgendes Schaubild erleichtert die Überprüfung der Heapeigenschaft. In den Kreisen stehen die Schlüssel, neben den Kreisen die Positionen in F. Wir gehen jedoch weiterhin davon aus, dass die Elemente in einem Array gespeichert sind.

Felddarstellung:

8	6	7	3	4	5	2	1

Baumdarstellung:



Heapsort

Das Element mit größtem Schlüssel ist $A[0]$.

Das nächst kleinere Element in einem Heap wird bestimmt, indem $A[0]$ aus F entfernt wird, und die Restfolge wieder zu einem Heap transformiert wird.

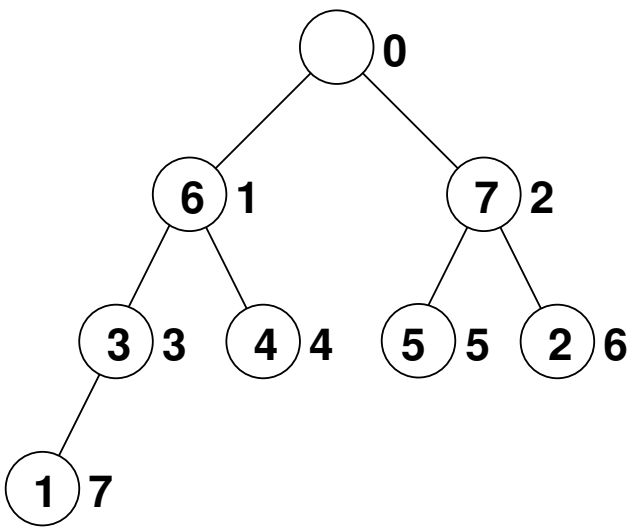
Bemerkung: nach dem Entfernen des Elementes mit größtem Schlüssel bleiben zwei Teilheaps zurück.

In unserem Beispiel: $F_1 = A[1], A[3], A[4], A[7]$ und $F_2 = A[2], A[5], A[6]$

-	6	-	3	4	-	-	1

und

-	-	7	-	-	5	2	-



Heapsort

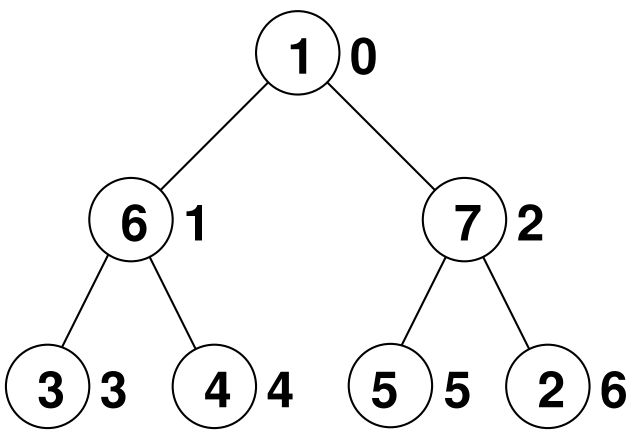
Wir machen daraus wieder einen Heap indem wir zuerst das letzte Element, das Element mit größtem Index, an die erste Position setzen. Dadurch wird jedoch die Heapeigenschaft verletzt.

Trick: Vertausche das erste Element mit dem letzten Element im aktuellen Heap. Das letzte Element gehört anschließend nicht mehr zum neuen Heap, ist aber weiterhin im Feld vorhanden.

Felddarstellung:

1	6	7	3	4	5	2	8

Baumdarstellung:



Heapsort

Dann lassen wir das neue Element versickern, indem wir es immer wieder mit dem Größeren seiner beiden Nachfolger vertauschen, bis die Schlüssel beider Nachfolger kleiner sind oder das zu versickernde Element unten (an einem Blatt) angekommen ist.

Felddarstellung:

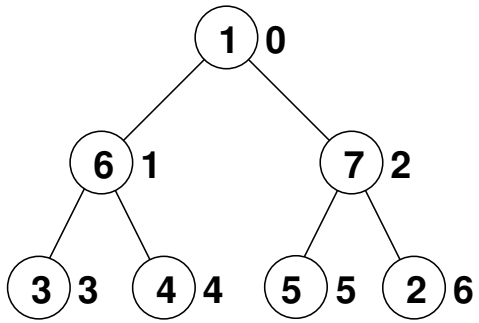
1	6	7	3	4	5	2	8

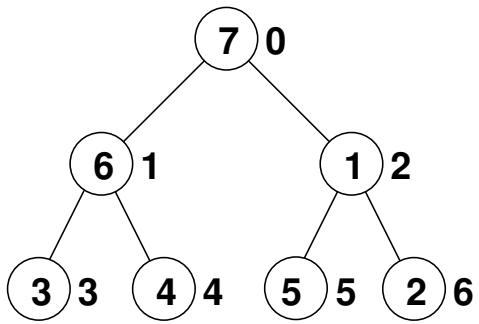
7	6	1	3	4	5	2	8

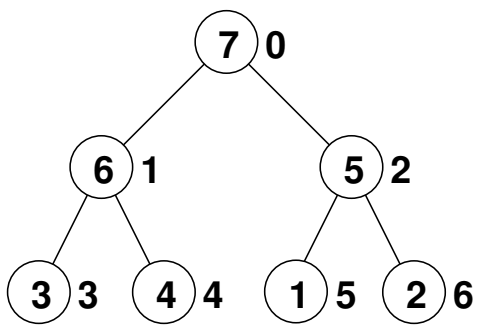
7	6	5	3	4	1	2	8

Heapsort

Baumdarstellung:







Heapsort

Algorithmische Umsetzung:

In diesem Algorithmus wird von A[i] bis höchstens A[m] versickert:

```
(1)  versickere (Element A [ ], int i, int m)
(2)  {
(3)     while (2*i+1 ≤ m) {
(4)        int j := 2*i+1;
(5)        if ((2*i+2 ≤ m) and (A[2*i+1].k < A[2*i+2].k))
(6)           j := 2*i+2;
(7)
(8)        if (A[i].k < A[j].k) {
(9)           vertausche (A, i, j);
(10)          i := j;
(11)       }
(12)       else
(13)          i := m; // break
(14)    }
(15) }
```

Heapsort

Analyse: Es erfolgen $n-1$ viele Vertauschungen außerhalb der Methode versickere( ). Beim Versickern wird ein Element wiederholt mit einem seiner Nachfolger vertauscht. Das Element wandert nach jedem Vertauschen eine Stufe tiefer, wobei sich die Anzahl der Elemente auf jeder Stufe verdoppelt.

Ein Heap mit n Elementen hat $\lceil \log(n+1) \rceil$ viele Stufen. Somit werden beim Versickern höchstens $\lceil \log(n+1) \rceil - 1$ viel Vertauschungen durchgeführt. Dadurch ergibt sich eine obere Schranke von $O(n \cdot \log(n))$ vielen Vergleichen und Vertauschungen.

Die Ausgangsfolge muss jedoch zuerst in einen Heap transformiert werden.

Idee: Wir lassen die Elemente $A[\lfloor \frac{n}{2} \rfloor - 1]$ bis $A[0]$ versickern. Man beginnt also dort zu versickern, wo Elemente als Nachfolger Blätter besitzen. Dadurch werden schrittweise immer größere Heaps aufgebaut, bis letztlich ein einziger Heap übrig bleibt.

Heapsort

Algorithmische Umsetzung:

```
(1) Heap_Sort (Element A [ ], int n)
(2) {
(3) // Heap herstellen
(4)     for (int i := (n/2)-1; i ≥ 0; i−−)
(5)         versickere (A, i, n-1);
(6) // Sortieren
(7)     for (int i := n-1; i > 0; i−−) {
(8)         vertausche (A, i, 0);
(9)         versickere (A, 0, i-1);
(10)    }
(11) }
```

Heapsort

Analyse: Sei j die Anzahl der Stufen des Heaps mit n Schlüsseln

$$2^{j-1} \leq n \leq 2^j - 1$$

Auf Stufe k, $1 \leq k \leq j$, gibt es höchstens 2^{k-1} Datensätze.

Die Anzahl der Bewegungen und Vergleiche zum Versickern eines Elementes der Stufe k ist proportional zu $j - k$.

Daraus ergibt sich für die Anzahl der Operationen zum Umwandeln einer unsortierten Folge in einen Heap:

$$\sum_{k=1}^{j-1} 2^{k-1}(j-k) \;=\; \sum_{k=1}^{j-1} 2^{j-k-1} \cdot k \;=\; 2^{j-1} \sum_{k=1}^{j-1} \frac{k}{2^k} \;\leq\; n \sum_{k=1}^{j-1} \frac{k}{2^k} \;<\; n \cdot 2 \in O(n)$$

(am 1. Gleichheitszeichen haben wir k durch $j - k$ ersetzt)

Heapsort – Einschub: Vollständige Induktion

Satz

Es gilt $\sum_{k=1}^{n} \frac{k}{2^k} = 2 - \frac{n+2}{2^n} < 2$.

Beweis.

Wir beweisen die Behauptung in zwei Schritten.

- *Zeige* $\sum_{k=1}^{n} \frac{k}{2^k} = 2 - \frac{n+2}{2^n}$ *per Induktion.*
- *Zeige* $2 - \frac{n+2}{2^n} < 2$.

Induktionsanfang (für $n = 1$*):*

$$\sum_{k=1}^{1} \frac{k}{2^k} = \frac{1}{2^1} = \frac{1}{2} = 2 - \frac{3}{2} = 2 - \frac{1+2}{2^1}$$

Die Behauptung gilt für $n = 1$.

Beweis Fortsetzung.

Induktionsschritt (für $n-1 \to n$):

Sei die Behauptung für alle natürlichen Zahlen $< n$ gezeigt.

$$\begin{aligned}
\sum_{k=1}^{n} \frac{k}{2^k} &= \sum_{k=1}^{n-1} \frac{k}{2^k} + \frac{n}{2^n} \\
&\overset{IV}{=} \left(2 - \frac{(n-1)+2}{2^{n-1}}\right) + \frac{n}{2^n} \\
&= 2 - \frac{(n+1)\cdot 2}{2^{n-1}\cdot 2} + \frac{n}{2^n} \\
&= 2 - \frac{2n+2-n}{2^n} \\
&= 2 - \frac{n+2}{2^n}
\end{aligned}$$

Heapsort – Einschub: Vollständige Induktion

Beweis Fortsetzung.

Im ersten Schritt haben wir gezeigt, dass $\sum_{k=1}^{n} \frac{k}{2^k} = 2 - \frac{n+2}{2^n}$ *gilt.*

Es verbleibt zu zeigen, dass $2 - \frac{n+2}{2^n} < 2$ *ebenfalls gilt.*

Für $n \geq 0$ *(was bei uns immer der Fall ist) gilt* $z = \frac{n+2}{2^n} > 0$*. Demnach ist* $2 - z < 2$. □

Heapsort

Alternativ kann man auch direkt durch vollständige Induktion zeigen, dass

$$\sum_{k=1}^{j-1} 2^{k-1}(j-k) \;=\; 2^j - (j+1)$$

gilt, und somit (wegen $2^{j-1} = n$):

$$\sum_{k=1}^{j-1} 2^{k-1}(j-k) \;\leq\; 2 \cdot n - (j+1) \;\in O(n).$$

Da der Aufbau eines Heaps aus einer unsortierten Folge in linearer Zeit möglich ist, ist die gesamte Anzahl der Vergleiche und Vertauschungen für Heapsort ebenfalls aus $O(n \cdot \log(n))$.

Bemerkungen:

Eine Vorsortierung der Eingabefolge schadet und nützt der Sortierung nichts.

Heapsort benötigt ebenfalls nur konstant viel zusätzlichen Speicherplatz (in-situ-Sortierverfahren).

Mergesort

Sortieren durch Vereinigen (häufig auch irreführend "durch Mischen").

Mergesort ist eines der ältesten (John von Neumann 1954) Sortierverfahren. Es arbeitet wie Quicksort mit „Teile und herrsche". Die Folge wird jedoch in einem Schritt in zwei gleich große Teilfolgen aufgeteilt. Die rekursiv sortierten Teilfolgen werden dann in linearer Zeit wieder vereinigt.

2-Wege-Mergesort:

Methode: Eine Folge $F = A[0], \ldots, A[n-1]$ von n Elementen wird sortiert, indem sie zunächst in zwei möglichst gleich große Teilfolgen $F_1 = A[0], \ldots, A[\lceil \frac{n}{2} \rceil - 1]$ und $F_2 = A[\lceil \frac{n}{2} \rceil], \ldots, A[n-1]$ aufgeteilt wird.

Dann wird jede dieser Teilfolgen rekursiv mit Mergesort sortiert.

Die Ergebnisse werden vereinigt, indem fortlaufend die Elemente mit kleinstem Schlüssel aus den sortierten Folgen entfernt und in die Ergebnisfolge eingefügt wird. Das Element mit kleinstem Schlüssel ist immer entweder das erste Element der sortierten Folge F_1 oder das erste Element der sortierten Folge F_2.

Mergesort

Das Vereinigen lässt sich im Feld einfach mit Hilfe von zwei Positionsvariablen realisieren.

	F_1						F_2						Ergebnisfolge
	1,	2,	3,	5,	9		4,	6,	7,	8,	10		
	$\uparrow i$						$\uparrow j$						
$1 < 4$		$\uparrow i$					$\uparrow j$						1
$2 < 4$			$\uparrow i$				$\uparrow j$						1,2
$3 < 4$				$\uparrow i$			$\uparrow j$						1,2,3
$5 > 4$				$\uparrow i$				$\uparrow j$					1,2,3,4
$5 < 6$					$\uparrow i$			$\uparrow j$					1,2,3,4,5
$9 > 6$					$\uparrow i$				$\uparrow j$				1,2,3,4,5,6
$9 > 7$					$\uparrow i$					$\uparrow j$			1,2,3,4,5,6,7
$9 > 8$					$\uparrow i$						$\uparrow j$		1,2,3,4,5,6,7,8
$9 < 10$						$\uparrow i$					$\uparrow j$		1,2,3,4,5,6,7,8,9
F_1 leer						$\uparrow i$						$\uparrow j$	1,2,3,4,5,6,7,8,9,10
F_2 leer						$\uparrow i$						$\uparrow j$	1,2,3,4,5,6,7,8,9,10

Mergesort

Wir speichern die sortierte Ergebnisfolge in einem Hilfsarray b und kopieren b anschließend nach a zurück.

Eine mögliche Implementierung:

Für den eigentlichen Algorithmus verwenden wir die Methode Merge (Element A [], int l, int m, int r), die die Teilfolgen A[l], ...,A[m-1] und A[m], ...,A[r] vereinigt.

```
(1)  Merge_Sort (Element A [ ], int l, int r)
(2)  {
(3)     if (l < r) {
(4)        int m := (l+r)/2; // m ist etwa die Mitte
(5)        Merge_Sort (A, l, m);
(6)        Merge_Sort (A, m+1, r); // beide Teilfolgen sind sortiert
(7)        Merge (A, l, m+1, r);
(8)     }
(8)  }
```

Mergesort

Analyse: Schlüsselvergleiche werden nur beim Vereinigen durchgeführt.

Für zwei Teilfolgen mit n_1 und n_2 Elementen werden mindestens $\min(n_1, n_2)$ und höchstens $n_1 + n_2 - 1$ Vergleiche durchgeführt.

Zum Vereinigen zweier etwa gleich großer Folgen mit n Elementen benötigen wir $\Theta(n)$ viele Vergleiche.

Somit ist die Anzahl der Vergleiche für das Sortieren von n Elementen (Master-Theorem!):

$$C_n = C_{\lceil \frac{n}{2} \rceil} + C_{\lfloor \frac{n}{2} \rfloor} + \Theta(n) \in \Theta(n \cdot \log(n)).$$

Das gilt für den schlechtesten ebenso wie für den besten (und damit auch für den mittleren) Fall.

Auch die Anzahl der Vertauschungen (Zuweisungen) ist aus $\Theta(n \cdot \log(n))$.

Das Verfahren ist besonders gut geeignet für den Fall, dass die Elemente in Listen oder auf Bändern gespeichert sind bzw. gespeichert werden können (externe Sortierverfahren).

Reines 2-Wege-Mergesort

Methode: Eine Folge von n Elementen wird sortiert, indem sortierte Teilfolgen zu immer längere Teilfolgen vereinigt werden.

Anfangs wird jedes einzelne Element als eine sortierte Folge betrachtet. Es werden also $(A[0])$ und $(A[1])$ vereinigt, $(A[2])$ und $(A[3])$ vereinigt, usw.

Danach werden die Teilfolgen mit zwei Elementen zu Teilfolgen mit 4 Elementen vereinigt usw.

Reines 2-Wege-Mergesort

Beispiel

2 | 1 | 3 | 9 | 5 | 6 | 7 | 4 | 8 | 10

Nach der ersten Runde:

1 2 | 3 9 | 5 6 | 4 7 | 8 10

Nach der zweiten Runde:

1 2 3 9 | 4 5 6 7 | 8 10

Nach der dritten Runde:

1 2 3 4 5 6 7 9 | 8 10

Nach der vierten Runde:

1 2 3 4 5 6 7 8 9 10

Reines 2-Wege-Mergesort

Algorithmische Umsetzung:

```
(1)  Straight_Merge_Sort (Element A [ ], int l, int r)
(2)  {
(3)     int s := 1;
(4)     while (s < r-l+1) {
(5)        int ll := l;
(6)        while (ll+s < r) {
(7)           int mm := ll+s;
(8)           int rr := mm+s-1;
(9)           if (rr > r)
(10)             rr := r;
(11)          Merge (A, ll, mm, rr);
(12)          ll := rr+1;
(13)       }
(14)       s := s*2;
(15)    }
(16) }
```

Analyse: Wie 2-Wege-Mergesort.

04. November 2016

Natürliches 2-Wege-Mergesort

Beim natürlichen 2-Wege-Mergesort wird versucht mit möglichst langen bereits sortierten Folgen zu beginnen.

Beispiel:

Die folgende Ausgangsfolge besteht aus vier sortierte Teilfolgen

$$2 \mid 1 \quad 3 \quad 9 \mid 5 \quad 6 \quad 7 \mid 4 \quad 8 \quad 10$$

Nach der ersten Runde:

$$1 \quad 2 \quad 3 \quad 9 \mid 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 10$$

Nach der zweiten Runde:

$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10$$

Methode: Teile zuerst die Ausgangsfolge $F = A[0], \ldots, A[n-1]$ in längstmögliche, sortierte Teilfolgen. Sortiere dann die Teilfolgen durch wiederholtes Vereinigen.

Natürliches 2-Wege-Mergesort

Algorithmische Umsetzung:

```
(1)  Natural_Merge_Sort (Element A [ ], int l, int r)
(2)  {
(3)     bool b := true;
(4)     while (b) {
(5)        b := false;
(6)        int ll := l;
(7)        while (ll < r) {
(8)           int mm := ll+1;
(9)           while ((mm ≤ r) and (A[mm-1].k ≤ A[mm].k))
(10)             mm++;
(11)          int rr := mm+1;
(12)          while ((rr ≤ r ) and (A[rr-1].k ≤ A[rr].k))
(13)             rr++;
(14)          rr−−;
(15)          if (mm ≤ r) {
(16)             Merge (A, ll, mm, rr);
(17)             b := true;
(18)          }
(19)          ll := rr+1;
(20)       }
(21)    }
(22) }
```

Natürliches 2-Wege-Mergesort

Analyse: Bei der Ermittlung der Anfangsaufteilung werden zusätzlich linear viele Vergleiche durchgeführt. Ansonsten besteht kein Unterschied zum reinen 2-Wege-Mergesort.

Die Anzahl der Vergleiche im schlechtesten Fall ist aus $\Theta(n \cdot \log(n))$.

Der Vorzug des natürlichen 2-Wege-Mergesorts liegt in der Ausnutzung einer Vorsortierung. Im günstigsten Fall ist die Folge bereits sortiert. Dann benötigt natürliches 2-Wege-Mergesort $\Theta(n)$ viele Vergleiche.

Im Mittel gilt für jede Position i, dass beide Ereignisse $A[i].k < A[i+1].k$ und $A[i].k > A[i+1].k$ gleich wahrscheinlich sind, falls alle Schlüssel verschieden sind.

Die Anzahl der Stellen, an denen eine Vorsortierung zu Ende ist, ist somit etwa $n/2$, woraus sich im Mittel etwa $n/2$ Teilfolgen ergeben. Beim reinen 2-Wege-Mergesort erhalten wir nach dem ersten Durchlauf ebenfalls $n/2$ Teilfolgen; daher sparen wir beim natürlichen 2-Wege-Mergesort im Mittel etwa einen Durchlauf.

Die Anzahl der Vertauschungen ist im besten Fall 0 und im Mittel sowie im schlechtesten Fall $\Theta(n \cdot \log(n))$.

Sortieren durch Fachverteilung

Achtung: Wir wechseln jetzt unserer Grundvoraussetzungen!

Beispiel

Counting-Sort: Die Schlüssel sind aus einem kleinen zusammenhängenden Zahlenbereich, zum Beispiel aus einer Indexmenge $\{0,1,\ldots,k\}$.

Methode: Initialisiere ein Array C [] mit k Einträgen:

```
(1)  for (int i := 0; i < k; i++)
(2)      C[i] := 0;
```

Zähle die Häufigkeiten der vorkommenden Schlüssel:

```
(1)  for (int i := 0; i < n; i++)
(2)      C[A[i].k]++;
```

Berechne nun in C [] die Anfangspositionen der Elemente mit gleichem Schlüssel:

Sortieren durch Fachverteilung

Beispiel

```
(1)  int s := 0;
(2)  for (int i := 0; i < k; i++)
(3)  {
(4)     int c := C[i];
(5)     C[i] := s;
(6)     s := s+c;
(7)  }
```

Schreibe das Ergebnis in Feld B []:

```
(1)  for (int i := 0; i < n; i++)
(2)  {
(3)     B[C[A[i].k]] := A[i];
(4)     C[A[i].k]++;
(5)  }
```

m-adische Zahlen

Zur Darstellung von Zahlen verwenden wir in der Regel 10 Ziffern (0, 1, ..., 9). Damit stellen wir Dezimalzahlen, also Zahlen zur Basis 10 dar.

Beispiel: $1234_{10} = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$

Genauso kann man Zahlen mit einer beliebigen anderen Anzahl Ziffern darstellen. Wenn wir m Ziffern verwenden, Stellen wir die Zahlen zur Basis m dar.

Seien die Ziffern $a_n, a_{n-1}, \ldots, a_1, a_0$ aus dieser Menge von Ziffern.

Dann gilt

$$a_n a_{n-1} \ldots a_1 a_0 = a_n \cdot m^n + a_{n-1} \cdot a^{n-1} + \cdots + a_1 \cdot m^1 + a_0 \cdot m^0$$

Beispiel:
$100_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 4_{10}$
$42_8 = 4 \cdot 8^1 + 2 \cdot 8^0 = 34_{10}$

Sortieren durch Fachverteilung

Nun etwas allgemeiner:

1. Die Schlüssel sind Wörter über einem Alphabet mit genau m-Elementen.
2. Ohne Beschränkung der Allgemeinheit setzen wir im Folgenden voraus, dass alle Schlüssel die gleiche Länge b haben. Falls nötig, fügen wir führende Nullen ein.

Die Schlüssel können als m-adische Zahl aufgefasst werden. Daher nennt man m auch die Wurzel (Radix) der Darstellung.

Sei $z_m(i, k)$ die Ziffer mit Gewicht m^i in Schlüssel k aufgefasst als Zahl zur Basis m.

$$z_{10}(0, 517) = 7$$
$$z_{10}(1, 517) = 1$$
$$z_{10}(2, 517) = 5$$

$$z_2(3, 1001) = 1$$

Radix-exchange-sort

Schlüssel sind Binärzahlen ($m = 2$).

Methode: Teile $A[0], \ldots, A[n-1]$ in zwei Teilfolgen. Alle Elemente mit führender 0 im Schlüssel kommen in die linke Teilfolge; alle Elemente mit führender 1 im Schlüssel kommen in die rechte Teilfolge. Die Teilfogen werden rekursiv auf dieselbe Weise sortiert, wobei nun das nächste Bit von links die Rolle des führenden Bits übernimmt.

Die Aufteilung in zwei Teilfolgen kann wie bei Quicksort "in situ" durch Vertauschen von Elementen des Feldes erreicht werden.

Beispiel

1011	0010	1101	1001	0011	0101	1010
0101	0010	0011 |	1001	1101	1011	1010
0011	0010 |	0101 |	1001	1010	1011 |	1101
0011	0010 |	0101 |	1001 |	1010	1011 |	1101
0010 |	0011 |	0101 |	1001 |	1010 |	1011 |	1101

Radix-exchange-sort

Analyse: Die Aufteilung benötigt $n + 1$ Vergleiche wie bei Quicksort. Die Rekursionstiefe ist immer b (Länge der Schlüssel).

Die Anzahl der Vergleiche ist immer $\Theta(n \cdot b)$.

Ist $b = \log(n)$, dann ist Radix-exchange-sort eine echte Alternative zu Quicksort. Hat man aber nur wenige, verschiedene aber dafür sehr lange Schlüssel, so ist Radix-exchange-sort schlecht.

Radixsort

Methode: Es werden b Verteilungsphasen und Sammelphasen durchlaufen. Für $t = 0, \ldots, b-1$ wird jede Phase einmal durchlaufen.

Verteilungsphase: In der Verteilungsphase werden die Elemente auf m Fächer verteilt. Das i-te Fach F_i nimmt in der $t+1$-ten Verteilungsphase alle Elemente auf, deren Schlüssel an Position t die Ziffer i haben. Die Elemente werden immer oben auf die im Fach bereits vorhandenen Elemente gelegt.

Sammelphase: In der Sammelphase werden die Elemente in den Fächern $F_0, \ldots, F_{m-1}$ gesammelt, indem für $i = m-2, \ldots, 0$ die Elemente im Fach F_{i+1} als Ganzes auf die Elemente im Fach F_i gelegt werden.

In der nachfolgenden Verteilungsphase werden die Elemente von unten nach oben auf die Fächer verteilt.

Radixsort

Beispiel

$m = 10$

$F = 37, 76, 10, 94, 23, 89, 45, 67, 30, 34, 55, 41, 42, 61$

Nach der ersten Verteilung bzgl. der Ziffer auf Position $t = 0$.

30	61			34	55		67		
10	41	42	23	94	45	76	37		89
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9

Nach der ersten Sammlung:

$$10, 30, 41, 61, 42, 23, 94, 34, 45, 55, 76, 37, 67, 89$$

Radixsort

Beispiel

Nach der zweiten Verteilung bzgl. der Ziffer auf Position $t = 1$.

			37	45					
			34	42		67			
	10	23	30	41	55	61	76	89	94
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9

Nach der zweiten Sammlung erhalten wir die sortierte Folge:

$$10, 23, 30, 34, 37, 41, 42, 45, 55, 61, 67, 76, 89, 94$$

Radixsort

Korrektheit: Beweis durch Induktion über die betrachtete Position.

Nach der Verteilung und Sammlung bzgl. Position $t = 0$ sind die Elemente bgzl. $t = 0$ sortiert.

Sind die Elemente aufsteigend sortiert, wenn man nur die Positionen $0, \ldots, t-1$ betrachtet, so bleiben sie nach der t-ten Verteilung in jedem Fach aufsteigend sortiert bzgl. der Positionen $0, \ldots, t-1$ sortiert. Nach der t-ten Sammlung sind sie zusätzlich bgzl. Position t sortiert, und somit sortiert bgzl. der Positionen $0, \ldots, t$.

Nach der Verteilung und Sammlung bzgl. Position $t = b-1$ ist die Folge sortiert.

Die Laufzeit von Implementierungen des Radixsort sind aus $\Theta(b \cdot (n+m))$.

Speziafall: Sollen m Elemente mit verschiedenen Schlüsseln im Bereich $0, \ldots, m-1$ sortiert werden, so ist $n = m$ und $b = 1$. Daraus folgt eine Sortierung in linearer Zeit.

Maße für Vorsortierungen

Beispiel

Einige vorsortierte Folgen:

$$
\begin{array}{lcccccccccc}
F_a & = & 2 & 1 & 4 & 3 & 6 & 5 & 8 & 7 & 9 \\
F_b & = & 6 & 7 & 8 & 9 & 1 & 2 & 3 & 4 & 5 \\
F_c & = & 5 & 1 & 7 & 4 & 9 & 2 & 8 & 3 & 6
\end{array}
$$

Maße für Vorsortierungen: Inversionszahl

$$\text{inv}(F) = |\{(i,j) \mid 0 \leq i < j \leq n-1 \text{ und } A[i].k > A[j].k\}|$$

Die Inversionszahl liegt immer zwischen 0 für (aufsteigend) sortierte Folgen und $\frac{n(n-1)}{2}$ für absteigend sortierte Folgen.

$$\begin{array}{lcll}
\text{inv}(F_a) & = & |\{ & (0,1),(2,3),(4,5),(6,7) \qquad\qquad\qquad\quad \}| = 4 \\
\text{inv}(F_b) & = & |\{ & (0,4),(0,5),(0,6),(0,7),(0,8) \\
 & & & (1,4),(1,5),(1,6),(1,7),(1,8) \\
 & & & (2,4),(2,5),(2,6),(2,7),(2,8) \\
 & & & (3,4),(3,5),(3,6),(3,7),(3,8) \quad \}| = 20 \\
\text{inv}(F_c) & = & |\{ & (0,1),(0,3),(0,5),(0,7),(2,3) \\
 & & & (2,5),(2,7),(2,8),(3,5),(3,7) \\
 & & & (4,5),(4,6),(4,7),(4,8),(6,7) \\
 & & & (6,8) \qquad\qquad\qquad\qquad\quad \}| = 16
\end{array}$$

Maße für Vorsortierungen: Inversionszahl

Offensichtlich gilt

$$\text{inv}(F) \;=\; |\{(i,j) \mid 0 \leq i < j \leq n-1 \text{ und } A[i].k > A[j].k\}| \;=\; \sum_{j=2}^{n} h_j$$

mit

$$h_j \;=\; |\{i \mid 0 \leq i < j \leq n-1 \text{ und } A[i].k > A[j].k\}|$$

h_j ist die Anzahl der Elemente $A[i]$ mit $i < j$, die einen Schlüssel größer als $A[j].k$ haben.

Beim Sortieren durch Einfügen muss das Element $A[j]$ im Abstand h_j vom Ende der bereits sortierten Teilfolge eingefügt werden.

Maße für Vorsortierungen: Inversionszahl

Beispiel

$$F_c = 5, 1, 7, 4, 9, 2, 8, 3, 6, \quad \text{inv}(F_c) = 16$$

Sortieren durch Einfügen:

i	k_i	h_i	
0	5	0	5
1	1	1	1, 5
2	7	0	1, 5, 7
3	4	2	1, 4, 5, 7
4	9	0	1, 4, 5, 7, 9
5	2	4	1, 2, 4, 5, 7, 9
6	8	1	1, 2, 4, 5, 7, 8, 9
7	3	5	1, 2, 3, 4, 5, 7, 8, 9
8	6	3	1, 2, 3, 4, 5, 6, 7, 8, 9

Maße für Vorsortierungen: Run-Zahl

$$\text{runs}(F) = |\{i \mid 0 \leq i < n-1 \text{ und } A[i].k > A[i+1].k\}| + 1$$

Die Run-Zahl liegt zwischen 1 für (aufsteigend) sortierte Folgen und n für absteigend sortierte Folgen.

$$\begin{aligned} \text{runs}(F_a) &= 5 \\ \text{runs}(F_b) &= 2 \\ \text{runs}(F_c) &= 5 \end{aligned}$$

Für welches Sortierverfahren könnte dieses Maß von Bedeutung sein?

Maße für Vorsortierungen: LAS-Zahl

Die Länge der längsten aufsteigenden Teilfolge

(longest ascending subsequence):

$$\mathrm{las}(F) = \max \left\{ t \;\middle|\; \begin{array}{l} \exists i_1, \ldots, i_t \text{ mit} \\ 0 \leq i_1 < i_2 < \cdots i_t \leq n-1 \text{ und} \\ A[i_1].k \leq A[i_2].k \leq \cdots A[i_t].k \end{array} \right\}$$

Die Länge der längsten aufsteigenden Teilfolge liegt zwischen n für (aufsteigend) sortierte Folgen und 1 für absteigend sortierte Folgen mit paarweise verschiedenen Schlüsseln.

$$\begin{array}{lcl} \mathrm{las}(F_a) & = & 5 \\ \mathrm{las}(F_b) & = & 5 \\ \mathrm{las}(F_c) & = & 4 \end{array}$$

Maße für Vorsortierungen: REM-Zahl

Die minimale Anzahl der Elemente, die entfernt werden müssen, um eine sortierte Teilfolge zu erhalten:

$$\mathrm{rem}(F) = n - \mathrm{las}(F)$$

Die REM-Zahl liegt zwischen 0 für aufsteigend sortierte Folgen und $n - 1$ für absteigend sortierte Folgen mit paarweise verschiedenen Schlüsseln.

08. November 2016

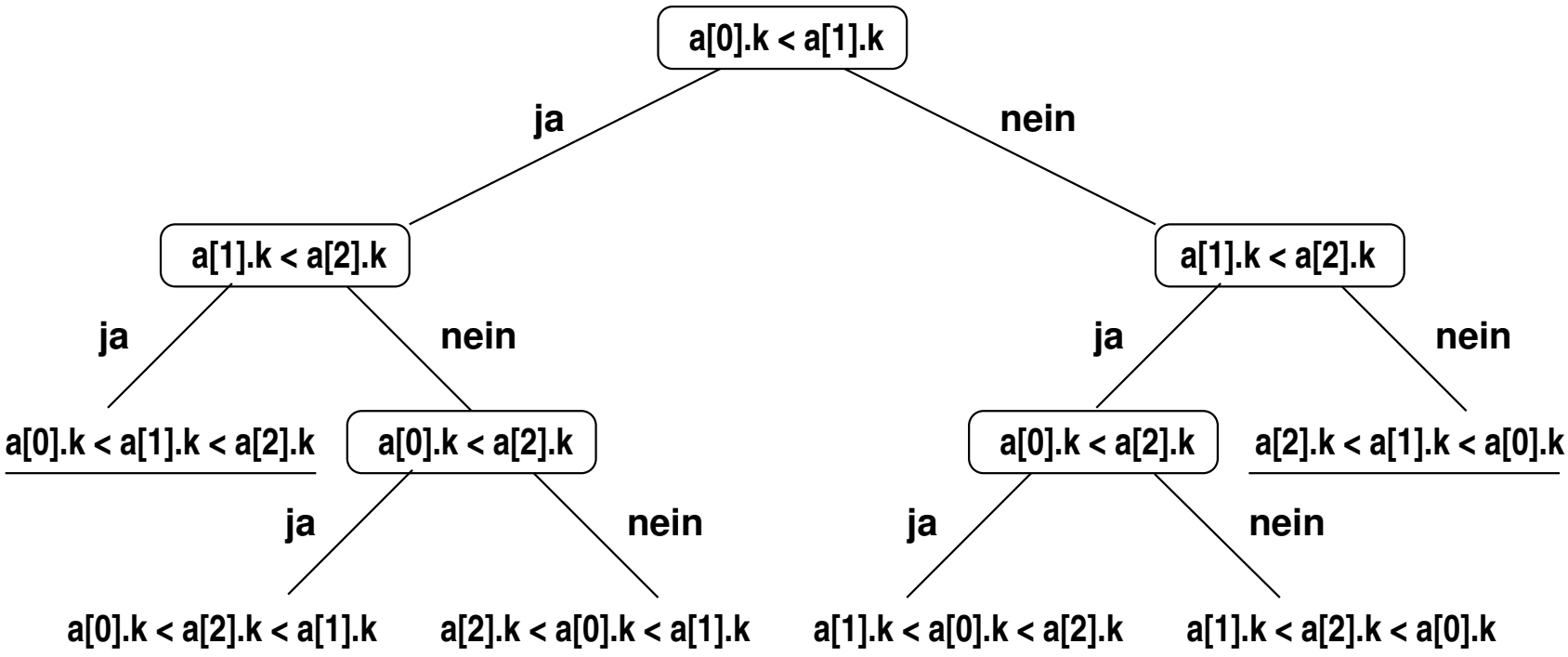
Sortieren: Untere Schranken

Voraussetzung: Alle Schlüssel sind verschieden.

Jeder Sortieralgorithmus definiert einen Entscheidungsbaum.

Jeder innere Knoten (Entscheidungsknoten) im Entscheidungsbaum repräsentiert einen Schlüsselvergleich. Der linke Sohn repräsentiert den nächsten Schlüsselvergleich bei Antwort "ja", der rechte Sohn repräsentiert den nächsten Schlüsselvergleich bei Antwort "nein". Die Blätter repräsentieren die Elemente der Ausgangsfolge in sortierter Reihenfolge.

Sortieren: Untere Schranken



Sortieren: Untere Schranken

Bemerkungen:

- Der Entscheidungsbaum ist ein Binärbaum, d.h., auf Höhe i befinden sich höchstens 2^i Knoten.
- Der Entscheidungsbaum hat $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$ Blätter, weil es $n!$ viele verschiedene Folgen mit n Elementen gibt.
- Die Anzahl der Entscheidungsknoten auf den Wegen von der Wurzel zu den Blättern entspricht der Anzahl der Vergleiche für das Sortieren der Folgen.
- Der längste Weg von der Wurzel zu einem Blatt enthält mindestens $\lceil \log(n!) \rceil$ Entscheidungsknoten. Daraus folgt, dass jedes Sortierverfahren im Worst Case mindestens $\log(n!) \in \Theta(n \cdot \log(n))$ viele Vergleiche benötigt.

Sortieren: Untere Schranken

Satz

$$\log(n!) \in \Theta(n \cdot \log(n))$$

Beweis.

$$\begin{aligned}
\left(\lfloor \tfrac{n}{2} \rfloor + 1\right)^{\lceil \frac{n}{2} \rceil} &= \overbrace{\left(\lfloor \tfrac{n}{2} \rfloor + 1\right) \cdot \left(\lfloor \tfrac{n}{2} \rfloor + 1\right) \cdot \ldots \cdot \left(\lfloor \tfrac{n}{2} \rfloor + 1\right) \cdot \left(\lfloor \tfrac{n}{2} \rfloor + 1\right)}^{\lceil \frac{n}{2} \rceil} \\
n! &= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot \left(\lfloor \tfrac{n}{2} \rfloor + 1\right) \cdot \left(\lfloor \tfrac{n}{2} \rfloor + 2\right) \cdot \ldots \cdot (n-1) \cdot n \\
n^n &= \underbrace{n \cdot n \cdot n \cdot \ldots \cdot n \cdot n \cdot \ldots \cdot n \cdot n}_{n}
\end{aligned}$$

Sortieren: Untere Schranken

Beweis Fortsetzung.

$$
\begin{aligned}
\Rightarrow \quad & \left(\left\lfloor \tfrac{n}{2} \right\rfloor + 1\right)^{\left\lceil \frac{n}{2} \right\rceil} && \leq \quad n! && \leq \quad n^n \\
\Rightarrow \quad & \log\left(\left(\left\lfloor \tfrac{n}{2} \right\rfloor + 1\right)^{\left\lceil \frac{n}{2} \right\rceil}\right) && \leq \quad \log(n!) && \leq \quad \log(n^n) \\
\Rightarrow \quad & \left(\left\lceil \tfrac{n}{2} \right\rceil\right) \cdot \log\left(\left\lfloor \tfrac{n}{2} \right\rfloor + 1\right) && \leq \quad \log(n!) && \leq \quad n \cdot \log(n)
\end{aligned}
$$

Da $\log(\frac{n}{2}) = \log(n) - 1$, *folgt* $\log(n!) \in \Theta(n \log(n))$. □

Sortieren: Untere Schranken

Ein Sortierverfahren nutzt eine Vorsortierung optimal aus, wenn im Entscheidungsbaum die Blätter der maximal vorsortierten Folgen so nah wie möglich bei der Wurzel sind.

Sei $m(F)$ ein Maß für die Vorsortierung einer Folge F.

Für eine Folge F sei $r_{m,F}$ die Anzahl aller Folgen F' gleicher Länge, die mindestens genau so gut vorsortiert sind wie F.

$$r_{m,F} \;=\; |\{F' \mid m(F') \leq m(F)\}|$$

Da der Entscheidungsbaum ein Binärbaum ist, gibt es mindestens eine Folge F_0 mit $m(F_0) \leq m(F)$, deren Abstand von der Wurzel mindestens $\lceil \log(r_{m,F}) \rceil$ ist.

Da jedoch immer mindestens $n-1$ Vergleiche notwendig sind, ist $\Omega(n + \log(r_{m,F}))$ eine untere Schranke für die Anzahl der Vergleiche zum Sortieren vorsortierter Folgen.