B Mathematische Grundlagen

B.1 Mengen, Tupel, Relationen

B.1.1 Mengen

Eine *Menge* fasst mehrere Elemente (z. B. Zahlen, Knoten, Strings, ...) zu einer Einheit zusammen. Üblicherweise werden die geschweiften Klammern „{“ und „}“ verwendet, um eine Menge darzustellen. Eine Menge, die kein Element enthält, wird als die *leere Menge* bezeichnet und üblicherweise durch das Symbol $\emptyset$ notiert. Die Notation einer Menge erfolgt entweder über das Aufzählen ihrer Elemente, wie etwa in folgenden Beispielen:

$$\{1, 10, 100, 2, 20, 200\} \qquad \{a, b, c\} \qquad \{\{1, 2\}, \{8, 9\}, \emptyset, 200\}$$

oder durch eine sog. *Mengenkomprehension*, wie etwa in folgenden Beispielen:

$$\{x \mid x \in \mathbb{N} \ \wedge (x \leq 100 \ \vee x \geq 1000) \ \} \qquad \{x^3 \mid x \in \mathbb{N} \ \wedge \ x \leq 10\}$$

Mengenkomprehensionen ähneln Python's Listenkomprehensionen.

Im Gegensatz zu (mathematischen und Python's) Tupeln und Python's Listen sind Mengen nicht geordnet, d. h. es gibt keine Reihenfolge der Elemente in einer Liste und zwei Mengen gelten als gleich, wenn sie die gleichen Elemente enthalten. Beispielsweise gilt also

$$\{1, 2, 3\} = \{3, 2, 1\} \quad \text{bzw. in Python} \quad \mathit{set}([1,2,3]) == \mathit{set}([3,2,1])$$

B.1.2 Tupel

Auch *Tupel* fassen mehrere Elemente zu einer Einheit zusammen. Im Gegensatz zu Mengen sind sie allerdings geordnet, d. h. die Reihenfolge in der die Elemente im Tupel notiert sind spielt eine Rolle. Daher gilt beispielsweise

$$(x, y) \neq (y, x) \quad \text{falls } x \neq y$$

Das *Kreuzprodukt* zweier Mengen A und B – notiert als $A \times B$ – bezeichnet die Menge aller Tupel, deren erste Komponente Elemente aus A und deren zweite Komponente Elemente aus B enthalten. Formaler kann das Kreuzprodukt folgendermaßen definiert werden:

$$A \times B := \{(x, y) \mid x \in A \ \wedge y \in B\}$$

Kreuzprodukte werden beispielsweise bei der Definition von gerichteten Graphen (siehe Abschnitt 5) verwendet oder bei der Definition von Produktionen einer Grammatik (siehe Abschnitt 6.1).

Mathematische Tupel entsprechen Pythons Tupel und Pythons Listen in der Hinsicht, dass die Reihenfolge der Elemente eine Rolle spielt. Mathematische Mengen entsprechen Objekten mit Pythons *set*-Typ.

B.1.3 Relationen

Formal definiert sich eine Relation über den Mengen A und B als eine Teilmenge des Kreuzproduktes $A \times B$ der beiden Mengen. Relationen werden dazu verwendet, Elemente aus A mit Elementen aus B in Beziehung zu setzen. Beispielsweise stellen folgende Mengen Relationen dar über der Menge $\mathbb{N}$ und der Menge $\{a, b, c\}$ dar:

$$\{(1, a), (2, b), (3, c)\} \quad , \quad \emptyset \quad , \quad \{(i, 1) \mid i \in \mathbb{N}\}$$

Im Folgenden beschreiben wir wichtige Eigenschaften, die eine Relation haben kann; insbesondere ein Verständnis davon, was „*transitiv*“ bedeutet ist eine Voraussetzung für das Verständnis einiger beispielsweise einiger Graphalgorithmen (etwa dem Warshall-Algorithmus – siehe Abschnitt 5.3.2). Eine Reflexion $\mathcal{R} \subseteq A \times A$ heißt ...

... reflexiv , falls $\forall x \in A : (x, x) \in \mathcal{R}$. Eine Relation heißt also genau dann reflexiv, wenn sich alle *alle* Tupel der Form (x, x) für $x \in A$ in $\mathcal{R}$ befinden.

... symmetrisch , falls $(x, y) \in \mathcal{R} \Rightarrow (y, x) \in \mathcal{R}$. Eine Relation heißt alos genau dann symmetrisch, wenn es zu jedem (x, y) in $\mathcal{R}$ auch ein (y, x) in $\mathcal{R}$ gibt.

... anti-symmetrisch , falls $(x, y) \in \mathcal{R} \wedge (y, x) \in \mathcal{R} \Rightarrow x = y$. Eine Relation heißt also genau dann anti-symmetrisch, wenn es keine zwei Elemente (x, y) und (y, x) mit $x \neq y$ in $\mathcal{R}$ gibt. Man beachte, dass „nicht symmetrisch“ nicht gleich „anti-symmetrisch“ ist.

... transitiv , falls $(x, y) \in \mathcal{R} \wedge (y, z) \in \mathcal{R} \Rightarrow (x, z) \in \mathcal{R}$. Eine Relation heißt also genau dann transitiv, wenn – falls zwei Elemente x und z indirekt miteinander in Relation stehen – sie automatisch auch immer direkt miteinander in Relation stehen müssen.

Einige Beispiele:

- $\mathcal{R}_1 = \{(1, 3), (1, 1)\}$ ist nicht reflexiv (z.B. $(2, 2)$ fehlt), nicht symmetrisch (z. B. $(3, 1)$ fehlt), anti-symmetrisch, und transitiv (die Transitivitäts-Bedingung kann mit den beiden vorhandenen Tupeln nicht verletzt werden).
- $\mathcal{R}_2 = \emptyset$ ist nicht reflexiv, symmetrisch (es sind keine Tupel in der Relation, die die Symmetriebedingung verletzen könnten), anti-symmetrisch und transitiv.
- $\mathcal{R}_3 = \{(x, y) \mid x, y \in \mathbb{N},\ x = y\}$ ist reflexiv, anti-symmetrisch und transitiv.

Aufgabe B.1

Betrachten Sie die folgenden Relationen und begründen Sie, ob diese reflexiv, symmetrisch, anti-symmetrisch oder transitiv sind.

(a) $\mathcal{R}_4 = \{(x, y) \mid x, y \in \mathbb{R}$ und x teilt $y\}$
(b) $\mathcal{R}_5 = \{(x, y) \mid x, y \in \mathbb{N} \wedge x \leq 10 \wedge y \geq 100\}$
(c) $\mathcal{R}_6 = \{(x, y) \mid x, y \in \{a, b, \ldots, z\}$ und x kommt im Alphabet vor $y\}$
(d) $\mathcal{R}_7 = \mathbb{N} \times \mathbb{N}$

Aufgabe B.2

Schreiben Sie eine Python-Funktion ...

(a) ... *isReflexive* (*A*,*R*), die testet, ob die als Sequenz von Paaren übergebene Relation *R* reflexiv ist. Der Parameter *A* soll hierbei die Grundmenge spezifizieren. Beispielanwendung:

```
>>> isReflexive ( [ 1,2,3,4 ], [    (1,1),(1,2),(2,2),(4,2),(3,3),(4,4)    ])
>>> True
```

(b) ... *isSymmetric*(*A*,*R*), die testet, ob die als Sequenz von Paaren übergebene Relation *R* symmetrisch ist. Der Parameter *A* soll hierbei die Grundmenge spezifizieren.

(c) ... *isAntiSymmetric*(*A*,*R*), die testet, ob die als Sequenz von Paaren übergebene Relation *R* anti-symmetrisch ist. Der Parameter *A* soll hierbei die Grundmenge spezifizieren.

(d) ... *isTransitive* (*A*,*R*), die testet, ob die als Sequenz von Paaren übergebene Relation *R* transitiv ist. Der Parameter *A* soll hierbei die Grundmenge spezifizieren.

Die *transitive Hülle* einer Relation $\mathcal{R}$ ist definiert als die „kleinste“ (betreffend der Ordnungsrelation „$\subseteq$“; d. h. mit möglichst wenig Elementen) transitive Relation, die $\mathcal{R}$ enthält.

Aufgabe B.3

Was ist die transitive Hülle der Relation ...

(a) $\mathcal{R} = \{(1, 2), (2, 1), (4, 1), (2, 3)\}$, über $A = \{1, 2, 3, 4, 5\}$
(b) $\mathcal{R} = \{(4, 2), (1, 2), (2, 3), (3, 4)\}$, über $A = \{1, 2, 3, 4, 5\}$

B.1.4 Vollständige Induktion

Die Beweistechnik der vollständigen Induktion wird in der Mathematik häufig verwendet, wenn es um Beweise von Aussagen über ganze Zahlen geht. Aussagen dieser Art sind in der diskreten Mathematik und der Zahlentheorie – und damit auch in der Algorithmik – häufig anzutreffen.

Außerdem lohnt sich ein Verstehen dieser Beweistechnik schon allein deshalb, weil diese eng verwandt mit der Implementierungstechnik der Rekursion ist.

Ein Induktionsbeweis einer über eine ganze Zahl parametrierten Aussage $A(n)$ – die im nächsten Abschnitt vorgestellte Summenformel ist etwa eine solche Aussage – gliedert sich in zwei Teile:

Induktionsanfang: Hier wird die Aussage zunächst für den Fall $n = 0$ bzw. $n = 1$ – bzw. je nachdem ab welchem n die zu zeigende Aussage gültig ist – gezeigt. Der Induktionsanfang ist eng verwandt mit dem Rekursionsabbruch.

Induktionsschritt: Hier wird die Implikation $A(k) \Rightarrow A(k+1)$ gezeigt. Man geht also zunächst hypothetisch davon aus, dass $A(k)$ gilt und versucht aus dieser Annahme (auch als *Induktionshypothese* bezeichnet) die Gültigkeit der Aussage $A(k+1)$ abzuleiten. Man beachte hier wiederum die Analogie mit der Rekursion: Auch bei der Programmierung des Rekursionsschritts muss man davon ausgehen, dass der Aufruf mit dem „kleineren“ Argument das richtige Ergebnis liefert; aus dieser Annahme versucht man dann, das Ergebnis für das größere Argument zu konstruieren.

Wir geben ein Beispiel und zeigen über vollständige Induktion, dass für alle $n \in \mathbb{N}$ der Ausdruck $4n^3 - n$ immer durch 3 teilbar ist.

- Induktionsanfang: Es gilt $4 \cdot 1^3 - 1 = 3$ ist durch 3 teilbar.
- Induktionsschritt: Wir nehmen als an, dass $4k^3 - k$ durch 3 teilbar sei und wollen unter Verwendung dieser Annahme zeigen, dass dann auch $4(k+1)^3 - (k+1)$ durch 4 teilbar ist:

$$\begin{aligned} 4(k+1)^3 - (k+1) &= 4(k^3 + 3k^2 + 3k + 1) - k - 1 = 4k^3 + 12k^2 + 11k + 3 \\ &= (4k^3 - k) + (12k^2 + 12k + 3) = (4k^3 - k) + 3(4k^2 + 4k + 1) \end{aligned}$$

  Da laut Induktionshypothese der linke Summand durch drei teilbar ist und auch der rechte Summand durch 3 teilbar ist, ist der Induktionsschritt gezeigt.

B.1.5 Summenformel

Satz 1

Es gilt für alle $n \in \mathbb{N}$, dass

$$\sum_{i=1}^{n} = \frac{n \cdot (n+1)}{2}$$

Am einfachsten lässt sich der Satz mit vollständiger Induktion über n beweisen. Ein konstruktiver Beweis, wie er wohl schon vom jungen Carl-Friedrich Gauß erfolgte, verwendet die Tatsache, dass sich die Summe der ersten n Zahlen zusammen mit der Summe der rückwärts gezählten ersten n Zahlen einfach berechnen lässt. Es gilt nämlich:

$$\begin{aligned}\sum_{i=1}^{n} + \sum_{i=n}^{1} &= 1 + 2 + \ldots n - 1 + n \\ &+ n + n - 1 + \ldots 2 + 1 \\ &= (n+1) + \ldots + (n+1) \\ &= n \cdot (n+1)\end{aligned}$$

B.2 Fibonacci-Zahlen

Leonardo da Pisa, auch unter dem Namen „Fibonacci“ bekannt, war ein italienischer Mathematiker und vielleicht einer der bedeutensten Mathematiker des Mittelalters. Er veröffentlichte das „Buch der Rechenkunst“ (Liber abbaci) das in seinem Anspruch und seiner theoretischen Durchdringung vieler mathematischer Fragestellungen (vor allem aus dem Bereich der natürlichen Zahlen) weit über Niveau anderer mittelalterlicher Werke hinausging.

Definition. Der Wert der i-ten Fibonacci-Zahl F_i (für $i \geq 0$) lässt sich wie folgt rekursiv definieren:

$$\begin{aligned}F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-2} + F_{i-1}, \quad \text{für } i \geq 2\end{aligned}$$

Wenden wir diese Definition an, so erhalten wir also:

$$\begin{aligned}F_2 &= F_0 + F_1 = 1, \\ F_3 &= F_2 + F_1 = 2, \\ F_4 &= F_3 + F_2 = 3, \\ &\ldots\end{aligned}$$

Folgende Pythonprozedur setzt direkt die Definition um und berechnet die n-te Fibonacci-Zahl:

```
def F(n):
  if n==0: return 0
  if n==1: return 1
 return F(n-2) + F(n-1)
```

Aufgabe B.4

(a) Erklären Sie, warum die Laufzeit von der eben vorgestellten Python-Funktion „*F*“ sehr ungünstig ist und schätzen Sie die Laufzeit ab.

(b) Implementieren Sie eine nicht-rekursive Funktion *fib* (n), die die Liste der ersten n Fibonacci-Zahlen berechnet. Anstatt rekursiver Aufrufe sollten die Fibonacci-Zahlen in einer Liste gespeichert werden und bei der Berechnung des nächstens Wertes auf die schon in der Liste gespeicherten Werte zurückgegriffen werden.

(c) Geben Sie unter Verwendung von *fib* einen Python-Ausdruck an, der überprüft, ob die Formel

$$F_{n+2} = 1 + \sum_{i=0}^{n} F_n$$

für alle $n \leq 1000$ gilt.

Eigenschaften. Um Laufzeit-Eigenschaften von Fibonacci-Heaps zu zeigen, benötigen wir einige Eigenschaften von Fibonacci-Zahlen.

Satz 2

Sei F_i die i-te Fibonacci-Zahl. Dann gilt, dass

$$F_{n+2} = 1 + \sum_{i=0}^{n} F_i$$

Wir zeigen Satz 2 durch vollständige Induktion über n.

$n = 0$: In diesem Fall ist zu zeigen, dass $F_2 = 1$; nach Definition der Fibonacci-Zahlen ist dies offensichlich der Fall.

$k - 1 \to k$: Es gilt

$$F_{k+2} = F_{k+1} + F_k \overset{\text{I.H.}}{=} (1 + \sum_{i=0}^{k-1}) + F_k = 1 + \sum_{i=0}^{k} F_i$$

womit der Induktionsschritt und damit die Aussage bewiesen ist.

Satz 3

Für alle $n \in \mathbb{N}$ gilt, dass $F_{n+2} \geq \varphi^n$, wobei $\varphi = (1 + \sqrt{5})/2$ (der „Goldene Schnitt“) ist.

Auch Satz 3 können wir einfach durch vollständige Induktion über $n \geq 2$ zeigen.

$n = 2$: Es gilt, dass $F_2 = 1 \geq \varphi^0 = 1$.

$> k \to k$: Es gilt:

$$F_{k+2} = F_k + F_{k+1} \overset{\text{I.H.}}{\geq} \varphi^{k-2} + \varphi^{k-1} = \varphi^{k-2}(1+\varphi) = \varphi^{k-2}\varphi^2$$

Zur Begründung des letzten Schritts bleibt zu zeigen, dass $1 + \varphi = \varphi^2$:

$$\varphi^2 = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{1+2\sqrt{5}+5}{4} = \frac{2(3+\sqrt{5})}{4} = \frac{2+1+\sqrt{5}}{2} = 1+\varphi$$

Damit ist auch der Induktionsschritt und somit die Aussage gezeigt.

B.3 Grundlagen der Stochastik

Die Stochastik befasst sich mit der Untersuchung von Zufallsexperimenten und mit Gesetzmäßigkeiten der Eintrittswahrscheinlichkeit von Ereignissen. Für unsere Belange genügen die Grundlagen der Stochastik etwa für die Analyse der Average-Case-Laufzeiten von Algorithmen (hierfür arbeitet man häufig mit Zufallsvariablen – wie etwa bei der Average-Case-Analyse von Quicksort – siehe Abschnitt 2.3.3) oder für das Verständnis von randomisierten Algorithmen und Datenstrukturen wie etwa Bloomfilter (siehe Abschnitt 3.5) oder Skip-Listen (siehe Abschnitt 3.6).

B.3.1 Wahrscheinlichkeitsraum

Bei der Modellierung „unsicherer“ Situationen definiert man sich einen *Wahrscheinlichkeitsraum*, der meist mit dem griechischen Buchstaben Ω („Omega“) bezeichnet wird und folgendermaßen definiert ist.

Definition B.1 *Wahrscheinlichkeitsraum, Elementarereignis*

Ein (diskreter) Wahrscheinlichkeitsraum ist bestimmt durch ...

- ... eine Menge $\Omega = \{e_0, e_1, \ldots\}$ von Elementarereignissen.
- ... eine Zuordnung der Elementarereignisse e_i zu einer Wahrscheinlichkeit $\Pr[e_i]$, wobei gelten muss:
  1. $0 \leq \Pr[e_i] \leq 1$
  2. $\sum_{e \in \Omega} \Pr[e] = 1$

Entscheidend ist insbesondere die Eigenschaft, dass die Summe der Wahrscheinlichkeiten aller Elementarereignisse immer eins sein muss, d. h. ein Wahrscheinlichkeitsraum muss insofern „vollständig“ sein, als dass immer sicher (eben mit Wahrscheinlichkeit „1“) eines der Elementarereignisse eintreten muss.

Beispielsweise könnte man für die Modellierung eines Zufallsexperiments „Würfeln mit einem sechseitigen Würfel“ den Wahrscheinlichkeitsraum $\Omega = \{1, 2, 3, 4, 5, 6\}$ mit $\Pr[e] = 1/6$ für alle $e \in \Omega$ wählen.

Neben Elementarereignissen ist auch der Begriff des „Ereignisses“ wichtig:

Definition B.2 *Ereignis*

Eine Menge $E \subseteq \Omega$ heißt Ereignis. Die Wahrscheinlichkeit $\Pr[E]$ ist definiert als

$$\Pr[E] = \sum_{e \in E} \Pr[e]$$

In gewissem Sinn ist also der „Operator“ Pr überladen und sowohl für Elementarereignisse als auch für Mengen von Elementarereignissen definiert.

Einfache Eigenschaften. Es gilt:

- $\Pr[\emptyset] = 0$
- $\Pr[\Omega] = 1$
- $\Pr[E] = 1 - \Pr[\overline{E}]$, wobei $\overline{E} = \Omega \setminus E$. Diese Tatsache ist beispielsweise relevant für Abschnitt 3.5.3.
- $\Pr[E'] \leq \Pr[E]$, falls $E' \subseteq E$.
- $\Pr[E] = \Pr[E_1] + \ldots + \Pr[E_n]$, falls $E = \bigcup_{i=1}^{n} E_i$ und die Ereignisse E_i paarweise disjunkt.

Unabhängigkeit von Ereignissen. Der Eintritt eines Ereignisses kann von dem eines anderen Ereignisses abhängen bzw. unabhängig sein. Hierfür definieren wir formal:

Definition B.3 *Unabhängigkeit von Ereignissen*

Zwei Ereignisse A und B nennt man *unabhängig*, wenn gilt:

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$$

Intuitiv bedeutet die Unabhängigkeit zweier Ereignisse A und B: Man kann aus dem Wissen, dass A eingetreten ist keine Rückschlüsse auf den Eintritt von B ziehen (und umgekehrt).

B.3.2 Laplacesches Prinzip

Spricht nichts dagegen, gehen wir (wie in obigem einfachen Beispiel eines Wahrscheinlichkeitsraums) davon aus, dass alle Elementarwahrscheinlichkeiten gleichwahrscheinlich sind und folglich gilt:

$$\Pr[e] = \frac{1}{|\Omega|} \qquad \text{bzw.} \qquad \Pr[E] = \frac{|E|}{|\Omega|}$$

Beispiel: Es gibt $\binom{49}{6}$ mögliche Lottoziehungen. Folglich ist die Wahrscheinlichkeit, 6 Richtige zu raten, genau $1/\binom{49}{6}$.

B.3.3 Zufallsvariablen und Erwartungswert

Oft will man jedem Ausgang eines Zufallsexperiments eine bestimmte Zahl zuordnen. Bei einem Würfelspiel würde etwa jedes Ereignis einem bestimmten Gewinn (bzw. negativem Gewinn bei Verlust) entsprechen; bei einem randomisierten Algorithmus würde jedes Ereignis bestimmten Rechen„kosten“ entsprechen. Hierfür definieren wir:

Definition B.4 *Zufallsvariable*

Sei ein Wahrscheinlichkeitsraum auf der Ergebnismenge Ω gegeben. Eine Abbildung $X : \Omega \to \mathbb{R}$ heißt Zufallsvariable.

Ein Beispiel: Wir modellieren einen 4-maligen Münzwurf einer Münze mit „Wappen“ W und Zahl Z und interessieren uns dafür, wo oft „Zahl“ fällt. Hierzu verwenden wir den Wahrscheinlichkeitsraum

$$\Omega = \{W, Z\}^4 \quad (:= \ \{W, Z\} \times \{W, Z\} \times \{W, Z\} \times \{W, Z\} \ )$$

d. h. Ω enthält alle möglichen 4-Tupel, deren Komponenten aus der Menge $\{W, Z\}$ kommen. Die Zufallsvariable $Y : \Omega \to \{0, 1, 2, 3, 4\}$ ordnet jedem Elementarereignis aus Ω die Anzahl der Zahlwürfe zu. Beispielsweise gilt $Y((K, Z, K, Z)) = 2$.

Oft interessiert man sich für die Wahrscheinlicheit, mit der eine Zufallsvariable X bestimmte Werte annimmt. Man schreibt:

- $\Pr[X = i]$ für $\Pr[\{e \in \Omega \mid X(e) = i\}]$
- $\Pr[X \leq i]$ für $\Pr[\{e \in \Omega \mid X(e) \leq i\}]$
- $\Pr[j \leq X \leq i]$ für $\Pr[\{e \in \Omega \mid j \leq X(e) \leq i\}]$
- $\Pr[X^2 \leq i]$ für $\Pr[\{e \in \Omega \mid (X(e))^2 \leq i\}]$

...

Für obige Beispiel-Zufallsvariable Y gilt etwa $\Pr[Y \leq 3] = 1 - \Pr[Y = 4] = 1 - (1/2)^4 = 15/16$.

Man kann jeder Zufallsvariablen zwei Funktionen zuordnen:

Dichte und Verteilung. Die Funktion $f_X : \mathbb{R} \to [0, 1]$ mit $f_X(i) = \Pr[X = i]$ heißt *Dichte* von X. Die Dichte f_X ordnet also jeder reellen Zahl i die Wahrscheinlichkeit zu, dass die Zufallsvariable diesen Wert i annimmt.

Die Funktion $F_X : \mathbb{R} \to [0, 1]$ mit $F_X(i) = \Pr[X \leq i]$ heißt *Verteilung* von X. Die Verteilung F_X ordnet also jeder reellen Zahl i die Warhscheinlichkeit zu, dass die Zufallsvariable einen Wert kleiner (oder gleich) i annimmt.

Die Abbildungen B.1 und B.2 zeigen jeweils ein Beispiel einer Dichte und Verteilung.

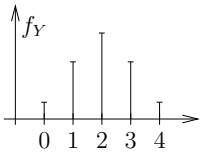

Abb. B.1: *Dichte der oben definierten Zufallsvariablen Y.*

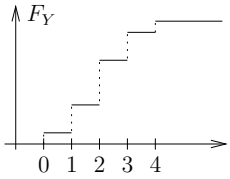

Abb. B.2: *Verteilung der oben definierten Zufallsvariablen Y.*

Erwartungswert. Oft interessiert man sich für die Frage, welchen Wert eine Zufallsvariable im Durchschnitt liefert. Hierzu wird der Erwartungswert definiert:

Definition B.5

Sei X eine Zufallsvariable mit $X : \Omega \to W_X$. Dann ist der Erwartungswert $\mathrm{E}[X]$ definiert als:

$$\mathrm{E}[X] := \sum_{i \in W_X} i \cdot \Pr[X = i]$$

Bemerkung: Man kann den Erwartungswert auch alternativ über die Elementarereignisse wie folgt berechnen (was tatsächlich in vielen Fällen der einfachere Weg ist):

$$\mathrm{E}[X] = \sum_{e \in \Omega} X(e) \cdot \Pr[e]$$

Ein Beispiel: Der Erwartungswert $\mathrm{E}[Y]$ der oben definierten Zufallsvariablen Y (die die Anzahl der Zahlwürfe bei einem 4-maligen Münzwurf zählt) ist gemäß obiger Definition:

$$\mathrm{E}[Y] = 1 \cdot \Pr[Y = 1] + 2 \cdot \Pr[Y = 2] + 3 \cdot \Pr[Y = 3] + 4 \cdot \Pr[Y = 4]$$

Aufgabe B.5

Berechnen Sie das Ergebnis obiger Summe, d. h. berechnen Sie den konkreten Wert für den Erwartungswert $\mathrm{E}[Y]$.

B.3.4 Wichtige Verteilungen

Zufallsvariablen sind eigentlich vollständig über ihre Dichten bzw. Verteilung bestimmt. Man kann daher auch die Verteilungen untersuchen, ohne auf ein konkretes Zufallsexperiment Bezug zu nehmen.

Die Bernoulli-Verteilung. Die Zufallsvariable $X : \Omega \to \{0, 1\}$ mit Dichte

$$f_X(i) = \begin{cases} p & \text{für} \quad i = 1 \\ 1 - p & \text{für} \quad i = 0 \end{cases}$$

heißt *Bernoulli-verteilt*. Der Parameter p heißt *Erfolgswahrscheinlichkeit*. Es gilt, dass $\mathrm{E}[X] = p$, d. h. der erwartete Wert ist p (der natürlich nie eintritt, aber der Erwartungswert selbst muss auch nicht notwendigerweise im Wertebereich der Zufallsvariablen liegen).

Binomialverteilung. Ist eine Zufallsvariable X als Summe $X := X_1 + \ldots + X_n$ von n unabhängigen Bernoulli-verteilten Zufallsvariablen (mit gleicher Erfolgswahrscheinlichkeit p) definiert, so heißt X *binomialverteilt* mit Parameter n und p. Man schreibt auch

$$X \sim Bin(n, p)$$

wenn man zum Ausdruck bringen möchte, dass die Zufallsvariable X binomialverteilt ist.

Für den Wertebereich W_X einer binomialverteilten Zufallsvariablen X gilt $W_X = \{0, 1, \ldots, n\}$. Für die Dichte f_X der Binomialverteilung gilt

$$f_X(i) = \binom{n}{i} \cdot p^i \cdot (1 - p)^{n-i}$$

Beispielsweise war die oben beispielhaft definierte Zufallsvariable Y, die die Zahlwürfe bei 4-maligem Münzwurf zählt, binomialverteilt mit Parameter $n = 4$ und $p = 1/2$.

Geometrische Verteilung. Diese Wahrscheinlichkeit ist insbesondere relevant bei der Bestimmung der Höhe eines neu einzufügenden Elements in einer Skip-Liste (siehe Abschnitt 3.6 auf Seite 93) und entsprechend bei der Laufzeitbetrachtung der Such-, Einfüge-, und Löschoperation auf Skip-Listen.

Eine geometrische Verteilung liegt dann vor, wenn bei einem Experiment eine Aktion so lange wiederholt wird, bis sie „erfolgreich“ ist. Sei p die Wahrscheinlichkeit, dass ein Versuch erfolgreich ist. Die Zufallsvariable X enthält als Wert die Anzahl der Versuche, bis Erfolg eintritt. Die Dichte der geometrischen Verteilung ist dann

$$f_X(i) = (1 - p)^{i-1} \cdot p$$

Für den Erwartungswert $\mathrm{E}[X]$ der geometrischen Verteilung gilt $\mathrm{E}[X] = 1/p$.

Aufgabe B.6

Rechnen Sie mit Hilfe der Definition des Erwartungswerts nach, dass bei einer geometrisch verteilten Zufallsvariablen X gilt, dass $\mathrm{E}[X] = 1/p$.

Ein Beispiel: Steht in einem Rechnernetz eine bestimmte Leitung nur mit einer Wahrscheinlichkeit von $p = 1/10$ zur Verfügung, dann sind durchschnittlich $1/p = 10$ Versuche notwendig, bis ein Datenpaket erfolgreich über die Leitung verschickt werden kann.

B.4 Graphen, Bäume und Netzwerke

In vielen Anwendungen (Kürzeste Wege, Optimale Flüsse in Netzwerken, Suchen) bilden Graphen das angemessenste mathematische Modell für denjenigen Ausschnitt der Wirklichkeit in dem ein bestimmtes Problem gelöst werden soll.

B.4.1 Graphen

Ein Graph $G = (V, E)$ besteht aus einer Menge V von *Knoten* und einer Menge E von Kanten (=Verbindungen) zwischen den Knoten. Man unterscheidet *gerichtete* Graphen, bei denen die Richtung der Verbindung zwischen zwei Knoten eine Rolle spielt und *ungerichtete* Graphen, bei denen diese Richtung keine Rolle spielt. Bei gerichteten Graphen werden Kanten mathematisch als Knotentupel repräsentiert; bei ungerichteten Graphen werden Kanten mathematisch als 2-elementige Teilmengen aus der Knotenmenge repräsentiert. Abbildung B.3 zeigt links ein Beispiel für einen gerichteten und rechts ein Beispiel für einen ungerichteten Graphen.

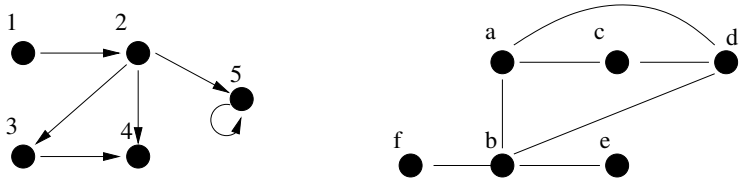

Abb. B.3: *Linkes Bild: eine graphische Veranschaulichung eines gerichteten Graphen* $G_1 = (V_1, E_1)$ *mit der Knotenmenge* $V_1 = \{1, 2, 3, 4, 5\}$ *und der Kantenmenge* $E_1 = \{(1, 2), (2, 3), (2, 4), (3, 4), (2, 5), (5, 5)\}$. *Rechtes Bild: eine graphische Veranschaulichung eines ungerichteten Graphen* $G_2 = (V_2, E_2)$ *mit der Knotenmenge* $V_2 = \{a, b, c, d, e, f\}$ *und der Kantenmenge* $E_2 = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, e\}, \{b, f\}, \{b, d\}, \{c, d\}\}$.

Definitionen.

Nachbarschaft Man definiert die Nachbarschaft $\Gamma(i)$ eines Knoten $i \in V$ in einem gerichteteten Graphen $G = (V, E)$ folgendermaßen:

$$\Gamma(i) := \{\ j \mid (i, j) \in E\ \}$$

Die Nachbarschaft eines Knotens in einem ungerichteten Graphen definiert man, indem man einfach (i, j) durch $\{i, j\}$ ersetzt.

Grad eines Knotens Die Größe der Nachbarschaft eines Knotens i bezeichnet man auch als *Grad* des Knotens und schreibt:

$$deg(i) := |\Gamma(i)|$$

Pfad Ein (ungerichteter) Pfad eines Graphen $G = (V, E)$ ist eine Folge $(v_0, v_1, \ldots, v_n)$ von Knoten mit $\{v_i, v_{i+1}\} \in E$. Ein (gerichteter) Pfad eines Graphen $G = (V, E)$ ist eine Folge $(v_0, v_1, \ldots, v_n)$ von Knoten mit $(v_i, v_{i+1}) \in E$.
Die Länge eines Pfades ist n.

Weg Ein (ungerichteter) Weg eines Graphen ist ein Pfad dieses Graphen, in dem alle Knoten paarweise verschieden sind. Ein (gerichteter) Weg eines Graphen ist ein Pfad dieses Graphen, in dem alle Knoten paarweise verschieden sind. Die Länge eines Weges ist n.

Zyklus / Kreis Ein (ungerichteter) Kreis ist ein Pfad $(v_0, \ldots, v_n)$ für den gilt, dass $\{v_0, v_n\} \in E$. Ein (gerichteter) Kreis ist ein Pfad $(v_0, \ldots, v_n)$, $n \geq 2$, für den gilt, dass $(v_0, v_n) \in E$.
Beispiele:

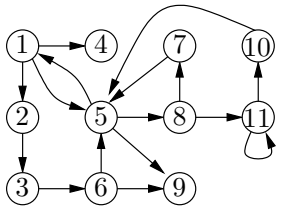

Abb. B.4: *Gerichteter Graph*
Pfade: (1,5,1,4), (7), (3,6,9), ...
Wege: (7), (3,6,9), ...
Kreise: (5,8,7), (1,5)

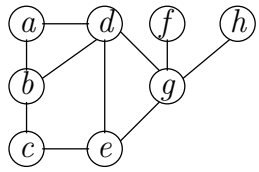

Abb. B.5: *Ungerichteter Graph*
Pfade: (d,e,g,d,a), (f), (a,b) ...
Wege: (f), (a,b), (b,c,e,g) ...
Kreise: (a,b,d), (d,e,g), ...

DAG Ein DAG (engl: Directed Acyclic Graph) bezeichnet einen gerichteten kreisfreien Graphen.

Baum Ein kreisfreier, zusammenhängender Graph. Für einen Baum $G = (V, E)$ gilt immer, dass $|E| = |V| - 1$.
Beispiel: Entfernt man etwa vom dem in Abbildung B.3 gezeigten ungerichteten Graphen die Kanten $\{a, d\}$ und $\{b, d\}$, so erhält man einen Baum – wie im linken Teil der Abbildung B.6 zu sehen. Der rechte Teil der Abbildung zeigt denselben Graphen – nur so gezeichnet, dass er als Wurzelbaum mit Wurzelknoten a gesehen werden kann.

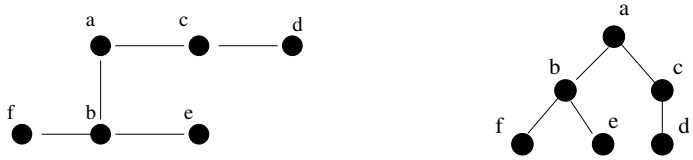

Abb. B.6: *Das linke Bild zeigt den Graphen G_3, der aus dem in Abbildung B.3 gezeigten Graphen G_2 nach Entfernen der Kanten $\{a, d\}$ und $\{b, d\}$ entstanden ist. Dieser Graph ist ein Baum. Das rechte Bild zeigt denselben Graphen G_3, der nun aber so gezeichnet ist, dass er als Wurzelbaum mit Wurzelknoten a interpretiert werden kann.*

Wurzelbaum In der Informatik werden Bäume häufig dazu verwendet, Informationen so abzulegen, dass sie schnell wiedergefunden werden können. Hierbei handelt es sich meist um sogenannte *Wurzelbäume*, in denen ein bestimmter Knoten als die Wurzel des Baumes definiert wird. Alternativ kann man einen Wurzelbaum auch definieren als einen kreisfreien gerichteten Graphen, bei dem ein spezieller Knoten als Wurzel gekennzeichnet ist.

Höhe eines Knotens (in einem Wurzelbaum) entspricht der Länge des längsten Pfades von diesem Knoten zu einem Blattknoten.

Spannbaum Als Spannbaum bezeichnet man einen Teilgraphen $G_T = (V_T, E_T)$ eines ungerichteten zusammenhängenden Graphen $G = (V, E)$, der ein Baum (also kreisfrei und zusammenhängend) ist. Der Teilgraph muss alle Knoten des Graphen enthalten, es muss also gelten: $V_T = V$ und $E_T = E$. Abbildung B.7 zeigt ein einfaches Beispiel eines Spannbaums (unter vielen Möglichen).

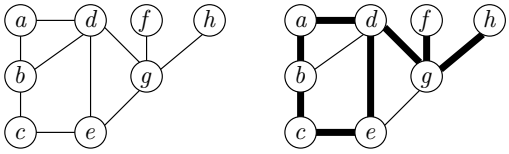

Abb. B.7: *Zwei verschiedene (von vielen möglichen) Spannbäume des in Abbildung B.5 gezeigten Graphen, zu sehen in Form der fett gezeichneten Kanten.*

Zusammenhang Ein ungerichteter Graph heißt *zusammenhängend*, wenn es für jedes Knotenpaar $i, j \in V, i \neq j$ einen Pfad von i nach j gibt.
Ein gerichteter Graph heißt schwach zusammenhängend, wenn der zugrundeliegende ungerichtete Graph (den man einfach dadurch erhält, in dem man jede Kante (i, j) durch eine entsprechende Kante $\{i, j\}$ ersetzt) zusammenhängend ist.
Ein gerichteter Graph heißt stark zusammenhängend (oder kurz einfach: zusammenhängend) wenn es für jedes Knotenpaar $i, j \in V, i \neq j$ einen Pfad von i nach j gibt.

Beispielsweise ist der Abbildung B.4 gezeigte Graph zwar schwach zusammenhängend, nicht jedoch stark zusammenhängend.

(Zusammenhangs-)Komponente Ein maximaler zusammenhängender Teilgraph eines ungerichteten Graphen G heißt *Zusammenhangskomponente* (oder oft auch nur: *Komponente*).

Aufgabe B.7

Bestimmen Sie für obige Beispielgraphen:

(a) $\Gamma(2)$ und $deg(2)$
(b) $\Gamma(1)$ und $deg(1)$

B.5 Potenzmengen

Die Potenzmenge $\mathcal{P}(M)$ einer Menge M ist definiert als die Menge aller Teilmengen von M; formaler:

$$\mathcal{P}(M) := \{N \mid N \subseteq M\}$$

Beispielsweise gilt, dass

$$\mathcal{P}(\{1,2,3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$$

Wir wollen uns überlegen, wie man die Potenzmenge eine Menge M in Python berechnen kann; wir repräsentieren hierbei Mengen als Listen. Systematisch kann man sich für das eben erwähnte Beispiel folgendes Vorgehen vorstellen: Zunächst erzeugt man alle Teilmengen, die die 1 nicht enthalten und danach alle Teilmengen, die die 1 enthalten, also

$$\mathcal{P}([1,2,3]) = [\overbrace{[\,],\quad [2],\quad [3],\quad [2,3]]}^{=\mathcal{P}([2,3])},\quad \overbrace{[\mathbf{1}], [\mathbf{1},2], [\mathbf{1},3], [\mathbf{1},2,3]]}^{=\xrightarrow{[1]+}\mathcal{P}([2,3])}$$

Man sieht, dass die erste Hälfte genau dem Wert von $\mathcal{P}([2,3])$ entspricht; auch die zweite Hälfte basiert auf den Werten aus $\mathcal{P}([2,3])$, nur dass vor jeder der Teilmengen die 1 angefügt wird. Daraus ergibt sich sehr direkt folgende Python-Implemetierung der Potenzmengen-Funktion:

```
1 def pot(l):
2   if l==[]: return [[]]
3   return pot(l[1:]) + map(lambda p: [l[0]] + p, pot(l[1:]))
```

Aufgabe B.8

(a) Wieviele Elemente hat $\mathcal{P}(M)$?

(b) Was ist der Wert von $len(pot(pot(pot([0,1]))))$?

B.5.1 Permutationen

Eine Permutation ist eine endliche bijektive Abbildung $\pi : X \to X$; endliche bedeutet: $|X| < \infty$, d. h. X enthält nur endlich viele Elemente; bijektiv bedeutet: für jedes $x_i \in X$ gibt es genau ein $x_j \in X$ mit $\pi(x_i) = x_j$, d. h. es gibt eins-zu-eins-Verhältnisse zwischen Bild- und einem Urbildwerten.

Da Permutationen endliche Abbildungen sind, können sie durch Auflistung aller möglichen Bild-Urbild-Paare dargestellt werden. Angenommen $X = \{1, \ldots n\}$, dann könnte man eine Permutation folgendermaßen darstellen:

$$\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix}$$

Ist klar und eindeutig, in welcher Reihenfolge die Bildwerte angeordnet werden können, so kann die erste Zeile auch weg gelassen werden.

Es gibt immer $n!$-viele verschiedene Permutation einer n-elementigen Menge. Dies kann man mit folgender Überlegung einfach nachvollziehen: Nimmt man das erste Elemente

aus der Menge, so gibt es n verschiedene Möglichkeiten dieses zu platzieren (nämlich an die Position 1 oder an die Position 2, usw.). Nimmt man anschließend das zweite Element aus der Menge, so gibt es noch $n-1$ verschiedene Möglichkeiten, dieses zu platzieren. Für *jede* der n Möglichkeiten, das erste Element zu platzieren gibt es also $n-1$ Möglichkeiten, das zweite Element zu platzieren, insgesamt also $n \cdot (n-1)$ Möglichkeiten, die ersten beiden Elemente zu platzieren, usw. Also gibt es insgesamt $n \cdot (n-1) \cdot \ldots \cdot 1 = n!$ Möglichkeiten die Elemente der n-elementigen Menge anzuordnen.

Mit Hilfe einer Listenkomprehension kann man relativ einfach eine Python-Funktion schreiben, die die Liste aller Permutation einer Menge (in Python repräsentiert als Liste) zurückliefert.

```
1 def perms(xs):
2   if xs == []: return [[]]
3   return [i for perm in perms(xs[1:]) for i in ins(xs[0],perm)]
```

Listing B.1: *Implementierung einer Funktion perms, die die Liste aller Permutationen der als Argument übergebenen Liste xs zurückliefert.*

Hierbei wird eine Hilfsfunktion $ins(x, xs)$ benötigt (siehe Aufgabe B.9), die die Liste aller möglichen Einfügungen des Elements x in die Liste xs zurückliefert.

Zeile 2 implementiert den Rekursionsabbruch: die einzige Permutation der leeren Liste ist wiederum die leere Liste. Bei der Implementierung des Rekursionsschrittes erfolgt der rekursive Aufruf $perms(xs[1:])$, angewendet auf die kürzere Liste $xs[1:]$. Wir nehmen an, der rekursive Auruf arbeitet korrekt – diese Annahme gehört zu der in Abschnitt 1.2.1 besprochenen Denk-Strategie für die Programmierung rekursiver Funktionen. Unter dieser Annahme fragen wir uns, wie wir die „kleinere“ Lösung $perms(x[1:])$ anreichern müssen, um $perms(xs)$ zu erhalten. Wir betrachten das in Abbildung B.8 gezeigte Beispiel: Um aus den Permutationen der Elemente aus $[2,3]$ die Permutationen der Elemente aus $[1,2,3]$ zu erhalten, muss mittels der Funktion ins das erste Element – in diesem Fall ist das die „1“ – in jede Position jeder Permutation eingefügt werden. Dies

$$
\begin{array}{rcl}
perms([2,3]) & = & \underset{\Downarrow\ ins(1,[2,3])}{[2,3]} \qquad\qquad \underset{\Downarrow\ ins(1,[3,2])}{[3,2]} \\
perms([1,2,3]) & = & \overbrace{[\mathbf{1},2,3] \quad [2,\mathbf{1},3] \quad [2,3,\mathbf{1}]} \quad \overbrace{[\mathbf{1},3,2] \quad [3,\mathbf{1},2] \quad [3,2,\mathbf{1}]}
\end{array}
$$

Abb. B.8: *Konstruktion von $perms([1,2,3])$ – der Liste aller Permutationen der Elemente aus $[1,2,3]$ – aus $perms([1,2])$: Auf jedes Element aus $perms([2,3]$ wird einfach $ins(1,...)$ ausgeführt; alle daraus entstehenden Listen bilden die Permutationen aus $[1,2,3]$.*

wird durch die in Zeile 3 in Listing B.1 gezeigte Listenkomprehension implementiert. Die Variable *perm* läuft über alle Permuationen von $xs[1:]$; für jede dieser Permutationen läuft die Variable i über alle Einfügungen des ersten Elements von xs. Alle diese „Einfügungen“ zusammengenommen ergeben die Liste aller gesuchten Permutationen.

Aufgabe B.9

Implementieren Sie die Funktion $ins(x, xs)$, die die Liste aller möglichen Einfügungen des Elements x in die Liste xs zurückliefert. Beispielanwendung:

```
>>> ins(1, [2,3,4,5])
>>> [[1,2,3,4,5], [2,1,3,4,5], [2,3,1,4,5], [2,3,4,1,5], [2,3,4,5,1]]
```

Tipp: Am einfachsten geht eine rekursive Implementierung. Es empfiehlt sich auch die Verwendung einer Listenkomprehension.

Aufgabe B.10

Implementieren Sie zwei Test-Funktionen, die (teilweise) überprüfen können, ob die Implementierung der in Listing B.1 korrekt war.

(a) Eine Funktion *allEqLen*(*xss*) die überprüft, ob alle in der als Argument übergebenen Liste *xss* enthaltenen Listen die gleiche Länge haben.

(b) Eine Funktion *allEqElems*(*xss*) die überprüft, ob alle in der als Argument übergebenen Liste *xss* enthaltenen Listen die selben Elemente enthalten.

B.5.2 Teilmengen und Binomialkoeffizient

Wie viele k-elementige Teilmengen einer n-elementigen Menge gibt es? Dies ist eine häufige kombinatorische Fragestellungen, die entsprechend häufig auch bei der Entwicklung von Optimierungs-Algorithmen auftaucht – bei der Entwicklung eines Algorithmus zur Lösung des Travelling-Salesman-Problems beispielsweise (siehe Kapitel 8.1.2 auf Seite 238).

Man kann sich wie folgt überlegen, wie viele k-elementige Teilmengen einer n-elementigen Menge es gibt. Betrachten wir zunächst eine verwandte und einfachere Fragestellung: Wie viele k-elementige Tupel aus einer n-elementigen Teilmenge gibt es? – Tupel sind, im Gegensatz zu Mengen, geordnet, d. h. die Reihenfolge, in der sich die Elemente innerhalb eines Tupels befinden, spielt eine Rolle. Für die erste zu besetzende Position haben wir noch n mögliche Elemente zur Wahl; für die zweite Position haben wir nur noch $n-1$ Elemente zu Auswahl, usw. Insgesamt gibt es also $n \cdot (n-1) \cdot \ldots \cdot (n-k+1) = n!/(n-k)!$ viele mögliche k-Tupel. Da jedes Tupel auf $k!$ viele Arten angeordnet werden kann, entsprechen immer genau $k!$ viele Tupel einer k-elementigen Teilmenge. Insgesamt gibt es also $n!/k!(n-k)!$ viele k-elementige Teilmengen einer n-elementigen Menge. Genau diese Zahl nennt man den *Binomialkoeffizienten* und schreibt dafür

$$\binom{n}{k} := \frac{n!}{k!(n-k)!} = \text{Anz. } k\text{-elementiger Teilmengen einer } n\text{-elem. Menge}$$

Für $\binom{n}{k}$ spricht man auch „n über k“.

Es gibt eine rekursive Formel, mit der man den Binomialkoeffizienten ohne Verwendung der Fakultätsfunktion berechnen kann. Diese rekursive Formel kann man sich durch folgende kombinatorische Überlegung herleiten. Die k-elementigen Teilmengen aus der n-elementigen Menge lassen sich aufteilen in zwei Klassen:

1. All die Teilmengen, die das Element „1“ enthalten. Diese Teilmengen bestehen also aus „1“ und einer $(k-1)$-elementigen Teilmenge der $(n-1)$-elementigen Menge $\{2,\ldots,n\}$. Davon gibt es genau $\binom{n-1}{k-1}$ viele.
2. All die Teilmengen, die das Element „1“ nicht enthalten. Diese Teilmengen sind also alle k-elementige Teilmengen der $(n-1)$-elementigen Menge $\{2,\ldots,n\}$. Davon gibt es genau $\binom{n-1}{k}$ viele.

Diese beiden Klassen sind überschneidungsfrei (disjunkt) und daher ist die Anzahl der k-elementigen Teilmengen einer n-elementigen Menge genau die Summe der Elemente der ersten und der zweiten Klasse, d. h. es gilt folgende rekursive Gleichung:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \tag{B.1}$$

Diese Überlegung war konstruktiv: Es ist möglich sich daraus einen Algorithmus abzuleiten. Die in folgendem Listing B.2 gezeigte Implementierung erzeugt gemäß obiger Überlegung alle k-elementigen Teilmengen der übergebenen Liste *lst*:

```
1 def choice( lst ,k):
2   if  lst  == []: return []
3   if  len( lst ) == k: return [lst]
4   if  len( lst ) <= k or k==0: return [[]]
5   return [[lst[0]] + choices for choices in choice( lst [1:],k-1)] + choice( lst [1:],k)
```

Listing B.2: *Implementierung der Funktion choice, die eine Liste aller k-elementigen Teilmengen der Elemente aus lst zurückliefert.*

Genau wie Gleichung B.1 enthält auch die Funktion *choice*(*lst* ,*k*) zwei rekursive Aufrufe die jeweils die um Eins kleinere Liste *lst* [1 :] verwenden: *choice*(*lst* [1 :],*k*-1) und *choice*(*lst* [1 :], *k*).