SS 2021

Algorithmen und Datenstrukturen

10. Kapitel

Greedy-Algorithmen: Prinzipien

Martin Dietzfelbinger

Juli 2021

Kapitel 10: Greedy-Algorithmen

Greedy*-Algorithmen sind anwendbar bei **Konstruktionsaufgaben**, deren Ziel es ist, eine **optimale Struktur zu finden**, die aus mehreren Komponenten besteht.

Sie finden eine Lösung, die sie schrittweise aufbauen.

In jedem Schritt wird eine „lokal“ oder aus der aktuellen Sicht optimale Entscheidung getroffen, ohne „an die Zukunft zu denken“.
(Was dies konkret bedeutet, versteht man besser anhand der später betrachteten Beispiele.)

Es werden dabei nicht mehr Teillösungen konstruiert als unbedingt nötig.

Es werden nie Entscheidungen korrigiert oder zurückgesetzt.

---

* greedy (*engl.*): gierig.

10.1 Zwei Beispiele

Beispiel 1: Hörsaalbelegung

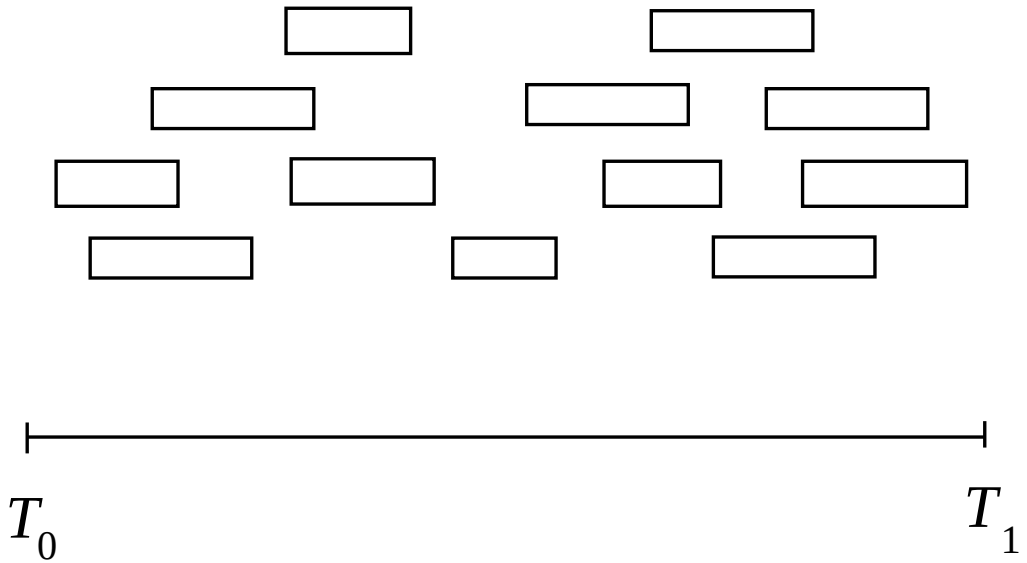
Gegeben: Veranstaltungsort (Hörsaal), Zeitspanne $[T_0, T_1)$ (z. B. 7 Uhr bis 21 Uhr) und eine Menge von n Aktionen (Vorlesungen oder ähnliches), die durch Start- und Endzeit spezifiziert sind:

$$[s_i, f_i) \quad \text{mit } T_0 \leq s_i < t_i \leq T_1, \text{ für } 1 \leq i \leq n.$$

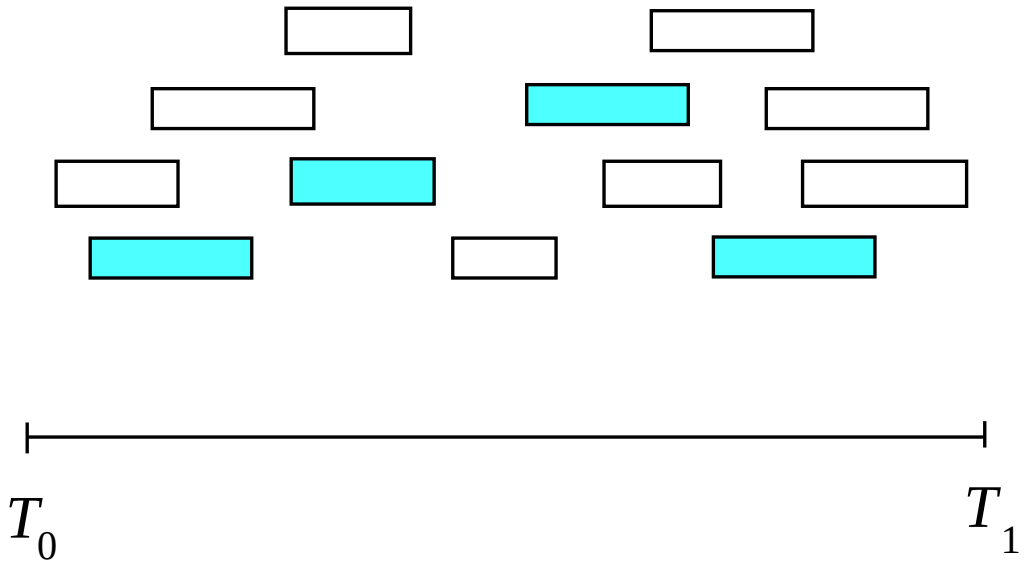
Gesucht: Belegung des Hörsaals, die **möglichst viele** Ereignisse mit disjunkten Zeitspannen stattfinden lässt.

$A \subseteq \{1, \ldots, n\}$ heißt **zulässig**, wenn alle $[s_i, f_i)$, $i \in A$, disjunkt sind.

Aufgabe, formal: Finde eine zulässige Menge A mit $|A|$ so groß wie möglich.

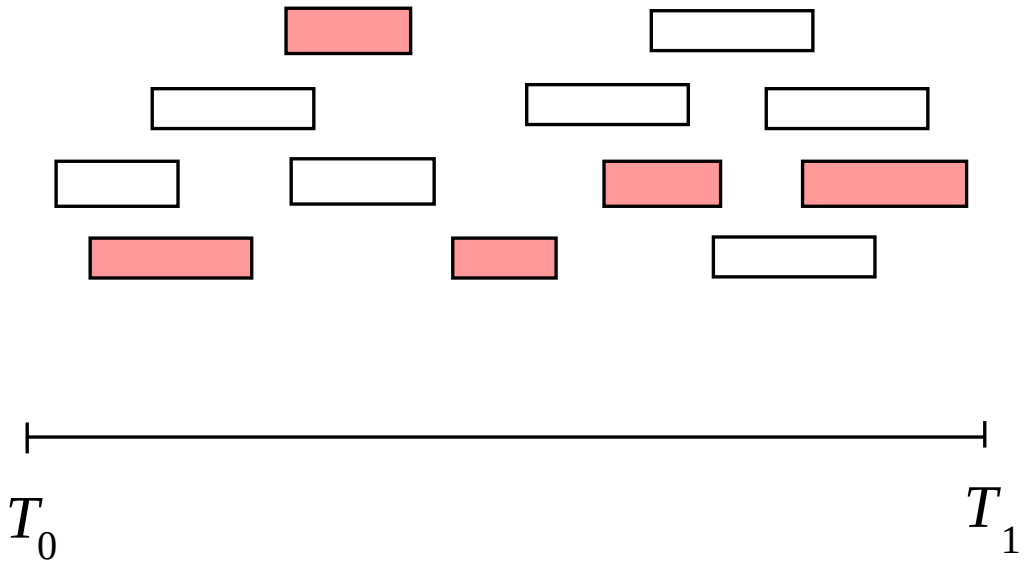


Eingabe: Aktionen mit Beginn und Ende.



Eingabe: Aktionen mit Beginn und Ende.

Zulässige Lösung mit **4** Aktionen. Ist sie optimal?

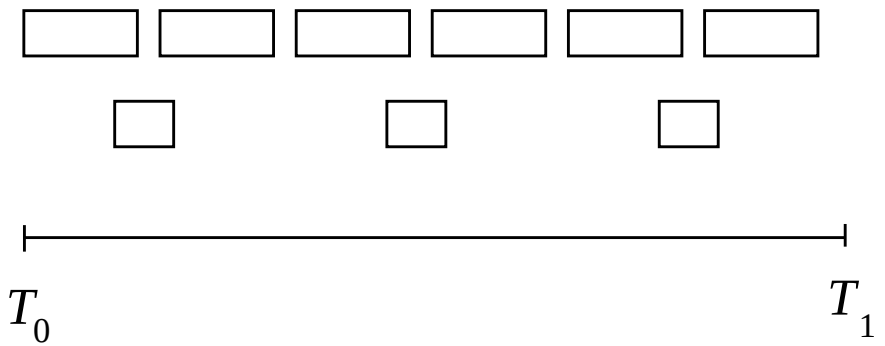


Eingabe: Aktionen mit Beginn und Ende.

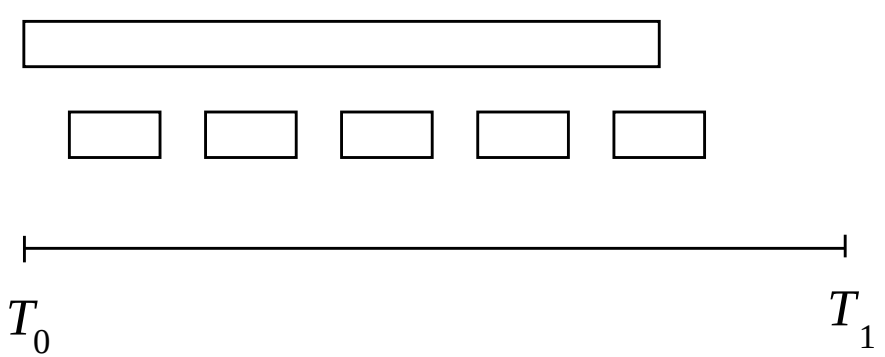
Zulässige Lösung mit **5** Aktionen. Ist sie optimal?

Ansätze, die nicht funktionieren:

- Zuerst kurze Ereignisse planen



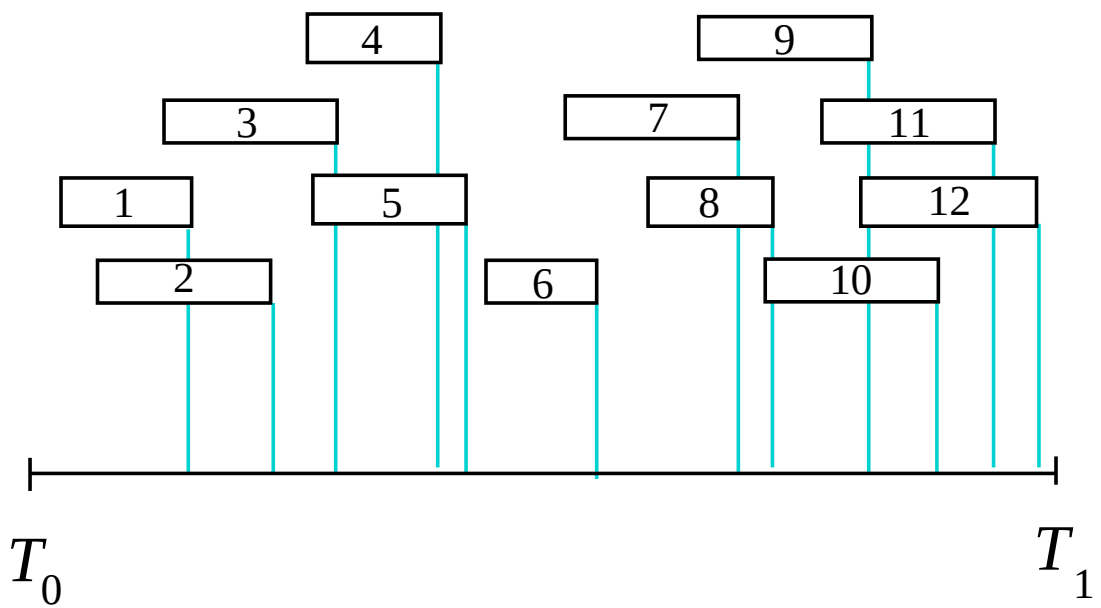
- Immer ein Ereignis mit möglichst früher Anfangszeit wählen



Trick: Bearbeite Ereignisse in der Reihenfolge **wachsender Schlusszeiten**.

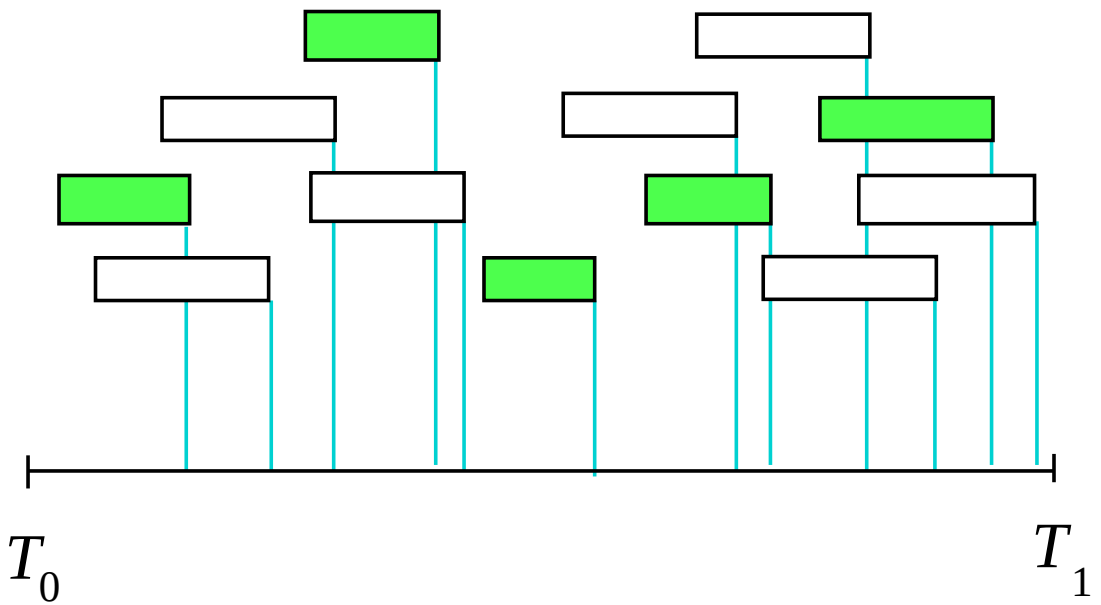
O.B.d.A.: Veranstaltungen nach Schlusszeiten aufsteigend sortiert (Zeitaufwand $O(n \log n)$), also:

$$f_1 \leq f_2 \leq \cdots \leq f_n.$$



Wiederhole: Wähle die **wählbare** Aktion mit der kleinsten Schlusszeit und füge sie zum Belegungsplan hinzu.

Eine Aktion ist **wählbar**, wenn ihre Startzeit mindestens so groß wie die Schlusszeit der letzten schon geplanten Veranstaltung ist.



Ausgabe von GS: Zulässige Lösung.

Algorithmus Greedy Scheduling (GS)

Eingabe: Reelle Zahlen $T_0 < T_1$, Paare $(s_1, t_1), \ldots, (s_n, f_n)$, mit
$T_0 \leq s_i < f_i \leq T_1$ für $1 \leq i \leq n$

Ausgabe: Maximal großes $A \subseteq \{1, \ldots, n\}$ mit $[s_i, f_i), i \in A$, disjunkt

(1) Sortiere Paare gemäß $f_1, \ldots, f_n$, nummeriere um, so dass $f_1 \leq \cdots \leq f_n$;
(2) $A \leftarrow \{1\}$;
(3) $f_{\text{last}} \leftarrow f_1$;
(4) **for** i **from** 2 **to** n **do**
(5) **if** $s_{\mathtt{i}} \geq f_{\text{last}}$ **then** // $[s_{\mathtt{i}}, f_{\mathtt{i}})$ wählbar
(6) $A \leftarrow A \cup \{\mathtt{i}\}$;
(7) $f_{\text{last}} \leftarrow f_{\mathtt{i}}$;
(8) **return** A

Satz 10.1.1

Der Algorithmus Greedy Scheduling (GS) hat lineare Rechenzeit (bis auf die Sortierkosten von in Zeile (1), höchstens $O(n \log n)$) und löst das Hörsaalplanungsproblem optimal.

Beweis: **Rechenzeit:** Sortieren kostet Zeit $O(n \log n)$; der restliche Algorithmus hat offensichtlich Laufzeit $O(n)$.

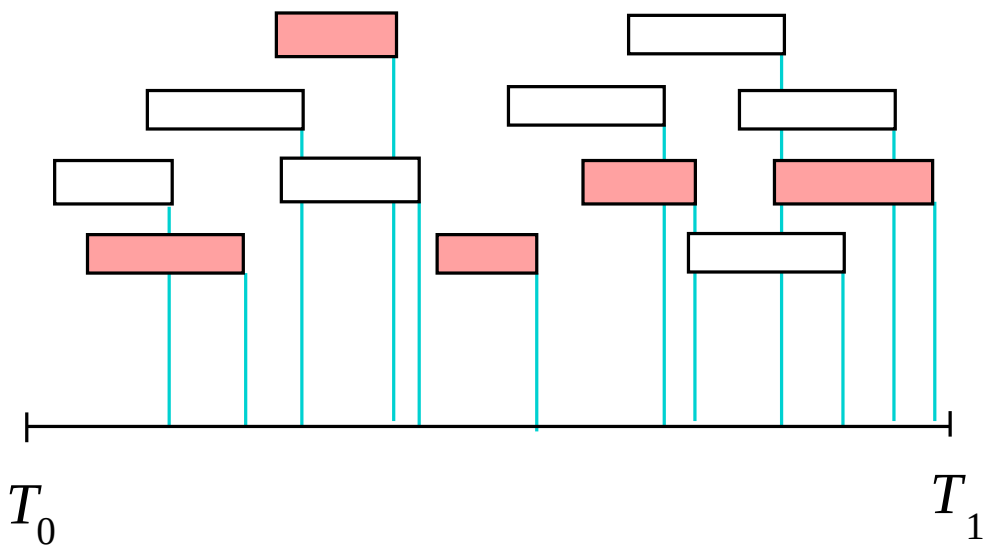
Korrektheit: Wir behaupten: Algorithmus GS liefert auf Inputs mit Größe n eine optimale Lösung. Dies beweisen wir durch **vollständige Induktion**.

Der Fall $n = 1$ ist trivial.

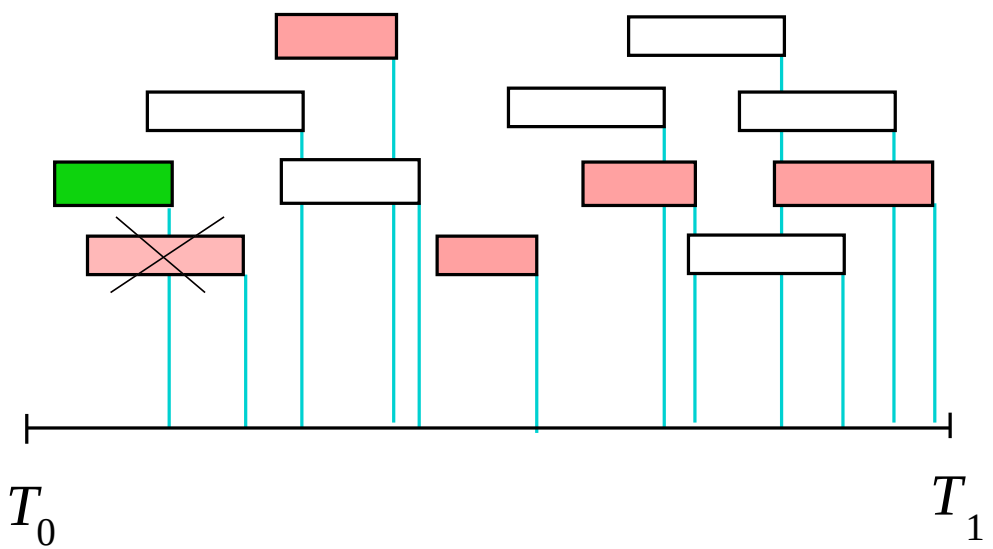
Sei nun $n > 1$.

I.V.: GS liefert für Inputs der Länge $n' < n$ eine optimale Lösung.

Ind.-Schritt: Sei $B \subseteq \{1, \ldots, n\}$ eine **optimale** Lösung, $|B| = r$.
(Im Beispiel: $r = 5$.)



1. Beobachtung:
Es gibt eine Lösung B', die mit dem Intervall $[s_1, f_1)$ (dem ersten Schritt des Greedy-Algorithmus) startet und ebenfalls r Ereignisse hat. (B' ist also auch optimal.)



Setze $B' := (B - \{\min(B)\}) \cup \{1\}$.
Intervalle aufsteigend sortiert $\Rightarrow f_1 \leq f_{\min(B)}$.

2. Beobachtung:

Die Menge $B - \{\min(B)\}$ **löst das Teilproblem**

$$\boldsymbol{f_1, T_1, (s_i, f_i), s_i \geq f_1} \qquad (*)$$

optimal. Anschaulich: In $[f_1, T_1)$ können maximal $r - 1$ dieser Ereignisse untergebracht werden.

Wieso? Sonst würden wir $[s_1, f_1)$ mit einer besseren Lösung für $(*)$ zu einer besseren Lösung für das Gesamtproblem kombinieren: Widerspruch zur Optimalität von B.

Wenn wir also eine beliebige optimale Lösung C für $(*)$ finden, so hat diese $r - 1$ Elemente.

Wenn wir zu einem solchen optimalen C das Ereignis (s_1, f_1) hinzufügen, also $C \cup \{1\}$ bilden, erhalten wir eine Lösung der Größe r für die ursprüngliche Eingabe. Diese ist optimal.

3. Beobachtung:

Algorithmus GS auf Eingabe $T_0, T_1, (s_i, f_i), 1 \leq i \leq n$, hat ab Iteration $i = 2$ **genau dasselbe Verhalten** wie wenn man GS auf f_1, T_1, $(s_i, f_i), s_i \geq f_1$, starten würde.

Nach I.V. liefert also dieser Teil des Algorithmus eine optimale Lösung C mit $r - 1$ Ereignissen für $(*)$.

(mit 2. Beob.) $\Rightarrow$ Greedy Scheduling liefert insgesamt eine optimale Lösung der Größe r. $\square$

Beispiel 2: Fraktionales („teilbares“) Rucksackproblem

Veranschaulichung: Ein Dieb stiehlt Säckchen mit Edelmetallkrümeln (Gold, Silber, Platin) und Edelsteinen. Er hat einen Rucksack mit Volumen b dabei. Durch teilweises Ausleeren der Säckchen kann er beliebige Teilvolumina herstellen. Der Wert pro Volumeneinheit ist unterschiedlich für unterschiedliche Materialien. Was soll er in seinen Rucksack mit Volumen b packen, um den Wert der Beute zu maximieren?

Dazu: Benenne den Bruchteil $\lambda_i \in [0, 1]$, den er von Säckchen i mitnehmen soll.

Gegeben: (n Objekte mit:)
Volumina $a_1, \ldots, a_n > 0$, **Nutzenwerte** $c_1, \ldots, c_n > 0$, **Volumenschranke** b.

Gesucht: Vektor $(\lambda_1, \ldots, \lambda_n) \in [0, 1]^n$, so dass

$$\lambda_1 a_1 + \ldots + \lambda_n a_n \leq b \quad \textbf{(„zulässig“)}$$

und

$$\lambda_1 c_1 + \ldots + \lambda_n c_n \text{ maximal.}$$

Beim **„0-1-Rucksackproblem“** werden nur 0-1-Vektoren mit $\lambda_i \in \{0, 1\}$ zugelassen.

Mitteilung (im Vorgriff auf „Automaten, Sprachen und Komplexität“, 3. Sem.): Die $\{0, 1\}$-Version ist **NP-vollständig**, besitzt also wahrscheinlich keinen effizienten Algorithmus.

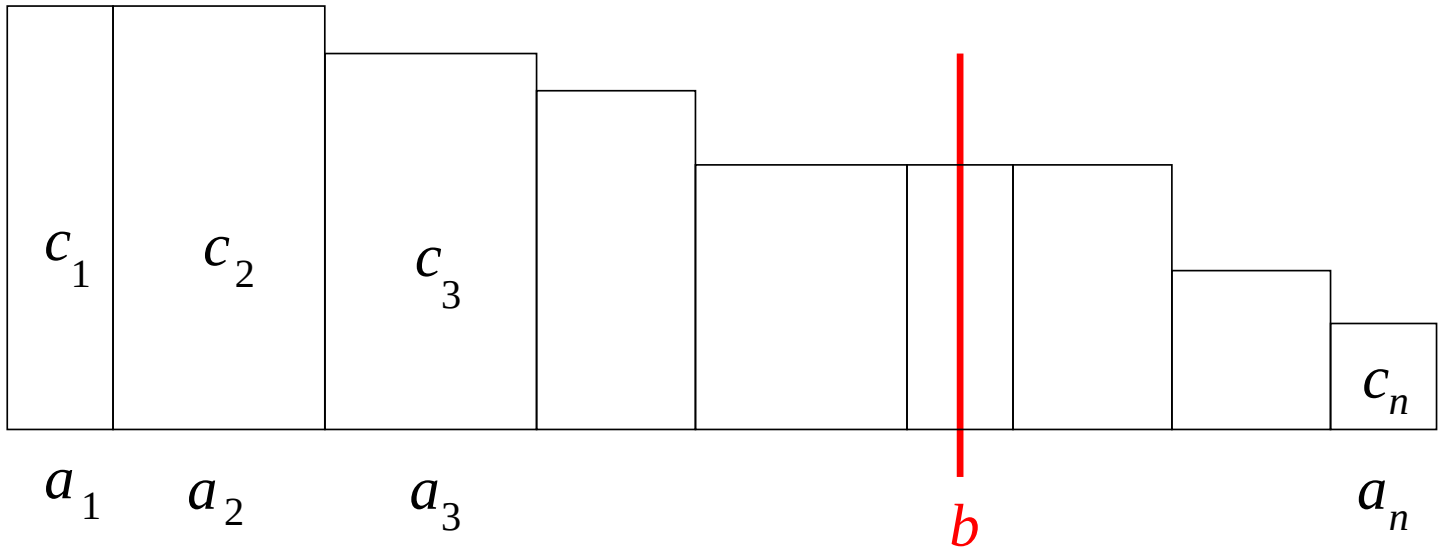
Das fraktionale Rucksackproblem ist mit einem Greedy-Algorithmus in Zeit $O(n \log n)$ lösbar.

Kern der Lösungsidee: Berechne **„Nutzendichten“**

$$d_i = \frac{c_i}{a_i}, 1 \leq i \leq n,$$

und sortiere die Objekte gemäß d_i fallend.

Nehme von vorne beginnend möglichst viele ganze Objekte, bis schließlich das letzte Objekt teilweise genommen wird, so dass die Volumenschranke vollständig ausgeschöpft wird.



Input, sortiert nach Nutzendichte $d_i = c_i/a_i$.

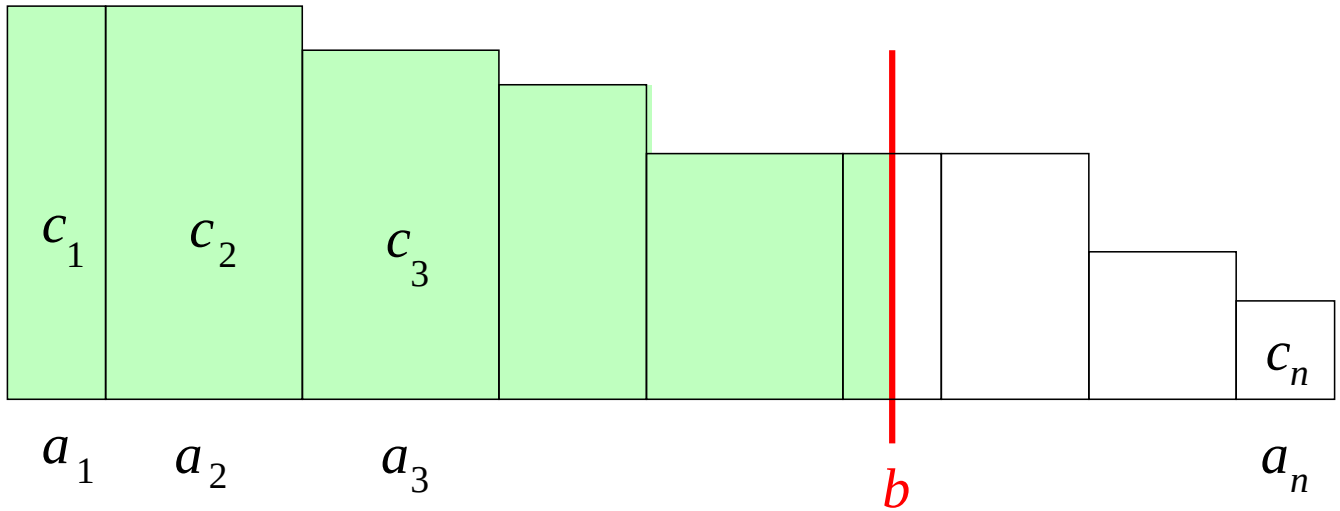
Höhe von Kasten Nummer i ist $\boldsymbol{d_i} = c_i/a_i$.

Breite ist $\boldsymbol{a_i}$.

Fläche ist $\boldsymbol{c_i}$.

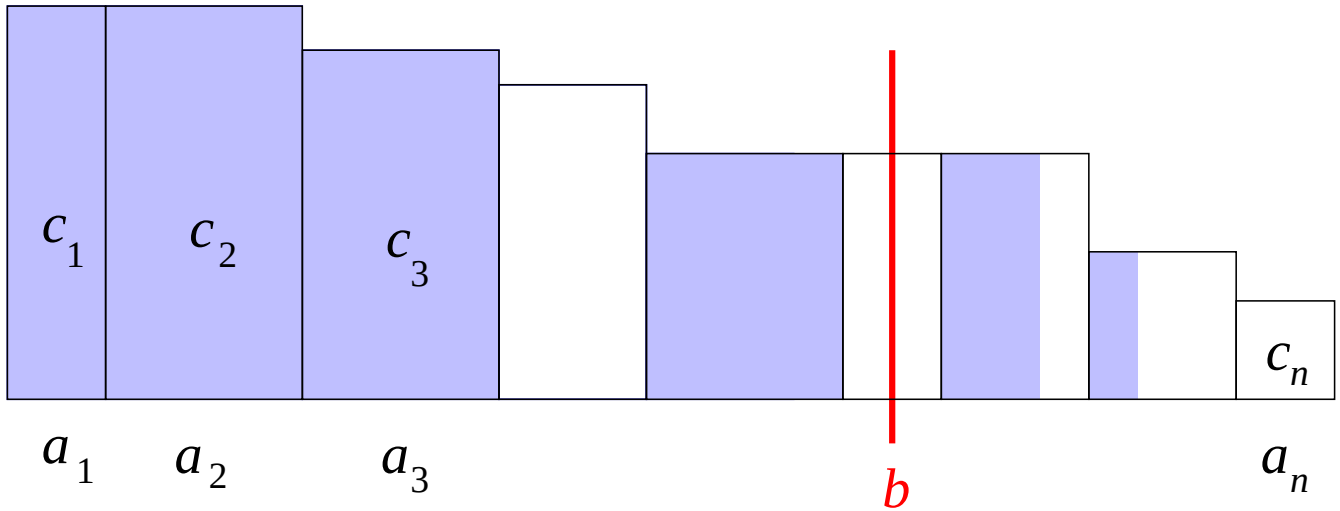
Algorithmus Greedy Fractional Knapsack (GFKS)

(1) **for** $\mathtt{i}$ **from** 1 **to** n **do**
(2) $\quad d_{\mathtt{i}} \leftarrow c_{\mathtt{i}}/a_{\mathtt{i}}$;
(3) $\quad \lambda_{\mathtt{i}} \leftarrow 0$;
(4) Sortiere Objekte gemäß d_i fallend;
(5) $\mathtt{i} \leftarrow 0$;
(6) $\mathtt{r} \leftarrow b$; // Inhalt r ist das verfügbare Rest-Volumen
(7) **while** $\mathtt{r} > 0$ **do**
(8) $\quad \mathtt{i}$++;
(9) $\quad$ **if** $a_{\mathtt{i}} \leq \mathtt{r}$
(10) $\quad\quad$ **then** $\lambda_{\mathtt{i}} \leftarrow 1$; $\mathtt{r} \leftarrow \mathtt{r} - a_{\mathtt{i}}$;
(11) $\quad\quad$ **else** $\lambda_{\mathtt{i}} \leftarrow \mathtt{r}/a_{\mathtt{i}}$;
(12) $\quad\quad\quad\quad \mathtt{r} \leftarrow 0$;

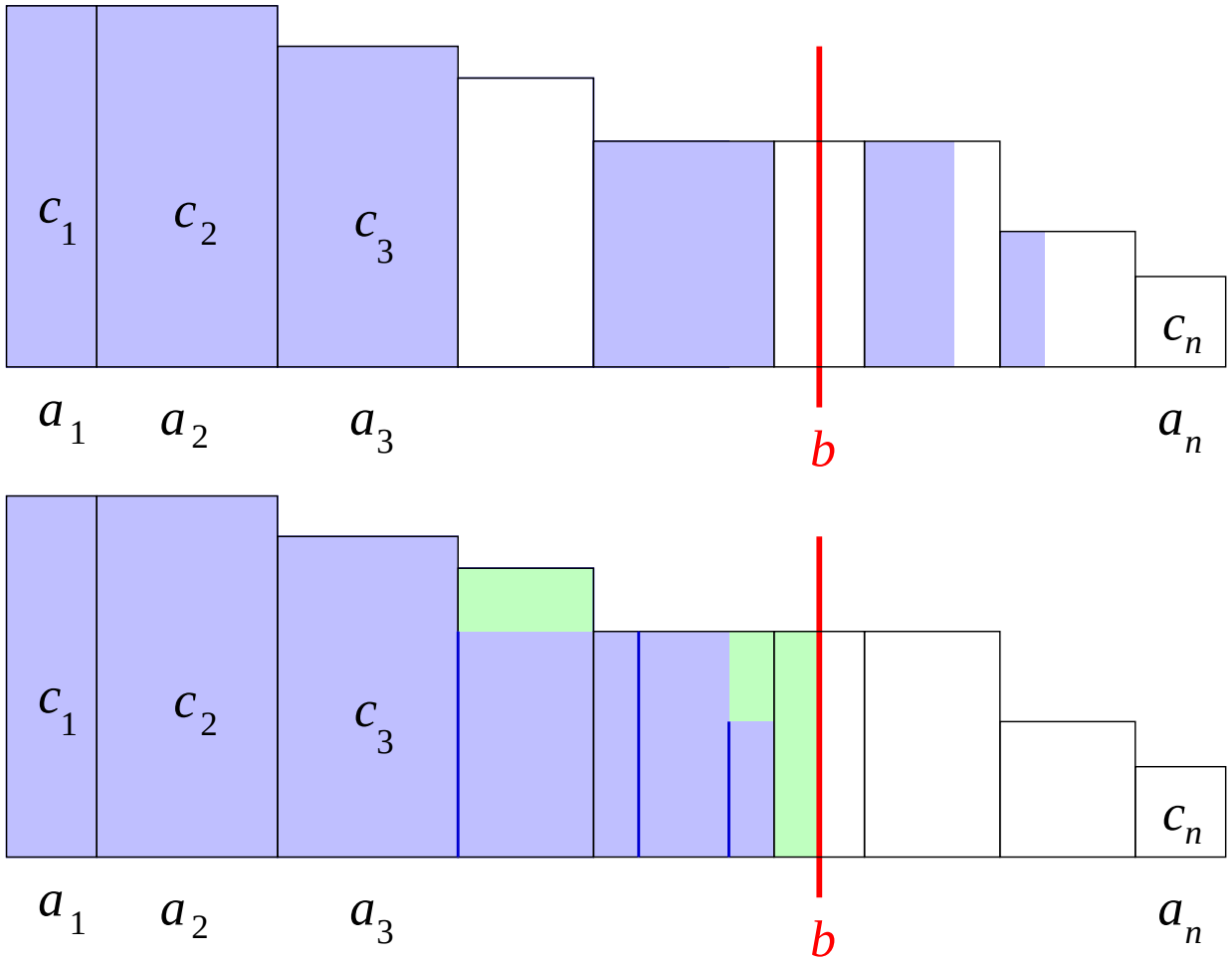


Vom Greedy-Algorithmus gelieferte Lösung.

Gesamtnutzen: grün.



Zulässige Lösung, nicht optimal. Gesamtnutzen: blau.



Anschaulich: **Greedy-Lösung** ist nie schlechter als **beliebige Lösung**.

Satz 10.1.2

Der Algorithmus GFKS ist korrekt (liefert eine zulässige Lösung mit maximalem Gesamtnutzen) und hat Rechenzeit $O(n \log n)$.

Für Interessierte: Vollständiger Beweis, der die anschauliche Argumentation in Formeln übersetzt, steht auf den Druckfolien.

Beweis: Sei $x = (a_1, \ldots, a_n, c_1, \ldots, c_n, b)$ die Eingabe.

Rechenzeit: klar.

Korrektheit: Dass die Lösung zulässig ist, ist klar. **Zu zeigen:** Optimalität.

O.B.d.A.: $\sum_i a_i > b$ (sonst ist $(\lambda_1, \ldots, \lambda_n) = (1, \ldots, 1)$ optimal und wird von GFKS gefunden).

Sei $(\lambda_1, \ldots, \lambda_n) \in [0, 1]^n$ die **Ausgabe** des Algorithmus.

Sei $(\lambda_1^*, \ldots, \lambda_n^*) \in [0, 1]^n$ eine **optimale** Lösung.

Offensichtlich: $\sum_i \lambda_i^* a_i = b$ (sonst könnten wir die Lösung durch Hinzufügen eines Bruchteil eines nicht ganz aufgebrauchten Objekts verbessern).

Durch Induktion über n zeigen wir: $\sum_i \lambda_i c_i = \sum_i \lambda_i^* c_i$.

I.A.: $n = 1$. Dann liefert der Algorithmus offensichtlich die optimale Lösung: Packe genau den Bruchteil $\lambda_1 = b/a_1$, der in den Rucksack passt.

Ind.-Schritt: Sei nun $n > 1$.

1. Fall: $a_1 > b$.

GFKS wählt $\lambda_1 = b/a_1$, und das ist optimal: Weil $d_1 \geq d_i$ für alle i, gilt:

$$\lambda_1 c_1 = b d_1 = \Big(\sum_i \lambda_i^* a_i\Big) d_1 \geq \sum_i \lambda_i^* a_i d_i = \sum_i \lambda_i^* c_i.$$

2. Fall: $a_1 \leq b$.

Behauptung: Wir können o.B.d.A. $\lambda_1^* = 1$ annehmen.
(„Der erste Schritt des Greedy-Algorithmus ist nicht falsch.“)
Denn: Wenn $\lambda_1^* < 1$, kann man wegen $\sum_{1 \leq i \leq n} \lambda_i^* a_i = b$ Werte $\lambda_1' = 1$, $0 \leq \lambda_2' \leq \lambda_2^*, \ldots, 0 \leq \lambda_n' \leq \lambda_n^*$ finden, so dass

$$(1 - \lambda_1^*) a_1 = \sum_{2 \leq i \leq n} (\lambda_i^* - \lambda_i') a_i. \qquad (*)$$

(Verringere Gewicht bei „späteren“ Objekten zugunsten von Objekt 1.)
Dann ist auch $(\lambda_1', \ldots, \lambda_n')$ zulässig und

$$\sum_i \lambda_i' c_i = \sum_i \lambda_i^* c_i + \Big((1 - \lambda_1^*) a_1 d_1 - \sum_{2 \leq i \leq n} (\lambda_i^* - \lambda_i') a_i d_i \Big).$$

Aus $(*)$ und $d_1 \geq d_i$ folgt, dass die letzte Klammer nichtnegativ ist, also $\sum_i \lambda_i' c_i \geq \sum_i \lambda_i^* c_i$ gilt, dass also auch $(\lambda_1', \ldots, \lambda_n')$ optimal ist.

Wir wissen nun: $1 = \lambda_1 = \lambda_1^*$.

Wir wenden die Induktionsvoraussetzung auf den modifizierten Input $x^- = (a_2, \ldots, a_n, c_2, \ldots, c_n, b - a_1)$ an: GFKS auf x^- liefert optimale Lösung $(\lambda_2, \ldots, \lambda_n)$. GFKS läuft in Schleifendurchläufen 2 bis n ebenso wie GFKS auf x^-, liefert also ebenfalls $(\lambda_2, \ldots, \lambda_n)$.

Klar: $(\lambda_2^*, \ldots, \lambda_n^*)$ muss für x^- optimal sein.

(Wenn $(\lambda_2', \ldots, \lambda_n')$ besser wäre, dann wäre $(1, \lambda_2', \ldots, \lambda_n')$ eine bessere Lösung für x als $(1, \lambda_2^*, \ldots, \lambda_n^*)$. Das kann aber nicht sein.)

Also gilt $\sum_{2 \leq i \leq n} \lambda_i c_i = \sum_{2 \leq i \leq n} \lambda_i^* c_i$.

Also haben auch die Lösungen $(1, \lambda_2^*, \ldots, \lambda_n^*)$ und $(1, \lambda_2, \ldots, \lambda_n)$ für x denselben Nutzenwert, und $(\lambda_1, \lambda_2, \ldots, \lambda_n) = (1, \lambda_2, \ldots, \lambda_n)$ ist optimal. □

Charakteristika der Greedy-Methode:

1. Der **erste Schritt** der Greedy-Lösung ist **nicht falsch**. Es gibt eine optimale Lösung, die als **Fortsetzung des ersten Schrittes** konstruiert werden kann.
2. **„Prinzip der optimalen Substruktur“:** Kombiniert man das Ergebnis des ersten „Greedy-Schritts“ mit einer beliebigen optimalen Lösung für die reduzierte Instanz (Wegnehmen des Teils, der durch den ersten Schritt erledigt ist), dann ergibt sich eine optimale Lösung für die Originaleingabe.
3. Der Algorithmus löst das verbleibende Teilproblem rekursiv (oder mit einem zur Rekursion äquivalenten iterativen Verfahren).

Die Korrektheit ergibt sich dann durch vollständige Induktion.

10.2 Huffman-Codes

Gegeben: **Alphabet A** mit $2 \leq |A| < \infty$ und „Wahrscheinlichkeiten“ oder „relativen Häufigkeiten“ $p(a) \in [0,1]$ für jedes $a \in A$.

Also: $\sum_{a \in A} p(a) = 1$.

Beispiel: $A = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{K}\}$, $|A| = 10$:

a	A	B	C	D	E	F	G	H	I	K
$p(a)$	0,15	0,08	0,07	0,10	0,21	0,08	0,07	0,09	0,06	0,09

Herkunft der Wahrscheinlichkeiten:
(1) Buchstabenhäufigkeit in natürlicher Sprache *oder*
(2) empirische relative Häufigkeiten in einem gegebenen Text $w = a_1 \dots a_m \in A^*$:
Anteil des Buchstabens a an w ist $(p(a) \cdot 100)\%$.

Gesucht: ein **„guter“ binärer Präfixcode** für (A, p). — *Beispiel*:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

Definition 10.2.1 Präfixcode:

Jedem $a \in A$ ist binärer „Code“ $c(a) \in \{0,1\}^*$ (Menge aller Binärstrings) zugeordnet, mit Eigenschaft

Präfixfreiheit: Für $a, b \in A$, $a \neq b$ ist $c(a)$ kein Präfix von $c(b)$.

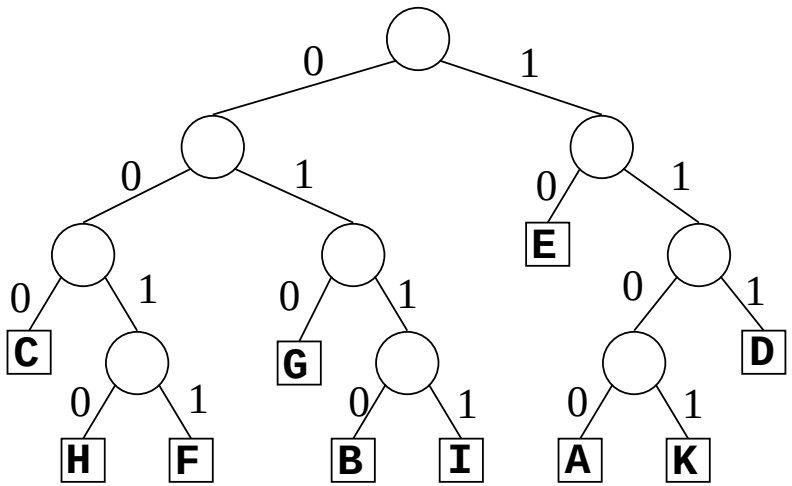
Codierung von Wörtern (Zeichenreihen):

$c(a_1 \dots a_n) = c(a_1) \cdots c(a_n) \in \{0,1\}^*$.

Zur **Codierung** benutzt man (konzeptuell) direkt die Tabelle.

Beispiel: $c(\texttt{F E I G E}) = 001110011101010$.

Kompakte Repräsentation des Codes als Binärbaum:



Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt. Wiederhole mit dem Restwort, bis nichts mehr übrig ist.

Präfixeigenschaft $\Rightarrow$ keine Zwischenräume nötig.

Beispiel: 001111000000010 liefert „`FACH`“.

1. Idee: Mache alle Codewörter $c(a)$ gleich lang.
Am besten ist dann eine Länge von $\lceil \log_2 |A| \rceil$ Bits.

$\Rightarrow c(a_1 \dots a_m)$ hat Länge $\lceil \log_2 |A| \rceil \cdot m$.

(Beispiele: 52 Groß- und Kleinbuchstaben plus Leerzeichen und Satzzeichen:
$\log 64 = 6$ Bits pro Codewort. Latin-1-Code: 8 Bits pro Codewort.)

2. Idee: Einsparmöglichkeit: Häufige Buchstaben mit kürzeren Wörtern codieren als seltenere Buchstaben.

Ein erster Ansatz zur **Datenkompression** (platzsparendes Speichern, zeitsparendes Übermitteln)!

Hier: **„verlustfreie Kompression“** – Information ist unverändert vorhanden.

Gegensatz: z. B. MP3: Informationsverlust bei der Kompression.

a	A	B	C	D	E	F	G	H	I	K
$p(a)$	0,15	0,08	0,07	0,10	0,21	0,08	0,07	0,09	0,06	0,09
$c_1(a)$	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
$c_2(a)$	1100	0110	000	111	10	0011	010	0010	0111	1101

Wir codieren eine Datei T mit 100000 Buchstaben aus A, wobei die relative Häufigkeit von $a \in A$ durch $p(a)$ gegeben ist.

Mit c_1 (fixe Codewortlänge): **400000** Bits.

Mit c_2 (variable Codewortlänge):

$$\big(4 \cdot (0,15 + 0{,}08 + 0{,}08 + 0{,}09 + 0{,}06 + 0{,}09) + 3 \cdot (0{,}07 + 0{,}10 + 0{,}07) + 2 \cdot 0{,}21\big) \cdot 100000 = \mathbf{334000} \text{ Bits.}$$

Das ist eine Ersparnis von 16%. Bei langen Dateien und wenn die Übertragung teuer oder langsam ist, lohnt es sich, die Buchstaben abzuzählen, die relativen Häufigkeiten $p(a)$ zu bestimmen und einen guten Code mit unterschiedlichen Codewortlängen zu suchen.

Definition 10.2.2

Ein **Codierungsbaum** für A ist ein Binärbaum T, in dem

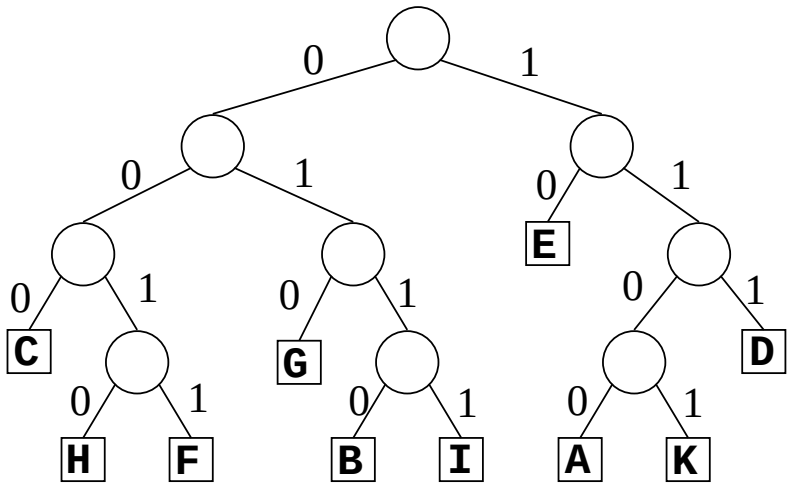
- die Kante in einem inneren Knoten zum linken bzw. rechten Kind (implizit) mit 0 bzw. 1 markiert ist;
- jedem Buchstaben $a \in A$ ein Blatt (externer Knoten) von T exklusiv zugeordnet ist.

Der Code $c_T(a)$ für a ist die Kanteninschrift auf dem Weg von der Wurzel zum Blatt mit Inschrift a.

Die **Kosten** von T unter p sind definiert als:

$$\boldsymbol{B(T,p)} = \sum_{a \in A} p(a) \cdot d_T(a),$$

wobei $d_T(a) = |c_T(a)|$ die Tiefe des a-Blatts in T ist.



Beispiel: Wenn T unser Beispielbaum ist und p die Beispielverteilung von oben, dann ist $B(T,p) = 3{,}34$ ($\ldots = 334\,000/100\,000$, s. Folie 28).

Leicht zu sehen:
$B(T,p) = |c_T(a_1 \ldots a_m)|/m$, wenn $p(a)$ die relative Häufigkeit von a in $w = a_1 \ldots a_m$ ist,
oder
$B(T,p) =$ die **erwartete** **Bitzahl** pro Buchstabe, wenn $p(a)$ die **Wahrscheinlichkeit** von a ist.

Definition 10.2.3

Ein Codierungsbaum T für A heißt **optimal** oder **redundanzminimal** für p, wenn für alle Codierungsbäume T' für A gilt: $B(T,p) \leq B(T',p)$.

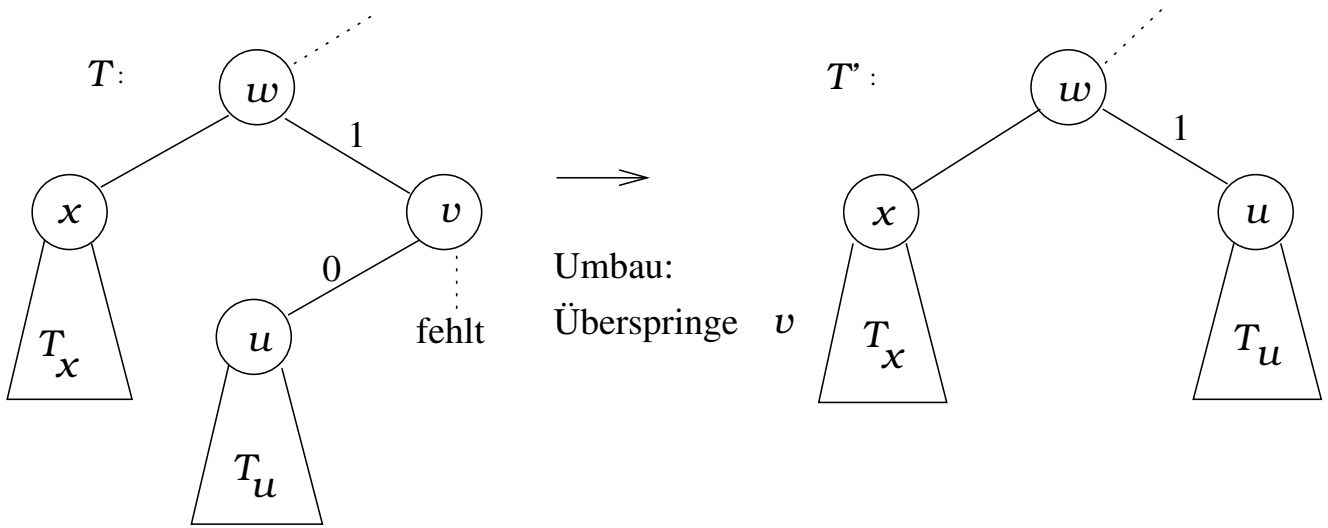
Aufgabe: Zu gegebenem $p\colon A \to [0,1]$ finde einen optimalen Baum T.

Existiert immer ein optimaler Baum? (Zu A gibt es unendlich viele Codierungsbäume!)

Lemma 10.2.4

Wenn T Codierungsbaum und p Verteilung für A ist, dann gibt es einen Codierungsbaum T' mit $B(T', p) \leq B(T, p)$, so dass in T' jeder innere Knoten zwei Kinder hat.

Beweisidee:



Resultat: T' für A mit denselben markierten Blättern wie T, und $B(T', p) \leq B(T, p)$.

Folgerung: Man kann sich bei der Suche nach optimalen Bäumen auf solche beschränken, in denen jeder innere Knoten zwei Kinder hat.

Weil es für festes A nur endlich viele Codierungsbäume gibt, in denen jeder innere Knoten zwei Kinder hat, gibt es für (A, p) optimale Codierungsbäume.

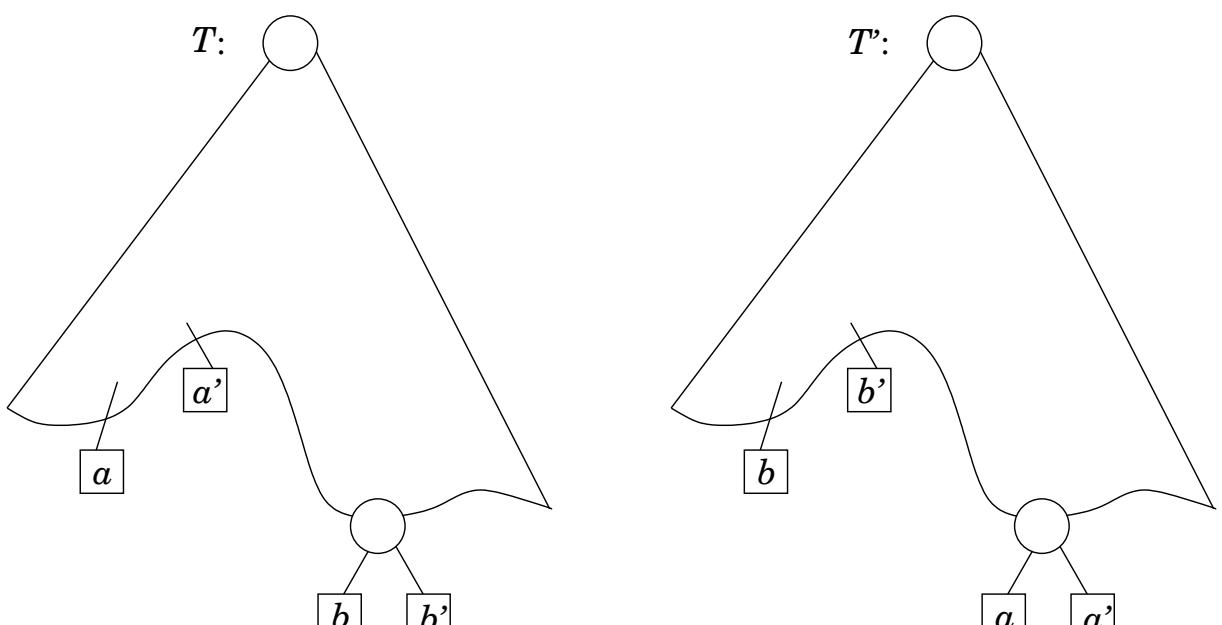
Optimale Bäume sind i. A. nicht eindeutig bestimmt.
(Man kann Knoten auf einem Level beliebig vertauschen.)

Aufgabe: Gegeben (A, p), finde einen optimalen Baum. **Methode: „Greedy“.**

Lemma 10.2.5

Es seien a, a' zwei Buchstaben mit $p(a) \leq p(a') \leq p(b)$ für alle $b \in A - \{a, a'\}$.
(a, a' sind zwei **„seltenste“** Buchstaben.)
Dann gibt es einen optimalen Baum, in dem die Blätter für a und a' „Geschwister“ sind, also Kinder desselben inneren Knotens.

Beweis:



Starte mit beliebigem optimalen Baum T.

O.B.d.A. (Lemma 10.2.4): Alle inneren Knoten haben zwei Kinder.

Es seien $d(a)$ und $d(a')$ die Tiefen der Knoten für a und a'.

Suche zwei Geschwisterknoten mit maximaler Tiefe $d = d(T)$, etwa mit Buchstaben b und b', wobei $p(b) \leq p(b')$.

Vertausche a mit b und a' mit b'!

Dies liefert Baum T', in dem a und a' in Geschwisterknoten sitzen.

Es gilt $B(T, p) \geq B(T', p)$, weil $p(a), p(a') \leq p(b) = p(b')$.

Also ist auch T' optimal für (A, p). □

Damit ist der erste Schritt zur Realisierung eines Greedy-Ansatzes getan!

Man beginnt den Algorithmus mit

„Mache die beiden seltensten Buchstaben zu Geschwistern.“

Dann ist sicher, dass dies stets zu einer optimalen Lösung ausgebaut werden kann.

Dann werden diese beiden Buchstaben „zusammengeklebt“, so dass man ein Alphabet erhält, das eine um 1 kleinere Größe hat.

Konzeptuell wendet man dann **Rekursion** an.

In der Realisierung wird der Algorithmus **iterativ** programmiert.

Algorithmus wurde 1951 von D. A. Huffman (1925–1999), gefunden.

Nette Geschichte: `http://www.huffmancoding.com/my-uncle/scientific-american`

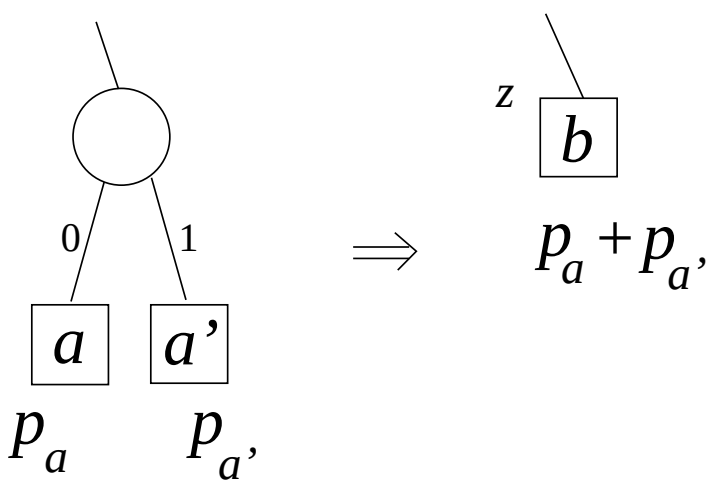
Huffman-Algorithmus (rekursiv):

Wir bauen **„bottom-up“** einen Baum auf.

Wenn $|A| = 1$: Baum hat nur einen (externen) Knoten.
(Code für den einen Buchstaben ist das leere Wort. Seltsam, aber als Rekursionsbasis geeignet.)

Die Kosten sind 0. (Optimalität ist klar.)

Wenn $|A| > 1$, werden zwei „seltenste“ Buchstaben a, a' aus A zu benachbarten Blättern gemacht.

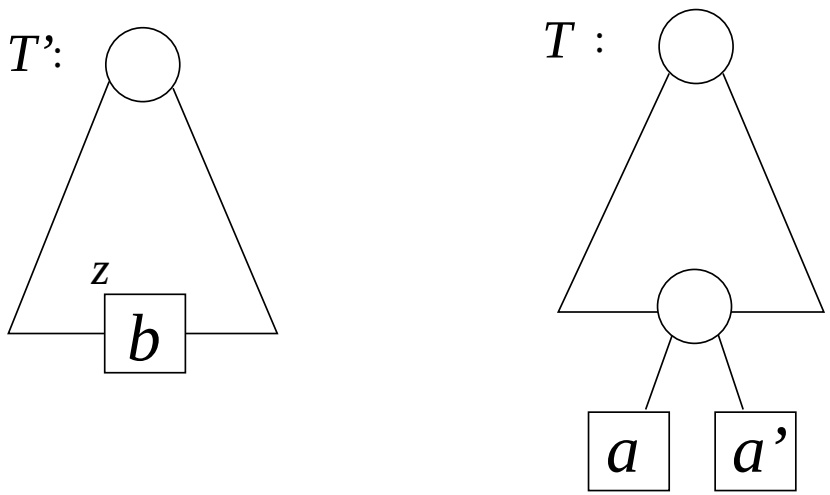


Die Wurzel des so erzeugten Mini-Baums wird als ein neuer Knoten z mit „Kunstbuchstaben“ b aufgefasst, mit Wahrscheinlichkeit $p(b) := p(a) + p(a')$.

Neues Alphabet: $A' := (A - \{a, a'\}) \cup \{b\}$; neue Verteilung:

$$p'(d) := \begin{cases} p(d), & \text{falls } d \neq b, \\ p(a) + p(a'), & \text{falls } d = b. \end{cases}$$

Wir rufen den Algorithmus **rekursiv** auf, um einen Baum T' für A' und p' zu erhalten.



In T' fügen wir an der Stelle des b-Knotens z den a, a'-Baum ein. Dies liefert einen **Codierungsbaum** T für A, mit

$$B(T,p) = B(T',p') + p(a) + p(a').$$

(Checken!)

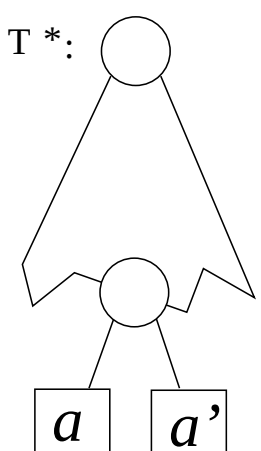
Lemma 10.2.6 T ist **optimaler** Baum für (A, p).

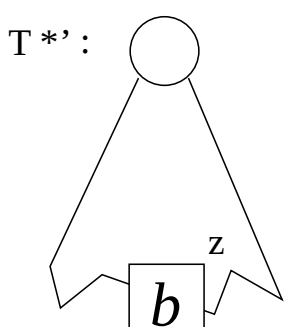
Beweis: Durch Induktion über die rekursiven Aufrufe. Der I.A. ($|A| = 1$) ist klar.

Sei nun $|A| \geq 2$, und seien a und a' seltenste Buchstaben in A mit p.

Nach Lemma 10.2.5. gibt es einen **optimalen** Baum T^* für (A, p), in dem die Knoten für a und a' Geschwister sind.

Aus T^* bilden wir $T^{*\prime}$ durch Ersetzen des a, a'-Teilbaums durch den neuen Knoten z mit dem Kunstbuchstaben b. Dann ist $T^{*\prime}$ **Codierungs**baum für A'.





Wir haben

$$B(T^*, p) = B(T^{*\prime}, p') + p(a) + p(a').$$

Nach I.V. für den rekursiven Aufruf ist T' optimal für (A', p'), also

$$B(T', p') \leq B(T^{*\prime}, p').$$

Daher:

$$\begin{aligned} B(T, p) &= B(T', p') + p(a) + p(a') \\ &\leq B(T^{*\prime}, p') + p(a) + p(a') \\ &= B(T^*, p). \end{aligned}$$

Weil T^* optimaler Baum für (A, p) ist, folgt $B(T, p) = B(T^*, p)$, und auch T ist optimal. □

Man *könnte* nach dem angegebenen Muster eine rekursive Prozedur programmieren.

PQ: Datenstruktur Prioritätswarteschlange,
Einträge: Buchstaben und Kunstbuchstaben;
Schlüssel: die Gewichte $p(b)$, b (Kunst-)Buchstabe.

Operationen: **PQ.insert**: Einfügen eines neuen Eintrags;
PQ.extractMin: Entnehmen des Eintrags mit kleinstem Schlüssel.

Beide Operationen benötigen **logarithmische Zeit** (s. Kap. 6).

Anfangs in **PQ**: Buchstaben $a \in A$ mit Gewichten $p(a)$ als Schlüssel.

Ermitteln und Entfernen der beiden „seltensten“ Buchstaben a, a' durch zwei Aufrufe **PQ.extractMin**;

Einfügen des neuen Kunstbuchstabens b durch **PQ.insert**(b).

Resultierende Rechenzeit: $O(n \log n)$, für $|A| = n$.

Iterative Implementierung mit ein paar Tricks wird **viel effizienter**.

Sei $A = \{a_1, \ldots, a_n\}$. Namen der Buchstaben spielen gar keine Rolle! O.B.d.A.: $1, \ldots, n$.

Wahrscheinlichkeiten: $p_1, \ldots, p_n$.

- Sortieren am Anfang sorgt dafür, dass $p(a_1) \leq \cdots \leq p(a_n)$ (benötigt Zeit $O(n \log n)$).
- Nummern $1, \ldots, n$ für Blätter (Buchstaben aus A),
  Nummern $n+1, \ldots, 2n-1$ für innere Knoten (Kunstbuchstaben).
- Spezielle Darstellung des Codierungsbaums (nur Indizes anstelle von Vorgängerzeigern).
- Beobachtung: Neu erzeugte Häufigkeiten $p(b)$ sind (schwach) monoton wachsend.

Datenstrukturen: Drei Arrays

$\texttt{p}[1..2n-1]$:	Wahrscheinlichkeiten der (Kunst-)Buchstaben,
$\texttt{pred}[1..2n-2]$:	Vorgängerknoten, $2n-1$ ist die Wurzel,
$\texttt{mark}[1..2n-2]$:	Eingangskante mit 0 oder mit 1 markiert?

Algorithmus Huffman(p[1..n])

Input: Gewichts-/Wahrscheinlichkeitsvektor p[1..n], sortiert: p[1] $\leq \cdots \leq$ p[n].

Output: Optimaler Baum T, dargestellt als pred[1..$2n-2$] und mark[1..$2n-2$]

(0) Erweitere den Vektor p[1..n] auf Länge $2n-1$;
(1) pred[1] $\leftarrow n+1$; pred[2] $\leftarrow n+1$;
(2) mark[1] $\leftarrow 0$; mark[2] $\leftarrow 1$;
(3) p[$n+1$] $\leftarrow$ p[1] + p[2];
(4) k $\leftarrow 3$ // erster nicht verarbeiteter Knoten in [1..n]
(5) h $\leftarrow n+1$ // erster nicht verarbeiteter Knoten in [$n+1$..$2n-2$]
(6) **for** b **from** $n+2$ **to** $2n-1$ **do**
(7) // Die folgenden beiden Zeilen finden, in i und j, die Positionen der
(8) // beiden noch nicht verarbeiteten Buchstaben mit kleinsten Gewichten
(9) **if** k $\leq n$ and p[k] $\leq$ p[h] **then** i $\leftarrow$ k; k++ **else** i $\leftarrow$ h; h++;
(10) **if** k $\leq n$ and (h = b or p[k] $\leq$ p[h]) **then** j $\leftarrow$ k; k++ **else** j $\leftarrow$ h; h++;
(11) pred[i] $\leftarrow$ b; pred[j] $\leftarrow$ b;
(12) mark[i] $\leftarrow 0$; mark[j] $\leftarrow 1$;
(13) p[b] $\leftarrow$ p[i] + p[j];
(14) **Ausgabe:** pred[1..$2n-2$] und mark[1..$2n-2$].

Aus pred$[1..2n-2]$ und mark$[1..2n-2]$ baut man den optimalen Huffman-Baum wie folgt:

Alloziere ein Array leaf$[1..n]$ mit Blattknoten-Objekten
und ein Array inner$[n+1..2n-1]$ mit Objekten für innere Knoten.

```
(1)   for i from 1 to n do
(2)     leaf[i].letter ← Buchstabe a_i.
(3)     if mark[i] = 0
(4)       then inner[pred[i]].left ← leaf[i]
(5)       else inner[pred[i]].right ← leaf[i]
(6)   for i from n + 1 to 2n − 2 do
(7)     if mark[i] = 0
(8)       then inner[pred[i]].left ← inner[i]
(9)       else inner[pred[i]].right ← inner[i]
(10)  return inner[2n − 1] // Wurzelknoten
```

Achtung: Wenn man den Algorithmus von Hand ausführt, ist es viel einfacher, den Baum von der Wurzel beginnend zu zeichnen.

a_i	A	B	C	D	E	F	G	H	I	K
i	1	2	3	4	5	6	7	8	9	10
p_i	0,15	0,08	0,07	0,10	0,21	0,08	0,07	0,09	0,06	0,09
pred										
mark										

Wir sortieren und nummerieren um.

(I.A.: $O(n \log n)$ Rechenzeit.)

Dann: Verlängerung der Arrays.

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred										
mark										

i	11	12	13	14	15	16	17	18	19
p_i									
pred									—
mark									—

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred	**11**	**11**								
mark	**0**	**1**								

i	11	12	13	14	15	16	17	18	19
p_i	**0,13**								
pred									—
mark									—

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred	11	11	12	12	13	13	**14**	**14**		
mark	0	1	0	1	0	1	**0**	**1**		

i	11	12	13	14	15	16	17	18	19
p_i	0,13	0,15	0,17	**0,19**					
pred									—
mark									—

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred	11	11	12	12	13	13	14	14	**15**	
mark	0	1	0	1	0	1	0	1	**1**	

i	11	12	13	14	15	16	17	18	19
p_i	0,13	0,15	0,17	0,19	**0,28**				
pred	**15**								—
mark	**0**								—

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred	11	11	12	12	13	13	14	14	15	
mark	0	1	0	1	0	1	0	1	1	

i	11	12	13	14	15	16	17	18	19
p_i	0,13	0,15	0,17	0,19	0,28	**0,32**			
pred	15	**16**	**16**						—
mark	0	**0**	**1**						—

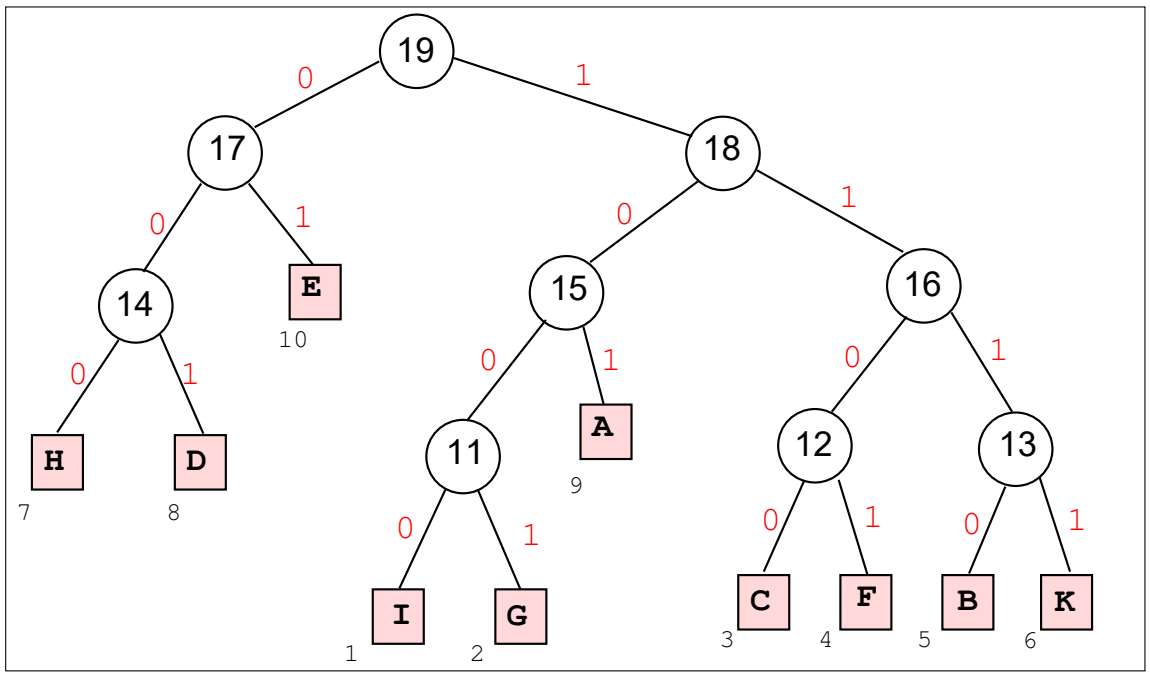
a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred	11	11	12	12	13	13	14	14	15	17
mark	0	1	0	1	0	1	0	1	1	1

i	11	12	13	14	15	16	17	18	19
p_i	0,13	0,15	0,17	0,19	0,28	0,32	0,40	0,60	1,00
pred	15	16	16	17	18	18	19	19	—
mark	0	0	1	0	0	1	0	1	—

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
pred	11	11	12	12	13	13	14	14	15	17
mark	0	1	0	1	0	1	0	1	1	1

i	11	12	13	14	15	16	17	18	19
pred	15	16	16	17	18	18	19	19	—
mark	0	0	1	0	0	1	0	1	—

Resultierender optimaler Codierungsbaum T, Kodierungsfunktion c_3:



	A	B	C	D	E	F	G	H	I	K
$p(a)$	0,15	0,08	0,07	0,10	0,21	0,08	0,07	0,09	0,06	0,09
c_3	101	1110	1100	001	01	1101	1001	000	1000	1111

$$B(T,p) = 0{,}21 \cdot 2 + (0{,}1 + 0{,}09 + 0{,}15) \cdot 3 + 0{,}45 \cdot 4 = \mathbf{3{,}24}.$$

Satz 10.2.7

Der Algorithmus **Huffman** ist korrekt und hat Laufzeit $O(n \log n)$, wenn n die Anzahl der Buchstaben des Alphabets A bezeichnet.

Das Material in diesem Schlussabschnitt ist interessant, aber **nicht prüfungsrelevant**.

Kann man $B(T,p)$ für einen optimalen Baum einfach aus den Häufigkeiten $p(a_1),\ldots,p(a_n)$ **berechnen**, ohne T zu konstruieren?

Antwort: Ja, zumindest näherungsweise.

Definition 10.2.8

Für $p_1,\ldots,p_n \geq 0$ mit $\sum_{i=1}^{n} p_i = 1$ setze

$$\boldsymbol{H(p_1,\ldots,p_n)} := \sum_{i=1}^{n} p_i \cdot \log(1/p_i).$$

$H(p_1,\ldots,p_n)$ heißt die (binäre) **Entropie** der Verteilung $p_1,\ldots,p_n$.

Beispiele: $H(\frac{1}{2},\frac{1}{4},\frac{1}{4}) = \frac{1}{2}\cdot 1 + 2\cdot\frac{1}{4}\cdot 2 = \frac{3}{2}$, und $H(\frac{1}{8},\ldots,\frac{1}{8}) = 8\cdot\frac{1}{8}\cdot\log(1/\frac{1}{8}) = \log 8 = 3$.

Wenn $p_i = 0$ ist, setzt man $p_i \log(1/p_i) = 0$, was vernünftig ist, weil $\lim_{x\searrow 0} x\cdot\log(1/x) = 0$.

Unser nächstes Ziel: Zusammenhang zwischen **Entropie** $H(p_1, \ldots, p_n)$ und der **erwarteten Bitlänge** der Codierung eines Textes, in dem $n = |A|$ Buchstaben mit Wahrscheinlichkeiten $p_1, \ldots, p_n$ auftreten.

Klassisches Resultat: (Minimaleigenschaft der Entropie.)

Lemma 10.2.9 (Ungleichung von Gibbs)

Es seien $p_1, \ldots, p_n \geq 0$ und $q_1, \ldots, q_n > 0$ mit $\sum_{i=1}^{n} q_i \leq 1 = \sum_{i=1}^{n} p_i$. Dann gilt:

$$\sum_{i=1}^{n} p_i \log(1/q_i) \geq \sum_{i=1}^{n} p_i \log(1/p_i) = H(p_1, \ldots, p_n).$$

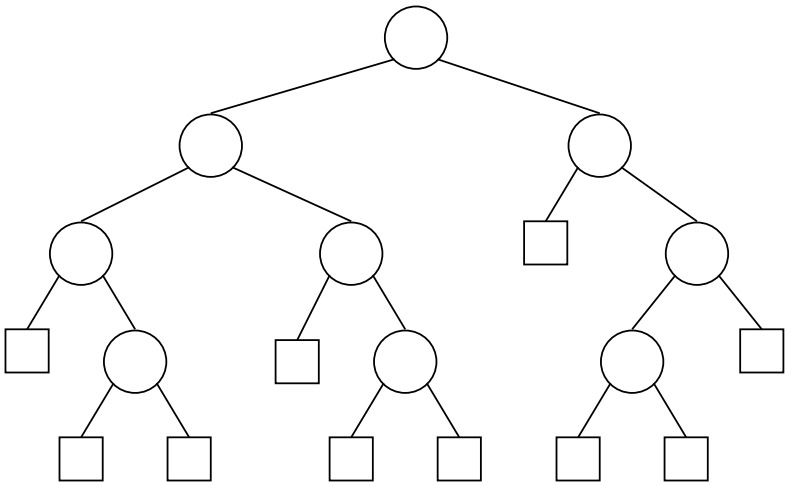
(Die Voraussetzung kann von $\forall i\colon\ q_i > 0$ zu $\forall i\colon\ p_i > 0 \Rightarrow q_i > 0$ abgeschwächt werden.)

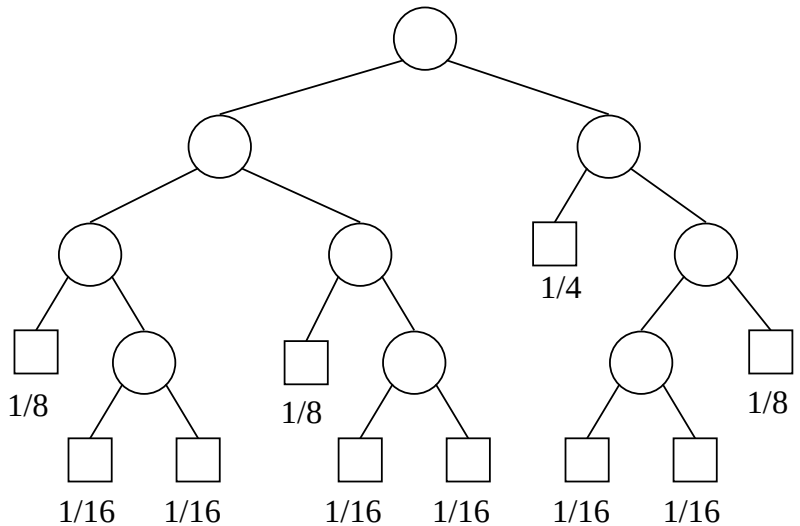
Beweis:

Weil $\log_2 x = (\ln x)/\ln 2$ ist, darf man mit dem natürlichen Logarithmus rechnen.
(Summanden für i mit $p_i = q_i = 0$ lässt man einfach weg.)

$$\begin{aligned}
&\sum_{i=1}^{n} p_i \ln\left(\frac{1}{p_i}\right) - \sum_{i=1}^{n} p_i \ln\left(\frac{1}{q_i}\right) \\
&\quad = \sum_{i=1}^{n} p_i \ln\left(\frac{q_i}{p_i}\right) \overset{(*)}{\leq} \sum_{i=1}^{n} p_i \left(\frac{q_i}{p_i} - 1\right) \\
&\quad = \sum_{i=1}^{n} (q_i - p_i) = \sum_{i=1}^{n} q_i - \sum_{i=1}^{n} p_i \\
&\quad \leq 0.
\end{aligned}$$

$(*)$: Es gilt $\ln x \leq x - 1$, für alle $x > 0$.





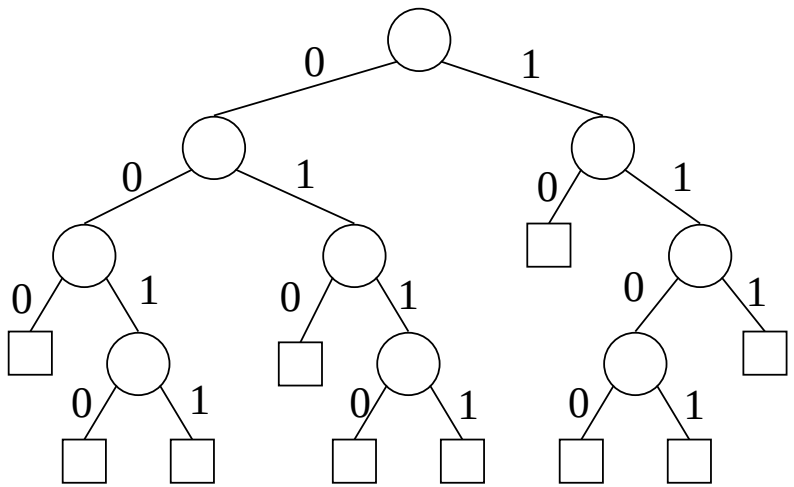
$$\frac{1}{4} + 3 \cdot \frac{1}{8} + 6 \cdot \frac{1}{16} = 1$$

Lemma 10.2.10

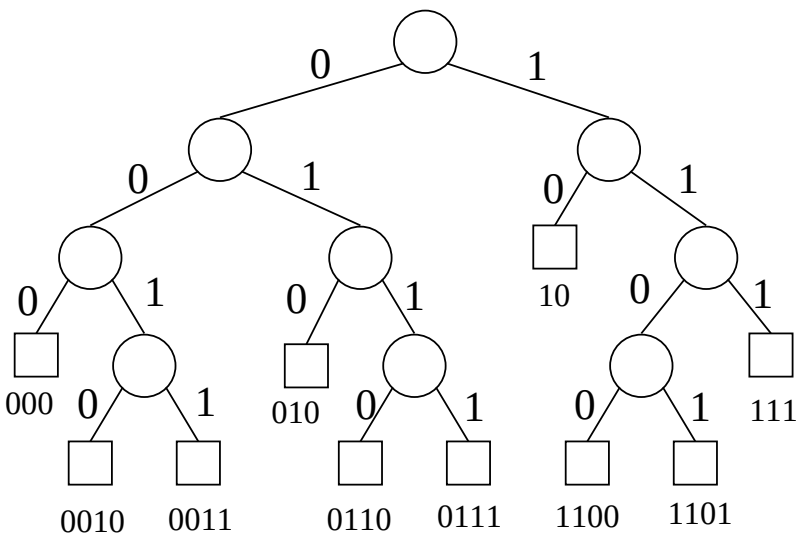
Es sei L die Menge der äußeren Knoten in einem Binärbaum T.

Dann gilt: $$\sum_{l \in L} 2^{-d(l)} = 1.$$

Beweis: Wie in einem Kodierungsbaum markiere in jedem inneren Knoten die Kante zum linken Kind mit 0, die zum rechten Kind mit 1.



Beweis: Wie in einem Kodierungsbaum markiere in jedem inneren Knoten die Kante zum linken Kind mit 0, die zum rechten Kind mit 1.



Jedem Blatt l entspricht dann eine Bitfolge $w_l = b_1 \cdots b_{d(l)}$.

Sei $D = d(T)$ die Tiefe des tiefsten Blatts.

Jedes w_l kann man auf $2^{D-d(l)}$ Arten zu einer Bitfolge der Länge D verlängern.

Alle so erzeugten Bitfolgen sind verschieden; alle 2^D Bitfolgen werden so erzeugt.

$$\Rightarrow \quad \sum_{l \in L} 2^{D-d(l)} = 2^D \quad \Rightarrow \quad \text{Behauptung.} \qquad \square$$

Satz 10.2.11 (Ungleichung von Kraft/McMillan)

Es seien $n \geq 1$, $\ell_1, \ldots, \ell_n \in \mathbb{N}$. Dann gilt:
Es gibt einen Präfixcode mit Codewortlängen $\ell_1, \ldots, \ell_n$ genau dann wenn

$$\sum_{i=1}^{n} 2^{-\ell_i} \leq 1.$$

Nach den Bemerkungen am Anfang von Abschnitt 10.2 kann man statt „Existenz eines Präfixcodes“ auch „Existenz eines Binärbaums mit verschiedenen Blättern auf Niveau $\ell_1, \ldots, \ell_n$ (und eventuell weiteren Blättern)“ sagen.

„$\Rightarrow$“: Sei ein Binärbaum mit n verschiedenen (externen) Blättern auf Tiefe $\ell_1, \ldots, \ell_n$ gegeben. (Eventuell gibt es weitere Blätter.) Aus Proposition 10.2.10 folgt $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$.

„$\Leftarrow$“: Nun seien $\ell_1, \ldots, \ell_n \geq 0$ gegeben, mit $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$.

O. B. d. A. gilt $\ell_1 \leq \cdots \leq \ell_n$ (sonst umordnen und umbenennen).

Wir bauen einen Binärbaum mit Blättern auf Niveaus $\ell_1, \ldots, \ell_n$.

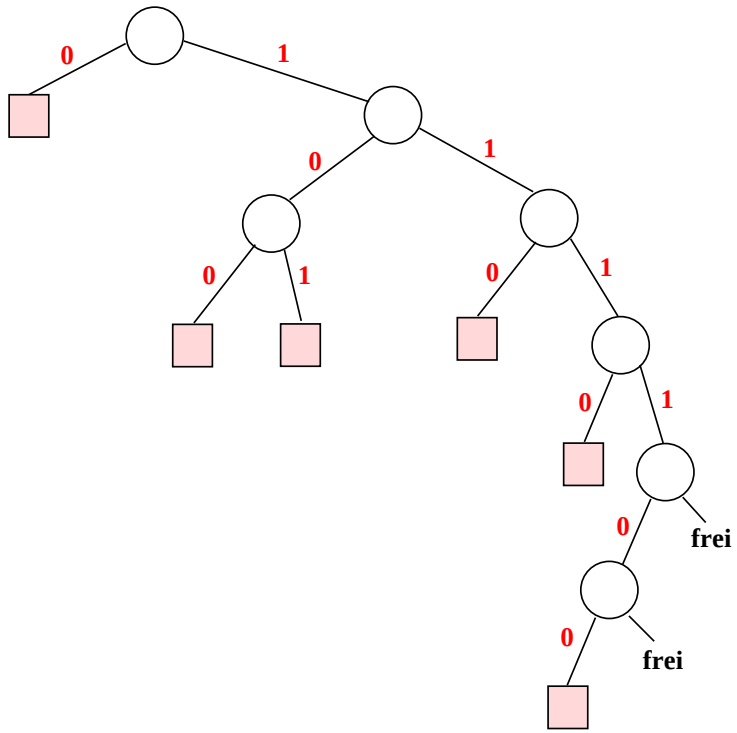
Falls $\ell_1 = 0$ ist, ist $n = 1$. Im Baum gibt es nur ein Blatt auf Niveau 0.

Sonst bauen wir iterativ einen Baum. Anfangs besteht er nur aus einer Wurzel. (Diese Wurzel hat zwei „freie Plätze“ für Kinder auf Niveau 1.)

Dann führen wir Runde k aus, für $k = 1, \ldots, n$:
Suche im Baum einen freien Platz für ein Kind auf Niveau $\ell \leq \ell_k$ und hänge an diese Stelle einen Weg der Länge $\ell_k - \ell + 1$ zu einem neuen Blatt auf Niveau ℓ_k.

Falls $\ell < \ell_k$, entstehen entlang des Wegs neue freie Plätze für Kinder auf Niveaus $\ell + 1, \ldots, \ell_k$.

Beispiel: $(\ell_1, \ldots, \ell_8) = (1, 3, 3, 3, 4, 6, 6, 6, 8)$.



Es werden nacheinander die Blätter mit folgenden Pfaden/Codewörtern angehängt: 0, 100, 101, 110, 1110, 111100.

Man überlege: Wo kommen die nächsten beiden Blätter mit Tiefe 6 hin?

Behauptung: Im Iterationsschritt wird immer ein freier Platz für ein Kind auf Niveau $\ell \leq \ell_k$ gefunden.

Beweis dafür: **Indirekt**. Da vor Runde k nur Blätter auf Niveaus $\leq \ell_k$ eingefügt wurden, sitzen innere Knoten nur auf Niveaus $0, 1, \ldots, \ell_k - 1$.

Annahme: Keiner von diesen inneren Knoten hat einen freien Platz für ein Kind.

Dann liegt ein Binärbaum mit Blättern auf Niveaus $\ell_1, \ldots, \ell_{k-1}$ (alle $\leq \ell_k$) vor, bei dem jeder innere Knoten zwei Kinder hat.

Nach Lemma 10.2.10 folgt $\sum_{1 \leq i \leq k-1} 2^{-\ell_i} = 1$.

Weil $2^{-\ell_k} > 0$, ist dann $\sum_{1 \leq i \leq n} 2^{-\ell_i} > 1$, **Widerspruch**. □

Satz 10.2.12 (Huffman versus Entropie)

Ist $p\colon A \to [0,1]$ mit $\sum_{a\in A} p(a) = 1$ gegeben, so gilt für einen **optimalen** Codierungsbaum T zu (A,p):

$$H(p_1,\ldots,p_n) \leq B(T,p) \leq H(p_1,\ldots,p_n) + 1.$$

(Informal: Setze $p_i = p(a_i)$, für $A = \{a_1,\ldots,a_n\}$. Dann gilt für die Codierung mit einem optimalen Cdierungsbaum: Die erwartete Zahl von Bits, die man braucht, um einen Text $t_1 \ldots t_m$ über A zu codieren, liegt zwischen $m \cdot H(p_1,\ldots,p_n)$ und $m \cdot (H(p_1,\ldots,p_n)+1)$.)

Beweis: 1. Ungleichung:

Es seien $l_1, \ldots, l_n$ die Tiefen der Blätter in T zu den Buchstaben $a_1, \ldots, a_n$.

Dann gilt $\sum_{i=1}^{n} 2^{-l_i} \leq 1$ (nach Satz 10.2.11, Satz von Kraft/McMillan).

Damit können wir Lemma 10.2.9 (Lemma von Gibbs) mit $q_i = 2^{-l_i}$ anwenden und erhalten

$$B(T,p) = \sum_{i=1}^{n} p_i \cdot l_i = \sum_{i=1}^{n} p_i \cdot \log(1/2^{-l_i}) \geq H(p_1, \ldots, p_n).$$

2. Ungleichung: Es genügt zu zeigen, dass ein Codierungsbaum T' für $a_1, \ldots, a_n$ existiert, in dem $B(T', p) \leq H(p_1, \ldots, p_n) + 1$ gilt.

(Ein optimaler Baum T erfüllt ja $B(T, p) \leq B(T', p)$.)

Wir setzen $l_i := \lceil \log(1/p_i) \rceil$, für $1 \leq i \leq n$, und beobachten:

$$\begin{aligned} \sum_{i=1}^{n} 2^{-l_i} &= \sum_{i=1}^{n} 2^{-\lceil \log(1/p_i) \rceil} \leq \sum_{i=1}^{n} 2^{-\log(1/p_i)} \\ &= \sum_{i=1}^{n} p_i = 1. \end{aligned}$$

Nach Satz 10.2.10 („$\Leftarrow$“) existiert also ein Präfixcode $\{x_1, \ldots, x_n\} \subseteq \{0,1\}^*$ mit Codewortlängen $l_1, \ldots, l_n$; im entsprechenden Codierungsbaum T' ordnen wir dem Blatt zu Codewort x_i den Buchstaben a_i zu. Dann ist

$$\begin{aligned} B(T', p) &= \sum_{i=1}^{n} p_i \cdot l_i \leq \sum_{i=1}^{n} p_i \cdot (\log(1/p_i) + 1) \\ &= H(p_1, \ldots, p_n) + \sum_{i=1}^{n} p_i \\ &= H(p_1, \ldots, p_n) + 1. \end{aligned}$$

Bemerkung: Es gibt bessere Kodierungsverfahren als das von Huffman (z. B. „arithmetische Kodierung“; diese vermeiden den Verlust von bis zu einem Bit pro Buchstabe), aber Huffman-Kodierung ist ein guter Anfang, der auch technisch noch eine große Rolle spielt.