Kapitel 1

Einführung

1.1 Berechnungsprobleme und Algorithmen

Definition 1.1. Ein *Berechnungsproblem* ist eine Relation der Form $P \subseteq X \times Y$ so dass für alle $x \in X$ ein $y \in Y$ existiert mit $(x, y) \in P$.

Dabei stellen wir uns X als Menge der möglichen Eingaben vor, Y als Menge der möglichen Ausgaben und $(x, y) \in P$ als die Aussage "bei Eingabe x ist y eine korrekte Ausgabe". Oft werden wir statt Berechnungsproblem einfach nur Problem sagen. Ein Berechnungsproblem ist aber zu unterscheiden von einem mathematischen Problem, bei welchem es sich (üblicherweise) um eine Frage der Form "Ist die Aussage ... wahr?" handelt. Beispiele für Berechnungsprobleme sind:

Bestimmung des ggT
Eingabe: positive ganze Zahlen n_1 und n_2
Ausgabe: der größte gemeinsame Teiler von n_1 und n_2

Sortierproblem
Eingabe: eine endliche Folge ganzer Zahlen $(a_1, \ldots, a_n)$
Ausgabe: eine Permutation $(a_{\pi(1)}, \ldots, a_{\pi(n)})$ so dass $a_{\pi(1)} \leq \cdots \leq a_{\pi(n)}$

Linearisierung
Eingabe: eine endliche partiell geordnete Menge $(A, \leq)$
Ausgabe: eine totale Ordnung $a_1, \ldots, a_n$ der Elemente von A so dass $a_i \leq a_j \Rightarrow i \leq j$

Wie man am dritten der obigen Beispiele sehen kann, muss ein Berechnungsproblem nicht unbedingt eine eindeutige Lösung haben. Falls $P \subseteq X \times Y$ ein Problem ist, dann heißt jedes $x \in X$ *Instanz von* P, beispielsweise ist $(3, 7, 2, 5, 8)$ eine Instanz des Sortierproblems.

Definition 1.2. Ein *Algorithmus* ist eine wohldefinierte Rechenvorschrift.

Wir geben hier keine präzisere Definition des Begriffs Algorithmus. Dies zu tun würde im Wesentlichen auf die Definition einer Programmiersprache hinauslaufen und damit am Thema dieser Vorlesung vorbeigehen. Wir werden konkrete Algorithmen in *Pseudocode* angeben, d.h. in einer der Situation angepassten Mischung aus natürlicher Sprache und üblichen Anweisungen und Kontrollstrukturen einer Programmiersprache wie Schleifen, Verzweigungen, usw. Wir verlangen auch nicht formell dass jeder Algorithmus *terminiert*, d.h. dass er für jede Eingabe nach endlicher Zeit stoppt. Allerdings werden wir in dieser Vorlesung fast ausschließlich terminierende Algorithmen betrachten.

Wir kennen bereits viele Algorithmen, zum Beispiel Algorithmen zur Addition und Multiplikation zweier natürlicher Zahlen in Dezimaldarstellung, wie sie in der Volksschule gelehrt werden, den Algorithmus zur Division mit Rest von Polynomen, den euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen, das gaußsche Eliminationsverfahren zur Lösung linearer Gleichungssysteme, usw. Algorithmen sind aus der Mathematik nicht wegzudenken. Anders als bisher werden wir in dieser Vorlesung ein systematisches Studium von Algorithmen betreiben. Dieses beschränkt sich nicht auf die bloße Definition und Verwendung von Algorithmen, sondern untersucht Fragen wie z.B. "Wie effizient ist dieser Algorithmus?" "Wie ist das Verhältnis zwischen zwei Algorithmen?" "Welche Ansätze zur Entwicklung guter Algorithmen gibt es?" "Ist dieser Algorithmus der beste zur Lösung dieses Problems? In welchem Sinn ist er das?" usw. Algorithmen werden also von einem Mittel zur Lösung von Problemen zu einem Objekt unserer Untersuchungen.

Oft operieren Algorithmen auf umfangreichen Daten. Ein wichtiger Aspekt ist dann die Frage in welcher Form bzw. Struktur die Daten gespeichert werden. Man spricht in diesem Zusammenhang auch von *Datenstrukturen*. In dieser Vorlesung werden wir eine Reihe nützlicher und effizienter Datenstrukturen kennenlernen. Eine der einfachsten und gleichzeitig wichtigsten Datenstrukturen ist das *Datenfeld* (engl. *array*). Mathematisch handelt es sich dabei um eine endliche Folge. Die Elemente der Folge werden in aufsteigender Reihenfolge im Speicher abgelegt. Wir schreiben $A, B, \ldots$ für Datenfelder, $A[i]$ für das i-te Element, der kleinste Index eines Datenfelds ist 1, wir schreiben $A.\mathit{Länge}$ für die Länge des Datenfelds (d.h. für die Anzahl von Elementen, die es enthält). Die Notation $A.\mathit{Länge}$ wird in der Informatik verwendet, um das Attribut *Länge* des Objekts A zu notieren. Manche Attribute (wie dieses) können nur gelesen werden (engl. *read-only*), manche können auch geschrieben werden, d.h. in Zuweisungen verwendet werden. Was davon der Fall ist wird üblicherweise aus dem Kontext heraus klar sein.

Algorithmen werden verwendet um Berechnungsprobleme zu lösen. Damit meint man Folgendes:

Definition 1.3. Sei $P \subseteq X \times Y$ ein Berechnungsproblem und $\mathcal{A}$ ein Algorithmus. Wir sagen dass $\mathcal{A}$ *das Problem* P löst falls für jede Instanz $x \in X$ gilt dass $y = \mathcal{A}(x)$ die Eigenschaft $(x, y) \in P$ hat.

Oft wird aus dem Kontext heraus klar sein, welches Berechnungsproblem wir lösen wollen, dann sprechen wir einfach von der *Korrektheit* eines Algorithmus.

1.2 Korrektheitsbeweise

Ein *Korrektheitsbeweis* ist ein Beweis der zeigt, dass ein bestimmter Algorithmus ein bestimmes Berechnungsproblem löst. Um diesen Begriff zu illustrieren wollen wir zunächst ein einfaches Beispiel betrachten. Der folgende Algorithmus erhält ein Datenfeld (dessen Einträge Zahlen sind) als Eingabe und liefert eine Zahl als Ausgabe.

Algorithmus 1 Prozedur P

```
1: Prozedur P(A)
2:     m := 0
3:     Für i = 1, ..., A.Länge
4:         m := m + A[i]
5:     Ende Für
6:     Antworte m/A.Länge
7: Ende Prozedur
```

Mit etwas Programmiererfahrung lässt sich leicht erkennen, dass Algorithmus 1 das folgende Berechnungsproblem löst:

Arithmetisches Mittel
Eingabe: eine endliche Folge $a_1, \ldots, a_n$
Ausgabe: arithmetischer Mittelwert von $a_1, \ldots, a_n$

Wir wollen diese Aussage nun präzise formulieren und beweisen. Ein wesentlicher Aspekt von Korrekheitsaussagen und Korrektheitsbeweisen besteht darin, dass wir es mit zwei unterschiedlichen Sprachebenen zu tun haben: einerseits die Sprachebene des Programms oder Pseudocodes und die in ihm vorkommenden *Programmvariablen*, andererseits die übliche mathematische Sprache in der wir Aussagen *über* den Pseudocode formulieren. Dieser Algorithmus erhält als Eingabe ein Datenfeld A, er benutzt die lokale Variable m und den Schleifenzähler i, der im Körper der Schleife ebenfalls eine lokale Variable ist. Die Programmvariablen sind also A, m und i. Für den Rest der Diskussion dieses Beispiels werden wir für Programmvariablen `Schreibmaschinenschrift` verwenden, d.h. $\mathtt{A}, \mathtt{m}, \mathtt{i}$. Wir können also formulieren:

Satz 1.1. *Algorithmus 1 ist korrrekt, d.h. er löst das Berechnungsproblem "Arithmetisches Mittel", d.h. P(*$\mathtt{A}$*) antwortet mit* $\frac{\sum_{j=1}^{\mathtt{A}.Länge} \mathtt{A}[j]}{\mathtt{A}.Länge}$.

Man beachte dass in obiger Formel die Variable j *keine* Programmvariable ist sondern eine Variable der üblichen mathematischen Sprache. Um diese Aussage formal zu beweisen benutzen wir eine *Schleifeninvariante*. Eine Invariante für eine bestimmte Schleife ist eine logische Aussage I über jene Programmvariablen, die zu Beginn eines Durchlaufs der Schleife verfügbar sind, mit den folgenden Eigenschaften:

1. I ist zu Beginn des ersten Durchlaufs der Schleife wahr und
2. I wird von der Schleife erhalten, d.h. wenn sie zu Beginn des i-ten Durchlaufs wahr ist, dann ist sie auch zu Beginn des $i+1$-ten Durchlaufs wahr.

Damit ist eine Schleifeninvariante auch nach dem letzten Durchlauf der Schleife wahr. Sinnvollerweise wählt man die Invariante so, dass sie am Ende der Schleife eine Form hat, die für den weiteren Korrektheitsbeweis nützlich ist. Deshalb enthält ein Korrektheitsbeweis im Kontext einer Schleifeninvariante noch den weiteren Teil der

3. Verwendung der Invariante im Korrektheitsbeweis.

Auf diese Weise werden in einem Programmablauf mit n Durchläufen einer bestimmte Schleife zusätzlich zum Beginn und zum Ende des Programms noch $n+1$ weitere Zeitpunkte definiert:

1. der Beginn des 1. Schleifendurchlaufs, ..., n. der Beginn des n-ten Schleifendurchlaufs, $n+1$. der Beginn des $n+1$-ten Schleifendurchlaufs der die Schleife beendet. Es ist oft nützlich den Wert einer Programmvariablen $\mathtt{x}$ zum i-ten dieser Zeitpunkte mit $\mathtt{x}_i$ zu bezeichnen. Für Algorithmus 1 erhalten wir so die Werte $\mathtt{m}_k$, $\mathtt{A}_k$ und $\mathtt{i}_k$ für $k = 1, \ldots, n+1$ wobei $n = \mathtt{A}.\mathit{Länge}$. Nun sieht man leicht dass $\mathtt{A}_1 = \cdots = \mathtt{A}_{n+1}$ so dass wir den Index von $\mathtt{A}$ einfach weglassen. Außerdem gilt $\mathtt{i}_k = k$ für $k = 1, \ldots, n+1$.

Beweis von Satz 1.1. Unsere Schleifeninvariante ist:

$$I(\mathtt{A}, \mathtt{m}, \mathtt{i}) \colon \mathtt{m} = \sum_{j=1}^{\mathtt{i}-1} \mathtt{A}[j].$$

Zunächst weisen wir nach, dass diese zu Beginn des ersten Durchlaufs der Schleife wahr ist. Dann ist $k = 1$ und damit erhalten wir:

1. $\mathtt{m}_1 = 0 = \sum_{j=1}^{0} \mathtt{A}[j]$.

Im nächsten Schritt weisen wir nach, dass die Invariante durch die Schleife erhalten wird. Dazu nehmen wir (ähnlich einer Induktionshypothese) an dass die Invariante zu Beginn des i-ten Durchlaufs wahr ist, d.h. $\mathtt{m}_k = \sum_{j=1}^{\mathtt{i}_k - 1} \mathtt{A}[j]$, und erhalten:

2. $\mathtt{m}_{k+1} = \mathtt{m}_k + \mathtt{A}[\mathtt{i}_k] = \sum_{j=1}^{\mathtt{i}_k} \mathtt{A}[j] = \sum_{j=1}^{\mathtt{i}_{k+1} - 1} \mathtt{A}[j]$.

Um den Beweis abzuschließen betrachten wir den Beginn des $n+1$-ten Schleifendurchlaufs. Bei diesem wird nach Inkrementierung von $\mathtt{i}$ festgestellt, dass die Schleife zu beenden ist und damit wird die Prozedur mit Zeile 6 fortgesetzt. Als Antwort der Prozedur erhalten wir also

3. $\mathtt{m}_{n+1}/n = \frac{\sum_{j=1}^{n} \mathtt{A}[j]}{n}$

was die Behauptung zeigt. □

Wie man an obigem Beweis exemplarisch sehen kann handelt es sich bei der Verwendung von Schleifeninvarianten um eine struktierte Vorgehensweise zur Führung von Induktionsbeweisen über imperative Programme. Bei dieser Vorgehensweise verwendet man für jede in einem Algorithmus vorkommende Schleife eine Invariante. Wie sich diese Invarianten zueinander verhalten hängt vom Verhältnis der Schleifen zueinander ab, so entsprechen etwa zwei verschachtelte Schleifen einem verschalteten Induktionsbeweis mittels Schleifeninvarianten. Es ist möglich, in einem präzisen logischen Sinn zu zeigen, dass Beweise mittels Schleifeninvarianten ausreichend sind um Eigenschaften von imperativen Programmen nachzuweisen, d.h. dass jede wahre Aussage über ein Programm durch einen Beweis mit Schleifeninvarianten gezeigt werden kann.

Eines der am häufigsten auftretenden Berechnungsprobleme ist das Sortieren eines Datenfelds. Folglich sind Sortieralgorithmen sehr gründlich untersucht worden. Als nächstes Beispiel wollen wir hier einen ersten Sortieralgorithmus betrachten: Einfügesortieren (engl. *insertion sort*). Die Grundidee besteht darin, ein Datenfeld zu unterteilen in einen bereits sortierten Bereich (links) und einen noch nicht sortierten Bereich (rechts). Der sortierte Bereich wird dann sukzessive vergrößert, indem das erste Element des unsortierten Bereichs in den sortierten Bereich (an der richtigen Stelle) eingefügt wird. Am Ende ist der unsortierte Bereich leer und das Datenfeld damit vollständig sortiert. Die Idee kann wie folgt als Pseudocode notiert werden:

Algorithmus 2 Einfügesortieren

```
1: Prozedur EINFÜGESORTIEREN(A)
2:     Für j := 2, ..., A.Länge
3:         x := A[j]
4:         i := j − 1
5:         Solange i ≥ 1 und A[i] > x
6:             A[i + 1] := A[i]
7:             i := i − 1
8:         Ende Solange
9:         A[i + 1] := x
10:    Ende Für
11: Ende Prozedur
```

Beispiel 1.1. siehe `bsp.einfuegesortieren.pdf`.

Um die Korrektheit von Einfügesortieren zu beweisen besprechen wir zunächst noch kurz einige Begriffe und Notationen die für Korrektheitsbeweise von Algorithmen im Allgemeinen von Bedeutung sind: Eine *Vorbedingung* eines Algorithmus ist eine Bedingung von der wir annehmen, dass sie beim Start des Algorithmus erfüllt ist. Eine *Nachbedingung* ist eine Aussage von der wir zeigen wollen, dass sie nach Ausführung des Algorithmus erfüllt ist falls die Eingabe die Vorbedingung erfüllt hat. Eine Korrektheitsaussage hat also oft die Form "Falls die Vorbedingung ... erfüllt ist, dann ist nach Ausführung des Algorithmus ... die Nachbedingung ..." erfüllt.

Beweise mittels Schleifeninvarianten werden oft systematisch in die oben erwähnten Punkte 1 bis 3 unterteilt. Damit ist an fast jeder Stelle klar auf den Wert zu welchen Zeitpunkt man sich bezieht wenn man eine Programmvariable erwähnt: bei 1 auf den Beginn der Schleife und bei 3 auf das Ende der Schleife. Die einzige Ausnahme ist 2 wo man sich auf den Wert zu zwei unterschiedlichen Zeitpunkte bezieht: zu Beginn der i-ten Iteration und zu Beginn der $i+1$-ten Iteration. Eine gebräuchliche Notation für diese Situation besteht darin den Wert einer Programmvariablen $\mathtt{X}$ zu Beginn der i-ten Iteration einfach als $\mathtt{X}$ zu notieren und jenen zu Beginn der $i+1$-ten Iteration als $\mathtt{X}'$. Auch Varianten davon wie zum Beispiel $\mathtt{X}_{\text{alt}}$ und $\mathtt{X}_{\text{neu}}$ sind in Gebrauch.

Wir wollen nun in dieser kompakteren Notation die Korrektheit von Einfügesortieren beweisen. Dazu beginnen wir mit der folgenden Aussage über die innere Schleife.

Lemma 1.1. *Sei* EINFÜGEN$(\mathtt{A},\mathtt{i},\mathtt{x})$ *eine Abkürzung für die Zeilen 5-9 von Algorithmus 2. Dann gilt für* EINFÜGEN$(\mathtt{A},\mathtt{i},\mathtt{x})$ *unter der Vorbedingung (V)*

$i_0 = \mathtt{i}$, $n = \mathtt{A}$.*Länge,* $\mathtt{A}[1,\ldots,i_0] = m_1,\ldots,m_{i_0}$ *ist sortiert*
und $\mathtt{A}[i_0+1,\ldots,n] = m_{i_0+1},\ldots,m_n$

die Nachbedingung (N)

$\mathtt{A}[1,\ldots,i_0+1] = m_1,\ldots,m_i,\mathtt{x},m_{i+1},\ldots,m_{i_0}$ *ist sortiert*
und $\mathtt{A}[i_0+2,\ldots,n] = m_{i_0+2},\ldots,m_n$.

Beweis. Der Körper der Schleife in EINFÜGEN$(\mathtt{A},\mathtt{i},\mathtt{x})$ induziert die Abbildung $(\mathtt{A},\mathtt{i}) \mapsto (\mathtt{A}',\mathtt{i}')$ wobei

$$\mathtt{A}'[k] = \begin{cases} \mathtt{A}[\mathtt{i}] & \text{falls } k = \mathtt{i}+1 \\ \mathtt{A}[k] & \text{sonst} \end{cases} \qquad \text{und} \qquad \mathtt{i}' = \mathtt{i} - 1.$$

Für diese Schleife verwenden wir die Invariante $I(\mathtt{A},\mathtt{i})$:

$\mathtt{i} \le i_0$,
$\mathtt{A}[1,\ldots,i_0+1] = m_1,\ldots,m_{\mathtt{i}},m_{\mathtt{i}+1},m_{\mathtt{i}+1},\ldots,m_{i_0}$,
$\mathtt{A}[i_0+2,\ldots,n] = m_{i_0+2},\ldots,m_n$
und falls $\mathtt{i} < i_0$ dann $x < m_{i+1}$.

und argumentieren wie folgt:

1. Zu Beginn der Schleife folgt $I(\mathtt{A},\mathtt{i})$ unmittelbar aus (V).
2. $I(\mathtt{A},\mathtt{i})$ impliziert $I(\mathtt{A}',\mathtt{i}')$ da erstens $\mathtt{i}' = \mathtt{i} - 1 \le^{I(\mathtt{A},\mathtt{i})} i_0 - 1 < i_0$ und damit bleibt $\mathtt{A}[i_0+2,\ldots,n]$ unverändert. Weiters ist

$$\begin{aligned} \mathtt{A}'[1,\ldots,i_0+1] &= \mathtt{A}[1,\ldots,\mathtt{i}],\mathtt{A}[\mathtt{i}],\mathtt{A}[\mathtt{i}+2,\ldots,i_0+1] \\ &=^{I(\mathtt{A},\mathtt{i})} m_1,\ldots,m_{\mathtt{i}},m_{\mathtt{i}},m_{\mathtt{i}+1},\ldots,m_{i_0} \\ &= m_1,\ldots,m_{\mathtt{i}'+1},m_{\mathtt{i}'+1},\ldots,m_{i_0}. \end{aligned}$$

Schließlich ist $\mathtt{i}' = \mathtt{i} - 1 < i_0$, also ist zu zeigen dass $x < m_{\mathtt{i}'+1} = m_{\mathtt{i}} =^{I(\mathtt{A},\mathtt{i})} \mathtt{A}[\mathtt{i}]$ was unmittelbar aus der Wahrheit der Schleifenbedingung folgt.

3. Bei Beendigung der Schleife ist $\mathtt{A}[1,\ldots,i_0+1] = m_1,\ldots,m_{\mathtt{i}},m_{\mathtt{i}+1},m_{\mathtt{i}+1},\ldots,m_{i_0}$ mit $x < m_{\mathtt{i}+1}$ und ($\mathtt{i} = 0$ oder ($\mathtt{i} \geq 1$ und $\mathtt{A}[\mathtt{i}] = m_i \leq x$)). Nach Ausführung von Zeile 9 ist damit $\mathtt{A}[1,\ldots,i_0+1] = m_1,\ldots,m_{\mathtt{i}},\mathtt{x},m_{\mathtt{i}}+1,\ldots,m_{i_0}$ sortiert und damit ist (N) gezeigt.

□

Satz 1.2. *Einfügesortieren ist korrekt.*

Beweis. Genauer gesagt wollen wir zeigen dass für EINFÜGESORTIEREN(A) unter der Vorbedingung (V*)

$n = \mathtt{A}.\mathit{Länge}$ und $A[1,\ldots,n] = p_1,\ldots,p_n$

die Nachbedingung (N*)

$\mathtt{A}[1,\ldots,n]$ ist sortierte Permutation von $p_1,\ldots,p_n$

gilt. Dazu verwenden wir für die äußere Schleife die Invariante $I(\mathtt{A},\mathtt{j})$:

$\mathtt{A}[1,\ldots,\mathtt{j}-1]$ ist sortierte Permutation von $p_1,\ldots,p_{\mathtt{j}-1}$ und
$\mathtt{A}[j,\ldots,n] = p_j,\ldots,p_n$.

und argumentieren wie folgt:

1. Zu Beginn der Schleife ist $\mathtt{j} = 2$ und damit folgt $I(\mathtt{A},\mathtt{j})$ unmittelbar aus (V*).

2. Aus $I(\mathtt{A},\mathtt{j})$ folgt (V) von Lemma 1.1 da $i_0 = \mathtt{j} - 1$ und $\mathtt{A}[1,\ldots,\mathtt{j}-1] = p_1,\ldots,p_{\mathtt{j}-1}$ sortiert ist. Also ist wegen Lemma 1.1 nach Ausführung von Zeile 9, und damit zu Beginn des nächsten Durchlaufs, die Nachbedingung (N) erfüllt. Also ist $\mathtt{A}'[1,\ldots,\mathtt{j}] = p_1,\ldots,p_{\mathtt{i}},\mathtt{x},p_{\mathtt{i}+1},\ldots,p_{\mathtt{j}-1}$ und sortiert und $\mathtt{A}'[\mathtt{j}+1,\ldots,n] = A[\mathtt{j}+1,\ldots,n] = p_{\mathtt{j}+1},\ldots,p_n$ woraus $I(\mathtt{A}',\mathtt{j}')$ folgt da $\mathtt{j} = \mathtt{j}' - 1$.

3. Bei Beendigung der Schleife ist $\mathtt{j} = n+1$ und damit folgt aus $I(\mathtt{A},\mathtt{j})$ dass $\mathtt{A}[1,\ldots,n]$ eine sortierte Permutation von $p_1,\ldots,p_n$ ist.

□

1.3 Aufwandsabschätzung

Der wichtigste Aspekt eines Algorithmus, neben seiner Korrektheit, ist typischerweise sein Ressourcenverbrauch, und hier vor allem: seine Laufzeit. Auch der Verbrauch anderer Ressourcen, zum Beispiel Speicherplatz, kann von Bedeutung sein, zentral ist aber in den allermeisten Situationen die Frage wie viel Zeit ein Algorithmus benötigt. Eine erste Antwort auf diese Frage bestünde darin, eine konkrete Implementierung des Algorithmus in einer bestimmten Programmiersprache anzufertigen, diese auf einem bestimmten Computer auf verschiedene Eingaben anzuwenden und die verbrauchte Zeit zu protokollieren. Derartige empirische Studien haben in der Informatik durchaus ihren Nutzen, allerdings haben sie die folgenden Nachteile:

1. Es ist unmöglich alle, typischerweise unendlich vielen, Eingaben auszuprobieren.

2. Die Ergebnisse hängen von der Implementierung, der Programmiersprache und vom Computer ab.

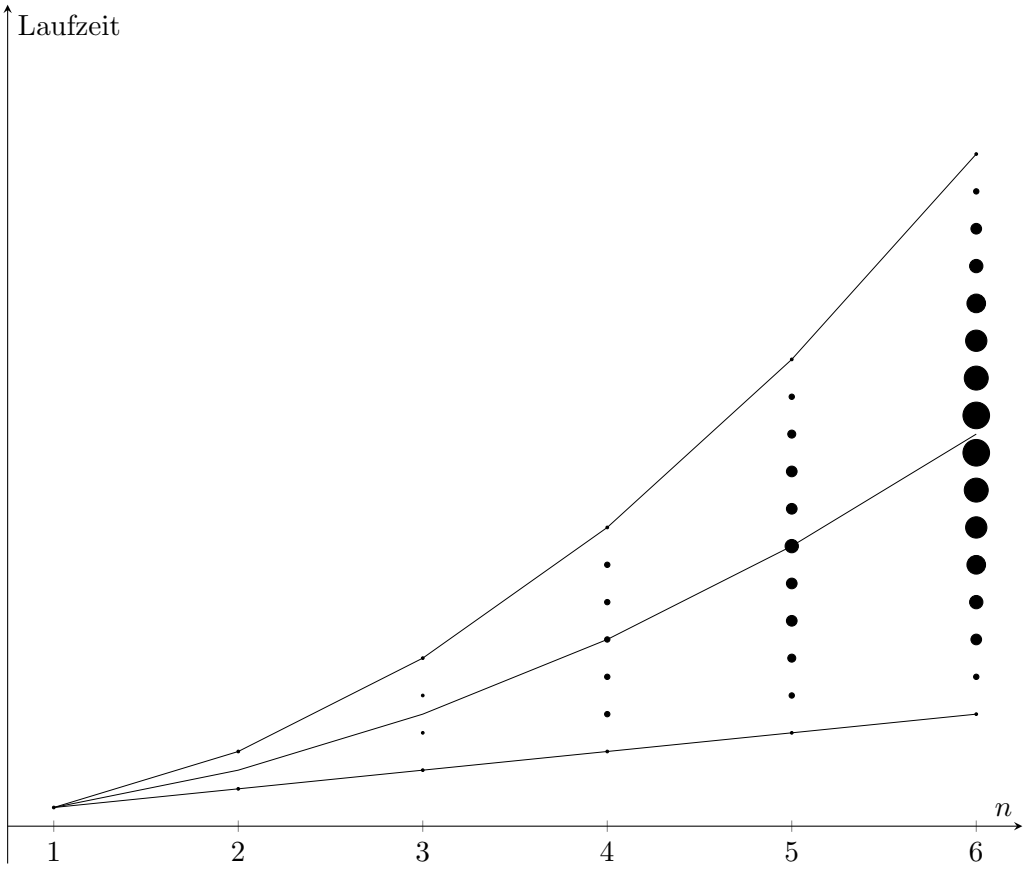


Abbildung 1.1: Laufzeit von Einfügesortieren für Datenfelder der Länge $n \leq 6$

Um zu erklären wie man diese Nachteile durch geeignete Abstraktionen vermeiden kann betrachten wir zunächst ein Beispiel. In Abbildung 1.1 ist die Laufzeit von Einfügesortieren für Eingabedatenfelder der Länge $n = 1, \ldots, 6$ dargestellt. Da es bei Sortieralgorithmen nur auf die Reihenfolge der Elemente in der Ordnung ankommt, wurden hier für festes n als Eingabe nur die $n!$ Permutationen von $\{1, \ldots, n\}$ betrachtet. Damit schränken wir unsere Betrachtung auch auf Fälle ein wo keine Elemente mehrfach vorkommen. Die Größe eines Punktes ist proportional zur Anzahl der Fälle mit dieser Laufzeit. Wir sehen dass es für festes n (abhängig von der Permutation) verschiedene Laufzeiten gibt und dass die Laufzeiten mit steigendem n wachsen. Zusätzlich sind in Abbildung 1.1 drei Kurven eingezeichnet: die Laufzeit im besten Fall (engl. *best case*), die Laufzeit im schlechtesten Fall (engl. *worst case*), sowie die Laufzeit im Durchschnittsfall (engl. *average case*).

Ein Verhalten wie das in Abbildung 1.1 dargestellte ist typisch für die Praxis: üblicherweise wächst die Laufzeit mit der Größe der Eingabe. Um zu einer abstrakteren Darstellung der Laufzeit zu kommen betrachten wir sie also nicht als eine Funktion der Menge erlaubter Eingaben, sondern als eine Funktion der Eingabegröße. Um der Tatsache Rechnung zu tragen, dass es für eine fixe Eingabegröße unterschiedliche Laufzeiten gibt betrachtet man die drei Laufzeiten im besten, im schlechtesten, und im durchschnittlichen Fall.

Zur Illustration führen wir nun eine Analyse der Komplexität von Einfügesortieren durch. Um die Analyse von Algorithmen zu vereinfachen und von Details der Implementierung und der Hardware zu abstrahieren (sh. oben) trifft man üblicherweise die folgenden Annahmen:

1. Die Instruktionen werden sequentiell ausgeführt (was auf Mehrprozessor-Systemen nicht immer der Fall sein muss).

2. Der Zeitbedarf jeder "elementaren Operation" ist konstant[1], also insbesondere ist er unabhängig von den Werten der Programmvariablen und wird nicht beeinflusst von Speicherhierachiekonzepten wie z.B. caching.

3. Wir arbeiten mit unbeschränkten Datentypen, also z.B. den natürlichen Zahlen und nicht, wie das in konkreten Implementierungen typischerweise der Fall ist mit, z.B., $\{0, \ldots, 2^{64} - 1\}$.

4. Wir nehmen an, dass wir mit den behandelten Zahlen exakt rechnen können. Der Einfluß von Rundungsfehlern bei der Verwendung von Gleitkommazahlen zur Darstellung reeller Zahlen spielt in der numerischen Mathematik eine wichtige Rolle, in dieser Vorlesung stellt er nur einen Nebenaspekt dar.

Wir können also voraussetzen dass für $i = 2, \ldots, 10$ eine Konstante c_i existiert, welche die Zeit angibt, die zur Ausführung von Zeile i benötigt wird. Sei A das Eingabedatenfeld und n die Länge von A. Dann belaufen sich die Kosten der Anwendung von Einfügesortieren auf das Datenfeld A auf

$$T(A) = (c_2 + c_{10})n + (c_3 + c_4 + c_9)(n-1) + (c_5 + c_8)\sum_{j=2}^{n} t_j + (c_6 + c_7)\sum_{j=2}^{n}(t_j - 1)$$

wobei t_j angibt, wie oft die Bedingung der inneren Schleife überprüft wird, wenn der äußere Schleifenzähler j ist. Die t_j hängen offensichtlich von A ab. Was sind nun mögliche Werte für die t_j ?

[1] Wir geben hier keine präzise Definition davon, was als elementare Operation zu verstehen ist. Jedenfalls dazu gehören: arithmetische Operationen wie Addition und Multiplikation, schreibender und lesender Zugriff auf Variablen sowie die Ausführung von Kontrollstrukturen wie Bedingungen, Schleifenköpfen, etc.

Im besten Fall ist das Eingabedatenfeld A bereits sortiert. Dann gilt nämlich $t_j = 1$ für $j = 2, \ldots, n$ und wir erhalten

$$T_{\mathrm{b}}(n) = (c_2 + c_{10})n + (c_3 + c_4 + c_5 + c_8 + c_9)(n-1),$$

d.h. also $T_{\mathrm{b}}(n) = kn + l$ für geeignete Konstanten k und l. Mit anderen Worten: die Laufzeit hängt linear von der Größe der Eingabe ab.

Im schlechtesten Fall muss jedes $A[j]$ mit *allen* Elementen in $A[1], \ldots, A[j-1]$ verglichen werden. Das geschieht dann wenn das Eingabedatenfeld absteigend sortiert ist. Dann ist also $t_j = j$ für $j = 2, \ldots, n$ und wir erhalten $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ sowie $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$ aus der gaußschen Summenformel und damit

$$T_{\mathrm{s}}(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} + \frac{c_8}{2})n^2 + (c_2 + c_3 + c_4 + c_9 + c_{10} + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + \frac{c_8}{2})n - c_3 - c_4 - c_5 - c_8 - c_9,$$

d.h. also $T_{\mathrm{s}}(n) = kn^2 + ln + m$ für geeignete Konstanten k, l und m. Im schlechtesten Fall hängt die Laufzeit also quadratisch von der Größe der Eingabe ab.

Bei der Analyse des Durchschnittsfalls eröffnet sich eine zusätzliche Schwierigkeit: es ist a priori nicht klar, welche Wahrscheinlichkeitsverteilung den Eingabedaten zugrunde liegt. Diese kann auch je nach Anwendung recht unterschiedlich ausfallen. Der Einfachheit halber arbeitet man oft mit einer (in geeignetem Sinn) uniformen Wahrscheinlichkeitsverteilung. In diesem Fall, der Sortierung von Datenfeldern, bedeutet das, dass wir für ein Eingabedatenfeld $A[1], \ldots, A[n]$ und seine Sortierung $A[\pi(1)], \ldots, A[\pi(n)]$ annehmen dass jede Permutation $\pi \in S_n$ gleich wahrscheinlich ist. D.h. jedes $\pi \in S_n$ tritt mit Wahrscheinlichkeit $\frac{1}{n!}$ auf. Man bezeichnet diese Annahme auch als *Permutationsmodell*. Dieses Modell für den allgemeinen Fall setzt auch die in Abbildung 1.1 betrachteten Eingabedaten für beliebige $n \geq 1$ fort. Wir verwenden die folgende Eigenschaft einer zufällig gewählten $\pi \in S_n$: Sei $1 \leq i \leq j \leq n$, dann ist

$$\mathrm{W}(\pi(j) \text{ ist das } i\text{-t-größte Element in } \{\pi(1), \ldots, \pi(j)\}) = \frac{1}{j}.$$

Diese Aussage werden wir im nächsten Kapitel beweisen. Die mittlere Anzahl von Aufrufen des inneren Schleifenkopfs ist dann also

$$\bar{t}_j = \frac{1}{j}1 + \frac{1}{j}2 + \cdots \frac{1}{j}j = \frac{1}{j}\frac{j(j+1)}{2} = \frac{j+1}{2}.$$

Damit erhalten wir

$$\begin{aligned}
\sum_{j=2}^{n} \bar{t}_j &= \frac{1}{2}\sum_{j=3}^{n+1} j = \frac{(n+1)(n+2)}{4} - \frac{3}{2} = \frac{n^2+3n}{4} - 1, \\
\sum_{j=2}^{n}(\bar{t}_j - 1) &= \frac{1}{2}\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{4} \text{ und} \\
T_{\mathrm{d}}(n) &= (\frac{c_5}{4} + \frac{c_6}{4} + \frac{c_7}{4} + \frac{c_8}{4})n^2 \\
&\quad + (c_2 + c_3 + c_4 + c_9 + c_{10} + \frac{3c_5}{4} - \frac{c_6}{4} - \frac{c_7}{4} + \frac{3c_8}{4})n \\
&\quad - c_3 - c_4 - c_5 - c_8 - c_9,
\end{aligned}$$

d.h. also $T_{\mathrm{d}}(n) = kn^2 + ln + m$ für geeignete Konstanten k, l und m.

Mit dieser Analyse von Einfügesortieren haben wir also in einem gewissen Sinn eine Darstellung der Laufzeit auf *allen* Eingaben erhalten, womit wir das oben angesprochene erste Problem als behandelt betrachten wollen.

Was das zweite Problem angeht ist die entscheidende Beobachtung, dass sich die Wahl einer Programmiersprache, eines Compilers, etc. nur auf die c_i auswirkt, nicht aber darauf wie die Laufzeit von n abhängt. Insbesondere haben wir rechnerisch gezeigt, dass diese Abhängigkeit, wie man aufgrund von Abbildung 1.1 bereits vermuten könnte, im besten Fall linear ist und im schlechtesten und durschnittlichen Fall quadratisch. Um diese Information auf formale Weise darzustellen verwendet man die Landau-Symbole, auch "Groß-O-Notation" genannt ($O(f)$ steht für "Ordnung von f").

Definition 1.4. Sei $f : \mathbb{N} \to \mathbb{R}$ eine Funktion. Wir definieren

$$O(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} \text{ so dass } \forall n \geq n_0 \colon |g(n)| \leq c \cdot |f(n)|\}.$$

Falls $g \in O(f)$ sagen wir dass f eine *asymptotisch obere Schranke* von g ist. Sei weiters

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} \text{ so dass } \forall n \geq n_0 \colon c \cdot |f(n)| \leq |g(n)|\}.$$

Falls $g \in \Omega(f)$ sagen wir dass f eine *asymptotisch untere Schranke* von g ist. Schließlich definieren wir noch

$$\Theta(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, d > 0, n_0 \in \mathbb{N} \text{ so dass } \forall n \geq n_0 \colon c \cdot |f(n)| \leq |g(n)| \leq d \cdot |f(n)|\}.$$

Falls $g \in \Theta(f)$ sagen wir dass f eine *asymptotisch scharfe Schranke* von g ist.

Beispiel 1.2. Seien $c, d, e > 0$, dann ist $cn^2+dn+e = \Theta(n^2)$. Einerseits ist nämlich $cn^2+dn+e \leq (c+d+e)n^2$ für $n \geq 1$. Andererseits ist auch $cn^2 \leq cn^2 + dn + e$ für $n \geq 0$.

In dieser Vorlesung wird diese Notation meist für positive Funktionen verwendet werden; dann sind die Betragsstriche überflüssig. In der Literatur wird diese Notation oft sinngemäß auch für $f : \mathbb{R} \to \mathbb{R}$ oder Funktionen anderen Typs verwendet. Deshalb wird in der Literatur häufig der Typ von f gar nicht erst angegeben. Zu dieser Definition lassen sich unmittelbar die folgenden Beobachtungen machen.

Lemma 1.2. *Für alle Funktionen f, g gilt:*

1. $\Theta(f) = O(f) \cap \Omega(f)$
2. $g \in O(f)$ *genau dann wenn* $f \in \Omega(g)$
3. $g \in O(f)$ *genau dann wenn* $\limsup_{n\to\infty} |\frac{g(n)}{f(n)}| < \infty$
4. $g \in \Omega(f)$ *genau dann wenn* $\liminf_{n\to\infty} |\frac{g(n)}{f(n)}| > 0$

wobei wir voraussetzen dass f und g nur endlich viele Nullstellen haben.

Beweis. 1. folgt direkt aus der Definition. Für 2. sei $c > 0$, $n_0 \in \mathbb{N}$ so dass $\forall n \geq n_0$: $|g(n)| \leq c \cdot |f(n)|$. Sei $d = \frac{1}{c}$, dann gilt $\forall n \geq n_0$: $d \cdot |g(n)| \leq |f(n)|$, d.h. also $f \in \Omega(g)$. Für die Gegenrichtung setzen wir $c = \frac{1}{d}$.

Für 3. sei wieder $c > 0, n_0 \in \mathbb{N}$ so dass $\forall n \geq n_0$: $|g(n)| \leq c \cdot |f(n)|$, d.h. $|\frac{g(n)}{f(n)}| \leq c$. Also ist $\{|\frac{g(n)}{f(n)}| \mid n \geq n_0\}$ im kompakten Intervall $[0, c]$ enthalten, besitzt also einen größten Häufungspunkt in $[0, c]$. Für die Gegenrichtung sei $n_0 - 1$ die größte Nullstelle von f. Dann ist $\left(|\frac{g(n)}{f(n)}|\right)_{n\geq n_0}$ beschränkt: falls $\left(|\frac{g(n)}{f(n)}|\right)_{n\geq n_0}$ nämlich unbeschränkt wäre, dann wäre $\limsup_{n\to\infty} |\frac{g(n)}{f(n)}| = \infty$. Also $\exists c > 0$ so dass $|\frac{g(n)}{f(n)}| \leq c$, d.h. $|g(n)| \leq c \cdot |f(n)|$ für $n \geq n_0$. 4. folgt aus 2. und 3. □

Oft schreiben wir $g(n) = O(f(n))$ statt $g \in O(f)$ um das Rechnen mit Termen wie z.B. $n^2+O(n)$ zu ermöglichen.

Satz 1.3. *Sei $f(n) = \sum_{i=0}^{k} a_i n^i$, $a_k \neq 0$, dann $f(n) = \Theta(n^k)$.*

Beweis. Man beachte dass

$$\left|\frac{\sum_{i=0}^{k} a_i n^i}{n^k}\right| = \left|a_k + \frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \cdots + \frac{a_0}{n^k}\right| \longrightarrow |a_k| \text{ für } n \to \infty.$$

Daraus folgt mit Lemma 1.2 das Resultat. □

Wir sagen dass eine Funktion f von polynomialem Wachstum ist falls ein $k \geq 1$ existiert so dass $f(n) = O(n^k)$.

Wir können nun die Laufzeit von Einfügesortieren im besten Fall angeben als $\Theta(n)$, jene im schlechtesten Fall als $\Theta(n^2)$ und jene im (uniform verteilten) Durchschnittsfall ebenfalls als $\Theta(n^2)$. Oft werden wir uns bei Angabe der Laufzeit im schlechtesten Fall auf die obere Schranke, d.h. $O(\cdot)$ beschränken und umgekehrt bei der Laufzeit im besten Fall auf die untere Schranke $\Omega(\cdot)$. Wenn wir also sagen dass der Algorithmus $\mathcal{A}$ Laufzeit $O(f)$ hat ist damit gemeint, dass er im schlechtesten Fall Laufzeit $O(f)$ hat und analog für Ω und den besten Fall.

Eng in Zusammenhang mit dieser asymptotischen Notation steht auch die Folgende:

Definition 1.5. Sei $f : \mathbb{N} \to \mathbb{R}$ eine Funktion. Wir definieren

$$o(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \forall c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 \colon |g(n)| \leq c \cdot |f(n)|\}.$$

Falls $g \in o(f)$ sagen wir dass g *asymptotisch kleiner als* f ist.

$$\omega(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \forall c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 \colon c \cdot |f(n)| \leq |g(n)|\}.$$

Falls $g \in \omega(f)$ sagen wir dass g *asymptotisch größer als* f ist.

Lemma 1.3. *Für alle Funktionen f, g gilt:*

1. $o(f) \subseteq O(f)$
2. $\omega(f) \subseteq \Omega(f)$
3. $f \notin o(f)$
4. $f \notin \omega(f)$
5. $g \in o(f)$ *genau dann wenn* $f \in \omega(g)$
6. $o(f) \cap \omega(f) = \emptyset$
7. $g \in o(f)$ *genau dann wenn* $\lim_{n\to\infty} \left|\frac{g(n)}{f(n)}\right| = 0$
8. $g \in \omega(f)$ *genau dann wenn* $\lim_{n\to\infty} \left|\frac{g(n)}{f(n)}\right| = \infty$

wobei wir wieder annehmen, dass f und g nur endlich viele Nullstellen haben.

Beweis. 1. und 2. folgen unmittelbar aus der Definition. Für 3. setzen wir $c = \frac{1}{2}$ und für 4. setzen wir $c = 2$. 5. kann analog zu Lemma 1.2 gelöst werden indem c auf $\frac{1}{d}$ und d auf $\frac{1}{c}$ gesetzt wird. 6. folgt durch Wahl geeigneter Konstanten ebenfalls direkt aus der Definition. Für 7. beachte man, dass die Definition von $g \in o(f)$ geschrieben werden kann als $\forall \varepsilon > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 \colon \left|\frac{g(n)}{f(n)}\right| \leq \varepsilon$. Für 8. schreibt man die Definition von $g \in \omega(f)$ als $\forall \varepsilon > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 \colon \left|\frac{f(n)}{g(n)}\right| \leq \varepsilon$, d.h. $\left|\frac{g(n)}{f(n)}\right| \geq \frac{1}{\varepsilon}$. □

Beispiel 1.3. Aus der Analysis wissen wir dass $\lim_{n\to\infty} \frac{n^k}{c^n} = 0$ für alle $k \in \mathbb{N}$ und $c > 1$, d.h. also $n^k \in o(c^n)$.

Definition 1.6. Für $f, g : \mathbb{N} \to \mathbb{R}$ schreiben wir $f \sim g$ genau dann wenn $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$. Falls $f \sim g$ sagen wir dass f und g *asymptotisch äquivalent sind.*

Kapitel 2

Elementare Kombinatorik

In diesem Kapitel werden wir einige elementare Begriffe und Resultate der Kombinatorik und der Graphentheorie kennen lernen bzw. wiederholen, da diese für das Studium von Algorithmen eine wichtige Grundlage bilden.

2.1 Abzählprobleme

Abzählprobleme sind klassische kombinatorische Fragestellungen. Bei einem Abzählproblem fragt man sich wie viele Elemente eine bestimmte endliche Menge hat. Abzählprobleme spielen bei der Analyse von Algorithmen oft eine wichtige Rolle. Dabei finden, für endliche Mengen A und B, oft die folgenden Überlegungen Anwendung:

1. Summenregel: Falls $A \cap B = \emptyset$, dann $|A \cup B| = |A| + |B|$
2. Produktregel: $|A \times B| = |A| \cdot |B|$
3. Gleichheitsregel: Falls eine Bijektion $f : A \to B$ existiert, dann ist $|A| = |B|$.

Für eine endliche Menge A wird eine bijektive Abbildung $\pi : A \to A$ auch als *Permutation* bezeichnet. Zur Erleichterung der Notation, geht man oft davon aus, dass $A = \{1, \ldots, n\}$. Eine solche Permutation kann in einer *zweizeiligen Darstellung* als

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

geschrieben werden. Eine alternative Darstellung ist die *Zyklendarstellung* als

$$\pi = (a_1\ \pi(a_1)\ \cdots\ \pi^{l_1-1}(a_1))(a_2\ \pi(a_2)\ \cdots\ \pi^{l_2-1}(a_2)) \cdots (a_k\ \pi(a_k)\ \cdots\ \pi^{l_k-1}(a_k))$$

wobei l_i definiert ist als das kleinste l so dass $\pi^l(a_i) = a_i$ und a_i so aus $\{1, \ldots, n\}$ gewählt wird, dass es in keinem der vorherigen Zyklen auftritt. Damit kommt also jedes $a \in \{1, \ldots, n\}$ genau ein Mal in der Zyklendarstellung vor.

Beispiel 2.1. Die Permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 2 & 5 & 1 & 3 & 4 \end{pmatrix}$$

wird in Zyklendarstellung als (164)(2)(35) geschrieben.

Die Anzahl von Permutation von $\{1,\dots,n\}$ ist $n! = n\cdot(n-1)\cdot(n-2)\cdots 1$. Die Funktion $n \mapsto n!$ kann auch rekursiv definiert werden durch $0! = 1$ und $(n+1)! = (n+1)n!$. Die Menge aller Permutationen von $\{1,\dots,n\}$ wird auch als S_n bezeichnet. Man kann sich leicht davon überzeugen, dass S_n mit der Komposition von Funktionen eine Gruppe bildet.

Eine *Variation ohne Wiederholung* besteht aus der k-fachen Auswahl eines von n Objekten wobei das Objekt nach seiner Auswahl nicht mehr für spätere Auswahlen zur Verfügung steht. Die Anzahl der Variationen ohne Wiederholung ist $n\cdot(n-1)\cdot\cdots\cdot(n-k+1) = \frac{n!}{(n-k)!}$

Beispiel 2.2. Für die ersten k Stellen einer Permutation $\pi \in S_n$ werden k aus n Objekten ohne Wiederholung ausgewählt, es gibt also $\frac{n!}{(n-k)!}$ Möglichkeiten. Für $k = n$ ergibt sich dann wie gehabt $\frac{n!}{(n-n)!} = n!$.

Eine *Variation mit Wiederholung* besteht aus der k-fachen Auswahl eines von n Objekten wobei das Objekt nach seiner Auswahl für spätere Auswahlen zur Verfügung steht. Die Anzahl der Variationen mit Wiederholung ist $n\cdot n\cdots n$, d.h. n^k.

Beispiel 2.3. Wenn ein Münzwurf entweder Kopf oder Zahl ergibt und eine Münze k mal hintereinander geworfen wird, gibt es insgesamt 2^k verschiedene Versuchsausgänge. Die Menge der Versuchsausgänge, d.h., der k-fachen Wiederholung von "Kopf" oder "Zahl" steht in Bijektion zu der Menge der Zeichenketten der Länge k die nur aus 0 und 1 bestehen, notiert als $\{0,1\}^k$. Diese wiederum steht in Bijektion zu den Teilmengen einer k-elementigen Menge. Somit gibt es auch davon jeweils genau 2^k.

Eine *Kombination ohne Wiederholung* besteht aus der k-fachen Auswahl eines von n Objekten wobei das Objekt nach seiner Auswahl nicht mehr für spätere Auswahlen zur Verfügung steht und die Reihenfolge der Wahl der Objekte irrelevant ist. Wir wählen also eine k-elementige Teilmenge einer n-elementigen Menge. Die Anzahl der Kombinationen ohne Wiederholung ist $\frac{n!}{(n-k)!k!} = \binom{n}{k}$, sie ergibt sich durch die Anzahl $\frac{n!}{(n-k)!}$ der Variationen ohne Wiederholung und der Überlegung, dass jeder Kombination $k!$ Variationen entsprechen.

Beispiel 2.4. Bei der Multiplikation von $(x+y)\cdots(x+y) = (x+y)^n$ müssen zur Bestimmung des Koeffizienten von $x^k y^{n-k}$ alle Möglichkeiten in Betracht gezogen werden in n Faktoren k mal x zu wählen und die anderen $n-k$ Mal y. Es gibt $\binom{n}{k}$ solche Möglichkeiten, also ergibt sich der binomische Lehrsatz

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}.$$

Das erklärt auch warum $\binom{n}{k}$ als *Binomialkoeffizient* bezeichnet wird.

Beispiel 2.5. Bei der Lottoziehung "6 aus 45" werden aus 45 Kugeln 6 Stück gezogen wobei die Reihenfolge für die Ermittlung des Gewinners egal ist. Es gibt also $\binom{45}{6} = 8145060$ Möglichkeiten für den Ausgang der Ziehung.

Eine *Kombination mit Wiederholung* besteht aus der k-fachen Auswahl eines von n Objekten wobei das Objekt nach seiner Auswahl für spätere Auswahlen zur Verfügung steht und die Reihenfolge der Wahl der Objekte irrelevant ist. Wir wählen also eine k-elementige Multimenge[1] mit Träger $\{1,\dots,n\}$. Eine solche Multimenge kann geschrieben werden als Tupel $(a_1,\dots,a_k)$ wobei $1 \le a_1 \le \cdots \le a_k \le n$. Nun gibt es eine Bijektion von der Menge der k-elementigen Multimengen mit Träger $\{1,\dots,n\}$ auf die k-elementigen Teilmengen von $\{1,\dots,n+k-1\}$, die durch

$$(a_1,\dots,a_k) \mapsto (a_1, a_2+1,\dots,a_k+k-1)$$

[1]Sei X eine Menge. Eine Multimenge M mit Träger X ist gegeben durch ihre charakteristische Funktion $\chi_M : X \to \mathbb{N}$. Eine Multimenge kann also, anders als eine Menge, ein Element mehrfach enthalten. Man definiert $|M| = \sum_{x\in X} \chi_M(x)$.

gegeben ist. Dann ist nämlich $1 \leq a_1 < a_2 + 1 < \cdots < a_k + k - 1 \leq n + k - 1$. Die Anzahl k-elementiger Teilmengen von $\{1, \ldots, n + k - 1\}$ kennen wir bereits: $\binom{n+k-1}{k}$.

Beispiel 2.6. Bei einem Brettspiel würfelt ein Spieler mit zwei Würfeln gleichzeitig. Bei den möglichen Würfen handelt es sich um eine Kombination mit Wiederholung mit $n = 6$ und $k = 2$. Es gibt also $\binom{7}{2} = 21$ verschiedene Würfe.

Lemma 2.1. *Sei $1 \leq i \leq j \leq n$, sei $\pi \in S_n$ uniform gewählt, dann ist*

$$\mathrm{W}(\pi(j) \textit{ ist das } i\textit{-t-größte Element in } \{\pi(1), \ldots, \pi(j)\}) = \frac{1}{j}.$$

Beweis. 1. Die Anzahl der Tupel $(\pi(1), \ldots, \pi(j))$ ist $\frac{n!}{(n-j)!}$. 2. Die Anzahl der Tupel $(\pi(1), \ldots, \pi(j))$ in denen $\pi(j)$ am i-t-größten ist ergibt sich a) durch Auswahl der Menge $\{\pi(1), \ldots, \pi(j)\} \subseteq \{1, \ldots, n\}$, dafür gibt es $\binom{n}{j} = \frac{n!}{(n-j)!j!}$ Möglichkeiten, b) durch Fixierung von $\pi(j)$ als das i-t-größte Element und c) durch Auswahl einer Reihenfolge für $\{\pi(1), \ldots, \pi(j-1)\}$, dafür gibt es $(j-1)!$ Möglichkeiten, d.h. also insgesamt $\frac{n!(j-1)!}{(n-j)!j!} = \frac{n!}{(n-j)!j}$. Wir erhalten also

$$\frac{\text{günstige}}{\text{mögliche}} = \frac{n! \cdot (n-j)!}{(n-j)! \cdot j \cdot n!} = \frac{1}{j}.$$

□

Für zwei endliche Mengen A und B mit $A \cap B = \emptyset$ gilt wie erwähnt $|A \cup B| = |A| + |B|$. Falls $A \cap B \neq \emptyset$ werden durch $|A|+|B|$ die Element im Durchschnitt doppelt gezählt. Durch Korrektur dieser Doppelzählung erhalten wir $|A \cup B| = |A|+|B|-|A \cap B|$. Im Fall von drei endlichen Mengen A, B und C werden durch $|A|+|B|+|C|$ alle Elemente die in zwei Mengen liegen zu oft gezählt. Eine erste Korrektur ergibt $|A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C|$. Nun werden aber die Elemente im Schnitt von drei Mengen gar nicht gezählt. Durch einen weiteren Korrekturschritt erhalten wir also $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$. Im Allgemeinen gilt:

Satz 2.1 (Prinzip von Inklusion und Exklusion)**.** *Seien $A_1, \ldots, A_n$ endliche Mengen, dann ist*

$$|\bigcup_{i=1}^{n} A_i| = \sum_{\emptyset \neq I \subseteq \{1,\ldots,n\}} (-1)^{|I|+1} |\bigcap_{i \in I} A_i|$$

Beweis. Sei $x \in \bigcup_{i=1}^n A_i$, $S = \{i \in \{1 \ldots, n\} \mid x \in A_i\}$ und $s = |S|$. Auf der rechten Seite wird x jetzt genau dann in einem Durchschnitt gezählt wenn $I \subseteq S$ und zwar, je nach Kardinalität von I entweder positiv oder negativ. Das Element x wird also

$$\binom{s}{1} - \binom{s}{2} + \binom{s}{3} - \cdots + (-1)^{s+1}\binom{s}{s} = \sum_{i=1}^{s} \binom{s}{i}(-1)^{i+1}$$

mal gezählt. Nun ist aber nach dem binomischen Lehrsatz $\sum_{i=0}^{s} \binom{s}{i}(-1)^i = (1-1)^s = 0$, sowie, nach Multiplikation mit -1, auch $\sum_{i=0}^{s} \binom{s}{i}(-1)^{i+1} = 0$. Weiters ist $\sum_{i=0}^{s} \binom{s}{i}(-1)^{i+1} = -1 + \sum_{i=1}^{s} \binom{s}{i}(-1)^{i+1}$. Damit wird x also $\sum_{i=1}^{s} \binom{s}{i}(-1)^{i+1} = 1$ mal gezählt. □

Das *Schubfachprinzip* (engl. *pigeonhole principle*) besagt Folgendes: Seien A und B endliche Mengen mit $|A| > |B|$, dann existiert keine injektive Funktion $f : A \to B$. Es kann mit einem einfachen Induktionsargument bewiesen werden. Ein Beispiel für seine Anwendung liefert der folgende Satz.

Satz 2.2. *Für jedes ungerade $q \in \mathbb{N}$ existiert ein $i \geq 1$ so dass $q \mid 2^i - 1$.*

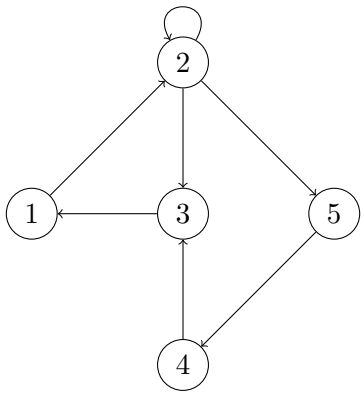
Beweis. Wir kürzen $2^i - 1$ als a_i ab. Wir betrachten $a_1, \ldots, a_q \pmod q$. Falls es darunter ein a_i gibt mit $a_i \equiv 0 \pmod q$ dann sind wir fertig. Falls nicht, dann gibt es nach dem Schubfachprinzip $i < j$ mit $a_i \equiv a_j \pmod q$, d.h. $q \mid a_j - a_i$. Nun ist aber $a_j - a_i = 2^j - 1 - 2^i + 1 = 2^j - 2^i = 2^i(2^{j-i} - 1)$ und da q ungerade muss $q \mid a_{j-i}$, Widerspruch. □

2.2 Graphen

Definition 2.1. Ein *gerichteter Graph* ist ein Paar $G = (V, E)$ wobei V eine beliebige Menge ist und $E \subseteq V \times V$.

Die Elemente von V heißen *Knoten*, die Elemente von E *Kanten*. Graphen werden oft aufgezeichnet indem die Kanten als Pfeile zwischen den Knoten dargestellt werden.

Beispiel 2.7. Der gerichtete Graph $G = (V, E)$ mit $V = \{1, 2, 3, 4, 5\}$ und $E = \{(1, 2), (2, 2), (2, 3), (2, 5), (3, 1), (4, 3), (5, 4)\}$ kann z.B. folgendermaßen gezeichnet werden.



Zwei Knoten $x, y \in V$ heißen *adjazent* falls $(x, y) \in E$ oder $(y, x) \in E$. Der *Ausgangsgrad* von $v \in V$ ist $\mathrm{d}^+(v) = |\{w \in V \mid (v, w) \in E\}|$. Der *Eingangsgrad* von $v \in V$ ist $\mathrm{d}^-(v) = |\{u \in V \mid (u, v) \in E\}|$. Ein Pfad ist eine endliche Folge $v_1, \ldots, v_n \in V$ mit $(v_i, v_{i+1}) \in E$ für $i = 1, \ldots, n-1$ und $i \neq j$ impliziert $v_i \neq v_j$.

Oft werden wir auch ungerichtete Graphen betrachten.

Definition 2.2. Ein *ungerichteter Graph* ist ein Paar (V, E) wobei V eine beliebige Menge ist und $E \subseteq \{\{x, y\} \mid x, y \in X, x \neq y\}$.

Damit ist $|E|$ in einem ungerichteten Graphen die Anzahl ungerichteter Kanten. Weiters definieren wir für $v \in V$ den Grad von v als $\mathrm{d}(v) = |\{w \in V \mid \{v, w\} \in E\}|$, es gibt also keinen getrennten Eingangs- und Ausgangsgrad mehr. Ein ungerichteter Graph kann als gerichteter Graph aufgefasst werden indem eine ungerichtete Kante $\{x, y\}$ durch die gerichteten Kanten (x, y) und (y, x) ersetzt wird. In diesem Sinn stellen die ungerichteten Graphen den allgemeineren Begriff dar. Von nun an werden wir mit "Graph" immer einen ungerichteten Graphen meinen. Wenn wir gerichtete Graphen betrachten wollen, werden wir das explizit erwähnen.

Ein *Pfad* in einem Graphen $G = (V, E)$ ist eine endliche Folge $v_1, \ldots, v_k \in V$ mit $\{v_i, v_{i+1}\} \in E$ für $i = 1, \ldots, k-1$ und $i \neq j$ impliziert $v_i \neq v_j$. G heißt *zusammenhängend* wenn es für alle $v, w \in V$ einen Pfad von v nach w gibt. G heißt *vollständig* falls $E = \{\{x, y\} \mid x, y \in V, x \neq y\}$.

Graphen treten in einer Unzahl von Anwendungskontexten und Berechnungsproblemen auf, wobei man es in der Informatik naturgemäß üblicherweise mit endlichen Graphen zu tun hat. Beispiele für Situationen die durch Graphen modelliert werden können sind: Verkehrsnetze wobei die Knoten z.B. Städten entsprechen und die Kanten Zugverbindungen, das WWW wobei die Knoten Webseiten und die Kanten Hyperlinks entsprechen oder auch Landkarten wobei jedem Land ein Knoten entspricht und zwei Konten durch eine ungerichtete Kante verbunden sind falls sie aneinander grenzen. Beispiele für Berechnungsprobleme aus der Graphentheorie sind:

Kürzester Pfad

Eingabe: endlicher zusammenhängender Graph $G = (V, E)$, Kostenfunktion $c : E \to \mathbb{R}_{>0}$, $s, t \in V$

Ausgabe: Pfad $v_1, \ldots, v_n$ mit $v_1 = s$, $v_n = t$ so dass $\sum_{i=1}^{n-1} c(\{v_i, v_{i+1}\})$ minimal ist

Problem des Handelsreisenden
(engl. *Travelling Salesman Problem (TSP)*)

Eingabe: endlicher vollständiger Graph $G = (V, E)$, Kostenfunktion $c : E \to \mathbb{R}_{>0}$

Ausgabe: Pfad $v_1, \ldots, v_n, v_{n+1} = v_1$ mit $\{v_1, \ldots, v_n\} = V$ so dass $\sum_{i=1}^{n} c(\{v_i, v_{i+1}\})$ minimal ist

Knotenfärbung

Eingabe: endlicher Graph $G = (V, E)$

Ausgabe: Abbildung $f : V \to \{1, \ldots, k\}$ so dass $(x, y) \in E \Rightarrow f(x) \neq f(y)$ und k minimal ist

Wir werden später effiziente Algorithmen zur Bestimmung eines kürzesten Pfades sehen, insb. wird deren Laufzeit polynomial in der Größe der Eingabe sein. Für das Problem des Handelsreisenden oder jenes der Knotenfärbung sind keine Algorithmen mit polynomialer Laufzeit bekannt. Aus der Existenz eines solchen Algorithmus würde **P** = **NP** folgen und damit die Lösung eines der bedeutendsten offenen Probleme der Mathematik. Das **P** vs. **NP** Problem wird später noch etwas detaillierter besprochen werden.

Zur algorithmischen Behandlung von Graphen müssen diese (etwa im Speicher eines Computers) repräsentiert werden. Es gibt verschiedene Datenstrukturen zur Repräsentation eines Graphen.

Definition 2.3. Die *Adjazenzmatrix* eines gerichteten Graphen $G = (\{1, \ldots, n\}, E)$ ist die Matrix $M = (m_{i,j})_{1 \leq i,j \leq n}$ wobei

$$m_{i,j} = \begin{cases} 1 & \text{falls } (i,j) \in E \\ 0 & \text{falls } (i,j) \notin E \end{cases}$$

Beispiel 2.8. Die Adjazenzmatrix des in Beispiel 2.7 angegebenen Graphen ist

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Eine Adjazenzmatrix wird im Speicher als ein Datenfeld von Datenfeldern abgelegt. Der für eine Adjazenzmatrix benötigte Speicherplatz ist $\Theta(|V|^2)$. Auf Basis einer Adjazenzmatrix kann von gegebenen Knoten v und w in Zeit $\Theta(1)$ festgestellt werden ob der Graph die Kante (v, w) enthält. Das Durchlaufen aller von einem gegebenen Knoten v aus wegführenden Kanten benötigt $\Theta(|V|)$ Zeit.

Definition 2.4. Die *Adjazenzliste* eines gerichteten Graphen $G = (\{1, \ldots, n\}, E)$ ist ein Datenfeld L der Länge n wobei $L[i]$ die Liste jener $j \in \{1, \ldots, n\}$ ist für die $(i, j) \in E$ gilt.

Beispiel 2.9. Die Adjazenzliste des in Beispiel 2.7 angegebenen Graphen ist:

1: 2

2: 2,3,5

3: 1

4: 3

5: 4

Der für eine Adjazenzliste benötigte Speicherplatz ist $\Theta(|E|)$ was im schlechtesten Fall auch $\Theta(|V|^2)$ ist. Für dünn besetzte Graphen, d.h. also solche die wesentlich weniger als $|V|^2$ viele Kanten haben, stellt die Verwendung einer Adjazenzliste aber eine Speicherersparnis dar. Auf Basis einer Adjazenzliste kann, gegeben Knoten v und w, in Zeit $O(\mathrm{d}^+(v))$ festgestellt werden, ob der Graph die Kante (v, w) enthält. Das Durchlaufen aller von einem gegebenen Knoten v wegführenden Kanten benötigt $\Theta(\mathrm{d}^+(v))$ Zeit.

Beispiel 2.10. Sei $G = (V, E)$ ein Graph wobei V die Menge der Kreuzungen in einer Großstadt ist und $(v, w) \in E$ falls eine Straße von v nach w führt ohne eine andere Kreuzung zu passieren. Der Graph G ist dünn besetzt, die Verwendung einer Adjazenzliste ist also empfehlenswert.

2.3 Bäume

Eine, insbesondere für Algorithmen, besonders wichtige Klasse von Graphen sind Bäume. Um Bäume näher zu untersuchen benötigen wir noch einige Begriffe: Sei $G = (V, E)$ ein Graph. Ein Graph $G' = (V', E')$ heißt *Teilgraph* von G falls $V' \subseteq V$ und $E' \subseteq E$. Sei $V' \subseteq V$, dann ist $G' = (V', E')$ der *von* V' *in* G *induzierte Graph* wobei $E' = \{\{v, w\} \in E \mid v, w \in V'\}$.

Definition 2.5. Sei $G = (V, E)$ ein Graph. $G' = (V', E')$ heißt *Zusammenhangskomponente* von G falls

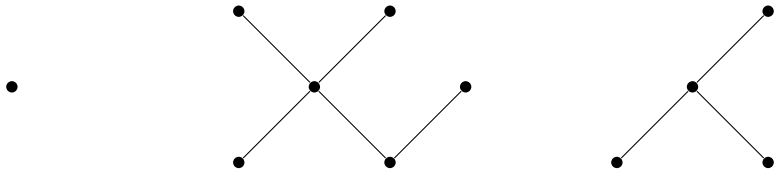
1. $V' \subseteq V$,
2. G' wird durch V' in G induziert,
3. G' ist zusammenhängend und

4. Für alle V'' mit $V' \subset V'' \subseteq V$ gilt: der durch V'' in G induzierte Graph ist nicht zusammenhängend.

Aus dieser Definition folgt unmittelbar dass jeder Graph als disjunkte Vereinigung seiner Zusammenhangskomponenten dargestellt werden kann. Ein *Zyklus* ist ein Pfad $v_1, \ldots, v_k$ mit $k \geq 3$ und $\{v_k, v_i\} \in E$.

Definition 2.6. Ein Graph $G = (V, E)$ heißt *Baum* falls er zusammenhängend und zyklenfrei ist. Ein Graph $G = (V, E)$ heißt *Wald* falls er zyklenfrei ist.

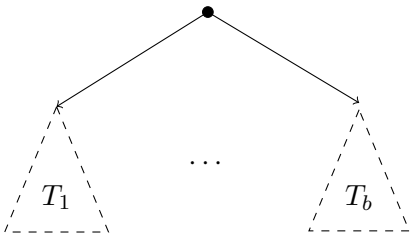
Beispiel 2.11. Ein Wald der aus 3 Bäumen besteht:



In dieser Definition eines Baums ist keine Wurzel ausgezeichnet, jeder Knoten kann als Wurzel designiert werden (je nachdem ändert sich dann die Form des Baums). Ein Wurzelbaum ist ein Tripel (V, E, r) wobei (V, E) ein Baum ist und $r \in V$. Ein Wurzelbaum kann als gerichteter Graph aufgefasst werden, indem, ausgehend von r, alle Kanten von r weg orientiert werden. Wurzelbäume werden uns, wie im folgenden Beispiel, oft auch in Form einer rekursiven Definition begegnen.

Beispiel 2.12. Sei $b \geq 2$. Ein *vollständiger Baum mit Arität* b und Tiefe 0 ist ein einzelner Knoten

Ein solcher Knoten heißt *Blatt*. Ein vollständiger Baum mit Arität b und Tiefe $d + 1$ ist ein gerichteter Graph der Form



wobei $T_1, \ldots, T_b$ vollständige Bäume mit Arität b und Tiefe d sind. Jeder in der Wurzel beginnende Pfad in einem vollständigen Baum kann identifiziert werden mit einem Tupel $(p_1, \ldots, p_d) \in \{1, \ldots, b\}^k$ wobei $k \leq d$. Also hat ein vollständiger Baum b^d *Blätter* sowie

$$b^0 + b^1 + \cdots + b^d = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}$$

Knoten. Ein vollständiger Binärbaum der Tiefe 3 hat also $2^3 = 8$ Blätter und $2^4 - 1 = 15$ Knoten.

Zum Zweck einer Charakterisierung der Bäume unter den Graphen machen wir zunächst noch einige vorbereitende Beobachtungen.

Lemma 2.2. *Sei $G = (V, E)$ ein endlicher, nicht-leerer, zyklenfreier Graph. Dann ist $|E| \leq |V| - 1$.*

Beweis. Wir gehen mit Induktion nach $|V|$ vor. Falls $|V| = 1$, dann muss $|E| = 0$ sein. Sei nun $|V| \geq 2$. Falls $E = \emptyset$ sind wir fertig. Falls $E \neq \emptyset$, dann existiert ein $v \in V$ mit $\mathrm{d}(v) \geq 1$. Seien $\{v, w_1\}, \ldots, \{v, w_k\}$ alle zu v inzidenten Kanten, dann gilt für alle $i, j \in \{1, \ldots, k\}$ mit $i \neq j$: jeder Pfad von w_i nach w_j enthält v, sonst würde nämlich G einen Zyklus enthalten. Für $i = 1, \ldots, k$ sei nun $G_i = (V_i, E_i)$ die Zusammenhangskomponente von w_i im Graphen der aus G entsteht wenn wir v sowie die Kanten $\{v, w_1\}, \ldots, \{v, w_k\}$ entfernen. Dann sind alle G_i zyklenfrei und mit der Induktionshypothese ist also

$$|E| = k + \sum_{i=1}^{k} |E_i| \leq^{\mathrm{IH}} k + \sum_{i=1}^{k} (|V_i| - 1) = \sum_{i=1}^{k} |V_i| = |V| - 1.$$

□