Teil 2: Sortieren

Strukturierte Schlüssel

Strukturierte Schlüssel

Bislang haben wir mit allgemeinen Schlüsseln gearbeitet. Wir konnten zwei miteinander vergleichen, aber es wurde keinerlei Information über ihre Struktur benutzt.

In vielen Fällen sind die Schlüssel aber stark strukturiert, d.h. besitzen eine ganz bestimmte Form, die zur Sortierung ausgenutzt werden kann.

Definition (Strukturierte Schlüssel)

$(\Sigma, \leq)$ sei ein geordnetes Alphabet, d.h. eine endliche, nicht-leere Menge mit einer linearen Ordnung $\leq$. Ein strukturierter Schlüssel S (über Σ) ist ein Wort aus

$$\Sigma^* := \{x_1 \dots x_k \mid k \in \mathbb{N} \text{ und } x_i \in \Sigma\}.$$

Beispiele:

1. $\Sigma = \{a \leq b \leq c \leq d \leq \cdots \leq z\}$ und $S_1 = quicksort, S_2 = bubblesort$.
2. $\Sigma = \{0 \leq 1 \leq \cdots \leq 9\}$ und $S_1 = 12345, S_2 = 314159, S_3 = 42$.

Die Lexikographische Ordnung

Definition (Die Lexikographische Ordnung)

$(\Sigma, \leq)$ sei ein geordnetes Alphabet und $x = x_1 \dots x_k$ und $y = y_1 \dots y_l$ seien zwei Wörter über Σ. Wir definieren die lexikographische Ordnung $<_{lex}$ auf Σ^* so, dass $x <_{lex} y$ genau dann, wenn

1. $x_i = y_i$ für $1 \leq i \leq k < l$ oder
2. es existiert ein $i, 1 \leq i \leq k, l$ mit $x_j = y_j$ für $1 \leq j < i$ und $x_i < y_i$.

D.h. entweder ist x ein echtes Präfix von y oder für das erste Zeichen x_i, y_i in dem sich x und y unterscheiden gilt $x_i < y_i$.

Die Lexikographische Ordnung - Beispiele

1. Auf $\Sigma = \{a, \ldots, z\}$ mit der alphabetischen Ordnung ergibt sich die „normale“ Ordnung:

$$a < aa < aaa < ab < ba < baa < bba < bbz$$

2. Auf $\Sigma = \{0, 1, \ldots, 9\}$ ergibt sich eine etwas andere Ordnung als „erwartet“

$$1 < 10 < 100 < 2 < 20 < 2234903 < 26 < 3$$

aber

$$0000001 < 0000002 < 0000003 < 0000010 < 2234903,$$

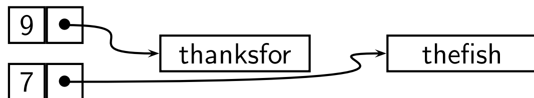
d.h. füllt man die zu betrachtenden Zahlen mit führenden Nullen auf die gleiche Länge auf, so ist die lexikographische Ordnung gleich der numerischen.

Speicherung von Wörtern

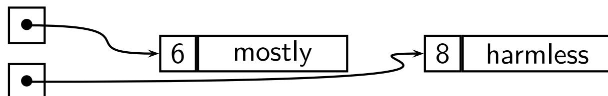
Strukturierte Schlüssel werden, wie allgemeine, nicht direkt verwendet. Sondern wir gehen davon aus, dass die eigentlichen Schlüssel als Zeichensequenzen an bestimmten, festen Stellen gespeichert werden, und im Rahmen der Algorithmen nur Verweise, bzw. Pointer auf die eigentlichen Inhalte „bewegt“ werden.

Dabei gibt es zwei mögliche Modelle:

1. Entweder enthält der Verweis auch die Länge des Strings,



2. oder die Länge ist beim String abgelegt.



Teil 2: Sortieren

Strukturierte Schlüssel

RADIXSORT

Das Sortieren von Buchstaben

Im Folgenden sei unser geordnetes Alphabet gegeben als $\Sigma := \{1, \ldots, m\}$ mit $m \in \mathbb{N}$ und der numerischen Ordnung.

Zuerst wollen wir uns mit der Sortierung von Wörtern $x^{(1)}, \ldots, x^{(n)}$ gleicher Länge l beschäftigen.

Betrachten wir zuerst den Fall $l = 1$, d.h. jedes Wort besteht nur aus einem Buchstaben $x^{(i)} = x_1^{(i)}$.

Idee

$B[1 \ldots m]$ sei ein Feld von Fächern (Buckets), d.h. jeder Eintrag ist (ein Verweis auf) eine Liste von Wörtern.

Für $i = 1, \ldots, n$ wirf das Element $x^{(i)}$ in das Fach $B[x_1^{(i)}]$, d.h. hänge es an das Ende der Liste an.

Anschließend enthält das Fach $B[j]$ alle Wörter mit $x^{(i)} = j$.

Um die sortierte Liste zu erhalten, hänge einfach die Listen $B[1], B[2], \ldots, B[m]$ in dieser Reihenfolge aneinander.

Das Sortieren von Buchstaben

Beobachtung

Dadurch, dass wir die Schlüssel an das Ende der Listen anhängen erhalten wir ein stabiles Verfahren, d.h. die Reihenfolge gleicher Schlüssel wird nicht verändert.

Frage

Wie lang benötigt dieses Verfahren?

1. Das Anhängen eines Schlüssels an eine Liste dauert $O(1)$, d.h. $O(n)$ für das Einfügen aller Schlüssel.
2. Das Zusammenfügen der Listen dauert $O(m)$.

Insgesamt ergibt sich somit eine Laufzeit von $O(n + m)$.

Wörter gleicher Länge

Wir wollen dieses Verfahren nun auf n Wörter $x^{(1)}, \ldots, x^{(n)}$ gleicher Länge $l \geq 1$ erweitern, d.h.

$$x^{(i)} = x_1^{(i)} \ldots x_l^{(i)}.$$

Wir könnten die Wörter mittels des oben vorgestellten Verfahrens nach dem ersten Buchstaben sortieren.

Dadurch erhalten wir $\leq m$ Gruppen, die wir nach dem zweiten Buchstaben sortieren usw.

Der wesentliche **Nachteil** dieses Verfahrens ist die starke Fragmentierung. Die zu sortierenden Gruppen werden immer kleiner, aber die Anzahl der Fächer bleibt, unabhängig von der Größe der Fragmente, immer m. Dadurch kann sich eine Laufzeit von $\Omega(n \cdot l \cdot m)$ ergeben.

RADIXSORT

Besser ist der folgende Ansatz:

Algorithmus (RADIXSORT)

Eingabe: *Wörter* $x^{(1)}, \ldots, x^{(n)}$ *der Länge* l *über* Σ
Ausgabe: *Die lexikographisch sortierte Folge der Wörter*
Daten: *Eine Liste* L *von Wörtern und ein Feld* $B[1 \ldots m]$ *von Wortlisten*

$L := (x^{(1)}, \ldots, x^{(n)})$
für $k = l \ldots 1$ **tue**
  Leere alle Listen $B[i]$
  solange L *nicht leer ist* **tue**
    Entferne das **erste** *Element* y *aus* L
    Hänge y *an die Liste* $B[y_k]$ *an*
  Ende
  $L := (B[1], \ldots, B[m])$
Ende

RADIXSORT

Satz (Laufzeit von RADIXSORT)

RADIXSORT benötigt zum Sortieren von n Wörtern der Länge l über einem geordneten Alphabet Σ mit m Buchstaben $O(l(n+m))$ Zeit.

Beweis.

Der Algorithmus führt insgesamt l Läufe über alle Wörter durch. jeder Lauf benötigt $O(n+m)$ Zeit. Damit ergibt sich die insgesamt benötigte Zeit als $O(l(n+m))$. □

RADIXSORT - Ein Beispiel

Wir betrachten ein Beispiel mit $m = 4, n = 5, l = 3$:

$$123, 124, 223, 324, 321$$

k	**Fach 1**	**Fach 2**	**Fach 3**	**Fach 4**
3	321		123,223	124,324
2		321,123,223,124,324		
1	123,124	223	321,324	
Ergebnis	123, 124, 223 ,321, 324			

Die Behandlung leerer Fächer

Beobachtung

Es werden nicht immer alle Fächer benutzt. Kann dies ausgenutzt werden?

Es wäre günstig zu wissen welche Fächer leer, bzw. nicht-leer sind. Dann würde sich unter Umständen die Laufzeit für das Zusammenfügen der einzelnen Fächer reduzieren (da man leere Fächer überspringen kann).

Dabei ist aber darauf zu achten, dass die notwendige Buchhaltung nicht aufwändiger wird als der Gewinn.

Um zu erfahren, welche Buchstaben an welchen Positionen auftreten bilden wir Paare

$$(j, x_j^{(i)}) \text{ mit } 1 \leq j \leq l \text{ und } 1 \leq i \leq n,$$

d.h. ein Paar (j, k) existiert, wenn es ein Wort $x^{(i)}$ gibt mit $x_j^{(i)} = k$.

Die Behandlung leerer Fächer

Diese Paare Sortieren wir zweimal:

1. Zuerst mittels der Fachverteilung nach der zweiten Komponente, d.h. dem Buchstaben.
2. Anschließend nach der ersten Komponente.

Danach liegt im j-ten Fach mit $1 \leq j \leq l$ die sortierte Liste der Buchstaben, die an Position j vorkommen in entsprechender Häufigkeit.
Durch Löschen der Duplikate erhält man für jede Runde die Liste der nicht-leeren Fächer.

Die Behandlung leerer Fächer

Für die Wörter 123, 124, 223, 324, 321 ergeben sich die folgenden Paare:

$2 \times (1,1), 5 \times (2,2), 2 \times (3,3), 2 \times (3,4), 1 \times (1,2), 2 \times (1,3), 1 \times (3,1)$

Nach der ersten Sortierung ergibt sich:

1	(1,1),(1,1),(3,1)
2	(2,2),(2,2),(2,2),(2,2),(2,2),(1,2)
3	(3,3),(3,3),(1,3),(1,3)
4	(3,4),(3,4)

Nach der zweiten Sortierung und dem Löschen der Duplikate ergibt sich:

1	(1,1),(1,2),(1,3)
2	(2,2)
3	(3,1),(3,3),(3,4)

Die Behandlung leerer Fächer

Die Laufzeiten sind:

1. $O(nl + m)$ für die erste Sortierung (nl Wörter, m Fächer)
2. $O(nl + l)$ für die zweite Sortierung (nl Wörter, l Fächer)

Insgesamt also $O(2nl + m + l) = O(nl + m)$.

Mit diesen Listen, können während des Zusammenfügens die leeren Fächer übersprungen werden, bzw. gezielt die nicht-leeren Fächer hintereinander gehängt werden.

Teil 2: Sortieren

Strukturierte Schlüssel

BUCKETSORT

Wörter beliebiger Länge

Wir wollen jetzt die oben beschriebenen Techniken dazu verwenden Wörter mit beliebigen Längen zu sortieren. D.h. unsere Eingabe besteht aus Wörtern

$$x^{(1)}, \ldots, x^{(n)} \text{ mit } |x^{(i)}| = l_i.$$

Die Grundidee basiert auf der Beobachtung, dass kurze Wörter nicht zu früh bearbeitet werden dürfen.

Konkret heißt dies, dass ein Wort der Länge l erst in die Sortierung einfließen darf, wenn nach dem l-ten Buchstaben sortiert wird. Vorher (d.h. weiter hinten) enthält es keine Zeichen, nach denen es sortiert werden kann.

Algorithmus (BUCKETSORT)

Eingabe: *Wörter* $x^{(i)}$ *für* $1 \leq i \leq n$

Ausgabe: *Eine lexikographisch sortierte Sequenz der Wörter*

1. *Bestimme die Länge* l_i *jedes Wortes* $x^{(i)}$ *und erzeuge Paare* $(l_i, x^{(i)})$. *Bestimme gleichzeitig das Maximum* l_{max} *der Längen.*
2. *Sortiere die Paare* $(l_i, x^{(i)})$ *durch Fachverteilung mit* l_{max} *Fächern nach der ersten Komponente und speichere die resultierende Liste der Längen in* $Länge[k]$ *für* $0 \leq k \leq l_{max}$.
3. *Erzeuge* $L := \sum_{i=1}^{n} l_i$ *Paare* $(j, x_j^{(i)})$ *für* $1 \leq i \leq n, 1 \leq j \leq l_i$.
4. *Erzeuge aus diesen Paaren nach dem oben beschriebenen Verfahren die Listen* $Nichtleer[j]$ *von nicht-leeren Fächern in Runde* j
5. *Sortiere die Wörter mittels DO_BUCKETSORT*

Algorithmus (DO_BUCKETSORT)

Eingabe: *Wörter $x^{(i)}$, die Listen Länge[k] und Nichtleer[j].*
Ausgabe: *Die lexikographisch sortierte Sequenz der Wörter.*

$W := \emptyset$ `// ∅ = leere Liste`
für $k = 1 \dots m$ **tue** $B[k] = \emptyset$
für $j = l_{\max} \dots 1$ **tue**
  $W = \textit{hänge_an}(\textit{Länge}[j], W)$ `// Füge Wörter der Länge j hinzu`
  `// Verteile Wörter der Länge ≥ j`
  solange $W \neq \emptyset$ **tue**
    $x = \textit{lösche_erstes}(W)$
    $B[x_j] = \textit{hänge_an}(B[x_j], x)$
  Ende
  `// Konkateniere nicht-leere Listen`
  solange $\textit{Nichtleer}[j] \neq \emptyset$ **tue**
    $k = \textit{lösche_erstes}(\textit{Nichtleer}[j])$
    $W = \textit{hänge_an}(W, B[k])$
    $B[k] = \emptyset$
  Ende
Ende

BUCKETSORT - Ein Beispiel

Wir sortieren: b, abc, ac, abb, ab.

Länge	
1	b
2	ac, ab
3	abc, abb

Nichtleer	
1	a, b
2	b, c
3	b, c

l	W	a	b	c
3	abc,abb		abb	abc
2	ac,ab,abb,abc		ab,abb,abc	ac
1	b,ab,abb,abc,ac	ab,abb,abc,ac	b	
Ergebnis	ab,abb,abc,ac,b			

Die Korrektheit von BUCKETSORT

Wir beweisen die folgende Invariante des Algorithmus:

> *Vor dem Durchlauf der zweiten Schleife mit Wert j, enthält W alle Wörter der Länge $\geq j+1$, sortiert in lexikographischer Ordnung nach den an Position $j+1$ beginnenden Suffixen.*

Zu Beginn ist W leer und somit ist die Invariante für $j = l_{\text{max}}$ erfüllt.

W erfülle nun die Invariante nach dem Durchlauf mit Wert $j+1$.

Zuerst werden alle Wörter der Länge j vor W gehängt, d.h. W enthält anschließend alle Wörter der Länge $\geq j$, sortiert ab der $j+1$-ten Position.

Danach wird nach dem j-ten Buchstaben sortiert. Dabei bleibt die relative Ordnung von W ab dem $j+1$-ten Buchstaben erhalten (stabile Sortierung).

Anschließend wird W zur Konkatenation der nicht-leeren Listen. Dadurch sind die Wörter in W ab Position j lexikographisch sortiert.

Die Laufzeit von DO_BUCKETSORT

Für die einzelnen Abschnitte ergeben sich die folgenden Laufzeiten:

1. Vorbereitung der Buckets: $O(m)$
2. j wird insgesamt $l_{\max}$-mal erhöht (zweite Schleife).
3. Die Liste W wird insgesamt $O(l_{\max})$- Male verlängert.
4. Das Wort $x^{(i)}$ wird l_i mal auf Fächer verteilt, d.h. insgesamt $O(L)$ Verteilungen.
5. Während der Konkatenierung der nicht-leeren Fächer, kann jedes Fach einem Buchstaben an der Position j eines bestimmten Wortes zugewiesen werden. Daher wird insgesamt höchstens $O(L)$ Zeit benötigt.

Damit ergibt sich die Laufzeit von DO_BUCKETSORT als
$O(m + l_{\max} + L) = O(m + L)$.

Die Laufzeit von BUCKETSORT

Es ergeben sich die folgenden Laufzeiten:

1. Erzeugung der Längen-Paare: $O(n)$
2. Sortierung der Längen-Paare: $O(n + l_{\max}) = O(n + L)$
3. Erzeugung der Positions-Buchstaben-Paare: $O(L)$
4. Sortierung der Positions-Buchstaben-Paare: $O(L + m)$
5. Laufzeit von DO_BUCKETSORT: $O(m + L)$

Satz

BUCKETSORT sortiert n Wörter der Gesamtlänge L über einem geordneten Alphabet der Größe m in Zeit $O(m + L)$.

Sortieren in linearer Zeit!

Konsequenz

Durch die Verwendung der Struktur der Schlüssel (wenn möglich), kann die prinzipielle untere Schranke von $\Omega(n \log n)$ Schlüsselvergleichen für die Sortierung durchbrochen werden.

Frage

Kann dieses Verfahren auf nicht-diskrete Schlüssel (d.h. unendliche Worte) erweitert werden?

Teil 2: Sortieren

Sortierung reeller Zahlen durch Fachverteilung

HYBRIDSORT

HYBRIDSORT

Problem

Eingabe: $x_1, \ldots, x_n \in (0, 1]$ *(durch Skalierung erreichbar)*
Sortiere die Sequenz der Zahlen.

Algorithmus (HYBRIDSORT)

1. *Lege k leere Fächer $B[1 \ldots k]$ an.*
2. *Hänge jedes x_i an das Fach $B[\lceil kx_i \rceil]$ an.*
3. *Sortiere jedes Fach einzeln mittels HEAPSORT.*
4. *Hänge die sortierten Inhalte der Fächer in sortierter Folge aneinander.*

Die Korrektheit von HYBRIDSORT ist offensichtlich.

HYBRIDSORT - die Laufzeit

Satz

Sei $0 < \alpha < 1$ konstant und „klein“ gewählt. HYBRIDSORT benötigt zur Sortierung von n reellen Zahlen $x_1, \ldots, x_n$ im Intervall $(0, 1]$ mit $k = \lceil \alpha n \rceil$ Fächern im schlechtesten Fall $O(n \log n)$.

Wenn die x_i unabhängig und gleichmäßig über $(0, 1]$ verteilt sind, dann hat HYBRIDSORT die mittlere Laufzeit $O(n)$.

Beweis

Das Anlegen der Fächer und die Einteilung der Werte benötigt $O(k + n)$.

Anschließend seien t_i Elemente im i-ten Fach für $1 \leq i \leq k$.

Mit der Vereinbarung $0 \log 0 = 0$ ergeben sich die Kosten der Sortierung der einzelnen Fächer mittels HEAPSORT im schlechtesten Fall als:

$$\sum_{i=1}^{k} t_i \log t_i \leq \sum_{i=1}^{k} t_i \log n = n \log n.$$

Beweis (Fortsetzung)

Um die mittlere Laufzeit zu ermitteln, müssen wir die erwartete Anzahl von Schlüsseln pro Fach berechnen.

Wir nehmen an, dass die Werte $x_1, \ldots, x_n$ *über das Intervall* $(0, 1]$ *gleichverteilt sind. D.h. die Wahrscheinlichkeit, dass ein Wert* x_j *im Fach* $(\frac{i-1}{k}, \frac{i}{k}]$ *mit* $1 \leq i \leq k$ *liegt ist*

$$P\left(\frac{i-1}{k} < x_j \leq \frac{i}{k}\right) = \frac{1}{k}.$$

Wir betrachten die folgenden Zufallsvariablen:

$$B_i = \text{ Anzahl der Element in Fach } i.$$

Beweis (Fortsetzung)

Da es sich um eine n-fache Wiederholung eines Bernoulli-Experimentes mit k verschiedenen Ausgängen jeweils mit Erfolgswahrscheinlichkeit $\frac{1}{k}$ handelt, handelt es sich um eine Multinomialverteilung:

$$P(B_1 = h_1, \ldots, B_k = h_k) = \frac{n!}{h_1! \ldots h_k!} \left(\frac{1}{k}\right)^n.$$

Damit ergibt sich die mittlere Laufzeit A für die Sortierung der Fächer mittels HEAPSORT als:

$$\begin{aligned} A &= \sum_{\sum h_i = n} P(B_1 = h_1, \ldots, B_k = h_k) \sum_{i=1}^{k} O(h_i \log h_i) \\ &= O\left(\frac{1}{k^n} \sum_{\sum h_j = n} \frac{n!}{h_1! \ldots h_k!} \sum_{i=1}^{k} h_i \log h_i\right) \end{aligned}$$

Beweis (Fortsetzung)

Wir betrachten diesen Term genauer:

$$A' = \frac{1}{k^n} \sum_{\sum h_j = n} \frac{n!}{h_1! \dots h_k!} \sum_{i=1}^{k} h_i \log h_i = \frac{1}{k^n} \sum_{i=1}^{k} \sum_{\sum h_j = n} h_i \log h_i \frac{n!}{h_1! \dots h_k!}$$

Da für $h_i = 0, 1$ der Ausdruck $h_i \log h_i$ Null ist, erhalten wir:

$$A' = \frac{1}{k^n} \sum_{i=1}^{k} \sum_{\substack{\sum h_j = n \\ h_i \geq 2}} h_i \log h_i \frac{n!}{h_1! \dots h_k!}$$

Wir betrachten nun einen der Summanden:

$$\begin{aligned} h_i \log h_i \frac{n!}{h_1! \dots h_k!} &\leq h_i^2 \frac{n!}{h_1! \dots h_k!} = (h_i(h_i - 1) + h_i) \frac{n!}{h_1! \dots h_k!} \\ &= h_i(h_i - 1) \frac{n!}{h_1! \dots h_k!} + h_i \frac{n!}{h_1! \dots h_k!} \end{aligned}$$

Beweis (Fortsetzung)

Da $h_i \geq 2$ ergibt sich für den ersten Teil dieses Summanden

$$h_i(h_i-1)\frac{n!}{h_1!\dots h_k!} = h_i(h_i-1)\frac{n(n-1)(n-2)!}{h_1!\dots h_i(h_i-1)(h_i-2)!\dots h_k!}$$
$$= n(n-1)\frac{(n-2)!}{h_1!\dots(h_i-2)!\dots h_k!}$$

Für den zweiten Teil ergibt sich:

$$h_i\frac{n!}{h_1!\dots h_k!} = h_i\frac{n(n-1)!}{h_1!\dots h_i(h_i-1)!\dots h_k!}$$
$$= n\frac{(n-1)!}{h_1!\dots(h_i-1)!\dots h_k!}$$

Beweis (Fortsetzung)

Damit ergibt sich

$$
\begin{aligned}
A' &\leq \frac{1}{k^n} \sum_{i=1}^{k} \sum_{\substack{\sum h_j = n \\ h_i \geq 2}} \left(n(n-1) \frac{(n-2)!}{h_1! \ldots (h_i - 2)! \ldots h_k!} \right. \\
&\qquad \left. + n \frac{(n-1)!}{h_1! \ldots (h_i - 1)! \ldots h_k!} \right) \\
&\leq \frac{n(n-1)}{k^n} \sum_{i=1}^{k} \sum_{\sum h_j = n-2} \frac{(n-2)!}{h_1! \ldots (h_i - 2)! \ldots h_k!} \\
&\qquad + \frac{n}{k^n} \sum_{i=1}^{k} \sum_{\sum h_j = n-1} \frac{(n-1)!}{h_1! \ldots (h_i - 1)! \ldots h_k!}
\end{aligned}
$$

$\ldots$

Beweis (Fortsetzung)

$$
\begin{aligned}
\dots \leq \frac{n(n-1)}{k^n} \sum_{i=1}^{k} \sum_{\sum h_j = n-2} \frac{(n-2)!}{h_1! \dots (h_i-2)! \dots h_k!} \\
+ \frac{n}{k^n} \sum_{i=1}^{k} \sum_{\sum h_j = n-1} \frac{(n-1)!}{h_1! \dots (h_i-1)! \dots h_k!} \\
= \frac{n(n-1)}{k^2} \sum_{i=1}^{k} \sum_{\sum h_j = n-2} \frac{1}{k^{n-2}} \frac{(n-2)!}{h_1! \dots (h_i-2)! \dots h_k!} \\
+ \frac{n}{k} \sum_{i=1}^{k} \sum_{\sum h_j = n-1} \frac{1}{k^{n-1}} \frac{(n-1)!}{h_1! \dots (h_i-1)! \dots h_k!}
\end{aligned}
$$

Beweis (Fortsetzung)

Die Summe

$$\sum_{\sum h_j = n-2} \frac{(n-2)!}{h_1! \ldots (h_i - 2)! \ldots h_k!} \frac{1}{k^{n-2}}$$

Ist die Summe aller Wahrscheinlichkeiten einer Multinomialverteilung mit $n-2$ *Experimenten und* k *gleichwahrscheinlichen Ausgängen. Damit ist sie* 1.

Das Gleiche gilt für die andere Summe, was insgesamt zu Folgendem führt:

$$A' \leq \frac{n(n-1)}{k^2} \sum_{i=1}^{k} 1 + \frac{n}{k} \sum_{i=1}^{k} 1 = \frac{n(n-1)}{k^2} k + \frac{n}{k} k = \frac{n(n-1)}{k} + n$$

Da $k = \lceil \alpha n \rceil$ *für ein* $0 < \alpha < 1$ *gewählt wurde, erhalten wir*

$$A = O(A') = O(\frac{n(n-1)}{\alpha n} + n) = O(\frac{n}{\alpha} + n) = O(n). \quad \square$$

Sortieren von reellen Zahlen in linearer Zeit?

Während der Analyse von HYBRIDSORT haben wir implizit diverse Annahmen gemacht:

- Die zu sortierenden Zahlen waren auf $(0, 1]$ skaliert.
- Die arithmetischen Operationen benötigen nur $O(1)$ Zeit.
- Das Aufrunden von reellen Zahlen benötigt nur $O(1)$ Zeit.

Dabei spielt insbesondere das Runden von reellen Zahlen eine kritische Rolle, denn es ist nicht mittels der vier Grundrechenarten $+, -, *, \div$ darstellbar.

Konsequenz

Ist das Runden reeller Zahlen nicht gestattet, so benötigt das Sortieren von n reellen Zahlen $\Omega(n \log n)$ Schritte im Mittel und im schlechtesten Fall.

Rationale Entscheidungsbäume

Um dies einzusehen, betrachten wir Rationale Entscheidungsbäume. Diese sind, wie die Entscheidungsbäume für vergleichsbasierte Verfahren, binäre Bäume, in denen jedem Knoten eine Menge von (noch) möglichen Permutationen zugeordnet ist.

Allerdings wird die Entscheidung, in welchen Zweig der Algorithmus läuft, nicht mittels eines direkten Vergleiches zweier Schlüssel getroffen, sondern mittels einer rationalen Funktion $B(x_1, \ldots, x_n)$, d.h. einer Funktion auf n Variablen, die nur die vier Grundrechenarten $+, -, *, \div$ verwendet.

Gilt $B(x_1, \ldots, x_n) \geq 0$ wird das eine Kind gewählt, gilt $B(x_1, \ldots, x_n) < 0$ das andere.

Dadurch ergibt sich wieder ein Baum, der mindestens $n!$ viele Blätter besitzen muss (eines für jede mögliche Eingabereihenfolge).

Das führt, wie bei den vergleichsbasierten Verfahren, wieder dazu, dass der Entscheidungsbaum im schlechtesten Fall und im Mittel eine Tiefe von $\Omega(n \log n)$ hat.