

Vh diese Werte: D_i, τ_i, D_0
 UE " $\bar{\tau}$

Quicksort

$$\begin{aligned}
 \bar{C}(n) &= \frac{1}{n!} \sum_{k=1}^n \sum_{\tau \in \Pi_k} C(\tau) \\
 &= \frac{1}{n!} \sum_{k=1}^n \left[n! + (n-1)! \bar{C}(k-1) + (n-1)! \bar{C}(n-k) \right] \\
 &= n + \frac{1}{n} \sum_{k=1}^n \bar{C}(k-1) + \frac{1}{n} \sum_{k=1}^n \bar{C}(n-k) \\
 &= n + \frac{2}{n} \sum_{k=1}^n \bar{C}(k-1)
 \end{aligned}$$

Diagramm zur Partitionierung: Ein Array der Größe n wird in drei Teile unterteilt: S_1 (Größe $k-1$), S_2 (Größe k) und S_3 (Größe $n-k$). Die Rekurrenzgleichung $\bar{C}(n) = \frac{1}{n!} \sum_{k=1}^n [n! + (n-1)! \bar{C}(k-1) + (n-1)! \bar{C}(n-k)]$ zeigt, dass S_1 $n!$ Operationen benötigt, S_2 $(n-1)! \bar{C}(k-1)$ und S_3 $(n-1)! \bar{C}(n-k)$.

Die Summen $\sum_{k=1}^n \bar{C}(k-1)$ und $\sum_{k=1}^n \bar{C}(n-k)$ sind äquivalent zu $\bar{C}(0) + \bar{C}(1) + \dots + \bar{C}(n-1)$.

$$\bar{c}(n) \leq n + \frac{2}{n} \sum_{k=2}^{n-1} \bar{c}(k)$$

$$\bar{c}(2) = 1$$

Rekursionsgleichung

für den mittleren

Aufwand von Quicksort

Lemma: Für die Lösung $r(n)$ der Rekursionsgleichung

$$r(n) = n + \frac{2}{n} \sum_{k=2}^{n-1} r(k) \quad \text{mit } r(0) = r(1) = 0 \\ r(2) = 1$$

$$\text{gilt: } r(n) \leq 2 \cdot n \cdot \ln n$$

Beweis: vollst. Ind. nach n

$$\text{Ansatz: } r(n) \leq c \cdot n \ln n$$

↑
bestimmen im Beweis

Ind. hyp.

$$n = 2 \quad r(2) = 1$$

$$c \cdot n \ln n = c \cdot 2 \cdot \ln(2) \approx c \cdot 1,39$$

} Ind. Anfang erfüllt
für $c \geq 1$

Ind. hyp. für $k = 2, 3, \dots, n-1$ sei $r(k) \leq c \cdot k \cdot \ln k$

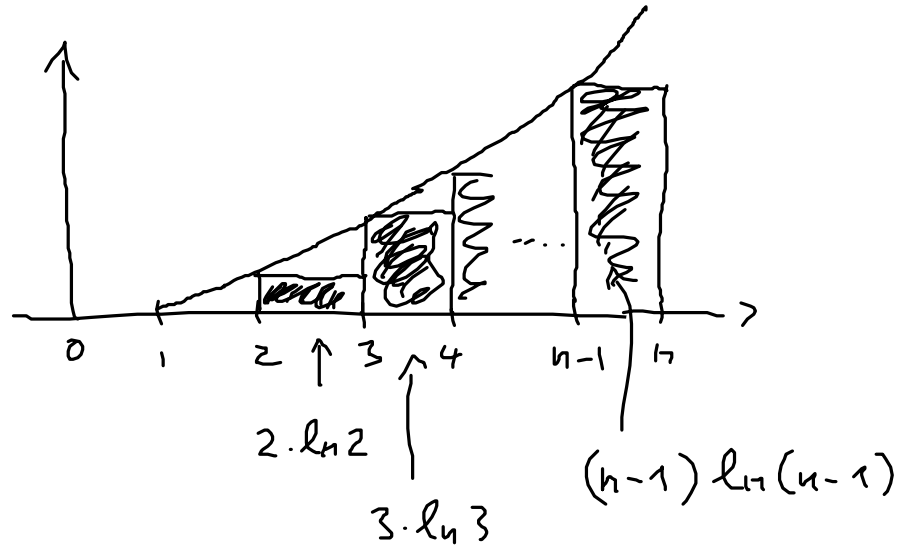
Schluss auf n

$$r(n) \stackrel{\text{Def}}{=} n + \frac{2}{n} \sum_{k=2}^{n-1} r(k) \stackrel{IV}{\leq} n + \frac{2}{n} \sum_{k=2}^{n-1} c \cdot k \ln k$$

$$= n + \frac{2c}{n} \sum_{k=2}^{n-1} k \ln k$$

? wie berechnen? $\stackrel{!}{=} \text{rote Fläche}$

Trick aus Analysis: Betrachte Fkt $f(x) = x \cdot \ln x$



$$\begin{aligned} \text{rote Fläche} &\leq \int_2^n x \ln x \, dx \\ &= \frac{x^2}{2} \ln x \Big|_2^n - \int_2^n \frac{x}{2} \, dx \\ &= \frac{n^2}{2} \ln n - \underline{2 \ln 2} - \left(\frac{n^2}{4} - 1 \right) \end{aligned}$$

$$1 - 2 \ln 2 \leq 0$$

$$\leq \frac{n^2}{2} \ln n - \frac{n^2}{4}$$

$$T(n) \leq n + \frac{2c}{n} \left[\frac{n^2}{2} \ln n - \frac{n^2}{4} \right]$$

$$= n + \frac{2c}{2} n \ln n - \frac{2c}{4} n$$

$$= c \cdot n \ln n + \underbrace{n - \frac{c}{2} n}$$

$$=: R(n)$$

$$R(n) \leq 0$$

$$\Leftrightarrow n \leq \frac{c}{2} n$$

$$\Leftrightarrow 2 \leq c \quad n \geq 2$$

$$\leq c \cdot n \ln n \quad \text{für } c \geq 2 \quad \square$$

Satz: Für mittlere Anzahl von Vergleichen $\bar{C}(n)$ zum Sortieren eines n -elem. Arrays gilt

$$\bar{C}(n) \in O(n \log n)$$

Beweis: $\bar{C}(n) \leq T(n) \leq 2 \cdot n \ln n = 2n \frac{\log n}{\log e}$

$$= \underbrace{\frac{2}{\log e}}_{\approx 1,386} \cdot n \log n \in O(n \log n)$$

mittlere
Anzahl der Zuweisungen $\bar{A}(n)$ ähnlich $A(n) \in O(n \log n)$

10.5 Heapsort

Heap (Priority Queue) = abstrakte Datenstruktur
mit folgenden Kennzeichen:

Wertebereich: Menge von Werten eines homogenen Komponententyps

Alle Komponenten besitzen einen Schlüssel

[bei Sortieren: Werte $\hat{=}$ $\text{array Komp. } \text{vec}[i]$
Schlüssel $\hat{=}$ $\text{vec}[i].\text{key}$]

Operationen:

1. Einfügen einer Komponente
2. Zugriff auf Komp. mit größtem Schlüssel
3. Entfernen der Komp. mit größtem Schlüssel
4. Ändern des Schlüssels einer Komponente

Großstruktur von Heapsort:

1. Gegeben sei array vec mit n Komponenten
2. Initialisiere den Heap mit Komponenten von vec
(alle einfügen) $O(n)$
3. für $i := n-1$ down to 0 do $n \text{ mal}$
 - 3.1 Greife auf das größte Element im Heap zu $O(1)$
 - 3.2 Wechse diese Komponente des $\text{array Komp. } \text{vec}[i]$ zu

3.3. Entferne das größte Element im Heap

$O(1)$

Klar das das also das Array sortiert

ähnelt Selection Sort mit zusätzlicher Datenstruktur "Heap"

$$\frac{O(\log n)}{\sum O(n \log n)}$$

Heiß speziellen Heap implementieren

- Initialisierung $O(n)$
- Zugriff auf größte Element in $O(1)$
- Entfernen des größten in $O(\log n)$

[Indizes Schlüssel nicht benötigt]

SATZ Mit dieser Heap-Implementierung benötigt

Heapsort $O(n \log n)$ Aufwand (Vergleiche und Zuweisungen)

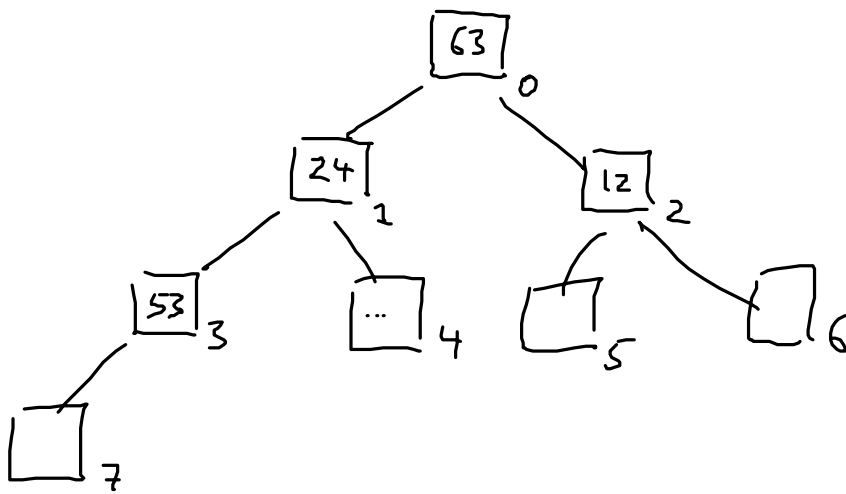
Implementierung des Heaps:

Hauptidee: Array als Baum vorstellen

63	24	12	53	72	18	44	35
0	1	2	3	4	5	6	7

Baum:

Baum schichtenweise voll machen



Lemma:

- $vec[0]$ ist Wurzel des Baumes
- linker Sohn von $vec[i]$ ist $vec[2i+1]$ (falls vorhanden)
- rechter ... $vec[2i+2]$ (...)
- nur Knoten $vec[i]$ mit $i \leq \lfloor \frac{n}{2} \rfloor - 1$ haben Söhne

Dieser Baum erfüllt Heapeigenschaft (ist ein Heap)

wenn gilt $vec[i].key \geq vec[2i+1].key$ falls $2i+1 < n$
... $\geq vec[2i+2].key$ $2i+2 < n$

Vater \geq Söhne

Heap = Array als Baum mit Heapeigenschaft

Lemma: Erfüllt vec die Heapeigenschaft, so gilt

- $vec[0].key$ ist der größte Schlüsselwert
- Entlang eines jeden Weges von Wurzel bis Blatt

sind die Schlüssel absteigend sortiert

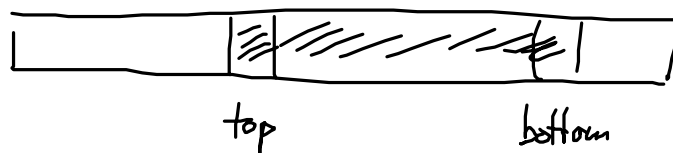
Wie Heapeigenschaft herstellen?

Methode `heapify()`

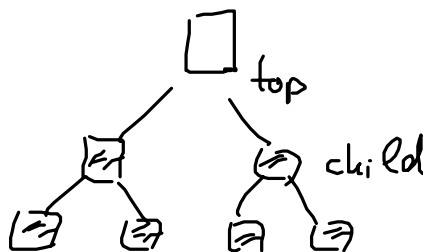
Voraussetzung: Heapeigenschaft gilt im Bereich
 $vec[top+1] \dots vec[bottom]$



Aufruf von `heapify(vec, top, bottom)` stellt dann
die Heapeigenschaft im Bereich von `top` ... `bottom` her



Baumvorstellung



`heapify()`:

1. Ermittle größeren der beiden Söhne von `top`, dies sei `child`
(falls kein Sohn exist, so ist Heapeig. erfüllt
falls nur ein Sohn exist, so nehme diesen

2. Vergleiche $vec[child].key$ mit $vec[top].key$
Falls $child$ den größeren Wert hat, so tausche
 $vec[child]$ und $vec[top]$

3. Wende $heapify$ rekursiv auf $child$ an
[Aufruf $heapify(vec, child, bottom)$]