Datenstrukturen & Algorithmen	9. April, 2014

Handout 7

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Splay Bäume, Huffman Bäume, Optimale Binäre Suchbäume

Referenz: Widmayer, Kapitel 5.4, 5.7, Cormen, Kapitel 15.5, 16.3

Links

- Hashing: ausfürliche Erklärungen zum Cuckoo-Hash-Verfahren
  http://www.it-c.dk/people/pagh/papers/cuckoo-undergrad.pdf (http://goo.gl/DWbKg)
- Splay Trees
  - Tarjan, Sleator: Self-adjusting binary search trees
    http://portal.acm.org/citation.cfm?id=3835 (http://goo.gl/pe1hu)
  - Demo Applet http://www.link.cs.cmu.edu/splay/
    (http://goo.gl/AnzMK)
  - Slides Belfrage: http://www.belfrage.net/eth/d&a/pdf/uebung12_h.pdf
    (http://goo.gl/a5o41)
  - Berkley Lecture: http://goo.gl/UQrSh
- Huffman Trees
  - Eine ausführliche Darstellung findet sich in dem Standardwerk *Elements of Information Theory*:
    http://goo.gl/knvgQ
  - Vorlesung Informationstheorie
    http://graphics.ethz.ch/teaching/infotheory08/notes.php
    (http://goo.gl/pgki3)
  - Huffman Tree Applet http://goo.gl/WP2ch (aus dem ETH Netz zugreifbar)
- Optimale Binäre Suchbäume
  - Jeff Erickson Lecture Notes
    http://compgeom.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/05-dynprog.pdf
    (http://goo.gl/54Hqk)
  - Video Lecture S. Albers: http://goo.gl/E0biZ

Splay Bäume

So you are lean and mean and resourceful and you continue to walk on the edge of the precipice because over the years you have become fascinated by how close you can walk without losing your balance. – Richard M. Nixon

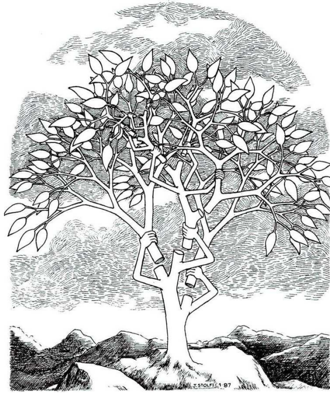
Motivation: Eine Folge von Zugriffsoperationen auf einer Menge von Elementen eines total geordneten Universums ausführen. Dabei sollen kürzlich aufgerufene Elemente wiederum schnell zugreifbar sein.

Splay Trees sind sich selbstanordnende binäre Suchbäume mit guten Balanceeigenschaften (amortisiert über eine Sequenz von Operationen). Sie wurden erstmals von Sleator und Tarjan 1985 eingeführt. In einer gewissen Weise vereinen sie die Eigenschaften von AVL-Bäumen (balancierend), und einer MFT-Liste (kürzlich zugegriffene Elemente sind sehr schnell erreichbar). Die Vorteile gegenüber selbstanordnenden Listen liegen auf der Hand. Suchanfragen können in (amortisiert) $\mathcal{O}(\log n)$ Zeit durchgeführt werden.

Eine Anfrage im Splay Tree zieht immer eine weitere Operation mit sich, das *Splaying* (siehe unten). Dabei wird der Baum so arrangiert, dass das aktuelle Element an die Wurzel platziert

wird. Dies wird mit Baumrotationen gemacht, die vom AVL-Baum her bekannt sein sollten. Ein Nachteil ist, dass der Baum komplett unbalanciert sein kann; die amortisierte Analyse zeigt jedoch trotzdem eine Performance von $\mathcal{O}(\log n)$ Zeit für Einfüge-, Such- und Lösch-Operationen.

(Für gleichmässig verteilte Zugriffshäufigkeiten ist ein Splay Tree jedoch sehr unpraktisch und es empfiehlt sich, einen 'normalen' binären Suchbaum zu verwenden.)



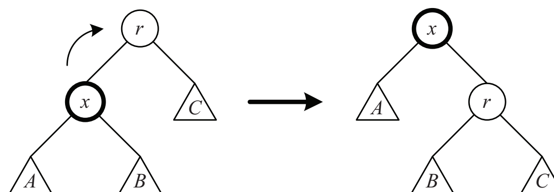
A Self-Adjusting Search Tree (Jorge Stolfi)

Im Folgenden sind die Operationen beschrieben. Diese werden manchmal auch genauer unterschieden. So bedeutet eine 'Zig' Operation eine Rechtsrotation und 'Zag' eine Linksrotation.

Analoges gilt für die beiden anderen Operationen.

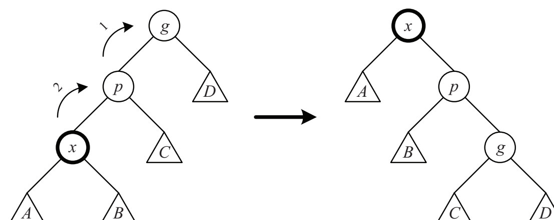
'Zig'

x ist das Element, auf das zugegriffen wird. Es muss nun also an die Wurzel rotiert werden. Die Zig-Operation wird nur ausgeführt, wenn x unmittelbar unter der Wurzel ist und es kann eine einfache Rotation über die Achse von x und der Wurzel vorgenommen werden. Diese Rotationen sind analog zu den einfachen Rotationen in einem AVL-Baum.



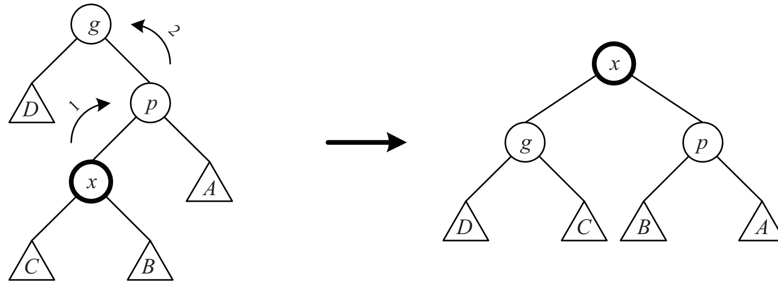
'Zig-Zig'

Diese Operation wird durchgeführt, wenn der Vater p vom aktuellen Element x nicht die Wurzel ist und sowohl p als auch x jeweils linke Kinder ihrer jeweiligen Väter sind. Es werden zwei Zig-Rotationen in die gleiche Richtung durchgeführt, um x zwei Level nach oben zu befördern. Zuerst p mit seinem Vater, dann x mit p. Diese Operation entspricht zwei einfachen Rotationen in die selbe Richtung in einem AVL-Baum.



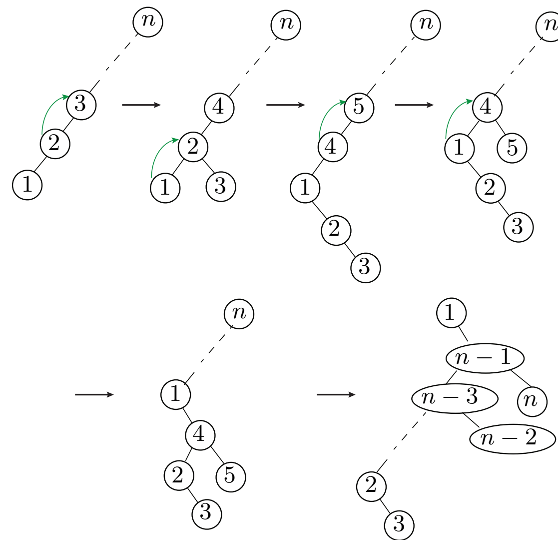
'Zig-Zag'

Diese Operation wird durchgeführt, wenn x ein linkes und p ein rechtes Kind seines Vaters ist. Zuerst wird eine Rotation über die Achse von x und p vorgenommen, danach eine über die Achse von x und seinem neuen Vater. Dies entspricht einer Doppelrotation in einem AVL-Baum (d.h. zwei einfache Rotationen in verschiedene Richtungen)



'Zag', 'Zag-Zag', 'Zag-Zig'

Analog zu den obengenannten, einfach spiegelverkehrt.



Nach dem Splaying des Schlüssels 1 *ist der Baum nur noch halb so gross*

Stop and Think

Wieso werden beim Zig-Zig zwei Rotationen durchgeführt? Was würde geschehen, wenn wie beim AVL nur eine Rotation gemacht würde? Was geschieht mit obigem Beispiel?

Einfügen

Um einen Schlüssel k in den Splay Tree einzufügen, wird zunächst $splay(k)$ ausgeführt und damit der symmetrische Vorgänger (oder Nachfolger) an die Wurzel des Baumes gebracht (ist das Element bereits im Baum, so ist dies nun an der Wurzel und es ist nichts weiter zu tun). Nun kann k einfach an die Wurzel gesetzt werden: die aktuelle Wurzel wird ein Kind von k.

Entfernen

Um einen Schlüssel k aus dem Splay Tree zu entfernen, wird zunächst wieder $splay(k)$ ausgeführt und damit das Element an die Wurzel gebracht. Ist k nicht im Baum ist nichts weiter zu tun. Ansonsten wird im linken Teilbaum der Wurzel k der symmetrische Vorgänger gesplayed und an die Wurzel des linken Teilbaumes von k gebracht. Dieser Vorgänger hat kein rechtes Kind und k kann deshalb einfach von der Wurzel entfernt werden.

Huffman Trees

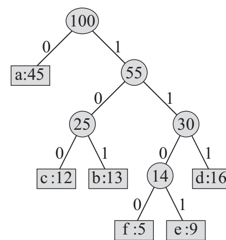
Ein *Greedy* Algorithmus, der zu einer gegebenen Häufigkeitsverteilung einen Präfixfreien-Code (*Huffman Code* genannt) mit minimaler mittlerer Codewortlänge konstruiert, stammt von D. A. Huffman. Huffman-Coding ist eines der bedeutendsten klassischen Codierungsverfahren. Gut merken! Es wird dabei ein Baum konstruiert, aus dem sich dann der Präfix-Code ergibt. Hierbei wird zuerst ein Wald erzeugt und dieser wird ann sukzessive in einen Baum umgewandelt. Im ersten Schritt erzeugt man für jedes Zeichen einen Wurzelbaum mit nur einem Knoten, der mit dem Zeichen und der entsprechenden Häufigkeit markiert ist. Der so entstandene Wald heisst W. In jedem weiteren Schritt werden nun D Bäume (oft $D = 2$, $D = 3$ für einen ternären Huffman Code) aus W zu einem einzigen vereinigt, bis W ebenfalls ein Baum ist. Dabei sucht man die Bäume $B_1, ..., B_D$ aus W, deren Wurzeln die kleinsten Markierungen haben. Ist dies auf mehrere Arten möglich, so hat die Auswahl der Bäume keinen Einfluss auf die erzielte mittlere Wortlänge der Codierung. Diese Bäume werden zu einem Baum verschmolzen. Die Wurzel dieses Baumes wird mit der Summe der Markierungen markiert. Die Nachfolger der Wurzel sind die Wurzeln der Bäume $B_1, ..., B_D$.

Der durch den Huffman-Algorithmus erzeugte Baum ist nicht eindeutig, denn es kann passieren, dass die Wurzeln von mehreren Bäumen die gleiche Gewichtung haben. In diesem Fall haben die entstehenden Codes die gleiche mittlere Codewortlänge.

HUFFMAN(C)
  // konstruiere binären Huffman-Code für Symbole $c \in C$ mit den Häufigkeiten $f[c]$
  $n = |C|$
  $W = C$ // verwende Min-Heap, alle Zeichen mit Wahrscheinlichkeiten einfügen

  for $i = 1$ **to** $n - 1$
    allocate a new node z
    $left[z] = x =$ EXTRACT-MIN(W)
    $right[z] = y =$ EXTRACT-MIN(W)
    $f[z] = f[x] + f[y]$
    INSERT(W, z)
  return EXTRACT-MIN(W) // Return the root of the tree

Beispiel eines optimalen Prefixcodes für die Symbole $a, ..., f$ mit den angegebenen Häufigkeiten.



Ist $D \geq 3$, hat man möglicherweise nicht immer genügend viele Symbole, so dass man jeweils D davon kombinieren kann. In einem solchen Fall fügt man *Dummy Symbols* ein. Die Dummy Symbols haben Wahrscheinlichkeit 0 und werden eingefügt um den Baum auszufüllen.

Da bei jedem Schritt die Anzahl der Knoten um $D - 1$ reduziert wird, muss die totale Anzahl Symbole $1 + k(D - 1)$ sein, wobei k die Anzahl Schritte ist. Deshalb, fügen wir genug Dummy Symbols ein, so dass die Gesamtanzahl Knoten diese Form hat. (Beachte dass maximal $D - 2$ Dummy Symbols eingefügt werden müssen)

Laufzeit-Analyse Bei Verwendung eines Min-Heaps kann ein Huffman-Baum in $\mathcal{O}(n \log n)$ konstruiert werden (*Think about it!*)

Optimale Binäre Suchbäume

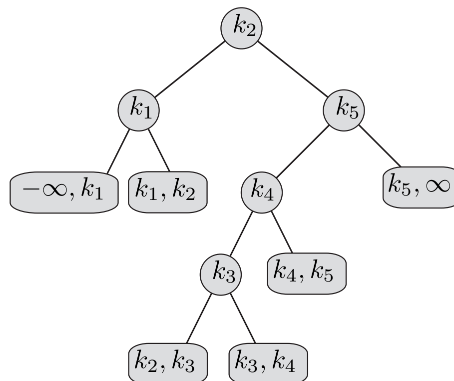
Gegeben:

- Eine Menge $S = k_1, ...k_n$ von n verschiedenen Schlüsseln, $k_1 < k_2 < ... < k_n$
- a_i ist die (absolute) Häufigkeit mit der nach k_i gesucht wird, $1 \leq i \leq n$
- (k_0, k_{n+1}) ist ein offenes Interval aller Schlüssel nach denen gesucht werden kann , sowohl erfolgreich als auch nicht erfolgreich. Es gilt $k_0 < k_1$ und $k_n < k_{n+1}$. Typische Werte sind $k_0 = -\infty$ und $k_{n+1} = +\infty$
- b_j ist die (absolute) Häufigkeit mit der ein Schlüssel $x \in (k_j, k_{j+1})$ (erfolglos) gesucht wird, $0 \leqslant j \leqslant n$

Das Ziel ist es einen optimalen Suchbaum zu finden, d.h. einen Suchbaum, der die (erwarteten) Suchkosten minimiert. Die Knoten enthalten die Schlüssel in S und die Intervalle zwischen den Schlüsseln werden durch die Blätter repräsentiert.

Beispiel: Optimaler Suchbaum für

i	0	1	2	3	4	5
a_i		15	10	5	10	20
b_i	5	10	5	5	5	10



Definitionen

Das Gewicht W eines Suchbaumes ist die summierte Häufgkeit seiner Knoten.

$$W = \sum_i a_i + \sum_j b_j$$

Die gewichtete Pfadlänge P eines Suchbaumes misst wie viele Schlüsselvergleiche total nötig sind für sowohl erfolgreiche als auch erfolglose Suchoperationen.

$$P = \sum_{i=1}^{n} (depth[k_i] + 1) \cdot a_i + \sum_{j=0}^{n} depth(leaf(k_j, k_{j+1})) \cdot b_j$$

Ein Suchbaum heisst *optimal*, falls diese totale Suchzeit P minimal ist.

$T(i,j)$	Optimaler Suchbaum für $(k_i, k_{i+1})k_{i+1} \dots k_j(k_j, k_{j+1})$
$W(i,j)$	Gewicht von $T(i,j)$, d.h. $W(i,j) = b_i + a_{i+1} + \cdots + a_j + b_j$
$P(i,j)$	gewichtete Pfadlänge von $T(i,j)$
$R(i,j)$	Wurzel von $T(i,j)$

Struktur der optimalen Lösung

Beobachtung: Jeder Teilbaum eines optimalen Suchbaums ist selbst ein optimaler Suchbaum.

Angenommen die Wurzel eines optimalen Suchbaumes hat den Schlüssel k_r. Die rekursive Definition von $S(T)$ impliziert direkt, dass der linke Teilbaum T_l ebenfalls ein optimaler Suchbaum sein muss für die Schlüssel $k_1, ..., k_{r-1}$ und die Intervalle $(-1, k_1), ..., (k_{r-1}, k_r)$. Ebenso muss der rechte Teilbaum T_r ein optimaler Suchbaum für die Schlüssel $k_{r+1}, ..., k_n$ und die Intervalle $(k_r, k_{r+1}), ..., (k_n, +1)$ sein (übliches Widerspruchs-Argument).

Wir müssen die optimale Substruktur benutzen um zu zeigen, dass man eine optimale Lösung für das Problem aus optimalen Lösungen von Teilproblemen erhalten kann.

Betrachte die Schlüssel $k_i, ..., k_j$. Einer dieser Schlüssel k_r $(i \leqslant r \leqslant j)$ ist die Wurzel eines optimalen Suchbaumes der diese Schlüssel enthält. Der linke Teilbaum enthält die Schlüssel $k_i, ..., k_{r-1}$ (und die Intervalle $(k_{i-1}, k_i), ..., (k_{r-1}, k_r)$ und der rechte Teilbaum enthält die Schlüssel $k_{r+1}, ..., k_j$ und die Intervalle $(k_r, k_{r+1}), ..., (k_j, k_{j+1})$. Solange wir alle möglichen Kandidaten $k_r, i \leqslant r \leqslant j$ für die Wurzel und alle optimales Suchbäume für $k_i, ..., k_{r-1}$ und $k_{r+1}, ..., k_j$ überprüfen ist garantiert, dass ein optimaler Suchbaum gefunden wird.

Rekursion

Wir können nun diese Überlegung rekursiv formulieren. Das Teilproblem ist das Finden eines optimalen Suchbaums für die Schlüssel $k_i, ..., k_j$ mit den entsprechenden Intervallen.

Der Dynamic Programming Algorithmus benutzt drei $(n+1) \times (n+1)$ Tabellen. Die erste $W(.,.)$ speichert die Gewichte, die zweite $P(.,.)$ speichert die gewichteten Pfadlängen und die dritte $R(.,.)$ speichert die *Wurzel* von $T(.,.)$.

Die Tabelle W, oder präziser die "rechte obere Hälfte" davon, kann rekursiv berechnet werden durch den oben erwähnten rekursiven Zusammenhang.

Für W gilt

- $W(i,i) = b_i =$"Häufigkeit von $x \in (ki, ki+1)$"
- $W(i,j) = W(i,j-1) + a_j + b_j (i < j)$

Um die gewichteten Pfadlänge $P(T)$ für den Baum T mit linkem Teilbaum T_l und rechtem Teilbaum T_r zu finden, kombinieren wir:

- die gewichteten Pfadlängen $P(T_l)$ und $P(T_r)$
- die Gewichte $W(T_l)$ und $W(T_r)$
- und a_{root}, die Häufigkeit der Wurzel

Es gilt:

- $P(T) = P(T_l) + P(T_r)$
- + Häufigkeit von Wurzel $(= a_{\text{root}})$
- + $\sum$ Häufigkeit in T_l (wie "häufig" geht Suche in linken Teil)
- + $\sum$ Häufigkeit in T_r (wie "häufig" geht Suche in rechten Teil)

Also: $P(T) = P(T_l) + P(T_r) + W(T)$

Dies führt zu der folgenden rekursiven Formel für P (wir wählen nun jeweils die optimale Aufteilung)

$P(i,i) = 0$

$P(i,j) = W(i,j) + \min_{i<l\leqslant j} \{P(i,l-1) + P(l,j)\}$

$R(i,j)$ wird benutzt um die Wahl l zu speichern, welche das minimum realisiert.

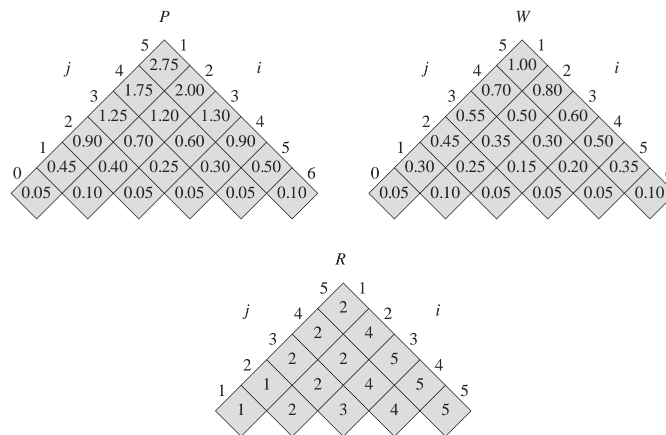
Bottom-Up

Um die gewichtete Pfadlänge für den ganzen Baum zu berechnen kann die Matrix P in drei verschiedenen Arten durchlaufen werden.

- Diagonal von der oberen linken Ecke zu der rechten unteren Ecke (oder vice versa)
- Zeile für Zeile bottom up, jede Zeile von links nach rechts (s. Algorithmus)
- Spaltenweise

Wichtig ist, dass alle Teilprobleme gelöst worden sind, bevor sie benutzt werden. Egal welche Ordnung verwendet wird, der Algorithmus braucht $\mathcal{O}(n^3)$ Laufzeit und $\mathcal{O}(n^2)$ Speicher.

Ausgefüllte Matrizen für obiges Beispiel (unter Verwendung der relativen Häufigkeiten (hier $\frac{a_i}{100}$ resp. $\frac{b_i}{100}$):




```
OPTIMAL_BINARY_SEARCHTREE(A[1..n], B[0..n])
  // Initialisierung
  for i = 0 to n
      W[i, i] = B[i]
      for j = i + 1 to n
          W[i, j] = W[i, j − 1] + A[j] + B[j]

  // Berechnung
  for i = n − 1 downto 0
      P[i, i] = 0
      for j = i + 1 to n
          COMPUTE_P_AND_R(i, j)


COMPUTE_P_AND_R(i, j)
  P[i, j] = ∞
  for r = i + 1 to j
      tmp = P[i, r − 1] + P[r + 1, j]
      if P[i, j] > tmp
          P[i, j] = tmp
          R[i, j] = r
  P[i, j] = P[i, j] + W[i, j]
```

Anmerkungen

Es gibt einen besseren Algorithmus der in $\mathcal{O}(n^2)$ läuft, der von D. Knuth vorgestellt wurde und Monotonieüberlegungen verwendet um den Algorithmus zu beschleunigen. Ein Verweis findet sich in `http://www.cs.uiuc.edu/class/fa05/cs473g/lectures/04-dynprog.pdf`.

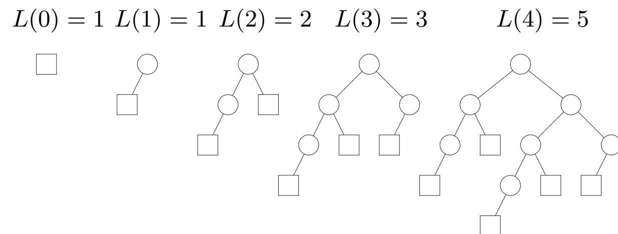
Bei einer leichten Modifikation der Problemstellung ist eine noch schnellere Lösung in $\mathcal{O}(n \log n)$ möglich (T. C. Hu and A. C. Tucker), vergleiche hierzu *Optimal Computer Search Trees and Variable-Length Alphabetical Codes* (`http://goo.gl/sPIDo`)

Stop and Think: The Best Among the Balanced

Betrachte das verallgemeinerte Problem, dass gegeben die Schlüssel und die Zugriffshäufigkeiten, der optimale *AVL*-Suchbaum gefunden werden soll. Was funktioniert im obigen Algorithmus nun nicht mehr? Wie kann der Algorithmus angepasst werden? Was ist die Laufzeit und der Speicherverbrauch des neuen Algorithmus?

Stop and Think: The Worst Among the Balanced

Gesucht ist der "schlechteste" AVL-Baum: Finde den kleinsten (wenigsten Knoten) AVL-Baum, der eine Höhe i hat. *Hint*: Wie kriegt man den kleinsten AVL-Baum der Höhe i aus den kleinsten AVL Bäumen der Höhen $i-1$ und $i-2$? Es sei $L(i)$ die Anzahl Blätter des kleinsten AVL Baumes der Höhe i. Was ist $L(i)$?



Brain Teaser: Kentucky Derby

Von 25 Rennpferden, von denen jedes eine konstante unbekannte Geschwindigkeit hat, die unterschiedlich ist zu jedem anderen Pferd (Geschwindigkeiten sind paarweise verschieden) sind die 3 schnellsten zu finden. Da die Rennstrecke nur 5 Bahnen hat, können in jedem Rennen höchstens 5 Pferde gegeneinander antreten. Wie viele Runden sind notwendig um die 3 schnellsten zu identifizieren (wenn man aus einem Rennen nur die relative Geschwindigkeit der Pferde (also das Ranking) untereinander ermitteln kann)?