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.

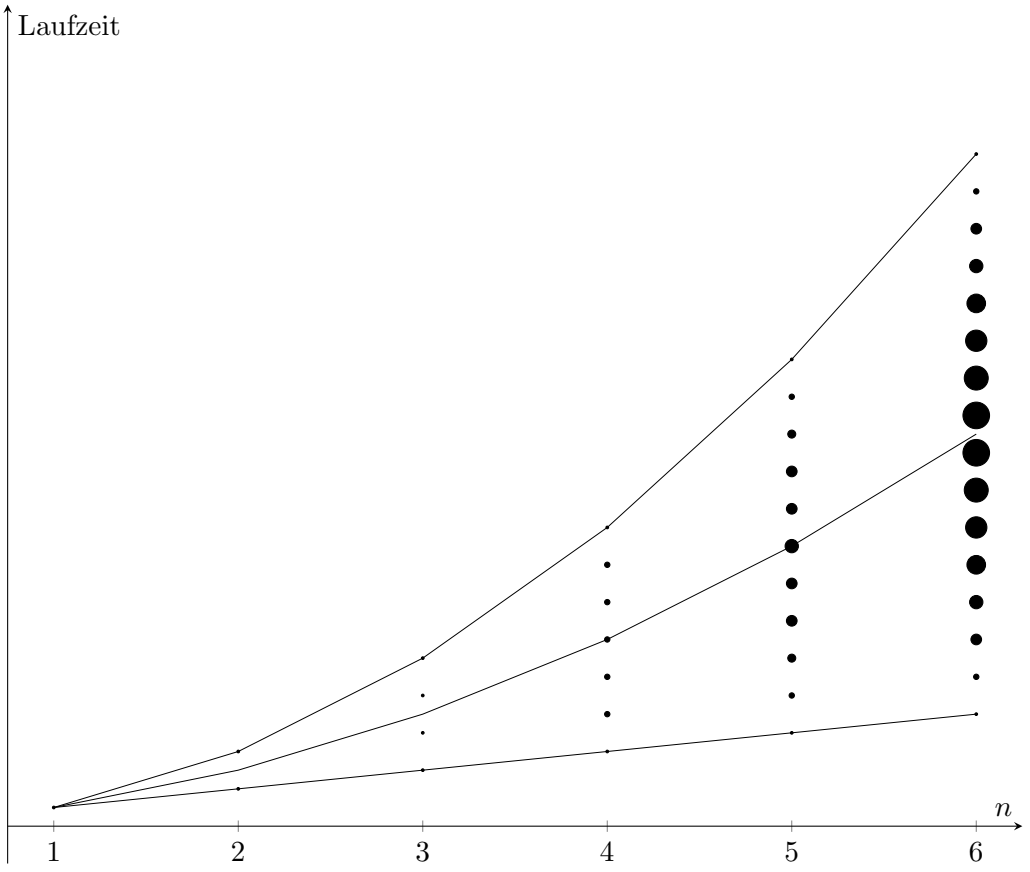


Abbildung 1.1: Laufzeit von Einfügesortieren für Datenfelder der Länge $n \leq 6$

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 $q \mid 2^i - 1$.*

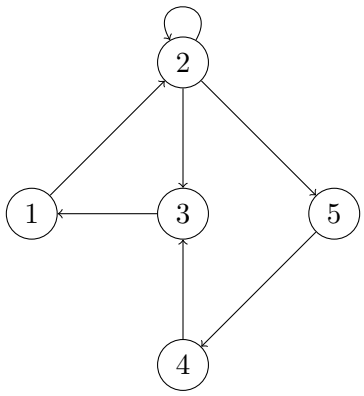
Beweis. Wir kürzen $2^i - 1$ als a_i ab. Wir betrachten $a_1, \ldots, a_q \pmod q$. Falls es darunter ein a_i gibt mit $a_i \equiv 0 \pmod q$ dann sind wir fertig. Falls nicht, dann gibt es nach dem Schubfachprinzip $i < j$ mit $a_i \equiv a_j \pmod q$, d.h. $q \mid a_j - a_i$. Nun ist aber $a_j - a_i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

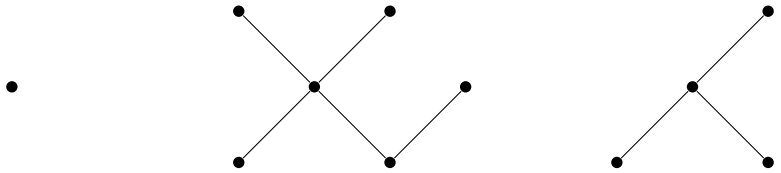
1. $V' \subseteq V$,
2. G' wird durch V' in G induziert,
3. G' ist zusammenhängend und

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.

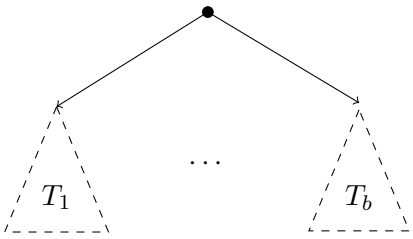
Beispiel 2.11. Ein Wald der aus 3 Bäumen besteht:



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

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

□

Lemma 2.3. *Sei $G = (V, E)$ ein endlicher, nicht-leerer, zusammenhängender Graph. Dann ist $|E| \geq |V| - 1$.*

Beweis. Wir gehen mit Induktion nach $|V|$ vor. Falls $|V| = 1$ ist das Resultat trivial. Sei nun $|V| \geq 2$. Da G zusammenhängend ist existiert ein Knoten v mit $\mathrm{d}(v) \geq 1$. Seien $\{v, w_1\}, \ldots, \{v, w_k\}$ alle zu v inzidenten Kanten. Sei $G' = (V', E')$ der Graph der aus G entsteht wenn wir v und sowie $\{v, w_1\}, \ldots, \{v, w_k\}$ löschen. Dann zerfällt G' in $l \leq k$ Zusammenhangskomponenten $(V_1, E_1), \ldots, (V_l, E_l)$ und mit der Induktionshypothese gilt

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

□

Definition 2.7. Ein Graph $G = (V, E)$ heißt *minimal zusammenhängend* falls er zusammenhängend ist und für alle $E' \subset E$ gilt: (V, E') ist nicht zusammenhängend.

Definition 2.8. Ein Graph $G = (V, E)$ heißt *maximal zyklenfrei* falls er zyklenfrei ist und für alle $E' \supset E$ gilt: (V, E') enthält einen Zyklus.

Satz 2.3. *Sei $G = (V, E)$ ein endlicher, nicht-leerer Graph. Dann sind äquivalent:*

1. *G ist ein Baum.*
2. *Je zwei Knoten von G sind durch genau einen Pfad verbunden.*
3. *G ist minimal zusammenhängend.*
4. *G ist zusammenhängend und $|E| = |V| - 1$.*
5. *G ist zyklenfrei und $|E| = |V| - 1$.*
6. *G ist maximal zyklenfrei.*

Beweis. 1. ⇒ 2: Da G zusammenhängend ist, gibt es von u nach v einen Pfad. Angenommen es gibt zwei unterschiedliche Pfade von u nach v. Dann gibt es einen ersten Knoten w_1 an dem sie sich unterscheiden und einen letzten Knoten w_2 an dem sie sich unterschieden. Die beiden Pfade von w_1 nach w_2 bilden dann einen Zyklus.

2. ⇒ 3.: Da je zwei Knoten durch einen Pfad verbunden sind ist G zusammenhängend. Sei nun $(u, v) \in E$, dann ist u, v der einzige Pfad von u nach v und damit ist $(V, E \setminus \{\{u, v\}\})$ nicht zusammenhängend.

3. ⇒ 4.: G ist zusammenhängend und damit ist $|E| \geq |V| - 1$ wegen Lemma 2.3. Es bleibt zu zeigen dass $|E| \leq |V| - 1$ ist. Angenommen $|E| > |V| - 1$ dann würde G wegen Lemma 2.2 einen Zyklus enthalten. Aus diesem könnte eine beliebige Kante gelöscht werden ohne den Zusammenhang zu zerstören, G wäre also nicht minimal zusammenhängend.

4. ⇒ 5.: Angenommen G enthält einen Zyklus $v_1, \ldots, v_k$. Für $l \in \{k, \ldots, |V|\}$ definieren wir einen Teilgraphen $G_l = (V_l, E_l)$ von G mit $|V_l| = |E_l| = l$ wie folgt: Falls $l = k$, dann ist besteht G_k aus dem Zyklus $v_1, \ldots, v_k$. Sei $l > k$. Da G zusammenhängend ist, existiert ein $\{v, w\} \in E$ so dass $v \in V_{l-1}$ und $w \notin V_{l-1}$. Wir definieren $V_l = V_{l-1} \cup \{w\}$ und $E_l = E_{l-1} \cup \{\{v, w\}\}$. Dann gilt $|V_l| = |E_l| = l$. Für $l = |V|$ erhalten wir also einen Teilgraphen $(V, E_{|V|})$ von (V, E). Damit ist $|E_V| = |V| \leq |E|$ was aber $|E| = |V| - 1$ widerspricht.

5.⇒ 6.: Angenommen es gäbe $E' \supset E$ so dass (V, E') zyklenfrei wäre, dann wäre wegen Lemma 2.2 ja $|E'| \leq |V| - 1$ was $|E'| > |E| = |V| - 1$ widerspricht.

6.⇒ 1.: Es reicht zu zeigen dass G zusammenhängend ist. Angenommen G wäre nicht zusammenhängend. Seien dann v und w Knoten aus verschiedenen Zusammenhangskomponenten und $G' = (V, E \cup \{\{v, w\}\})$. Dann ist G' immer noch zyklenfrei, denn jeder Zyklus in G' wäre entweder Zyklus in G (Widerspruch) oder er würde die Kante $\{v, w\}$ mindestens zwei Mal enthalten woraus sich ein Zyklus in jeder der beiden Zusammenhangskomponenten bilden ließe (Widerspruch). G ist also nicht maximal zyklenfrei, Widerspruch. □

Korollar 2.1. *Ein endlicher Wald (V, E) besteht aus $|V| - |E|$ Bäumen.*

Beweis. Angenommen (V, E) besteht aus den m Bäumen $(V_1, E_1), \ldots, (V_m, E_m)$. Dann ist

$$|E| = \sum_{i=1}^{m} |E_i| \overset{\text{Satz 2.3}}{=} \sum_{i=1}^{m} (|V_i| - 1) = \sum_{i=1}^{m} |V_i| - m = |V| - m$$

und damit $m = |V| - |E|$. □

Wir kommen jetzt zu einer Anwendung von Bäumen. Angenommen wir wollen eine Menge V von Orten (z.B. Computern in einem Gebäude oder Pins auf einer Leiterplatte) verbinden. Die Knoten V bilden gemeinsam mit möglichen Verbindungen $E \subseteq V \times V$ einen ungerichteten Graphen. Zusätzlich sind die Kosten des Legens einer Verbindung bekannt, d.h. eine Kostenfunktion $c : E \to \mathbb{R}_{\geq 0}$ ist gegeben. Der Graph der Verbindungen soll also $G' = (V, E')$ sein wobei $E' \subseteq E$ ist, G' zusammenhängend ist und $\sum_{e \in E} c(e)$ minimal sein soll.

Korollar 2.2. *Sei $G = (V, E)$ ein zusammenhängender Graph und $E' \subseteq E$ so dass $G' = (V, E')$ minimal zusammenhängend ist. Dann ist G' ein Baum.*

Beweis. Folgt unmittelbar aus Satz 2.3. □

Der gesuchte Verbindungsgraph ist also ein Baum. Das motiviert die folgenden Definitionen und das folgende Berechnungsproblem.

Definition 2.9. Sei $G = (V, E)$ ein zusammenhängender Graph. Ein *Spannbaum* von G ist ein Baum $B = (V, E')$ mit $E' \subseteq E$.

Der Baum B spannt also G auf.

Definition 2.10. Sei $G = (V, E)$ ein zusammenhängender Graph und sei $c : E \to \mathbb{R}_{\geq 0}$. Ein *minimaler Spannbaum* von G bezüglich c ist ein Spannbaum $B = (V, E')$ von G so dass $\sum_{e \in E'} c(e)$ minimal ist unter aller Spannbäumen von G.

Minimaler Spannbaum
Eingabe: Ein zusammenhängender Graph G und eine Kostenfunktion $c : E \to \mathbb{R}_{\geq 0}$
Ausgabe: Ein minimaler Spannbaum von G bezüglich c

Wir werden später effiziente Algorithmen zur Bestimmung eines minimalen Spannbaums kennen lernen.

Kapitel 3

Teile und Herrsche

In der Praxis auftretende Berechnungsprobleme haben oft die Eigenschaft dass ihre Instanzen zerlegt werden können und Lösungen der kleineren Instanzen zur Lösung der ursprünglichen Instanz hilfreich sind. Diese Vorgehensweise zur Lösung eines Berechnungsproblems ist so weit verbreitet, dass sich dafür ein eigener Begriff eingebürgert hat: die *teile-und-herrsche Strategie*. Algorithmen, die der teile-und-herrsche Strategie folgen bestehen üblicherweise aus drei Phasen:

1. *Aufteilung* der Eingabeinstanz in mehrere Instanzen kleinerer Größe
2. *Lösung* der kleineren Instanzen durch rekursiven Aufruf des Algorithmus
3. *Kombination* der Lösungen der kleineren Instanzen zu einer Lösung der ursprünglichen Instanz

Die Basis dieser Rekursion wird normalerweise dadurch gebildet, dass Instanzen deren Größe eine Konstante ist trivial gelöst werden können.

3.1 Sortieren durch Verschmelzen

Das Sortierproblem kann durch einen teile-und-herrsche Algorithmus gelöst werden. Die Grundidee dazu ist 1. das Eingabedatenfeld in zwei (zirka) gleich große Teile zu teilen, 2. jeden dieser beiden Teile unabhängig vom anderen durch einen rekursiven Aufruf zu sortieren und 3. die beiden sortierten Datenfelder zu einem einzigen sortierten Datenfeld zu verschmelzen. Deshalb wird dieser Algorithmus auch als "Sortieren durch Verschmelzen" (engl. *merge sort*) bezeichnet. Von den beiden Schritten 1 und 2 sollte klar sein wie sie umgesetzt werden können. Der Ansatz zur Realisierung des 3. Schritts besteht darin zwei Indizes zu verwenden, jeweils einen für jedes der Eingabedatenfelder, und sie so durch die Eingabedatenfelder laufen zu lassen, dass das aktuelle Minimum immer unter einem der beiden steht. Dieses aktuelle Minimum wird in das Ausgabedatenfeld übertragen. In Algorithmus 3 ist diese Idee als Pseudocode ausformuliert.

Basierend auf der Prozedur Verschmelzen kann nun Sortieren durch Verschmelzen wie in Algorithmus 4 als Pseudocode formuliert werden. Für $i \leq j$ ist die Notation $A := B[i, \ldots, j]$ eine Abkürzung für die Herstellung eines Datenfeldes A dessen Inhalt genau den Einträgen i bis j des Datenfeldes B entspricht. Eine solche Kopie kann durch eine einfache Schleife in Laufzeit $\Theta(j - i)$ hergestellt werden.

Wir wollen nun die Laufzeit von Sortieren durch Verschmelzen analysieren. Die Prozedur "Verschmelzen" benötigt Laufzeit $\Theta(A.Länge + B.Länge)$. Sei $T(n)$ die Laufzeit von Sortieren durch

Algorithmus 3 Verschmelzen

Prozedur VERSCHMELZEN(A, B)
  Sei C ein neues Datenfeld der Länge $A.Länge + B.Länge$
  $i := 1$
  $j := 1$
  Für $k := 1, \ldots, A.Länge + B.Länge$
    Falls $j > B.Länge$ **oder** ($i \leq A.Länge$ **und** $A[i] \leq B[j]$) **dann**
      $C[k] := A[i]$
      $i := i + 1$
    sonst
      $C[k] := B[j]$
      $j := j + 1$
    Ende Falls
  Ende Für
  Antworte C
Ende Prozedur

Algorithmus 4 Sortieren durch Verschmelzen (engl. *merge sort*)

Prozedur VSORTIEREN(A)
  Falls $A.Länge = 1$ **dann**
    Antworte A
  sonst
    $m := \left\lceil \frac{A.Länge}{2} \right\rceil$
    $L := A[1, \ldots, m]$
    $R := A[m + 1, \ldots, A.Länge]$
    $L' :=$ VSORTIEREN(L)
    $R' :=$ VSORTIEREN(R)
    Antworte VERSCHMELZEN(L', R')
  Ende Falls
Ende Prozedur

Verschmelzen wenn das Eingabedatenfeld die Länge n hat. Klar ist bereits dass $T(1) = \Theta(1)$. Für $n \geq 2$ beobachten wir: Die Aufteilung in Teilprobleme benötigt Laufzeit $\Theta(n)$, das Lösen der Teilprobleme benötigt $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$, das Verschmelzen der beiden Lösungen, die ja Größe m und $n-m$ haben, benötigt $\Theta(n)$. Insbesondere können wir hier beobachten, dass die Laufzeit von Sortieren durch Verschmelzen, anders als die von Einfügesortieren, nur von $n = A.\mathit{Länge}$ abhängt, nicht aber vom Inhalt von A. Somit ist die Laufzeit im besten Fall gleich der Laufzeit im schlechtesten Fall und gleich der Laufzeit im Durchschnittsfall.

Insgesamt erhalten wir für die Laufzeit

$$T(n) = \begin{cases} \Theta(1) & \text{für } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{für } n \geq 2 \end{cases}.$$

Bei dieser Darstellung der Laufzeit handelt es sich um eine *Rekursionsgleichung*. Die Analyse eines teile-und-herrsche Algorithmus führt typischerweise auf eine Rekursionsgleichung einer solchen Form. In Kapitel 4 werden wir zeigen, dass für die obige Rekursionsgleichung $T(n) = \Theta(n \log n)$ gilt[1]. Für den Augenblick wollen wir uns damit begnügen die folgende einfache Beobachtung zu machen, die den Kern der obigen Rekursionsgleichung enthält.

Satz 3.1. *Sei* $S : \{2^k \mid k \geq 1\} \to \mathbb{N}$ *definiert durch* $S(n) = \begin{cases} 2 & \textit{falls } n = 2 \\ 2S(\frac{n}{2}) + n & \textit{falls } n > 2 \end{cases}$, *dann ist* $S(n) = n \log n$.

Beweis. Zu zeigen ist also, für alle $k \geq 1$, dass $S(2^k) = k \cdot 2^k$. Für $k = 1$ ist das per definitionem erfüllt. Für $k \geq 2$ haben wir $S(2^k) = 2 \cdot S(2^{k-1}) + 2^k =^{\text{IH}} 2 \cdot (k-1) \cdot 2^{k-1} + 2^k = k \cdot 2^k$. □

3.2 Matrixmultiplikation

Wir werden annehmen, dass eine Matrix im Speicher abgelegt wird als ein Datenfeld, dessen Elemente Datenfelder einer festen Länge sind. Falls also A eine Matrix ist, so ist $A[i]$ die i-te Zeile von A und damit $A[i][j]$ das j-te Element der i-ten Zeile. Um die Notation etwas abzukürzen, schreiben wir stattdessen auch $A[i, j]$. Wir betrachten das folgende Problem:

Matrixmultiplikation
Eingabe: eine $n \times m$-Matrix A und eine $m \times l$-Matrix B
Ausgabe: $A \cdot B$

Seien $n, m, l \geq 1$ und $A = (a_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}}$ und $B = (b_{i,j})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq l}}$ Matrizen. Dann ist deren Produkt $A \cdot B = C = (c_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq l}}$ bekannterweise durch

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}$$

definiert. Diese Formel induziert sofort Algorithmus 5. Dessen Laufzeit kann aufgrund der drei verschachtelten Schleifen sofort als $\Theta(n \cdot l \cdot m)$ erkannt werden.

[1] In dieser Vorlesung werden wir mit log immer den Logarithmus zur Basis 2 notieren

Algorithmus 5 Matrixmultiplikation (direkt)

Vorbedingung: $n, m, l \geq 1$, A ist $n \times m$-Matrix, B ist $m \times l$-Matrix

```
Prozedur MATMULT(A, B)
    Sei C eine neue n × l-Matrix
    Für i := 1, ..., n
        Für j := 1, ..., l
            C[i, j] := 0
            Für k := 1, ..., m
                C[i, j] := C[i, j] + A[i, k] · B[k, j]
            Ende Für
        Ende Für
    Ende Für
    Antworte C
Ende Prozedur
```

Um die Analyse zu vereinfachen, werden wir von nun an annehmen, dass $n = m = l$ und dass n eine Zweierpotenz ist. Für $n = 2n'$ kann eine $n \times n$-Matrix in vier $n' \times n'$-Matrizen geteilt werden.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \quad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \quad C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Damit kann eine $n \times n$-Matrix mit Elementen aus einer Menge R identifiziert werden mit einer 2×2-Matrix deren Elemente $n' \times n'$-Matrizen mit Elementen aus R sind (Diese Beobachtung kann präzisiert werden, z.B. für einen Ring R, indem man zeigt, dass die Matrizenringe $R^{n \times n}$ und $(R^{n' \times n'})^{2 \times 2}$ isomorph sind). Für $C = A \cdot B$ erhalten wir damit die Darstellung

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

der $n \times n$-Multiplikation durch $n' \times n'$-Multiplikation. Diese Beobachtung induziert das in Algorithmus 6 angegebene einfache teile-und-herrsche Verfahren. Hier schreiben wir im Sinne der soeben diskutierten Teilung $A_{1,1}$ für $A[1, \ldots, n'; 1, \ldots, n']$, $A_{1,2}$ für $A[1, \ldots, n'; n' + 1, \ldots, n]$, usw. auch für B und C. Die Laufzeit dieses Algorithmus erfüllt also:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \\ 8T(\frac{n}{2}) + \Theta(n^2) & \text{falls } n \geq 2 \end{cases}$$

Diese Rekursionsgleichung hat die asymptotische Lösung $T(n) = \Theta(n^3)$ wie wir im Kapitel 4 sehen werden. Dieser teile-und-herrsche Algorithmus ist also asymptotisch nicht effizienter als das direkte Verfahren. Man könnte nun glauben, dass der teile-und-herrsche-Ansatz für die Matrixmultiplikation keine Verbesserung des direkten Verfahren erlaubt, oder sogar dass ein Algorithmus mit einer Laufzeit von weniger als $\Theta(n^3)$ gar nicht möglich ist, da dies ja der natürlichen Definition des Produkts entspricht. Beides wäre allerdings ein Irrtum wie der Algorithmus von Strassen zeigt.

Algorithmus 6 Matrixmultiplikation (rekursiv)

Vorbedingung: A, B sind $n \times n$ Matrizen, n ist Zweierpotenz

Prozedur MATMULT(A, B)
  Sei C eine neue $n \times n$-Matrix
  Falls $n = 1$ **dann**
    $C[1,1] := A[1,1] \cdot B[1,1]$
  sonst
    $C_{1,1} :=$ MATMULT$(A_{1,1}, B_{1,1}) +$ MATMULT$(A_{1,2}, B_{2,1})$
    $C_{1,2} :=$ MATMULT$(A_{1,1}, B_{1,2}) +$ MATMULT$(A_{1,2}, B_{2,2})$
    $C_{2,1} :=$ MATMULT$(A_{2,1}, B_{1,1}) +$ MATMULT$(A_{2,2}, B_{2,1})$
    $C_{2,2} :=$ MATMULT$(A_{2,1}, B_{1,2}) +$ MATMULT$(A_{2,2}, B_{2,2})$
  Ende Falls
  Antworte C
Ende Prozedur

Der Algorithmus von Strassen folgt ebenso wie Algorithmus 6 der teile-und-herrsche-Methode. Er unterscheidet sich von ihm dadurch, dass er eine geschicktere Darstellung der $C_{i,j}$ findet, die mit sieben Multiplikationen (von $n' \times n'$-Matrizen) auskommt. Seien nämlich

$$\begin{aligned}
P_1 &= A_{1,1} \cdot (B_{1,2} - B_{2,2}),\\
P_2 &= (A_{1,1} + A_{1,2}) \cdot B_{2,2},\\
P_3 &= (A_{2,1} + A_{2,2}) \cdot B_{1,1},\\
P_4 &= A_{2,2} \cdot (B_{2,1} - B_{1,1}),\\
P_5 &= (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}),\\
P_6 &= (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}), \text{ und}\\
P_7 &= (A_{1,1} - A_{2,1}) \cdot (B_{1,1} + B_{1,2}).
\end{aligned}$$

Dann gilt

$$\begin{aligned}
C_{1,1} &= P_5 + P_4 - P_2 + P_6,\\
C_{1,2} &= P_1 + P_2,\\
C_{2,1} &= P_3 + P_4, \text{ und}\\
C_{2,2} &= P_5 + P_1 - P_3 - P_7,
\end{aligned}$$

wovon man sich durch eine kurze Rechnung überzeugen kann. Der Strassen-Algorithmus ist dann wie folgt wobei wir (wie oben) die Abkürzungen $A_{i,j}$, $B_{i,j}$ und $C_{i,j}$ auch im Pseudocode verwenden.

Algorithmus 7 Strassen-Algorithmus

Vorbedingung: A, B sind $n \times n$ Matrizen, n ist Zweierpotenz

Prozedur STRASSEN(A, B)
  Sei C eine neue $n \times n$-Matrix
  Falls $n = 1$ **dann**
    $C[1,1] := A[1,1] \cdot B[1,1]$
  sonst
    $P_1 := \text{STRASSEN}(A_{1,1}, B_{1,2} - B_{2,2})$
    $P_2 := \text{STRASSEN}(A_{1,1} + A_{1,2}, B_{2,2})$
    $P_3 := \text{STRASSEN}(A_{2,1} + A_{2,2}, B_{1,1})$
    $P_4 := \text{STRASSEN}(A_{2,2}, B_{2,1} - B_{1,1})$
    $P_5 := \text{STRASSEN}(A_{1,1} + A_{2,2}, B_{1,1} + B_{2,2})$
    $P_6 := \text{STRASSEN}(A_{1,2} - A_{2,2}, B_{2,1} + B_{2,2})$
    $P_7 := \text{STRASSEN}(A_{1,1} - A_{2,1}, B_{1,1} + B_{1,2})$
    $C_{1,1} := P_5 + P_4 - P_2 + P_6$
    $C_{1,2} := P_1 + P_2$
    $C_{2,1} := P_3 + P_4$
    $C_{2,2} := P_5 + P_1 - P_3 - P_7$
  Ende Falls
  Antworte C
Ende Prozedur

Die Laufzeit vom Strassen-Algorithmus erfüllt also:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \\ 7T(\frac{n}{2}) + \Theta(n^2) & \text{falls } n > 1 \end{cases}$$

Im nächsten Kapitel werden wir zeigen dass die asymptotische Lösung dieser Rekursionsgleichung $T(n) = \Theta(n^{\log 7})$ ist. Da $\log 7 \approx 2{,}807$ wird dadurch eine Verbesserung des direkten sowie des rekursiven Verfahrens erreicht. Da die Konstanten des Strassen-Algorithmus aber größer sind als bei den beiden anderen Verfahren, wird in der Praxis meist eine Kombination verwendet: für hinreichend große Matrizen wird der Strassen-Algorithmus eingesetzt, für kleinere ein direkteres Verfahren.

Der Algorithmus von Strassen ist ein gutes Beispiel für einen teile-und-herrsche Algorithmus mit einer trickreichen Aufteilungs- und damit auch Kombinationsphase.

Bemerkung 3.1. Die Einschränkung auf quadratische Matrizen und n einer Zweierpotenz ist weder für den einfachen teile-und-herrsche Algorithmus, noch für den Strassen-Algorithmus notwendig. Diese Einschränkung dient lediglich der einfacheren Darstellung da durch sie einige Sonderfälle nicht behandelt werden müssen. In der Tat kann für beliebige n, m, l die Multiplikation einer $n \times m$- mit einer $m \times l$-Matrix durch Multiplikationen von Matrizen kleinerer Größe dargestellt werden, durch eine Zerlegung der Form $n = n_1 + n_2$, $m = m_1 + m_2$, $l = l_1 + l_2$. Setzt man zum Beispiel für $x \in \{n, m, l\}$ jeweils $x_1 = \lceil \frac{x}{2} \rceil$ und $x_2 = \lfloor \frac{x}{2} \rfloor$ erhält man einen allgemeineren teile-und-herrsche Algorithmus. Ähnliches gilt für den Strassen-Algorithmus. Für diesen ist allerdings notwendig dass alle $A_{i,j}$ die selbe Größe haben und dass alle $B_{i,j}$ die selbe Größe haben, d.h. also dass n, m und l gerade sind. Das kann erreicht werden durch Anfügen einer Nullzeile oder Nullspalte im Bedarfsfall.

Bemerkung 3.2. Der Algorithmus von Strassen wurde im Jahr 1969 publiziert und hat, wie beschrieben, eine Laufzeitkomplexität von $O(n^{2{,}807\cdots})$. Seitdem konnten weitere Verbesserungen der asymptotischen Laufzeit der Matrixmultiplikation erreicht werden. Ein signifikanter

Schritt vorwärts war der Algorithmus von Coppersmith-Winograd aus dem Jahr 1990 mit einer Laufzeitkomplexität von $O(n^{2,376})$. Über diesen hinaus konnten bis heute nur geringe Verbesserungen erreicht werden, so z.B. $O(n^{2,374})$ von Stothers 2010 und $O(n^{2,373})$ von Williams 2011. Diese weiteren Algorithmen haben allerdings so große Konstanten, dass sie in der Praxis keine Bedeutung haben.

Untere Schranken zur Matrixmultiplikation sind kaum bekannt. Klar ist, dass $\Omega(n^2)$ eine triviale untere Schranke ist, da die Eingabe der Größe $2n^2$ ja gelesen und die Ausgabe der Größe n^2 geschrieben werden muss. Eine untere Schranke von $\Omega(n^2 \log n)$ auf der Größe einer gewissen, eingeschränkten, Klasse von Schaltkreisen wurde von Raz 2003 bewiesen.

3.3 Dichtestes Punktepaar

Wir betrachten jetzt ein erstes fundamentales geometrisches Berechnungsproblem. Für Punkte $p_1, p_2 \in \mathbb{R}^2$ sei $\mathrm{d}(p_1, p_2)$ die übliche euklidische Distanz, d.h.

$$\mathrm{d}(\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Wir wollen das folgende Problem lösen:

Dichtestes Punktepaar
Eingabe: Eine endliche Menge $P \subseteq \mathbb{R}^2$
Ausgabe: $p, q \in P$ so dass $p \neq q$ und $\mathrm{d}(p, q)$ minimal

Klar ist, dass ein auf erschöpfender Suche basierender Algorithmus existiert, der einfach alle Paare von Punkten ausprobiert, siehe Algorithmus 8. Es gibt $\binom{n}{2}$ Paare die alle in jedem Fall

Algorithmus 8 Erschöpfende Suche nach dichtestem Punktepaar

```
Prozedur DPP-Suche(P)
    d_min := ∞
    Für i := 1, ..., P.Länge
        Für j := i + 1, ..., P.Länge
            Falls d(P[i], P[j]) < d_min dann
                d_min := d(P[i], P[j])
                a := (P[i], P[j])
            Ende Falls
        Ende Für
    Ende Für
    Antworte a
Ende Prozedur
```

durchlaufen werden, damit hat dieser Algorithmus Laufzeit $\Theta(n^2)$.

Wir werden sehen, dass es möglich ist mit einem Algorithmus der dem teile-und-herrsche Prinzip folgt (selbst im schlechtesten Fall) eine Laufzeit von $O(n \log n)$ zu erreichen.

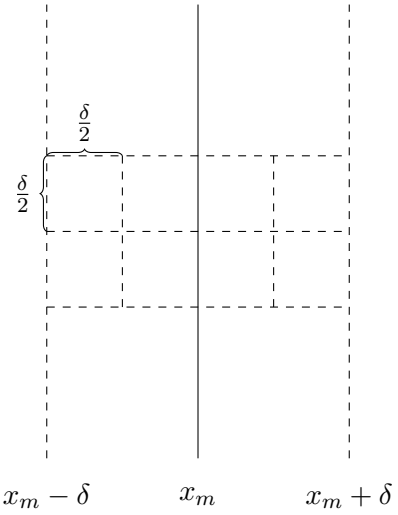


Abbildung 3.1: Kombination der Lösungen der Teilprobleme

Sei $P \subseteq \mathbb{R}^2$ eine endliche Menge von Punkten. Die Grundidee des Verfahrens ist wie folgt:

1. *Aufteilung*: Wir teilen das Problem entlang einer vertikalen Gerade, genauer: Sei $x_m \in \mathbb{R}$ und $P = Q \uplus R$ so dass $|Q| = \left\lceil \frac{|P|}{2} \right\rceil$, $|R| = \left\lfloor \frac{|P|}{2} \right\rfloor$, für alle $\begin{pmatrix} x \\ y \end{pmatrix} \in Q$: $x \leq x_m$ und für alle $\begin{pmatrix} x \\ y \end{pmatrix} \in R$: $x \geq x_m$. Man beachte, dass sowohl P als auch Q Punkte mit x-Koordinate x_m enthalten können.

2. *Lösung*: Mit Hilfe rekursiver Aufrufe bestimmen wir das dichteste Punktepaar $q_1, q_2 \in Q$ sowie das dichteste Punktepaar $r_1, r_2 \in R$. Sei $\delta = \min\{\mathrm{d}(q_1, q_2), \mathrm{d}(r_1, r_2)\}$.

3. *Kombination*: Das dichteste Punktepaar von P ist nun entweder q_1, q_2 oder r_1, r_2 oder ein Paar q, r wobei $q \in Q$ und $r \in R$. Natürlich können wir jetzt nicht alle (quadratisch vielen) Paare in $Q \times R$ durchsuchen wenn wir nur Laufzeit $O(n \log n)$ verwenden wollen. Mit einer kurzen Überlegung kann man aber zeigen, dass es ausreicht linear viele Paare zu überprüfen. Der Schlüssel dazu ist, dass wir δ bereits kennen. Falls nämlich ein Paar $q = \begin{pmatrix} q_x \\ q_y \end{pmatrix} \in Q$, $r = \begin{pmatrix} r_x \\ r_y \end{pmatrix} \in R$ das dichteste Punktepaar von P ist, muss nämlich $q_x, r_x \in [x_m - \delta, x_m + \delta]$ sein, d.h. dass q und r in einem Schlauch der Breite 2δ um x_m liegen. Weiters muss natürlich auch $|q_y - r_y| < \delta$ sein. Also liegen q und r in einem $2\delta \times \delta$ großen Rechteck das um die Gerade $x = x_m$ zentriert ist. Dieses Rechteck stellen wir uns nun als unterteilt in 8 Zellen der Größe $\frac{\delta}{2} \times \frac{\delta}{2}$ vor, siehe Abbildung 3.1. Jeder der Zellen (als Produkt geschlossener Intervalle) auf der linken Seite (von $x = x_m$) enthält höchstens einen Punkt von Q: würde eine Zelle nämlich zwei Punkte $q_1', q_2' \in Q$ enthalten, dann wäre $\mathrm{d}(q_1', q_2') \leq \sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \frac{\delta}{\sqrt{2}} < \delta$ was ein Widerspruch ist. Analog dazu enthält auch jede Zelle auf der rechten Seite höchstens einen Punkt von R. In diesem Rechteck gibt es also höchstens 8 Punkte von P. Um alle für das dichteste in Frage kommenden Punktepaare $q \in Q, r \in R$ zu überprüfen reicht es also, die Punkte im Schlauch rund um die Gerade $x = x_m$ nach y-Koordinate zu sortieren und für jeden Punkt den Abstand zu seinen 7 nächsten in der sortierten Liste zu überprüfen. Falls auf diese Weise ein Punktepaar $q \in Q, r \in R$ mit $\mathrm{d}(q, r) < \delta$ gefunden wird, so ist dieses das dichteste in P, falls nicht, dann ist das dichteste Punktepaar in P je nachdem entweder q_1, q_2 oder r_1, r_2. Damit haben wir uns also geometrisch davon überzeugt, dass diese Idee für einen teile-und-herrsche-Algorithmus sinnvoll ist und wir können uns an die konkrete Realisierung machen.

Ein Aspekt der durch die obige Diskussion noch nicht vollständig festgelegt ist, ist was passieren soll wenn mehrere Punkte in P auf der Gerade $x = x_m$ liegen. In solchen nicht-deterministischen Situationen ist es häufig nützlich, eine einfach zu berechnende Determinisierung festzulegen. Dazu definieren wir:

Definition 3.1. Die *lexikographische Ordnung* $<_{\text{lex}}$ auf $\mathbb{R}^2$ ist festgelegt durch:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} <_{\text{lex}} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \text{ genau dann wenn } 1.\ x_1 < x_2 \text{ oder}$$
$$2.\ x_1 = x_2 \text{ und } y_1 < y_2.$$

Wir schreiben $\leq_{\text{lex}}$ für die reflexive Hülle von $<_{\text{lex}}$ und beobachten, dass $<_{\text{lex}}$ eine totale Ordnung auf $\mathbb{R}^2$ ist. Wir können die Aufteilung des Problems für die Menge P nun also deterministisch spezifizieren als: Sei $p_m \in P$, $Q = \{p \in P \mid p \leq_{\text{lex}} p_m\}$, $R = \{p \in P \mid p >_{\text{lex}} p_m\}$ so dass $|Q| = \left\lceil \frac{|P|}{2} \right\rceil$ und $R = \left\lfloor \frac{|P|}{2} \right\rfloor$. Dann ist x_m die x-Koordinate von p_m. Falls also mehrere Punkte in P die x-Koordinate x_m haben gibt es auf der Gerade $x = x_m$ einen Punkt so dass alle darunter liegenden Punkte in Q sind und alle darüber liegenden in R.

Wir benötigen also eine Darstellung der Punkte sortiert nach $\leq_{\text{lex}}$ sowie eine in der die selben Punkte nach y-Koordinate sortiert sind. Das können wir erreichen, indem wir zwei Datenfelder verwenden die die selben Punkte enthalten: eines wird nach $\leq_{\text{lex}}$ sortiert und eines nach der y-Koordinate. Ein wichtiger Punkt ist nun dass wir es uns nicht leisten können, nach der Teilung erneut zu sortieren. Das würde in jedem Schritt Zeit $O(n \log n)$ verbrauchen und um eine Gesamtlaufzeit von $O(n \log n)$ zu erreichen müssen wir in jedem Schritt mit Zeit $O(n)$ auskommen. Dieses Hindernis kann überwunden werden, indem wir diese beiden Sortierungen einmal zu Beginn des Algorithmus (in Zeit $O(n \log n)$) berechnen und dann sicherstellen, dass die Sortierung durch den gesamten Algorithmus hindurch beibehalten wird.

Zur Realisierung dieses Ansatzes benötigen wir noch eine weitere Operation. Falls wir aus einer Menge X alle Elemente x auswählen wollen, die eine Eigenschaft $P(x)$ haben, dann schreiben wir diese Menge als $\{x \in X \mid P(x)\}$. Eine analoge Operation auf Datenfeldern kann wie folgt definiert werden: Sei A ein Datenfeld, dann ist $\langle x \in A \mid P(x) \rangle$ ein Datenfeld das alle Elemente von A enthält, die die Eigenschaft P erfüllen und zwar *in der Reihenfolge, in der sie in A vorkommen*. Algorithmus 9 führt diese Auswahl durch. Falls die Überprüfung der Eigenschaft

Algorithmus 9 Auswahl aus einem Datenfeld

```
Prozedur AUSWAHL_P(A)
    Sei B ein neues Datenfeld
    j := 1
    Für i = 1, ..., A.Länge
        Falls P(A[i]) dann
            B[j] := A[i]
            j := j + 1
        Ende Falls
    Ende Für
    Antworte B
Ende Prozedur
```

P konstante Zeit erfordert (was üblicherweise bei uns der Fall sein wird), dann läuft dieser Algorithmus in Zeit $\Theta(n)$ wobei $n = A.\text{Länge}$.

Der gesamte Algorithmus zur Bestimmung des dichtesten Punktepaares kann als Pseudocode wie in Algorithmus 10 geschrieben werden.

Algorithmus 10 Teile-und-herrsche Algorithmus für dichtestes Punktepaar

Prozedur DPP-TH(P)
  $P := P$ aufsteigend nach $\leq_{\text{lex}}$ sortiert
  $P_y := P$ aufsteigend nach y-Koordinate sortiert
  Antworte DPP-TH-Rek(P, P_y)
Ende Prozedur

Prozedur DPP-TH-Rek(P, P_y)
  Falls $P.Länge \leq 3$ **dann**
    Antworte DPP-Suche(P)
  Ende Falls

  $m := \left\lceil \frac{P.Länge}{2} \right\rceil$
  $Q := P[1, \ldots, m]$
  $R := P[m+1, \ldots, P_x.Länge]$
  $Q_y := \langle p \in P_y \mid p \leq_{\text{lex}} P[m] \rangle$
  $R_y := \langle p \in P_y \mid p >_{\text{lex}} P[m] \rangle$

  $(q_1, q_2) :=$ DPP-TH-Rek(Q, Q_y)
  $(r_1, r_2) :=$ DPP-TH-Rek(R, R_y)
  $\delta := \min\{\mathrm{d}(q_1, q_2), \mathrm{d}(r_1, r_2)\}$

  $S_y := \langle p \in P_y \mid P[m].x - \delta \leq p.x \leq P[m].x + \delta \rangle$
  $s_1 := (0, 0)$
  $s_2 := (\infty, \infty)$
  Für $i := 1, \ldots, S_y.Länge - 1$
    Für $j := i+1, \ldots, \min\{i+7, S_y.Länge\}$
      Falls $\mathrm{d}(S_y[i], S_y[j]) < \mathrm{d}(s_1, s_2)$ **dann**
        $s_1 := S_y[i]$
        $s_2 := S_y[j]$
      Ende Falls
    Ende Für
  Ende Für

  Falls $\mathrm{d}(s_1, s_2) < \delta$ **dann**
    Antworte (s_1, s_2)
  sonst falls $\mathrm{d}(r_1, r_2) < \mathrm{d}(q_1, q_2)$ **dann**
    Antworte (r_1, r_2)
  sonst
    Antworte (q_1, q_2)
  Ende Falls
Ende Prozedur

Für die Laufzeitanalyse von DPP-TH sei $n = |P|$. Die Laufzeit von DPP-TH ist $\Theta(n \log n)$ für das Sortieren plus der Laufzeit von DPP-TH-Rek. Für die Laufzeitkomplexität von DPP-TH-Rek erhalten wir

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 3 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{if } n > 3 \end{cases}$$

Diese Rekursionsgleichung ist beinahe identisch zu jener die wir aus Sortieren durch Verschmelzen erhalten haben. In der Tat gilt auch hier $T(n) = \Theta(n \log n)$ wie wir in Kapitel 4 zeigen werden. Somit ist auch die Laufzeit des Gesamtalgorithmus DPP-TH $\Theta(n \log n)$.

Wir sehen also, dass die Kombinationsphase in einem teile-und-herrsche-Algorithmus durchaus auch trickreicher sein kann als das beim Sortieren durch Verschmelzen der Fall ist.

Kapitel 4

Rekursionsgleichungen

Eine Rekursionsgleichung definiert eine Folge $(x_n)_{n\geq 0}$ durch eine Gleichung, die x_n basierend auf n und den x_i mit $i < n$ definiert sowie hinreichend vielen Anfangswerten. Typischerweise ist man daran interessiert, einen geschlossenen Ausdruck für x_n zu finden, d.h. einen, in dem keine x_i mehr vorkommen.

Beispiel 4.1. Die bekannte Rekursionsgleichung

$$F_n = F_{n-1} + F_{n-2} \text{ für } n \geq 2 \text{ mit } F_0 = 0 \text{ und } F_1 = 1$$

definiert die Folge der Fibonacci-Zahlen $0, 1, 1, 2, 3, 5, 8, 13, \ldots$.

4.1 Lineare Rekursionsgleichungen erster Ordnung

Definition 4.1. Die *Ordnung* einer Rekursionsgleichung ist das kleinste k so dass die Definition von x_n von $\{x_{n-1}, \ldots, x_{n-k}\}$ abhängt.

So hat zum Beispiel die Fibonacci-Gleichung die Ordnung 2.

Definition 4.2. Eine *lineare Rekursionsgleichung erster Ordnung* ist eine Rekursionsgleichung der Form

$$x_n = a_n x_{n-1} + b_n \text{ für } n \geq 1$$

mit festgelegten Anfangswert x_0.

Beispiel 4.2. Die Rekursionsgleichung

$$x_n = x_{n-1} + b_n \text{ für } n \geq 1 \text{ mit } x_0 = 0$$

hat als Lösung $x_n = \sum_{i=1}^{n} b_i$.

Die Rekursionsgleichung

$$x_n = a_n x_{n-1} \text{ für } n \geq 1 \text{ mit } x_0 = 1$$

hat als Lösung $x_n = \prod_{i=1}^{n} a_i$.

Diese beiden Beispiele können zu dem folgenden Resultat verallgemeinert werden:

Satz 4.1. *Die Rekursionsgleichung $x_n = a_n x_{n-1} + b_n$ für $n \geq 1$ mit festgelegten Anfangswert $x_0 = b_0$ hat die Lösung*

$$x_n = \sum_{i=0}^{n} b_i \prod_{j=i+1}^{n} a_j.$$

Beweis. Mit Induktion: Für $n = 0$ gilt $x_0 = b_0$. Für $n > 0$ haben wir

$$x_n = a_n x_{n-1} + b_n = a_n \left(\sum_{i=0}^{n-1} b_i \prod_{j=i+1}^{n-1} a_j \right) + b_n = \left(\sum_{i=0}^{n-1} b_i \prod_{j=i+1}^{n} a_j \right) + b_n = \sum_{i=0}^{n} b_i \prod_{j=i+1}^{n} a_j.$$

□

4.2 Lineare Rekursionsgleichungen k-ter Ordnung

Wir wollen nun lineare Rekursionsgleichungen beliebiger Ordnung betrachten, schränken uns dabei aber auf den Fall konstanter Koeffizienten ein.

Definition 4.3. Eine *homogene lineare Rekursionsgleichung mit konstanten Koeffizienten k-ter Ordnung* ist von der Form

$$x_n = c_{k-1} x_{n-1} + \cdots + c_0 x_{n-k} \text{ für } n \geq k \tag{4.1}$$

Falls die Konstanten $c_0, \ldots, c_{k-1}$ Elemente eines Körpers K sind, können wir die Rekursionsgleichung (4.1) über diesem Körper K auffassen. Wir sagen dass $(a_n)_{n\geq 0}$ eine Lösung von (4.1) in K ist falls alle $a_n \in K$ sind und $a_n = c_{k-1} a_{n-1} + \cdots + c_0 a_{n-k}$ für alle $n \geq k$. Wir definieren $K^\omega = \{(a_n)_{n\geq 0} \mid a_n \in K \text{ für alle } n \geq 0\}$ und stellen fest, dass K^ω ein Vektorraum unendlicher Dimension über K ist.

Lemma 4.1. *Sei K ein Körper, seien $c_0, \ldots, c_{k-1} \in K$, dann ist die Lösungsmenge von (4.1) in K ein Untervektorraum von K^ω mit Dimension k.*

Beweis. Klar ist, dass die Lösungsmenge eine Teilmenge von K^ω ist. Weiters ist die Lösungsmenge nicht leer, da z.B. $(0)_{n\geq 0}$ eine Lösung von (4.1) ist. Seien nun $(a_n)_{n\geq 0}, (b_n)_{n\geq 0}$ Lösungen von (4.1) und $\lambda, \mu \in K$, dann ist für $n \geq k$

$$\begin{aligned} a_n &= c_{k-1} a_{n-1} + \ldots + c_0 a_{n-k}, \\ b_n &= c_{k-1} b_{n-1} + \ldots + c_0 b_{n-k}, \end{aligned}$$

und damit auch

$$\lambda a_n + \mu b_n = c_{k-1}(\lambda a_{n-1} + \mu b_{n-1}) + \ldots + c_0(\lambda a_{n-1} + \mu b_{n-1}),$$

d.h. also auch $(\lambda a_n + \mu b_n)_{n\geq 0}$ ist eine Lösung von (4.1) und damit ist die Lösungsmenge ein Unterraum von K^ω. Seien $a_0, \ldots, a_{k-1}$ beliebig, dann ist $(a_n)_{n\geq 0}$ eindeutig bestimmt, und kann geschrieben werden als $(a_n)_{n\geq 0} = a_0 e_0 + \cdots + a_{k-1} e_{k-1}$ wobei $e_i \in K^\omega$ jene Lösung von (4.1) ist, die durch $e_{i,j} = \delta_{i,j}$ für $j = 0, \ldots, k-1$ bestimmt wird. Also ist $e_0, \ldots, e_{k-1}$ Basis des Lösungsraums und damit ist seine Dimension k. □

Für den Spezialfall $k = 1$ und $x_0 = 1$ von (4.1) haben wir in Beispiel 4.2 bereits festgestellt, dass die Lösung $x_n = c_0^n$ ist. Es liegt also nahe, Lösungen von der Form $(\alpha^n)_{n\geq 0}$ zu betrachten. Für eine solche Lösung gilt

$$\alpha^n - c_{k-1}\alpha^{n-1} - \cdots - c_0\alpha^{n-k} = 0.$$

für alle $n \geq k$ und insbesondere für $n = k$:

$$\alpha^k - c_{k-1}\alpha^{k-1} - \cdots - c_1\alpha - c_0 = 0.$$

Diese Beobachtung motiviert die folgende Definition:

Definition 4.4. Das *charakteristische Polynom* der Rekursionsgleichung (4.1) ist

$$\chi(z) = z^k - c_{k-1}z^{k-1} - \cdots - c_1 z - c_0$$

Das heißt also: falls $(\alpha^n)_{n\geq 0}$ Lösung von (4.1) ist, dann ist α Nullstelle von $\chi(z)$. Es gilt auch eine (stärkere) Implikation in die andere Richtung so dass wir das folgende Resultat erhalten:

Satz 4.2. *Sei $\chi(z)$ das charakteristische Polynom von (4.1) mit Nullstellen $\alpha_1, \ldots, \alpha_r$ und Vielfachheiten $m_1, \ldots, m_r$. Dann ist $\{(n^j\alpha_i^n)_{n\geq 0} \mid 1 \leq i \leq r, 0 \leq j < m_i\}$ eine Basis des Lösungsraums von (4.1).*

Beweis. Sei α eine Nullstelle von $\chi(z)$, dann ist

$$\alpha^k = c_{k-1}\alpha^{k-1} + \cdots + c_1\alpha + c_0$$

und nach Multiplikation mit α^{n-k}

$$\alpha^n = c_{k-1}\alpha^{n-1} + \cdots + c_1\alpha^{n-k+1} + c_0\alpha^{n-k}.$$

Somit ist $(\alpha^n)_{n\geq 0}$ eine Lösung.

Sei α Doppelnullstelle von $\chi(z)$, dann ist α auch Nullstelle von $\chi'(z)$ und damit vom Polynom $\psi(z) = (n-k)\chi(z) + z\chi'(z)$. Nun ist aber

$$\begin{aligned}\alpha\chi'(\alpha) &= k\alpha^k - (k-1)c_{k-1}\alpha^{k-1} - \cdots - 2c_2\alpha^2 - c_1\alpha \text{ und}\\ (n-k)\chi(\alpha) &= (n-k)\alpha^k - (n-k)c_{k-1}\alpha^{k-1} - \cdots - (n-k)c_1\alpha - (n-k)c_0\end{aligned}$$

und damit

$$\psi(\alpha) = n\alpha^k - (n-1)c_{k-1}\alpha^{k-1} - \cdots - (n-k+1)c_1\alpha - (n-k)c_0.$$

Nach Multiplikation mit α^{n-k} erhalten wir

$$n\alpha^n = (n-1)c_{k-1}\alpha^{n-1} - \cdots - (n-k+1)c_1\alpha^{n-k+1} - (n-k)c_0\alpha^{n-k}$$

also ist auch $(n\alpha^n)_{n\geq 0}$ eine Lösung.

Falls α eine dreifache Nullstelle von $\chi(z)$ ist, ist α eine Doppelnullstelle von $\psi(z)$ und wir wenden die Abbildung $\chi \mapsto \psi$ ein zweites Mal, diesmal auf ψ, an. Induktiv folgt damit: falls α eine m-fache Nullstelle von $\chi(z)$ ist, dann sind $(\alpha^n)_{n\geq 0}, (n\alpha^n)_{n\geq 0}, \ldots, (n^m\alpha^n)_{n\geq 0}$ Lösungen.

Diese k Lösungen sind linear unabhängig, also handelt es sich wegen Lemma 4.1 um eine Basis des Lösungsraums. □

Die allgemeine Lösung von (4.1) hat also die Form

$$x_n = \sum_{i=1}^{r} \sum_{j=0}^{m_i-1} c_{i,j} n^j \alpha_i^n$$

wobei die $c_{i,j}$ Konstanten sind, die von den Anfangswerten abhängen.

Beispiel 4.3. Mit Hilfe des obigen Satzes lässt sich leicht eine explizite Darstellung der Fibonacci-Zahlen berechnen. Das charakteristische Polynom von $F_n = F_{n-1} + F_{n-2}$ ist $\chi(z) = z^2 - z - 1$. Dieses hat die Nullstellen

$$z_{1,2} = \frac{1}{2} \pm \sqrt{\frac{1}{4} + 1} = \frac{1 \pm \sqrt{5}}{2},$$

woraus sich die allgemeine Lösung

$$F_n = c_1 \left(\frac{1+\sqrt{5}}{2} \right)^n + c_2 \left(\frac{1-\sqrt{5}}{2} \right)^n$$

ergibt, wobei c_1 und c_2 von den Anfangsbedingungen abhängen. Für $F_0 = 0$ und $F_1 = 1$ ergibt sich das lineare Gleichungssystem

$$\begin{aligned} c_1 + c_2 &= 0 \\ c_1 \left(\frac{1+\sqrt{5}}{2} \right) + c_2 \left(\frac{1-\sqrt{5}}{2} \right) &= 1 \end{aligned}$$

für c_1 und c_2, das die Lösung $c_1 = \frac{1}{\sqrt{5}}, c_2 = -\frac{1}{\sqrt{5}}$ hat. Wir erhalten also die Darstellung

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n.$$

Bemerkung 4.1. Die obige Darstellung der Fibonacci-Zahlen ist nicht nur von rein theoretischem Interesse, sie bietet auch einen effizienteren Algorithmus zur Berechnung von F_n. Der direkt auf der Definition $F_n = F_{n-1} + F_{n-2}$ basierende Algorithmus benötigt $\Theta(n)$ Zeit zur Berechnung von F_n. Zur Verwendung der obigen Darstellung muss im Wesentlichen a^n (für $a = \frac{1\pm\sqrt{5}}{2}$) berechnet werden. Das ist in Zeit $O(\log n)$ wie folgt möglich: zunächst berechnen wir a^n für hinreichend viele Zweierpotenzen n, also $a^0, a^1, a^2, a^4, a^8, \ldots$. Aus a^i kann $a^{2i} = (a^i)^2$ mit einer einzigen Multiplikation berechnet werden. Wir schreiben dann n binär und multiplizieren die entsprechenden Potenzen, also z.B. $(37)_\mathrm{b} = 100101$ und damit $a^{37} = a^{32}a^4a^1$.

Zum Abschluss dieses Abschnitts über lineare Rekursionsgleichungen mit konstanten Koeffizienten wollen wir nun auch noch den inhomogenen Fall betrachten. Dieser ist leicht zu behandeln.

Definition 4.5. Eine *inhomogene lineare Rekursionsgleichung mit konstanten Koeffizienten k-ter Ordnung* ist von der Form

$$x_n = c_{k-1}x_{n-1} + \cdots + c_0 x_{n-k} + d \text{ für } n \geq k \tag{4.2}$$

Satz 4.3. *Sei K ein Körper, seien $c_0, \ldots, c_{k-1}, d \in K$, dann hat (4.2) die Lösung $(x_n)_{n\geq 0}$ definiert durch $x_n = y_n - \frac{d}{C-1}$ wobei $C = \sum_{i=0}^{k-1} c_i$ und y_n ist Lösung von*

$$\begin{aligned} y_n &= x_n + \frac{d}{C-1} \text{ für } 0 \leq n < k \\ y_n &= c_{k-1}y_{k_1} + \cdots + c_0 y_{n-k} \text{ für } n \geq k \end{aligned}$$

Beweis. Wir gehen mit Induktion nach n vor. Der Fall $n < k$ folgt direkt aus der Definition von x_n und y_n. Für $n \geq k$ haben wir

$$\begin{aligned} x_n &= c_{k-1}x_{n-1} + \cdots + c_0 x_{n-k} + d = c_{k-1}(y_{n-1} - \frac{d}{C-1}) + \cdots + c_0(y_{n-k} - \frac{d}{C-1}) + d \\ &= c_{k-1}y_{k-1} + \cdots + c_0 y_{n-k} - C \cdot \frac{d}{C-1} + d = y_n - \frac{d}{C-1}. \end{aligned}$$

□

4.3 Die Substitutionsmethode

Als Substitutionsmethode bezeichnet man die aus den folgenden beiden Schritten bestehende Vorgehensweise zum Auflösen einer Rekursionsgleichung:

1. Errate die Form der Lösung.
2. Beweise dass die Form korrekt ist.

Üblicherweise wird der zweite Schritt mit Induktion gemacht wobei die Verwendung der Induktionshypothese die Form einer *Substitution* eines T-Terms durch die erratene rechte Seite hat. Man verwendet dabei einen Lösungsansatz mit unbekannten Konstanten und bestimmt diese Konstanten erst im Lauf des zweiten Schritts (bzw. deren Existenz zu zeigen). Eine ähnliche Vorgehensweise (Beweisansatz mit unbekannten Konstanten, die erst nach Abschluss des Beweises bestimmt werden) ist in vielen Situationen zum Beweis von Abschätzungen nützlich.

Beispiel 4.4. Eine etwas vereinfachte Variante der Rekursionsgleichung von Sortieren durch Verschmelzen ist

$$T(n) = T(\left\lceil \frac{n}{2} \right\rceil) + T(\left\lfloor \frac{n}{2} \right\rfloor) + n.$$

Wir wollen uns jetzt nicht mehr für die exakte Lösung interessieren, sondern nur noch für eine asymptotische Lösung. Deswegen sind die konkreten Zahlenwerte eines endlichen Anfangs von T irrelevant und werden gar nicht mehr angegeben. Zunächst erraten wir die obere Schranke $T(n) = O(n \log n)$. Wir müssen also zeigen dass $\exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : T(n) \leq cn \log n$. Die Konstanten c und n_0 lassen wir noch offen und setzen zu einem Induktionsbeweis an.

Bezüglich des Induktionsanfangs bemerken wir, dass die Funktion $n \mapsto n \log n$ die Nullstellen 0 und 1 hat, danach ist sie positiv. Nachdem wir nicht garantieren können dass $T(0) = T(1) = 0$ müssen wir uns auf $n_0 \geq 2$ einschränken. Dann muss als Induktionsbasis ein hinreichend großes Intervall $[n_0, b[$ gewählt werden, so dass sowohl $n \mapsto \lceil \frac{n}{2} \rceil$ als auch $n \mapsto \lfloor \frac{n}{2} \rfloor$, angewandt auf $n \geq b$ immer noch größer gleich n_0 ist. Offensichtlich reicht dafür $b = 2n_0$. Damit schränken wir uns ein auf c mit $T(m) \leq cm \log m$ für $m \in [n_0, 2n_0[$.

Für den Induktionsschritt sei $n \geq 2n_0$. Wir nehmen an dass $T(m) \leq cm \log m$ für alle $m \in [n_0, n[$ und setzen an mit

$$T(n) \leq c \left\lceil \frac{n}{2} \right\rceil \log \left\lceil \frac{n}{2} \right\rceil + c \left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor + n = (*).$$

Falls $n = 2k$, dann

$$\begin{aligned} (*) &= cn \log \frac{n}{2} + n = cn \log n - cn \log 2 + n. \\ &= cn \log n - cn + n \end{aligned}$$

Nun schränken wir uns ein auf $c \geq 1$ und erhalten damit

$$\leq cn \log n.$$

Falls $n = 2k + 1$, dann

$$(*) = c(k+1)\log(k+1) + ck \log k + n \leq cn \log(k+1) + n$$

und da $k = \frac{n-1}{2}$ ist $k + 1 = \frac{n+1}{2}$ und damit

$$= cn \log \frac{n+1}{2} + n = cn \log(n+1) - cn + n.$$

Wir wollen $(*) \leq cn \log n$ erreichen und müssen uns dazu wieder einschränken auf $c \geq 1$. Das allein reicht aber noch nicht, der Term $-cn$ muss zusätzlich zu n auch $cn(\log(n+1) - \log n)$ entfernen. Wir brauchen also ein n_0 so dass für alle $n \geq n_0$ gilt

$$cn \log(n+1) - cn + n \leq cn \log n, \text{ d.h.}$$
$$\log(n+1) - 1 + \frac{1}{c} \leq \log n, \text{ d.h.}$$
$$\log(n+1) - \log n \leq 1 - \frac{1}{c}, \text{ d.h.}$$
$$\log \frac{n+1}{n} \leq 1 - \frac{1}{c}.$$

Wir müssen also c, n_0 finden so dass die folgenden Bedingungen erfüllt sind:

1. $n_0 \geq 2$
2. $\forall m \in [n_0, 2n_0[: T(m) \leq cm \log m$
3. $c \geq 1$
4. $\forall n \geq 2n_0 : \log \frac{n+1}{n} \leq 1 - \frac{1}{c}$

Dazu wählen wir zuerst ein $c > 1$, dann gibt es n_0 das 4. wahr macht da $c > 1$ impliziert dass $1 - \frac{1}{c} > 0$ und $\frac{n+1}{n} \searrow 1$ und damit $\log \frac{n+1}{n} \searrow 0$ für $n \to \infty$. Falls dieses n_0 zu klein war erhöhen wir es um $n_0 \geq 2$ zu erfüllen wodurch 4. beibehalten wird. Es bleibt noch, 2. zu erfüllen. Das geschieht durch hinreichende Erhöhung von c bei gleichbleibendem n_0 und die Beobachtung dass dadurch 1., 3. und 4. beibehalten werden.

Damit haben wir also gezeigt dass c und n_0 existieren die den Bedingungen 1.-4. genügen und damit dass $T(n) = O(n \log n)$.

Beispiel 4.5. Sei

$$T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + 1.$$

Eine Vermutung ist $T(n) = O(n)$, d.h. $\exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : T(n) \leq cn$. Auf direkte Weise würde das zum folgenden Ansatz für den Induktionsschritt führen:

$$T(n) \leq c \left\lfloor \frac{n}{2} \right\rfloor + c \left\lceil \frac{n}{2} \right\rceil + 1 = cn + 1$$

Wir wollen $T(n) \leq cn$ erreichen was offensichtlich unmöglich ist. Jetzt ist aber $T(n) = O(n)$ trotzdem wahr was durch den folgenden, verbesserten, Ansatz auch gezeigt werden kann: $\exists c > 0, d > 0, n_0 \in \mathbb{N} \forall n \geq n_0 : T(n) \leq cn - d$. Dann erhalten wir

$$T(n) \leq c \left\lfloor \frac{n}{2} \right\rfloor - d + c \left\lceil \frac{n}{2} \right\rceil - d + 1 = cn - 2d + 1 \leq cn - d$$

unter der Voraussetzung $d \geq 1$. Der Induktionsanfang wird wie oben behandelt und so erhält man $T(n) = O(cn - d) = O(n)$.

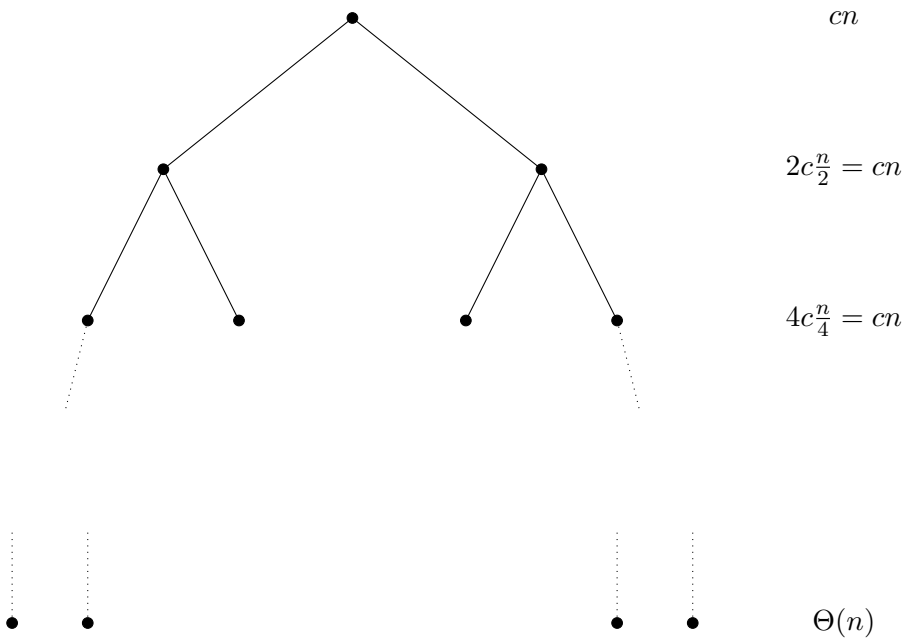


Abbildung 4.1: Rekursionsbaum von $T(n) = 2T(\frac{n}{2}) + cn$

4.4 Die Rekursionsbaummethode

Die Rekursionsbaummethode besteht im Aufzeichnen des durch eine Rekursionsgleichung induzierten Rekursionsbaums sowie in der anschließenden Summierung der im gesamten Baum anfallenden Kosten. Sie wird oft verwendet, um zu einer Vermutung zu gelangen, die danach mit der Substitutionsmethode bewiesen wird. Deshalb wird mit Rekursionsbäumen oft recht ungenau gearbeitet, z.B. lässt man die Rundungsoperatoren $\lfloor\cdot\rfloor$ und $\lceil\cdot\rceil$ weg oder man ersetzt Ausdrücke wie $\Theta(f(n))$ durch $cf(n)$.

Beispiel 4.6. Die Rekursion von Sortieren durch Verschmelzen war

$$T(n) = \begin{cases} \Theta(1) & \text{für } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{für } n \geq 2 \end{cases}.$$

Wir vereinfachen diese zu $T(n) = 2T(\frac{n}{2}) + cn$ und zeichnen den Rekursionsbaum auf, siehe Abbildung 4.1.

In der ersten Ebene, d.h. am Wurzelknoten, treten Kosten von cn auf. In der zweiten Ebene treten an jedem Knoten Kosten von $c\frac{n}{2}$ auf. Da es in der zweiten Ebene zwei Knoten gibt, treten insgesamt in der zweiten Ebene Kosten von cn auf. In der dritten Ebene gibt es vier Konten mit Kosten jeweils $c\frac{n}{4}$, also hat auch diese Ebene Kosten von cn, usw. Jede Ebene hat also Kosten cn. Wie viele Ebenen gibt es? Bis n auf eine Konstante (z.B. konkret auf 1) reduziert wurde benötigen wir $\log n$ Ebenen. Der Rekursionsbaum ist also ein vollständiger Binärbaum der Tiefe $\log n$ wobei wir auch hier, im Sinne einer vereinfachenden Annahme, die Tatsache vernachlässigen dass $\log n$ nicht notwendigerweise eine natürliche Zahl ist und nicht klar ist was mit einem Baum reeller Tiefe gemeint sein soll. Ein solcher Baum hat $\Theta(n)$ Blätter. So kommen wir also zur Vermutung $T(n) = cn \log n + \Theta(n) = \Theta(n \log n)$.

Beispiel 4.7. Betrachten wir nun die vereinfachte Rekursionsgleichung

$$T(n) = 3T(\frac{n}{4}) + cn^2$$

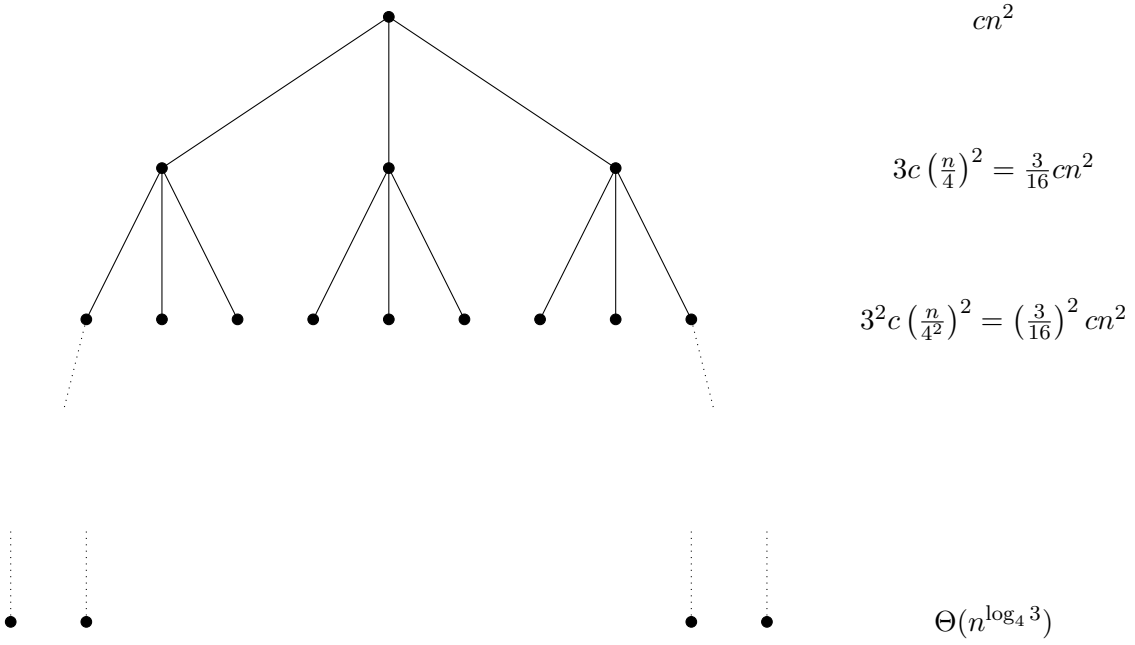


Abbildung 4.2: Rekursionsbaum von $T(n) = 3T(\frac{n}{4}) + cn^2$

Der Rekursionsbaum für diese Gleichung ist in Abbildung 4.2 zu finden. Die erste Ebene hat Kosten cn^2, die zweite $\frac{3}{16}cn^2$, die dritte $\left(\frac{3}{16}\right)^2 cn^2$ usw. Dieser Rekursionsbaum ist ein vollständiger Baum der Arität 3 mit Tiefe $\log_4 n$, er hat also $3^{\log_4 n} = n^{\log_4 3}$ Blätter. Nun beobachten wir dass

$$\sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3} < cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + n^{\log_4 3} = cn^2 \frac{1}{1-\frac{3}{16}} + n^{\log_4 3} = O(n^2)$$

und erhalten somit die Vermutung $T(n) = O(n^2)$.

4.5 Die Mastermethode

Summenbildungen über Rekursionsbäume, wie sie in den beiden Beispielen im vorherigen Abschnitt auftreten, folgen immer dem selben Muster. Wir werden nun einen allgemeinen Satz über solche Summen beweisen, der dann erlaubt die Lösungen der allermeisten Teile-und-herrsche Rekursionsgleichungen als Korollare zu erhalten. Dazu betrachten wir $a_0, a_1 \in \mathbb{N}$ mit $a = a_0 + a_1 \geq 1$, $b > 1$ und eine Rekursionsgleichung der Form

$$T(n) = a_0 T(\left\lfloor \frac{n}{b} \right\rfloor) + a_1 T(\left\lceil \frac{n}{b} \right\rceil) + f(n) \tag{4.3}$$

Da wir uns nur für asymptotische Lösungen interessieren, entfällt die Angabe von Anfangswerten. Der Anfang des Rekursionsbaums der Gleichung (4.3) ist in Abbildung 4.3 dargestellt. Um das iterierte auf- und abrunden darzustellen verwenden wir Zeichenketten im Alphabet $\{0, 1\}$ wobei 0 für Abrunden und 1 für Aufrunden steht.

Definition 4.6. $\{0,1\}^*$ ist die Menge aller endlich langen aus 0 und 1 bestehenden Zeichenketten (Wörtern) inklusive dem Leerwort ε. Die Länge $|w|$ eines Wortes $w \in \{0,1\}^*$ ist die

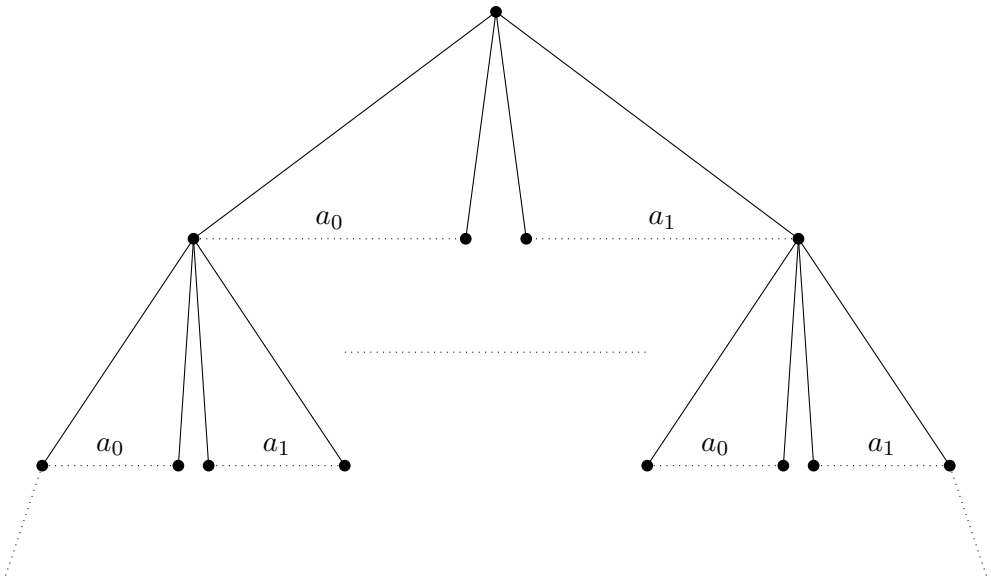


Abbildung 4.3: Anfang des Rekursionsbaums von Gleichung (4.3)

Anzahl der Zeichen aus denen w besteht. Die Länge des Leerwortes ist $|\varepsilon| = 0$. Für $w \in \{0,1\}^*$ bezeichnet $n_0(w)$ die Anzahl von Nullern in w und $n_1(w)$ die Anzahl von Einsern in w. Für einen Buchstaben x und ein $i \in \mathbb{N}$ bezeichnet x^i das Wort das aus i Vorkommen des Buchstaben x besteht.

Definition 4.7. Sei $b > 1$. Wir definieren für $n \in \mathbb{N}$ und $w \in \{0,1\}^*$ die Zahl $n_w \in \mathbb{N}$ als

$$n_w = \begin{cases} n & \text{falls } w = \varepsilon \\ \left\lfloor \frac{n_v}{b} \right\rfloor & \text{falls } w = 0v \\ \left\lceil \frac{n_v}{b} \right\rceil & \text{falls } w = 1v \end{cases}$$

Jeder Knoten im Baum entspricht also T angewandt auf ein bestimmtes n_w. Ein w kommt im Baum an $a_0^{n_0(w)} a_1^{n_1(w)}$ vielen Stellen vor. Die lokalen Kosten an einem Knoten der $T(n_w)$ entspricht sind $f(n_w)$. Wir wollen diese lokalen Kosten über alle Knoten im Baum summieren.

Wie tief ist dieser Baum? Falls n eine Potenz von b ist, und somit niemals gerundet wird, ist klar: nach $\log_b n$ Schritten in die Tiefe ist $T(1)$ erreicht. Aber auch der Fall wo n keine Potenz von b ist, ist nicht wesentlich komplizierter: n_w verhält sich bis auf eine additive Konstante so wie $\frac{n}{b^{|w|}}$, siehe Abbildung 4.4.

Lemma 4.2. *Sei $b > 1$, dann existiert ein $c > 0$ so dass für alle $n \in \mathbb{N}$ und $w \in \{0,1\}^*$: $\frac{n}{b^{|w|}} - c < n_w < \frac{n}{b^{|w|}} + c$.*

Beweis. Sei $|w| = j$, dann ist $n_{0^j} \leq n_w \leq n_{1^j}$. Wir behaupten weiters dass

$$n_{0^j} \geq \frac{n}{b^j} - \sum_{i=0}^{j-1} \frac{1}{b^i} \quad \text{und} \quad n_{1^j} \leq \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i}.$$

Wir zeigen die Ungleichung für n_{1^j} mit Induktion nach j. Falls $j = 0$, dann ist $n_\varepsilon = n$. Für den Induktionsschritt gilt

$$n_{1^{j+1}} = \left\lceil \frac{n_{1^j}}{b} \right\rceil \leq \frac{n_{1^j}}{b} + 1 \leq^{\text{IH}} \frac{1}{b} \left(\frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} \right) + 1 = \frac{n}{b^{j+1}} + \sum_{i=0}^{j} \frac{1}{b^i}.$$

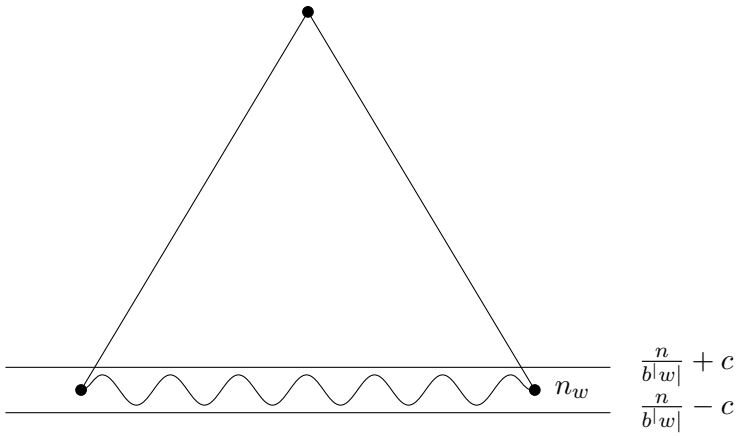


Abbildung 4.4: Illustration von Lemma 4.2 für $w \in \{0,1\}^j$

Analog dazu wird die Ungleichung für n_{0^j} gezeigt. Das Resultat folgt nun aus

$$\sum_{i=0}^{j-1} \frac{1}{b^i} < \sum_{i=0}^{\infty} \left(\frac{1}{b}\right)^i = \frac{1}{1-\frac{1}{b}} = \frac{b}{b-1} = c.$$

□

Für die Tiefe $d(n)$ des Baums können wir nun eine beliebige Funktion wählen, die $\frac{n}{b^{d(n)}} = \Theta(1)$ erfüllt. Dann ist nämlich nach Lemma 4.2 auch für alle $w \in \{0,1\}^{d(n)}$ erreicht dass $n_w = \Theta(1)$ und damit auch für alle $w \in \{0,1\}^{d(n)}$ dass $T(n_w) = \Theta(1)$. Wir setzen $d(n) = \lfloor \log_b n \rfloor$ und beobachten dass $\frac{n}{b^{d(n)}} = \Theta(1)$. Dementsprechend definieren wir die Indizes der inneren Knoten und die Indizes der Blattknoten als:

Definition 4.8. $X_I(n) = \{0,1\}^{\leq d(n)-1}$ und $X_B(n) = \{0,1\}^{d(n)}$.

Die Kosten des gesamten Baums setzen sich zusammen aus den Kosten an den inneren Knoten $\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w)$ und den Kosten an den Blättern. Jedes Blatt verursacht konstante Kosten. Für die Bestimmung der Anzahl der Blätter spielt der Unterschied zwischen a_0 und a_1 keine Rolle und somit können wir zu diesem Zweck den Baum als vollständigen Baum der Arität a mit Tiefe $d(n)$ betrachten. Dieser hat $a^{d(n)} = a^{\lfloor \log_b n \rfloor} = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$ Blätter. Also erhalten wir

$$T(n) = \Theta(n^{\log_b a}) + \sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w).$$

für die Summe über den gesamten Baum.

Eine weitere nützliche Beobachtung ist: Falls wir eine Funktion φ schichtweise summieren wollen die nur von der Länge von w, nicht aber von w selbst abhängt, dann spielt der Unterschied zwischen a_0 und a_1 ebenfalls keine Rolle und wir können ebenfalls den Baum als einen vollständigen Baum mit Arität a betrachten und entsprechend aufsummieren, d.h.

$$\sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} \varphi(|w|) = a^j \varphi(j).$$

Damit ist also

$$\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} \varphi(|w|) = \sum_{j=0}^{d(n)-1} a^j \varphi(j).$$

Satz 4.4 (Master-Theorem für Teile-und-herrsche-Rekursionsgleichungen)**.** *Seien $a_0, a_1 \in \mathbb{N}$ mit $a = a_0 + a_1 \geq 1$, sei $b > 1$ und sei $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Dann hat die Rekursionsgleichung*

$$T(n) = a_0 T(\left\lfloor \frac{n}{b} \right\rfloor) + a_1 T(\left\lceil \frac{n}{b} \right\rceil) + f(n)$$

die folgende asymptotische Lösung.

1. *Falls $f(n) = O(n^{\log_b a - \varepsilon})$ für ein $\varepsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$.*
2. *Falls $f(n) = \Theta(n^{\log_b a})$, dann $T(n) = \Theta(n^{\log_b a} \log n)$.*
3. *Falls es ein $c < 1$ gibt, so dass für alle bis auf endlich viele $n \in \mathbb{N}$ die Ungleichung $a_0 f(\lfloor \frac{n}{b} \rfloor) + a_1 f(\lceil \frac{n}{b} \rceil) \leq c f(n)$ gilt, dann ist $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für ein $\varepsilon > 0$ und $T(n) = \Theta(f(n))$.*

Wie man sehen kann ist für dieses Resultat vor allem a wesentlich, a_0 und a_1 spielen eine untergeordnete Rolle. Deshalb werden solche Rekursionsgleichungen oft auch geschrieben als $T(n) = aT(\frac{n}{b}) + f(n)$ wobei jedes der a Vorkommen von $\frac{n}{b}$ für $\lceil \frac{n}{b} \rceil$ oder $\lfloor \frac{n}{b} \rfloor$ steht.

Beweis. Sei $g(n) = \sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w)$. In Fall 1 haben wir

$$g(n) = O\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} n_w^{\log_b a - \varepsilon} \right) = O\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} \left(\frac{n}{b^{|w|}} \right)^{\log_b a - \varepsilon} \right)$$

weil $n_w = \Theta(\frac{n}{b^{|w|}})$ und weiters

$$\begin{aligned}
&= O\left(\sum_{j=0}^{d(n)-1} a^j \left(\frac{n}{b^j} \right)^{\log_b a - \varepsilon} \right) = O\left(n^{\log_b a - \varepsilon} \sum_{j=0}^{d(n)-1} \left(\frac{a}{b^{\log_b a - \varepsilon}} \right)^j \right) \\
&= O\left(n^{\log_b a - \varepsilon} \sum_{j=0}^{d(n)-1} (b^\varepsilon)^j \right) = O\left(n^{\log_b a - \varepsilon} \frac{b^{\varepsilon d(n)} - 1}{b^\varepsilon - 1} \right) \\
&= O(n^{\log_b a - \varepsilon} n^\varepsilon) = O(n^{\log_b a})
\end{aligned}$$

weil $b^{d(n)} = \Theta(n)$. Insgesamt erhalten wir also $T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a})$.
In Fall 2 haben wir

$$g(n) = \Theta\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} n_w^{\log_b a} \right) = \Theta\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} \left(\frac{n}{b^{|w|}} \right)^{\log_b a} \right)$$

weil $n_w = \Theta(\frac{n}{b^{|w|}})$ und weiters

$$\begin{aligned}
&= \Theta\left(\sum_{j=0}^{d(n)-1} a^j \left(\frac{n}{b^j} \right)^{\log_b a} \right) = \Theta\left(n^{\log_b a} \sum_{j=0}^{d(n)-1} \left(\frac{a}{b^{\log_b a}} \right)^j \right) \\
&= \Theta\left(n^{\log_b a} d(n) \right) = \Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \log n).
\end{aligned}$$

Insgesamt erhalten wir also $T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \log n) = \Theta(n^{\log_b a} \log n)$.
In Fall 3 ist $cf(n) \geq a_0 f(\lfloor \frac{n}{b} \rfloor) + a_1 f(\lceil \frac{n}{b} \rceil)$ für alle $n \geq n_0$. Sei nun $d_0(n) = d(n) - p$ wobei $p \in \mathbb{N}$ so gewählt ist, dass $\frac{n}{b^{d_0(n)}} \geq n_0$ für alle $n \geq n_0$. Außerdem ist $\frac{n}{b^{d_0(n)}} = \Theta(1)$. Sei $n \geq n_0$. Wir zeigen zunächst

$$c^j f(n) \geq \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w) \text{ für } j = 0, \ldots, d_0(n) \qquad (*)$$

mit Induktion nach j. Falls $j = 0$, dann ist $f(n) = f(n_\varepsilon)$. Für den Induktionsschritt haben wir

$$\begin{aligned}
c^{j+1} f(n) &\geq c \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w) \geq \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} (a_0 f(n_{0w}) + a_1 f(n_{1w})) \\
&= \sum_{v \in \{0,1\}^{j+1}} a_0^{n_0(v)} a_1^{n_1(v)} f(n_v).
\end{aligned}$$

Dann ist $c^{d_0(n)} f(n) \geq \sum_{w \in \{0,1\}^{d_0(n)}} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w)$, da aber $\frac{n}{b^{d_0(n)}} = \Theta(1)$ ist wegen Lemma 4.2 auch $\min\{f(n_w) \mid w \in \{0,1\}^{d_0(n)}\}$ eine Konstante q und somit $c^{d_0(n)} f(n) \geq q a^{d_0(n)}$. Damit erhalten wir

$$f(n) \geq q \left(\frac{a}{c}\right)^{\lfloor \log_b n \rfloor - p} = \Theta\left(\left(\frac{a}{c}\right)^{\log_b n}\right) = \Theta(n^{\log_b \frac{a}{c}}).$$

Nun ist $n^{\log_b \frac{a}{c}} = n^{\log_b a - \log_b c}$ und da $c < 1$ war ist $\log_b c < 0$, also ist $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für $\varepsilon = -\log_b a > 0$.

Es bleibt zu zeigen dass $g(n) = \Theta(f(n))$. Zunächst einmal ist klar dass $g(n) = \Omega(f(n))$ da $f(n) = f(n_\varepsilon)$ ja ein Summand von $g(n)$ ist. Für die andere Richtung betrachten wir den Rekursionsbaum bis zur Tiefe $d_0(n)$. Die Anzahl von Blättern ist $a^{d_0(n)} = a^{\lfloor \log_b n \rfloor - p} = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$. Für $j \in \{d_0(n), \ldots, d(n) - 1\}$ haben wir: $\forall w \in \{0,1\}^j$: $\frac{n}{b^{|w|}} = \Theta(1)$, also $\forall w \in \{0,1\}^j$: $n_w = \Theta(1)$, also $\forall w \in \{0,1\}^j$: $f(n_w) = \Theta(1)$. Weiters ist die Anzahl der $f(n_w)$-Summanden an einem Blatt der Tiefe $d_0(n)$ konstant. Also sind die Kosten an einem Blatt der Tiefe $d_0(n)$ konstant und wir erhalten:

$$g(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{d_0(n)-1} \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w) \leq \Theta(n^{\log_b a}) + \sum_{j=0}^{d_0(n)-1} c^j f(n)$$

$$< \Theta(n^{\log_b a}) + f(n) \sum_{j=0}^{\infty} c^j = \Theta(n^{\log_b a}) + \frac{1}{1-c} f(n) = \Theta(f(n))$$

da $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für ein $\varepsilon > 0$. Also ist $g(n) = O(f(n))$. Insgesamt erhalten wir also $T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n))$. □

Korollar 4.1. *Die Rekursionsgleichung*

$$T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + \Theta(n)$$

die sowohl die Laufzeit von Sortieren durch Verschmelzen als auch jene von unserem Teile-und-herrsche Algorithmus für das dichteste Punktepaar beschreibt hat die Lösung $T(n) = \Theta(n \log n)$.

Beweis. In der Notation des master-Theorems gilt $a = b = 2$ und es trifft der zweite Fall zu. Die Lösung ist also $\Theta(n^{\log_b a} \log n) = \Theta(n \log n)$. □

Korollar 4.2. *Die Rekursionsgleichung*

$$T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$$

die die Laufzeit des einfachen rekursiven Algorithmus zur Matrixmultiplikation beschreibt hat die Lösung $T(n) = \Theta(n^3)$.

Beweis. In der Notation des master-Theorems ist $a = 8, b = 2$ und es trifft der erste Fall zu da $n^2 = O(n^{3-\varepsilon})$ für ein $\varepsilon > 0$. Die Lösung ist also $\Theta(n^{\log_b a}) = \Theta(n^3)$. □

Korollar 4.3. *Die Rekursionsgleichung*

$$T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$$

die die Laufzeit des Strassen-Algorithmus zur Matrixmultiplikation beschreibt hat die Lösung $T(n) = \Theta(n^{2{,}807\ldots})$.

Beweis. In der Notation des master-Theorems ist $a = 7, b = 2$. Es ist $\log_2 7 = 2{,}807\ldots$. Da $n^2 = O(n^{2{,}807\ldots-\varepsilon})$ für ein $\varepsilon > 0$ trifft der erste Fall zu. Die Lösung ist also $\Theta(n^{\log_b a}) = \Theta(n^{2{,}807})$. □

Beispiel 4.8. Auf die Rekursionsgleichung

$$T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + n \log n$$

ist das master-Theorem in dieser Form nicht anwendbar. Es ist nämlich $a = b = 2$, so dass $n \log n$ mit n verglichen werden muss. Nun ist aber $n \log n \neq \Theta(n)$, $n \log n \neq \Omega(n^{1+\varepsilon})$ sowie $n \log n \neq O(n^{1-\varepsilon})$. Damit trifft keiner der Fälle des master-Theorems zu. Diese Rekursionsgleichung hat die asymptotische Lösung $T(n) = \Theta(n \log^2 n)$.

Kapitel 5

Datenstrukturen

In diesem Kapitel werden wir uns mit Datenstrukturen beschäftigen. Eine Datenstruktur ist eine Struktur zur Speicherung und Organisation von Daten, die typischerweise gewisse Operationen (auf effiziente Weise) zur Verfügung stellt. Eine einfache Datenstruktur, das Datenfeld, haben wir bereits kennengelernt. Es erlaubt Lese- und Schreibzugriff auf ein beliebiges Element dessen Index bekannt ist in konstanter Zeit. Algorithmen und Datenstrukturen gehen Hand in Hand: jede Datenstruktur benötigt Algorithmen, welche die gewünschten Operationen zur Verfügung stellen und umgekehrt: oft sind gewisse Datenstrukturen notwendig, damit ein Algorithmus eine bestimmte Laufzeit erreicht. So haben wir zum Beispiel im teile-und-herrsche Algorithmus zur Lösung des dichtesten Punktepaarproblems (Algorithmus 10) eine Menge auf eine Weise repräsentiert, die das Durchlaufen anhand zweier unterschiedlicher Ordnungen erlaubt hat. Auch diese Repräsentation kann als Datenstruktur betrachtet werden.

Datenstrukturen sind darüber hinaus auch deswegen von großer Bedeutung in der Informatik, weil die Betrachtung von Berechnungsproblemen und Algorithmen als Lösungen dieser zwar für viele aber bei weitem nicht für alle Anwendungen adäquat ist. Oft befindet man sich in der Praxis in einer Situation die dynamisch ist, also nicht durch eine Eingabe-Ausgabe-Relation vollständig beschrieben werden kann.

5.1 Das Wörterbuchproblem

In diesem Abschnitt wollen wir das *Wörterbuchproblem* betrachten. Das *Wörterbuchproblem* besteht darin eine möglichst effiziente Organisation einer endlichen Menge von Datensätzen zu finden die abgefragt und verändert(!) werden kann. Wir nehmen dabei an, dass jeder Datensatz D durch einen eindeutigen Schlüssel $D.x$ identifiziert wird. Wir wollen, dass unser Wörterbuch M zumindest die folgenden Operationen unterstützt:

1. $M.Suche(x)$ gibt jenes D aus M zurück dessen Schlüssel x ist falls es existiert.
2. $M.Einfügen(D)$ fügt den neuen Datensatz D zu M hinzu.
3. $M.Löschen(x)$ entfernt den Datensatz mit Schlüssel x aus M.

Zum Beispiel könnte man an einer Repräsentation aller Studenten dieser Universität interessiert sein. Diese Menge verändert sich im Laufe der Zeit. Die Matrikelnummer kann als eindeutiger Schlüssel dienen. Als Lösung für das Wörterbuchproblem erwarten wir eine geeignete Datenstruktur gemeinsam mit zumindest drei Algorithmen, die die oben beschriebenen Operationen (möglichst effizient) durchführen.

Die einfachste Art ein solches Wörterbuch zu realisieren besteht darin, M als Datenfeld aufzufassen. Die Laufzeitkomplexität der Operationen (wobei $n = |M|$) ist dann wie folgt:

1. $M.\mathit{Suche}(x)$ benötigt Zeit $O(n)$, d.h. genauer: $\Theta(n)$ im schlechtesten Fall und $\Theta(1)$ im besten Fall.
2. $M.\mathit{Einfügen}(D)$ benötigt Zeit $\Theta(n)$ da sichergestellt werden muss, dass kein Duplikat erzeugt wird und dafür im schlechtesten Fall ($D.x$ existiert noch nicht) das gesamte Datenfeld durchlaufen werden muss.
3. $M.\mathit{Löschen}(x)$ benötigt Zeit $\Theta(n)$ da D mit $D.x = x$ zunächst gefunden werden muss und dann das Datenfeld verkürzt werden muss.

Eine bessere Darstellung von M besteht darin, ein nach Schlüssel (aufsteigend) sortiertes Datenfeld zu verwenden. Dann haben wir die folgenden Operationen:

1. $M.\mathit{Suche}(x)$ in Zeit $O(\log n)$ mit binärer Suche (engl. *binary search*), siehe Algorithmus 11.
2. $M.\mathit{Einfügen}(D)$ in Zeit $\Theta(n)$ da das Datenfeld verlängert werden muss.
3. $M.\mathit{Löschen}(x)$ in Zeit $\Theta(n)$ da das Datenfeld verkürzt werden muss.

Algorithmus 11 Binäre Suche

```
Prozedur BSuche(A, x)
    Antworte BSucheRek(A, x, 1, A.Länge)
Ende Prozedur

Prozedur BSucheRek(A, x, l, r)
    Falls l > r dann
        Antworte "nicht gefunden"
    sonst
        m := ⌈(l+r)/2⌉
        Falls A[m] = x dann
            Antworte m
        sonst falls A[m] < x dann
            Antworte BSucheRek(A, x, m + 1, r)
        sonst                                          ▷ A[m] > x
            Antworte BSucheRek(A, x, l, m − 1)
        Ende Falls
    Ende Falls
Ende Prozedur
```

Mit einem sortierten Datenfeld ist die Suche also effizient, Änderung hingegen nicht.

Eine Datenstruktur in der lokale Änderungen auf effiziente Weise gemacht werden können sind verkettete Listen. Hier unterscheidet man zwischen einfach verketteten und doppelt verketteten Listen. Die Grundidee ist, dass die Elemente einer Liste jeweils einzeln an einer beliebigen Stelle im Speicher abgelegt werden und jedes Element auf das nächste (einfache Verkettung) bzw. auf das nächste und das vorherige verweist (doppelte Verkettung). Diese Verweise auf Speicherpositionen werden als *Zeiger* (engl. *pointer*) bezeichnet. Ein Zeiger verweist entweder auf einen bestimmten Ort im Speicher oder er hat den Wert Nil, den Nullzeiger. Realisiert wird ein

einzelner Knoten durch ein Tupel v (für Vertex), dessen Elemente im Fall einer einfach verketteten Liste $v.D$ und $v.nächster$ sind, im Fall einer doppelt verketteten Liste $v.D$, $v.nächster$ und $v.vorheriger$ sind. Der Vorteil einer doppelt verketteten Liste gegenüber einer einfach verketteten besteht darin, dass sie in beide Richtungen durchlaufen werden kann, ihr Nachteil darin, dass sie (konstant) mehr Speicherplatz benötigt. Doppelt verkettete Listen erlauben Einfügen und löschen in konstanter Zeit durch Umsetzen der pointer Eine doppelt verkette Liste als Darstellung von M erlaubt die folgenden Operationen:

1. $M.Suche(x)$ benötigt Zeit $O(n)$ genauer: $\Theta(n)$ im schlechtesten Fall, $\Theta(1)$ im besten Fall und $\Theta(n)$ im Durchschnittsfall.
2. $M.Einfügen(D)$ in Zeit $\Theta(n)$ da sichergestellt werden muss, dass kein Duplikat erzeugt wird
3. $M.Löschen(x)$ wie die Suche in Zeit $O(n)$ da D mit $D.x = x$ gefunden werden muss.

Doppelt verkettete Listen erlauben zusätzlich noch die folgenden effizienten Operationen:

1'. $M.Einfügen(D)$ in Zeit $\Theta(1)$ unter der Annahme dass D noch nicht in M vorkommt.

2'. $M.Löschen(v)$ in Zeit $\Theta(1)$ durch Umsetzen der Zeiger unter der Voraussetzung dass v ein Element von M ist.

Bei verketteten Listen ist zwar im Vergleich zum unsortierten Datenfeld nichts gewonnen was die Suche angeht, allerdings erhalten wir bei den dynamischen Operationen Einfügen und Löschen neue Möglichkeiten.