Wir analysieren nun den durchschnittlichen Fall. Im durchschnittlichen Fall treten bessere und schlechtere Aufteilungen gemischt auf. Wir gehen, wie immer bisher, davon aus dass alle Permutationen des Eingabedatenfelds gleich wahrscheinlich sind. Weiters nehmen wir zur Vereinfachung der Analyse an, dass die Schlüssel paarweise unterschiedlich sind. Dann können wir zeigen:

Satz 9.2. *Die Laufzeit von Quicksort im durchschnittlichen Fall ist $O(n \log n)$.*

Beweis. Ein Aufruf von Quicksort auf einem Datenfeld der Länge n führt zu höchstens n Aufrufen der Prozedur TEILEN. Der Aufwand eines Aufrufs von TEILEN ist $\Theta(v)$ wobei v die Anzahl der durchgeführten Vergleiche ("$A[j] \leq x$") ist. Weiters wird ja immer mit dem Pivotelement verglichen das in späteren Aufrufen von TEILEN keine Rolle mehr spielt, also wird kein Paar zwei Mal verglichen. Sei X die Anzahl der Vergleiche die über alle Aufrufe von TEILEN hinweg insgesamt durchgeführt werden. Dann ist die Gesamtlaufzeit von Quicksort $O(X + n)$. Wir wollen nun $\mathrm{E}X$ bestimmen.

Seien dazu $\{z_1, \ldots, z_n\}$ die Elemente des Eingabedatenfeldes mit $z_1 < \cdots < z_n$. Für $i \leq j$ sei weiters $Z_{i,j} = \{z_i, \ldots, z_j\}$. Der Algorithmus beginnt mit der Auswahl eines Pivotelements z_k aus $Z_{1,n}$ das mit $z_1, \ldots, z_{k-1}, z_{k+1}, \ldots, z_n$ verglichen wird um die erste Teilung des Datenfelds zu berechnen. Nach Teilung des Datenfelds werden $z_1, \ldots, z_{k-1}$ sowie $z_{k+1}, \ldots, z_n$ nur noch untereinander verglichen, nicht aber miteinander. Dieser Prozess wiederholt sich dann.

Allgemein können wir also feststellen, dass z_i mit z_j verglichen wird genau dann wenn z_i oder z_j als erstes in $Z_{i,j}$ als Pivotelement gewählt wird und damit also kein (früher gewähltes) Pivotelement zwischen z_i und z_j steht. Sei

$$X_{i,j} = \begin{cases} 1 & \text{falls } z_i \text{ mit } z_j \text{ verglichen wird} \\ 0 & \text{sonst} \end{cases}$$

Dann ist $X = \sum_{1 \leq i < j \leq n} X_{i,j}$ und wir erhalten

$$\begin{aligned}
\mathrm{E}X &= \sum_{1 \leq i < j \leq n} \mathrm{E}X_{i,j} = \sum_{1 \leq i < j \leq n} \mathrm{W}(z_i \text{ wird mit } z_j \text{ verglichen}) \\
&= \sum_{1 \leq i < j \leq n} \mathrm{W}(z_i \text{ oder } z_j \text{ ist erstes Pivotelement aus } Z_{i,j}) \\
&= \sum_{1 \leq i < j \leq n} \left(\frac{1}{j-i+1} + \frac{1}{j-i+1} \right) \\
&= 2 \sum_{1 \leq i < j \leq n} \frac{1}{j-i+1} \\
&= 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k+1} \\
&< 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k} \\
&= 2 \sum_{i=1}^{n-1} O(\ln n) \\
&= O(n \log n).
\end{aligned}$$

Somit ist auch die Gesamtlaufzeit im durchschnittlichen Fall $O(n \log n)$. □

Um diese durchschnittliche Laufzeit als erwartete Laufzeit auch auf ungünstigen Eingaben zu garantieren bietet sich die in Algorithmus 40 angegebene Randomisierung an. Hier wird die

Algorithmus 40 Randomisiertes Quicksort

```
Prozedur QuicksortRandomisiert(A, l, r)
    Falls l < r dann
        i := Zufall(l, r)
        Vertausche A[i] und A[r]
        m := Teilen(A, l, r)
        Quicksort(A, l, m − 1)
        Quicksort(A, m + 1, r)
    Ende Falls
Ende Prozedur
```

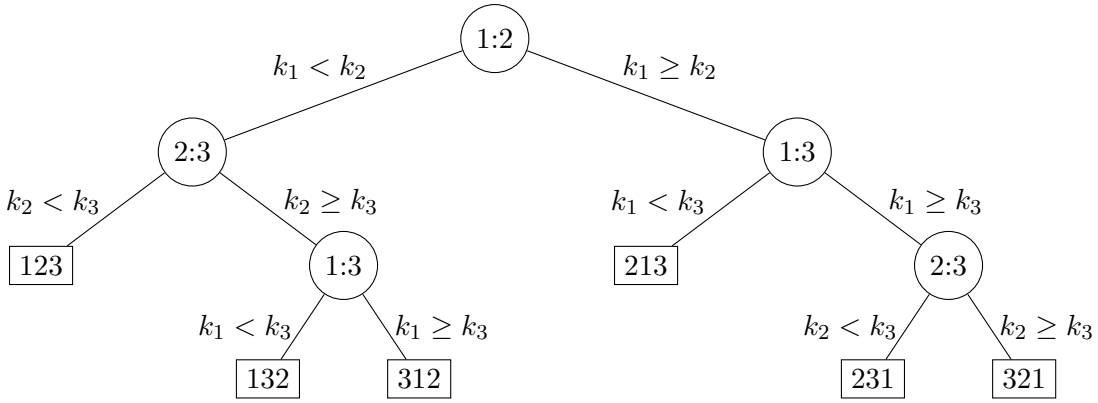
Auswahl eines zufälligen Elements als Pivotelement dadurch realisiert, dass das zufällig gewählte Element mit $A[r]$ vertauscht wird bevor die Teilung des Datenfelds durchgeführt wird. Ein zufällig gewähltes Pivotelement teilt das aktuelle Teildatenfeld mit großer Wahrscheinlichkeit gut auf. Diese randomisierte Variante von Quicksort hat, unabhängig vom Eingabedatenfeld, die erwartete Laufzeit $O(n \log n)$.

9.3 Eine untere Schranke auf Sortieralgorithmen

Wir haben nun einige Sortierverfahren gesehen deren Laufzeit immer mindestens $n \log n$ war. Es ist natürlich zu fragen, ob es auch besser möglich ist. Eine positive Antwort auf diese Frage bestünde in einem schnelleren Sortierverfahren (das in gewissen Spezialfällen tatsächlich existiert, siehe unten). Eine negative Antwort darauf zu geben fällt aber viel schwerer da ein Beweis über alle Sortierverfahren notwendig wäre. Wir werden hier eine untere Schranke für ein bestimme *Klasse von Sortierverfahren* beweisen: für vergleichsbasierte Sortierverfahren. Diese zeichnen sich dadurch aus, dass sie lediglich Schlüsselvergleiche verwenden können, um die korrekte Reihenfolge der Eingabedaten zu bestimmen (nicht aber z.B. Schlüssel addieren können). Alle bisher in dieser Vorlesung betrachteten Sortierverfahren sind von dieser Art.

Die Entscheidungen die ein vergleichsbasiertes Sortierverfahren trifft können durch einen Entscheidungsbaum dargestellt werden. Bei einer Eingabe von Schlüssel $k_1, \ldots k_n$ stellt ein Knoten mit Beschriftung $i; j$ einen Vergleich von k_i mit k_j dar. Ein Blatt stellt dann die sortierende Permutation der Eingabe dar. Auf diese Weise induziert ein vergleichsbasiertes Sortierverfahren eine Folge $(T_n)_{n\geq 2}$ von Entscheidungsbäumen. Der Entscheidungsbaum T_n hat dann $n!$ Blätter, denn einerseits muss ja jede Permutation vorkommen, da sie als Eingabe möglich ist. Andererseits unterscheiden sich die Permutationen an zwei verschiedenen Blättern und so kann keine Permutation zwei Mal vorkommen.

Beispiel 9.2. Ein Entscheidungsbaum eines vergleichsbasierten Sortierverfahrens für $n = 3$ ist:



Wir können dann zeigen:

Satz 9.3. *Jedes vergleichsbasierte Sortierverfahren benötigt im schlechtesten Fall Laufzeit* $\Omega(n \log n)$.

Beweis. Wir betrachten ein Eingabedatenfeld der Länge n und ein Sortierverfahren $\mathcal{A}$. Sei b die Anzahl der Blätter des Entscheidungsbaums von $\mathcal{A}$ für Eingabe der Länge n, dann ist $b = n!$. Sei d die maximale Tiefe eines Blattes, dann ist $b \leq 2^d$ und so erhalten wir $n! \leq 2^d$, d.h. $\log(n!) \leq d$. Da $n! \geq (\frac{n}{2})^{\frac{n}{2}} = 2^{\frac{n}{2} \log \frac{n}{2}}$, ist $\log(n!) \geq \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n)$. □

Um den Durchschnittsfall zu behandeln beweisen wir zunächst ein Lemma. Für einen Baum T bezeichnen wir den Durchschnitt der Tiefen der Blätter von T als *mittlere Tiefe* $\bar{d}(T)$ *von* T.

Lemma 9.2. *Ein binärer Baum* T *mit* $b \geq 2$ *Blättern hat mittlere Tiefe* $\bar{d}(T) \geq \log b$.

Beweis. Wir gehen mit Induktion nach b vor. Der Fall $b = 2$ ist trivial. Für den Induktionsschritt sei T_1 der linke Teilbaum von T mit b_1 Blättern und T_2 der rechte Teilbaum von T mit b_2 Blättern. Dann ist $b = b_1 + b_2$ und

$$\begin{aligned}
\bar{d}(T) &= \frac{1}{b}(b_1(\bar{d}(T_1) + 1) + b_2(\bar{d}(T_2) + 1)) \\
&\geq^{\text{IH}} \frac{1}{b}(b_1(\log b_1 + 1) + b_2(\log b_2 + 1)) \\
&= \frac{1}{b_1 + b_2}(b_1 \log 2b_1 + b_2 \log 2b_2).
\end{aligned}$$

Diese Funktion erreicht ihr Minimum bei $b_1 = b_2 = \frac{b}{2}$, also

$$\begin{aligned}
&\geq \frac{1}{b}(\frac{b}{2} \log b + \frac{b}{2} \log b) \\
&= \log b.
\end{aligned}$$

□

Satz 9.4. *Jedes vergleichsbasierte Sortierverfahren benötigt im durchschnittlichen Fall Laufzeit* $\Omega(n \log n)$.

Beweis. Sei $\mathcal{A}$ ein vergleichsbasiertes Sortierverfahren und sei T_n der Entscheidungsbaum von $\mathcal{A}$ für Eingaben der Länge n, dann hat T_n genau $n!$ Blätter und damit ist, nach Lemma 9.2, $\bar{d}(T_n) \geq \log n!$. Wie im Beweis von Satz 9.3 gezeigt ist $\log n! = \Omega(n \log n)$. □

Man kann durchaus auch sinnvolle Sortierverfahren angeben, die keine vergleichsbasierten Sortierverfahren sind und, zumindest für gewisse Spezialfälle, eine bessere Laufzeit haben. *Zählsortieren* geht davon aus, dass die Schlüssel in einen bestimmten Bereich $\{1, \ldots, k\}$ fallen und erlaubt dann eine Sortierung in Zeit $\Theta(n + k)$ wobei n die Länge des Eingabedatenfelds ist. Für hinreichend kleine k ist damit also ein Sortierverfahren mit Laufzeit $\Theta(n)$ angegeben. Ein Einsatz dieses Verfahrens ist nur sinnvoll wenn k nicht größer als zirka $n \log n$ ist.

Beispiel 9.3. Bei dem Eingabedatenfeld

$A =$

a	b	c	d	e	f	g	h
2	4	3	2	1	4	3	3

sind alle Schlüssel in $\{1, \ldots, 4\}$, d.h. also $k = 4$. Zunächst zählen wir, für alle $j \in \{1, \ldots, k\}$ wie viele Elemente den Schlüssel j haben und erhalten.

1	2	3	2

Daraus berechnen wir, für $j \in \{1, \ldots, k\}$ den letzten Index mit Schlüssel j im sortierten Datenfeld.

1	3	6	8

Das induziert eine Unterteilung

1	2	2	3	3	3	4	4

des Ausgabedatenfelds. Dieses wird dann, der Unterteilung folgend, mit den Elementen von A befüllt und wir erhalten

e	a	d	c	g	h	b	f
1	2	2	3	3	3	4	4

Zählsortieren ist in Algorithmus 41 als Pseudocode ausformuliert. Man erkennt leicht, dass diese Prozedur Laufzeit $\Theta(n + k)$ hat.

9.4 Primzahltests

Wir wollen uns nun mit dem folgenden Entscheidungsproblem beschäftigen:

Primzahlen

Eingabe: $n \in \mathbb{N}$

Ausgabe: “ja” falls $n \in \mathbb{P}$, “nein” sonst

Algorithmus 41 Zählsortieren

Prozedur ZÄHLSORTIEREN(A, k)
  Sei $B[1, \ldots, A.Länge]$ ein neues Datenfeld
  Sei $C[1, \ldots, k]$ ein neues Datenfeld, überall mit 0 initialisiert.
  Für $i := 1, \ldots, A.Länge$
    $C[A[i].x] := C[A[i].x] + 1$
  Ende Für
  Für $i := 2, \ldots, k$
    $C[i] := C[i] + C[i-1]$
  Ende Für
  Für $i := A.Länge, \ldots, 1$
    $B[C[A[i].x]] := A[i]$
    $C[A[i].x] := C[A[i].x] - 1$
  Ende Für
  Antworte B
Ende Prozedur

Dieses Problem hat wichtige Anwendungen in der Kryptographie, zum Beispiel bildet es die Grundlage der Schlüsselerzeugung beim RSA-Verfahren das wir in Abschnitt 9.5 genauer besprechen werden. Man kann sich leicht überlegen, dass eine Zahl $n \in \mathbb{N}$ zusammengesetzt ist genau dann wenn sie einen Teiler $\leq \sqrt{n}$ hat, so dass es für einen Primzahltest ausreichend ist, Probedivisionen von 1 bis $\sqrt{n}$ durchzuführen. Dieses auf Probedivisionen basierende Verfahren hat also Laufzeit $O(\sqrt{n})$. In der Praxis hat man es aber oft mit Primzahlen zu tun die etwa 1000 bis 3000 Bits haben. Nun ist z.B. $\sqrt{2^{2000}} = 2^{1000}$ was zu hoch ist, um zu einem praktisch einsetzbaren Algorithmus zu führen.

Seit 2002 ist ein Algorithmus bekannt der in Zeit $O((\log n)^k)$ für ein $k \in \mathbb{N}$ entscheidet ob ein gegebenen $n \in \mathbb{N}$ eine Primzahl ist (AKS-Verfahren). Das derzeit kleinste bekannte solche k ist $k = 7$. Dieses Resultat hat bisher nur theoretische Bedeutung. Für die praktische Anwendung ist diese Klasse von Algorithmen nicht effizient genug. In der Praxis wendet man stattdessen probabilistische Primzahltests an die effizient sind und mit beliebig großer Wahrscheinlichkeit < 1 eine korrekte Antwort liefern. Um diese näher zu besprechen rufen wir uns zunächst den kleinen Satz von Fermat in Erinnerung:

Satz 9.5 (Kleiner Satz von Fermat)**.** *Sei $n \in \mathbb{P}$, $a \geq 1$, $\mathrm{ggT}(a, n) = 1$, dann ist*

$$a^{n-1} \equiv 1 \pmod{n}.$$

Auf Basis dieses Satz ist ein einfacher Primzahltest möglich: Gegeben $n \in \mathbb{N}$ wählen wir zufällig ein $a \in \{2, \ldots, n-1\}$ und überprüfen (mit dem euklidischen Algorithmus) ob $\mathrm{ggT}(a, n) = 1$. Ist das nicht der Fall ist a ein Teiler von n und damit $n \notin \mathbb{P}$. Falls $\mathrm{ggT}(a, n) = 1$ ist dann überprüfen wir ob $a^{n-1} \equiv 1 \pmod{n}$. Falls $a^{n-1} \not\equiv 1 \pmod{n}$ dann ist $n \notin \mathbb{P}$ (auch wenn wir keinen Teiler von n kennen). In diesem Fall heißt a auch *Fermat-Zeuge gegen die Primalität von n*. Ist hingegen $a^{n-1} \equiv 1 \pmod{n}$ dann endet der Algorithmus *mit der Vermutung dass $n \in \mathbb{P}$*. Die Laufzeit dieses Verfahrens beträgt bei Verwendung effizienter Exponentation $O(\log n)$.

Nun hat dieser Primzahltest allerdings einen bedeutenden Defekt: Es gibt zusammengesetzte Zahlen n so dass für alle $a \in \{2, \ldots, n-1\}$ mit $\mathrm{ggT}(a, n) = 1$ gilt: $a^{n-1} \equiv 1 \pmod{n}$. Solche Zahlen heißen Carmichael-Zahlen. Es gibt unendlich viele Carmichael-Zahlen, die kleinste ist $3 \cdot 11 \cdot 17 = 561$. Wenn wir diese aber ausklammern, so können wir zeigen:

Lemma 9.3. *Sei $n \in \mathbb{N}$ eine zusammengesetzte Zahl die keine Carmichael-Zahl ist. Dann gibt*

es in $\{1, \ldots, n-1\}$ *höchstens* $\frac{n-1}{2}$ *Zahlen die zu* n *teilerfremd sind aber keine Fermat-Zeugen gegen die Primalität von* n *sind.*

Beweis. Sei $B = \{b \in (\mathbb{Z}/n\mathbb{Z})^\times \mid b^{n-1} \equiv 1 \pmod{n}\}$. Dann ist B eine Untergruppe der multiplikativen Gruppe $(\mathbb{Z}/n\mathbb{Z})^\times$. Nach dem Satz von Lagrange gilt also $|B| \mid n-1$. Da n zusammengesetzt ist aber keine Carmichael-Zahl gibt es ein $a \in \{2, \ldots, n-1\}$ mit $a^{n-1} \not\equiv 1 \pmod{n}$. Also ist B eine echte Untergruppe von $(\mathbb{Z}/n\mathbb{Z})^\times$ und damit ist $|B| \leq \frac{n-1}{2}$. □

Wenn wir also für ein zusammengesetztes $n \in \mathbb{N}$ das keine Carmichael-Zahl ist ein $a \in \{1, \ldots, n-1\}$ uniform verteilt wählen, so ist $\mathrm{W}(\mathrm{ggT}(a, n) = 1 \text{ und } a^{n-1} \equiv 1 \pmod{n}) \leq \frac{1}{2}$. Falls also der Fermattest auf einem $n \in \mathbb{N}$ das keine Carmichael-Zahl ist mit "vermutlich ist $n \in \mathbb{P}$" antwortet, so irrt er sich mit einer Wahrscheinlichkeit von $\leq \frac{1}{2}$. Dieser Test kann nun für verschiedene, unabhängig voneinander zufällig gewählte, a wiederholt werden um so die Fehlerwahrscheinlichkeit beliebig nahe an 1 zu bringen.