28. BWINF, Endrundenvorbeitung

Matthias Springer

3. Oktober 2010

Inhaltsverzeichnis

1 Vorwort

Dies ist meine Vorbereitung auf die Endrunde des 28. Bundeswettbewerbs Informatik. Ich habe sie für zukünftige Endrundenteilnehmer veröffentlicht, die sich gerade Gedanken darüber machen, wie sie sich auf die Endrunde vorbereiten könnten.

Ich habe mir bei meiner BWINF-Vorbereitung Bücher über Algorithmen und theoretische Informatik durchgelesen. Die Kapitel, die ich als wichtig empfunden hatte, habe ich dann kurz zusammengefasst. Kurz vor der Endrunde habe ich dann die Zusammenfassung nochmals gründlich durchgelesen.

Achtung: Es handelt sich nur um die Kapitel, die ich persönlich als wichtig empfand und um die Kapitel, die mich besonders interessieren. Dementsprechend habe ich Schwerpunkte gesetzt. Außerdem kann ich nicht garantieren, dass die Zusammenfassung keine Fehler enthält! Es kann gut sein, dass ich das ein oder andere Thema einfach falsch verstanden habe! Ich denke der größte Nutzen für jemanden, der sich auf die Endrunde vorbereitet, liegt deshalb darin, sich einen Überblick über die verschiedenen Themengebiete zu verschaffen. Danach kann man sich über die einzelnen Themen im Internet informieren oder Bücher lesen. Diese Zusammenfassung diente mir wirklich nur als Denkstütze, um kurz das wichtigste nachschlagen zu können.

2 Graphentheorie

2.1 Adjazenzmatrix vs. Adjazenzliste

Aktion	**Matrix**	**Liste**	**sortierte Liste**
Speicher	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\| + \|E\|)$
Kante hinzufügen	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\|E\|)$ / $\mathcal{O}(log\|E\|)$?
Kante entfernen	$\mathcal{O}(1)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|E\|)$ / $\mathcal{O}(log\|E\|)$?
Kantenexistenz prüfen	$\mathcal{O}(1)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|E\|)$ / $\mathcal{O}(log\|E\|)$?
Nachbarschaft iterieren	$\mathcal{O}(\|V\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|E\|)$

2.2 Eigenschaften von Graphen

- **gerichteter Graph (Digraph):** Kanten besitzen Pfeile.
- **gefärbter Graph:** Knoten oder Kanten sind mit einer Farbe eingefärbt.
- **gewichteter Graph:** Kanten besitzen ein Gewicht (numerischer Wert).
- **Multigraph:** Mehrfachkanten zwischen zwei verschiedenen Knoten sind zulässig.
- **zyklisch (/ azyklisch):** Man kann im Kreis laufen.
- **k-Knoten- bzw. Kanten-zusammenhängend:** Es müssen mindestens k Knoten bzw. Kanten entfernt werden, damit der Graph in mehrere Komponenten zerfällt.
- **Kanten in Bäumen:** forward edge, back edge

2.3 Tiefensuche (Depth-First-Search, DFS)

- Tiefensuchbaum kann in $\mathcal{O}(|V| + |E|)$ generiert werden.
- Zuerst in die Tiefe (depth-first).
- Implementation über Rekursion oder Stack.
- Test auf Zyklen mit visited-Array möglich.
- Problem bei *unendlich* tiefen Graphen: iterative Tiefensuche mit maximaler Suchtiefe.
- Vergleiche auch BFS mit Schlange und Best-First-Search mit Heap (priority queue).

2.4 Artikulationsknoten

- Graph zerfällt bei Entfernung eines Artikulationsknoten in mindestens zwei Komponenten.
- Betrache hier nur ungerichteten Fall.
- Tiefensuche (DFS) auf dem Graphen, speichere discovery-time für jeden Knoten (einfach durchnummerieren).
- Ein Knoten ist ein Artikulationsknoten, wenn der kleinste erreichbare Knoten größer oder gleich der eigenen discovery-time ist (direkten Vorgängerknoten und nur über direkten Vorgängerknoten erreichbare Knoten nicht beachten!).
- Rückgabewert der visit-Prozedur: minimale discovery-time eines erreichbaren Knotens, also Minimum über alle rekursiven visit-Aufrufe und dem eigenen discovery-time-Wert.
- Spezialfälle: Blätter im Tiefensuchbaum sind keine Artikulationsknoten, Wurzel ist Artikulationsknoten, wenn sie mindestens 2 direkte Kinderknoten hat.
- Laufzeit wegen Tiefensuche $\mathcal{O}(|V| + |E|)$.

2.5 Topologische Sortierung

- Gegeben ein gerichteter Graph, der Prioritätsabhängigkeiten zwischen Aktivitäten darstellt, finde eine zulässige Reihenfolge der Aktivitäten (z.B. Pfeil von *Hemd* nach *Mantel* bedeutet: zuerst das Hemd anziehen, dann den Mantel).
- Tiefensuche (DFS) auf dem Graph anwenden und Knoten am Ende des Aufrufs (nachdem auch die rekursiven Aufrufe zurückgekehrt sind), am Anfang einer Liste einfügen.
- Laufzeit wegen Tiefensuche $\mathcal{O}(|V| + |E|)$.

2.6 Kürzeste Wege: Dijkstra-Algorithmus

- Sei MK die Menge der markierten Knoten, $D[i]$ der Abstand des Knotens i vom Startknoten a und $R[i]$ der Vorgänger des Knotens i im küresten Pfad.

1. $MK = \{a\}; D[a] = 0; D[i] = \infty, \forall i \neq a$.
2. Wähle Knoten i aus MK mit kleinster Entfernung zu a.
3. Für alle Nachfolger n von i: Falls $D[i] + dist[i][n] < D[n]$, aktualisiere $D[n]$, $R[n]$ und markiere n.
4. Entferne i aus MK und gehe zu Schritt 2.

- Funktioniert nicht mit negativen Kanten!
- Laufzeit $\mathcal{O}(|V|^2 + |E|)$, besser mit Heap.
- single-source shortest-path

2.7 Kürzeste Wege: Dijkstra ohne Computer

- Lege Netz (Schnur) mit Knoten und Kanten auf die Landkarte.
- Hebe Startpunkt hoch, bis Zielknoten auch oben. Gespannter Zug ist kürzester Weg.
- **Dijkstra:** Berechne $d[v]$ für alle Knoten, die von S aus erreichbar sind.
- Alle Knoten sind wartend außer S (hängend).
- Wähle Knoten mit minimalem Abstand $d[v]$, der wartend ist. Passe Abstände zu Nachfolgerknoten an (anfangs alle unendlich) und setze Knoten auf hängend. Implementierung mit Priority Queue.

2.8 Kürzeste Wege: Bellman-Ford-Algorithmus

1. $D[a] = 0; D[i] = \infty, \forall i \neq a$.
2. Wiederhole $|V|-1$ mal: Für alle Kanten $(u, v) \in E$: Wenn $D[u]+dist[u][v] < D[v]$, aktualisiere $D[v]$ und $R[v]$.
3. Für alle Kanten (u, v): Wenn $D[u] + dist[u][v] < D[v]$, gibt es einen negativen Zyklus.

- Funktioniert nicht mit negativen Zyklen!
- Laufzeit $\mathcal{O}(|V| \cdot |E|)$, im Allgemeinen aber langsamer als Dijkstra.
- single-source shortest-path

2.9 Kürzeste Wege: Floyd-Warshall-Algorithmus

- Dynamische Programmierung: $D^k[u][v]$ ist Länge des kürzesten Weges von u nach v, wobei über keine Knoten mit Index größer als k gegangen werden darf.
- $D^{k+1}[u][v] = \min\{D^k[u][v]; D^k[u][k+1] + D^k[k+1][v]\}$.
- Algorithmus: Baue DP-Tabelle auf, es wird immer nur auf $k-1$ zurückgegriffen, deshalb wird keine Dimension für k benötigt. Erhöhe k in der äußeren Schleife.
- Setze $D[u][v]$ anfangs auf die Kantengewichte der entsprechenden Kanten.

- Keine negativen Zyklen erlaubt.
- Laufzeit $\mathcal{O}(|V|^3)$, all-pairs shortest-paths.

2.10 Minimum spanning tree: Algorithmus von Prim

1. Beginne mit beliebigem Knoten, dieser stellt den Startgraph T dar.
2. Solange T nicht alle Knoten enthält: Füge Kante mit minimalem Gewicht zu einem Knotem außerhalb von T hinzu, sodass dieser Knoten Bestandteil von T wird.

- Effiziente Implementierung mit Prioritätswarteschlange, die alle nicht in T enthaltenen Knoten enthält.

2.11 Minimum spanning tree: Algorithmus von Kruskal

1. Beginne mit leerem Graphen T.
2. Füge die nächste Kante mit minimalem Gewicht ein. Wenn dadurch ein Zyklus entsteht, lösche die Kante wieder. (Sortiere Kantenliste ggf. vorher.)

- Algorithmus kann so angepasst werden, dass Laufzeit für das Sortieren dominiert.

2.12 Maximaler Fluss: Ford-Fulkerson-Algorithmus

1. Bilde den Residuengraphen mit Kanten $a \xrightarrow{x} b$, wenn in Richtung b noch x Fluss möglich ist und $a \xleftarrow{x} b$, wenn in Richtung b z.Zt. x Einheiten fließen (diese können dann zurückgeschickt werden).
2. Suche einen augmentierenden Pfad von der Quelle zur Senke.
3. Augmentiere entlang des gefundenen Pfades, passe Residuennetzwerk an.

- Laufzeit hängt von den Gewichten ab (!)
- Algorithmus terminiert nicht immer (Spezialfall mit rellen Zahlen und goldenem Schnitt).

2.13 Maximaler Fluss: MKM-Algorithmus

- **Geschichtetes Netzwerk:** Nur Forward-Kanten, Konstruktion über BFS.
- Laufzeit $\mathcal{O}(|V|^3)$.

1. Bilde den geschichteten Residuengraphen.
2. Wähle Knoten mit niedrigstem Potential (Minimum aus möglichem eingehendem und ausgehendem Fluss).

3. Führe push/pull auf diesen Knoten aus (dann rekursiv), sättige Kanten immer komplett, wenn möglich.

4. Gehe zu Schritt 1. Maximaler Fluss, wenn es keinen Pfad mehr von der Quelle zur Senke gibt.

2.14 Maximaler Fluss: Push-and-relabel (Goldberg, Tarjan)

- Wenn bei Knoten v: Schiebe genzen vorhandenen Fluss weiter nach vorne (so viel wie durch die Kanten geht): $h[v] \geq h[Nachfolger]$, d.h. immer bergab schieben.
- Wenn nichts mehr geht und zuviel Fluss vorhanden ist: Hebe eigene Höhe an und schiebe Fluss zurück.
- Überflüssiger Fluss kehrt zu S zurück.
- Höhe eines Knotens soll sich von Nachbarn immer nur um 1 unterscheiden.
- Laufzeit $\mathcal{O}(|V|^2|E|)$.

2.15 Zyklen finden

- **Ungerichtet:** mit DFS/BFS und visited-Array.
- **Gerichtet:** Speichere Zustand für Knoten: unbesucht, in Berarbeitung, fertig (am Ende eines visit-Aufrufs). Nur dann Zyklus gefunden, wenn auf einen Knoten getroffen wird, der noch in Bearbeitung ist.

2.16 SCC: Starke Zusammenhangskomponenten

- Tiefensuche mit Zustand *noch nicht besucht*, *in Bearbeitung* und *fertig* für jeden Knoten. Knoten ist in Bearbeitung, wenn der visit-Aufruf noch nicht endgültig zu Ende ist (also noch nicht alle rekursiven Aufrufe zurückgekehrt sind).
- Speichere zu jedem Knoten discovery-time und Komponentennummer.
- visit-Aufruf für alle Nachfolger: Wenn das erste Mal besucht, setze Komponentennummer = discovery-time und rekursiver Aufruf. Wenn Knoten in Bearbeitung: wenn dessen Komponentennummer kleiner als die eigene Komponentennummer: setze eigene Komponentennummer auf dessen Nummer. Wenn ein rekursiver Aufruf zurückkehrt: verfahre mit Komponentennummer wie bei Knoten in Bearbeitung.
- Am Ende eines Aufrufs: Wenn discovery-time = Komponentennummer: SCC gefunden.
- Laufzeit in $\mathcal{O}(|V| + |E|)$ mit einer Tiefensuche.

2.17 Kantengraph (Line Graph)

- Für einen Graphen G konstruiert man den Kantengraph $L(G)$.
- Jeder Knoten von $L(G)$ repräsentiert eine Kante in G.
- Zwei Knoten in $L(G)$ sind verbunden, wenn die Kanten in G gemeinsame Endpunkte haben.
- Chromatische Kantennummer in G ist die chromatische Knotennummer in $L(G)$.
- Wenn es in G einen Euler-Zyklus gibt, dann hat $L(G)$ einen Hamilon-Zyklus (Umkehrschluss gilt nicht!).
- Nicht jeder Graph ist ein Kantengraph eines anderen Graphen (Beineke's 9 verbotene Teilgraphen).

2.18 Färbung von Graphen

- Kantenfärhung ist Knotenfärbung auf dem Kantengraphen.
- Jeder planare Graoh ist 4-färbbar (erster computergestützer Beweis 1976).
- Praktische Anwendung: Registerzuteilung (Variablen im Prozessor, Compiler): Knoten sind Variablen, Interferenz-Kanten: gleichzeitig aktive Variablen, k-Färbung bei k Registern.
- Praktische Anwendung: Scheduling: Knoten sind Jobs, Kanten verbinden Jobs, die nicht gleichzeitig stattfinden dürfen. Chromatische Zahl ist die Anzahl der benötigten Zeitslots.
- NP-vollständiges Problem.
- Graph ist 2-färbbar, wenn er bipartit ist, also keinen Kreis ungerader Länge enthält.

2.19 Eulerzyklus (Eulertour)

- Alle Knoten haben geraden Grad und sind zusammenhängend $\Leftrightarrow$ Es gibt eine Eulertour.
- Starte bei beliebigem Knoten und gehe so lange beliebig über noch nicht besuchte Kanten, bis man wieder am Anfang ist.
- Entferne begangene Kanten und starte bei Knoten mit Verbindung zum gefundenen Zyklus, gehe wieder im Kreis.
- Füge Zyklen zusammen, wiederhole, bis alle Kanten besucht wurden.
- **Eulerpfad:** Verbinde die beiden Knoten mit ungeradem Grad, Start- und Endknoten des Pfades sind Kniten ungeraden Grades.

2.20 Briefträgerproblem (Eulertouren)

- Finde im Graph die Kanten mit ungeradem Grad (immer eine gerade Anzahl!).
- Bilde aus diesen Knoten den vollständigen gewichteten Graphen mit minimalem Abstand (im ursprünglichen Graph) als Kosten.
- Finde perfektes, kostenminimales (gewichtet, nicht-bipartit!) Matching (Paar), also $\frac{r}{2}$ Kanten, die dann in den ursprünglichen Graph eingefügt werden.
- Bilde die Eulertour mit dem bekannten Algorithmus.

2.21 Matching (Paarung) in Graphen

- Matching M: keine 2 Kanten aus M haben einen gemeinsamen Endknoten.
- v ist Endknoten einer Kante in M: gesättigt.
- Perfektes Matching: jeder Knoten ist M-gesättigt.
- Maximum-Matching: Es gibt kein Matching N das mehr Kanten als M hat.
- M-alternierender Weg: abwechselnd Kanten aus M und $E \setminus M$.
- M-erweiternder Weg: Anfangs- und Endknoten sind M-ungesättigt.
- **Beispiele:** Immobilienmakler: Zuweisung Wohnung-Kunde, evtl. mit Provision für jede Kante Haus-Kunde.

2.22 Bipartites kantengewichtetes Matching: Max-Flow

- Fluss-Algorithmen: z.B. Ford-Fulkerson.
- Bipartit: zwei knotendisjunkte Teilmengen von V.
- Baue Flussnetzwerk auf: Kanten von der Senke zu jedem Knoten in der einen Teilmenge mit Kapazität unendlich, ebenso Kanten von der anderen Teilmenge zur Quelle.
- Maximaler Fluss ist gewichtsmaximales Matching.

2.23 Kantenmaximales bipartites Matching ohne Kantegewichte

1. Markiere eine Zuordnung $(v_1 \in V_1) \leftrightarrow (v_2 \in V_2)$ beliebig.
2. Wiederhole, solange es einen ungesättigten Knoten v gibt:
3. Prüfe (direkte) Nachbarschaft von v und kombiniere, wenn möglich.

4. Gehe alle passenden Partner durch und frage nach, ob diese tauschen können, also einen anderen Partner finden (rekursiv weiterfragen bis der erste Fall, also passender Partner, eintritt).

- Betrache jeden Knoten maximal einmal.
- Algorithmus sucht nach einem M-erweiternden Weg bei aktuellem Matching M, der dann umgefärbt wird.
- Laufzeit $\mathcal{O}(|V||E|)$, $\mathcal{O}(\sqrt{(}|V|)|E|)$ möglich.

2.24 Maximum Cut (NP-vollständig)

- Teile Knoten in zwei Mengen V_1 und V_2.
- Zu maximierender Wert ist die Summe der Kantengewichte $v_1 \in V_1 \leftrightarrow v_2 \in V_2$.
- Näherungsweise mit simulierter Abkühlung berechnbar (z.B. 1 Bit für $v \in V_1$).
- Betrachte nur Anzahl der Kanten: 0,5-Approximation: Weise jeder Menge zufällig Knoten zu (derandomisiert: starte mit beliebiger Partition und vertausche Mengenzugehörigkeit von Knoten, wenn sich ein besserer Schnitt ergibt, $\mathcal{O}(|E|)$).

2.25 Maximum Independent Set (NP-vollständig)

- Teilmenge aus V, sodass keine Knoten daraus adjazent sind.
- Teilmenge ist independent $\Leftrightarrow$ Teilmenge im inversen Graph zu G ist eine Clique $\Leftrightarrow$ Inverse Teilmenge ist ein Vertex Cover (jede Kante in G hat einen Endpunkt in der (inversen) Teilmenge).
- α: Größe des Independent Sets, β: Größe des entsprechenden Vertex Covers, $\alpha+\beta = |V|$.
- Maximum Independent Set und Größte Clique sind (polynomiell) äquivalent.
- Vertex Cover lässt sich mit der Formel berechnen (Minimale Anzahl der Knoten).

2.26 Travelling Salesman Problem (NP-vollständig)

- **Brute-Force:** $(n-1)!$ viele Touren, eigentlich $\frac{1}{2}(n-1)!$, weil Umkehrungen der Touren weggelassen werden können (Symmetrie).
- **DP:** $L(i, A)$: kürzester Weg von i zu 1, der jede Stadt $a \in A$ genau einmal besucht.
- Trivialfälle: $L(i, \emptyset) = DIST[i, 1]$.
- $L(i, A) = \min\{L(j, A - j) + DIST[i, j] | j \in A\}$.

- $L(i, \{2, 3, \ldots, n\}$ ist die Länge der kürzesten TSP-Tour.
- Tabellengröße $n \cdot 2^n$, Laufzeit $\mathcal{O}(n^2 \cdot 2^n)$.
- **MST-Heuristik:** Jede Tour ist ein Hamilton-Zyklus (Achtung: Jeder TSP-Graph ist vollständig, also keine polynomielle Reduktion). Durch Weglassen einer Kante entsteht ein Hamilton-Pfad, also ein Spanning Tree. Im schlimmsten Fall ist diese Kante so lang wie der Rest der Kanten zusammen (metrisches TSP). Ein MST stellt also eine TSP-Tour dar, die maximal zweimal so lang wie das Optimum ist. Füge Abkürzungen ein, sodass jeder Knoten nur einmal besucht wird (Teste ob $a \to b \to c$ durch $a \to c$ abgekürzt werden kann, ohne dass b isoliert wird).
- **Nearest-Neighbor-Eröffnungverfahren:** Wähle vom aktuellen Knoten aus den nächsten (minimale Distanz) nicht besuchten Knoten bzw. den Ausgangspunkt, wenn kein solcher verfügbar ist. Vgl. auch Farthest-Neighbor. Polynomielle Laufzeit (quadratisch in $|V|$, beliebig schlecht.
- **Simulierte Abkühlung:** Lokale Operationen: z.B. Vertauschen zweier Städte, eine Stadt an anderer Stelle einfügen, Teilstück in umgekehrter Reihenfolge durchlaufen (z.B. $a \to b$ und $c \to d$ wird zu $a \to d$ und $c \to b$).
- **Lösung mit Metaheuristiken oder lokaler Suche.**

2.27 Graphenisomorphie

- Zeige, dass zwei Graphen gleich sind, also eine bijektive Abbildung $\alpha\colon V_1 \to V_2$ existiert, sodass $(u, v) \in E_1 \Leftrightarrow (\alpha(u), \alpha(v)) \in E_2$.
- Liegt vermutlich nicht in P, NP-Vollständigkeit wurde aber auch noch nicht gezeigt.

3 Datenstrukturen

3.1 Primitive Datenstrukturen

- **Stapel (Stack):** push/pop immer oben. Implementation z.B. mit verketteter Liste (Heap, dynamisch allokiert) oder Array.
- **Schlange (Queue):** push_back hinten, pop vorne. Implementation z.B. mit verketteter Liste oder Ringpuffer fester Größe (Array mit Zeiger auf Anfang und Ende des Datenbereichs).
- **Schlange mit zwei Enden (Deque):** push/pop vorne und hinten. Implementation z.B. mit doppelt verketteter Liste oder Ringpuffer.
- **Prioritätswarteschlange (Priority queue):** Implementation z.B. mit Binärheap.
- **Weitere Datenstrukturen:** Bäume, balancierte Bäume, Hashtabellen.

3.2 Heap

- Max-Heap/Min-Heap mit max./min. Element ganz oben.
- Verwendung in Priority Queue.
- Nachfolger eines Elements sind kleiner (gleich) dem aktuellen Element.
- **Einfügen:** ganz unten und UpHeap in $\mathcal{O}(\log n)$.
- **Entfernen des Maximums/Minimums:** Nehme unterstes Element und ersetze es mit der Wurzel, dann DownHeap in logarithmischer Zeit.
- **Priorität ändern:** UpHeap und DownHeap.
- Bei fester maximaler Größe verwende Array.

3.3 Binärer Suchbaum (unballanciert)

- Element links ist kleiner/gleich dem aktuellen Element, analog für rechtes Element.
- **Einfügen:** Suche Element und füge es an dieser Stelle ein, wenn nicht gefunden. Laufzeit $\mathcal{O}(h = n)$.
- Baum kann zur Liste entarten.
- **Entfernen:** Ersetze Element durch kleinstes Element im rechten Teilbaum (ganz links unten).

3.4 AVL-Baum (balancierter Suchbaum)

- Für jeden Knoten gilt: beide Teilbäume unterscheiden sich in der Höhe höchstens um 1.
- Höhe des gesamten Baums liegt garantiert bei $\mathcal{O}(logn)$.
- Speichere Balance-Werte für jeden: Höhenunterschied der Teilbäume.
- Aktualisiere bei Einfügen und Löschen die Balance-Werte rekursiv (Abbruchbedingung: Einfügen: wenn Balance-Wert zu 0 wird, Löschen: wenn Balancewert zu plus/minus 1 wird. Höhe des gesamten Teilbaums ändert sich dadruch nicht.)
- Rebalancierung: Rotiere den Baum entweder durch Einfachrotation oder durch Doppelrotation.

3.5 Rot-Schwarz-Baum (balancierter Suchbaum)

- Jeder Knoten ist entweder rot oder schwarz.
- Die Wurzel ist schwarz.
- Blattknoten (NIL, am Ende) sind schwarz.
- Ist ein Knoten rot, so sind seine direkten Nachfolger schwarz.
- Jeder Pfad von irgendeinem Knoten zu seinen Blattknoten hat die gleiche Anzahl an schwarzen Knoten.
- Von der Wurzel: längster Pfad zu einem Blatt ist höchstens doppelt so hoch wie kürzester Pfad (annähernd balanciert).
- Einfügen und Löschen: betrachte viele verschiedene Fälle und färbe Knoten um und rotiere Teilbäume nach festen Regeln.

3.6 2-3-4-Baum (balancierter Suchbaum)

- Knoten haben 2, 3 oder 4 Nachfolger, also bis zu 3 Elemente.
- Wenn ein Element hinzugefügt wird: Fülle Knoten auf, bis er 3 Elemente enthält. Wenn er 4 Elemente enthält: teile ihn in 2 Knoten auf und reiche ein Element nach oben weiter. Wenn der Elternknoten voll ist, reiche weiter, bis an der Wurzel. Wenn nötig die Wurzel aufteilen.
- Beim Löschen evtl. mehrere Knoten zusammenlegen.
- 2-3-4-Bäume werden durch Rot-Schwarz-Bäume implementiert.

4 Algorithmen

4.1 Insertion-Sort

- In-place Sortierverfahren mit Laufzeit $\mathcal{O}(n^2)$ bei n Elementen.
- Der Bereich (Indizes) $[0; i-1]$ sei bereits sortiert. Füge jetzt Element mit Index i an der richtigen Stelle ein. Vertausche dazu das Element so lange mit seinem Vorgänger, bis es keinen größeren Vorgänger mehr gibt.
- Für Worst-Case-Laufzeitbetrachtung beachte allgemein immer Spezialfälle (z.B. Liste bereits sortiert oder rückwärts sortiert).

4.2 Selection-Sort

- In-place Sortierverfahren mit Laufzeit $\mathcal{O}(n^2)$.
- Der Bereich $[0; i-1]$ sei bereits sortiert. Suche aus dem Bereich $[i; n-1]$ das nächstgrößere Element heraus und vertausche es mit dem Element auf Platz i.

4.3 Merge-Sort

- Sortieren durch Mischen, kein in-place Sortierverfahren, Laufzeit $\mathcal{O}(n \log n)$.
- Teile-und-Herrsche (divide-and-conquer) Verfahren: Teile die Elemente (das Array) in zwei gleich große Bereiche auf, wende Merge-Sort auf beide Teile an. Wenn beide rekursiven Aufrufe fertig sind, mische die Arrayhälften zusammen, d.h. wähle jeweils das kleinere Element aus beiden Arrayhälfen (dieses Element befindet sich am Anfang, da bereits sortiert) und füge es am Ende einer Liste ein.

4.4 Quick-Sort

1. Wähle das Pivotelement zufällig (randomisierter Quick-Sort-Algorithmus) und setze es an den rechten Rand.
2. partition-Funktion teilt das Array in linearer Zeit in zwei Teile, wobei die Elemente im linken Teil kleiner sind (Zwei Zeiger auf die zu vergleichenden Elemente). Setze das Pivotelement in die richtige Arrayhälfte.
3. Beide Subarrays mit Quick-Sort sortieren.

- Worst-Case-Laufzeit $\mathcal{O}(n^2)$, bei Randomisierung $\mathcal{O}(n \log n)$ erwartet.
- Kleine Subarrays evtl. mit einfacherem Verfahren sortieren.

4.5 Primzahlen: Sieb des Eratosthenes

1. Schreibe alle Zahlen von 1 bis n auf.
2. Streiche 1 durch und unterstreiche 2.
3. Streiche alle Vielfachen der letzten unterstrichenen Zahl durch.
4. Unterstreiche die nächste freie Zahl (bis $\sqrt{n}$).

4.6 Greedy-Algorithmus zur Stammbruchdarstellung

- Finde eine Stammbruchdarstellung $\frac{1}{c_1} + \frac{1}{c_2} + \ldots + \frac{1}{c_n}$ eines Bruches $\frac{a}{b}$.

1. Finde den größten Stammbruch $\frac{1}{c}$ kleiner/gleich $\frac{a}{b}$.
2. Bilde die Differenz $\frac{a_1}{b_1} = \frac{a}{b} - \frac{1}{c}$.
3. Verfahre mit $\frac{a_1}{b_1}$ wie mit $\frac{a}{b}$.
4. Verfahren ist zu Ende, wenn sich in Schritt 2 eine Null ergibt.

- Verfahren ist nicht optimal, d.h. es wird nicht immer die kürzeste Stammbruchdarstellung gefunden.
- Verfahren terminiert immer.

4.7 Rekursion: Türme von Hanoi

1. Lege die obersten $n-1$ Scheiben mit dem Zielstab auf den Hilfsstab (rekursiv mit dem aktellen Algorithmus).
2. Lege die unterste Scheibe direkt auf den Zielstab.
3. Lege die restlichen $n-1$ Scheiben mit dem Anfangsstab auf den Zielstab (rekursiv).

- Es werden immer genau $2^n - 1$ Züge benötigt. Der Algorithmus ist optimal.

4.8 Backtracking: Das n-Damen-Problem

- $n \times n$ Schachbrett mit n Damen, pro Spalte und Zeile eine Dame
- Setze in jedem Zug eine Dame pro Spalte, in einer Zeile, sodass es keine Kollisionen gibt.
- Wenn es (zwangsläufig) zu Kollisionen kommt, gehe so weit zurück wie notwendig und probiere die nächste Platzierungsmöglichkeit für die Dame aus.
- Brute-Force-Lösung mit Backtracking: Breche ab, wenn klar ist, dass die aktuelle Teillösung nicht funktionieren kann.

4.9 Schnelle Multiplikation: Karazuba

- Schulbuchmethode: $2n^2$ elementare Multiplikationen.
- $a \cdot b = (10p + q)(10r + s) = 100pr + 10(ps + qr) + qs$ bei $n = 2$.
- $u = pr$, $v = (q - p)(s - r)$, $w = qs$.
- Karazuba: $u + w - v = pr + qs - (q - p)(s - r) = ps + qr$, also $a \cdot b = 100u + 10(u + w - v) + w$.
- Nur noch 3 Multiplikationen statt 4 notwendig und ein paar Additionen.
- Für $n = 4$: $a = 100p + q$, $b = 100r + s$, wobei p, q, r, s vierstellig sind.
- Verfahren funktioniert für $n = 2^k$, ansonsten mit Nullen auffüllen.

4.10 Schnelles Potenzieren (nicht-negativ)

- Bei geradzahligen Exponenten gilt: $a^n = (a^2)^{\frac{n}{2}}$.
- $a^n = a \cdot a^{n-1}$ falls n ungerade.
- Obere Schranke für Anzahl der Multiplikationen: $2\lceil \log_2(n + 1) \rceil$.
- **Beispiel** 2^10^10**:** Naiver Ansatz $10^10 - 1$, schneller Ansatz 44 Multiplikationen.

4.11 Polynomauswertung: Horner-Schema

- Auswertung von Polynomen an der Stelle x: $p(x)$.
- $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$.
- Anzahl der Multiplikationen bei naiver Methode: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- Horner-Schema: $p(x) = (((\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_3)x + a_2)x + a_1)x + a_0$.
- Anzahl der Multiplikationen: n.

4.12 Kompression: Huffman-Codierung

- Schreibe alle Zeichen Σ mit ihrer Wahrscheinlichkeit in einer Liste auf.
- Fasse 2 Knoten mit niedrigster Wahrscheinlichkeit zusammen zu einem Knoten, bis nur noch ein einziger Baum vorhanden ist.
- Linke Teilbäume mit 0 kodieren, rechte Teilbäume mit 1 (oder andersherum).
- Greedy-Algorithmus ist immer optimal.
- Entropiekodierung: Jedem Zeichen wird eine unterschiedlich lange Folge an Bits zugeordnet.

4.13 Kompression: Lempel-Ziv-77 (LZ77)

- Ganze Wörter und Wordteile komprimieren.
- Vorschau-Puffer und Textfeld (schon komprimierter Text) konstanter Größe.
- Komprimierte Daten als Tripel (a, b, c), wobei a die Position des Lexikon-Eintrags, b die Länge der zu komprimierenden Daten und c das nachfolgende Zeichen sind.
- **Beispiel: ANANAS**

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**:**	**0**	**1**	**2**	**3**	**Tripel**
								:	A	N	A	N	$(0, 0, A)$
							A	:	N	A	N	A	$(0, 0, N)$
						A	N	:	A	N	A	S	$(6, 2, A)$
			A	N	A	N	A	:	S				$(0, 0, S)$

- Kompressionsgüte hängt von der Größe des Lexikons ab, Laufzeit steigt dann aber auch an!
- Huffman und Lempel-Ziv sind Grundlage für Deflate (siehe auch PNG).
- Verbesserung mit dem Lempel-Ziv-Storer-Szymanski (LZSS): Wenn ein Lexikonverweis (Tripel) länger als die zu ersetzende Zeichenfolge ist, füge stattdessen die Zeichenfolge ein (es wird ein Bit/Flag zur Unterscheidung benötigt).

4.14 Kompression: Lempel-Ziv-Welch (LZW)

- Wörterbuch mit 12 Bit langen Indizes, also $2^{12} = 4096$ Wörtern.
- Indizes 0-255: ASCII-Zeichen (Bytes).
- Wort (Muster): Anderes Wort (Präfix aus Wörterbuch) + Zeichen (Suffix).
- **Beispiel:** LZWLZ78LZ77.
- Finde L, Ausgabe L, LZ $\rightarrow$ (256).
- Finde Z, Ausgabe Z, ZW $\rightarrow$ (257).
- Finde W, Ausgabe W, WL $\rightarrow$ (258).
- Finde LZ, Ausgabe (256), LZ7 $\rightarrow$ (259) bzw. (256) 7 $\rightarrow$ (259).
- Finde 7, Ausgabe 7, 78 $\rightarrow$ (260), usw.
- Zum Schluss: LZW (256) 78 (259).....
- Beim Dekomprimieren wird das Wörterbuch automatisch miterzeugt.
- Patentprobleme (siehe GIF).
- LZC-Algorithmus: Variable Wörterbuchgröße, 7 Prozent bessere Kompression.

4.15 GGT: Euklidischer Algorithmus

- **Beobachtung:** Bei $a > b$ kann der das Stück $a - b$ genauso *aufgeteilt* werden wie a. Beachte auch Spezialfall *Vielfache*.
- **Einfach:** Zwei Zahlen $a > b$: Bestimme den GGT von $a-b$ und b: Zerteile so lange, bis $a = b$. Laufzeit linear im Zahlenwert, da die a in jedem Schritt um mindestens 1 verringert wird.
- **Schneller:** Nimm den Rest der ganzzahligen Division $a \div b$. Laufzeit linear in der Anzahl der Ziffern, logarithmisch im Zahlenwert.

4.16 Geometrische Algorithmen

4.16.1 Counter-Clockwise

- 3 Punkte P_0, P_1, P_2.
- Bewegung $P_0 \to P_1 \to P_2$ gegen den Uhrzeigersinn: return 1.
- Vergleiche Steigungen der Geraden P_0P_1 und P_0P_2. $P_0P_2 > P_0P_1$, dann return 1.
- Kollinear (Gerade): return 0.
- Für konvexe Polygone: Enthaltenseit eines Punktes *ccw(A, B, P) == ccw(B, C, P) == ccw(C, A, P)*.
- *Schneiden sich 2 Geraden?* mit ccw.

4.16.2 Pick's Theorem

- **Voraussetzungen:** Polygon, dessen Ecken alle ganzzahlig sind.
- A: Fläche, I: Gitterpunkte im Inneren des Polygons, R: Anzahl der Gitterpunkte auf dem Rand des Polygons.
- $A = I + \frac{R}{2} - 1$.

4.16.3 Einfacher geschlossener Pfad

- Gegeben eine Menge von Punkten, finde einen Weg (Zyklus), sodass sich der Weg nicht überschneidet.
- Beginne bei einem beliebigen Punkt A.
- Berechne für alle Punkte den Winkel zwischen der Waagrechten, A und dem aktuellen Punkt.
- Sortiere die Punkte nach dem Winkel.
- Verbinde die Punkte in dieser Reihenfolge.

4.16.4 Enthaltensein in einem Polygon

- Lege eine beliebige Gerade durch den Punkt und zähle Anzahl der Schnitte in eine Richtung.
- Wenn es eine gerade Anzahl ist, liegt der Punkt außerhalb, ansonsten liegt er im Polygon.
- Funktioniert nicht, wenn die Gerade durch eine Ecke geht oder genau auf einer Seite verläuft (neue Gerade legen).

4.16.5 Konvexe Hülle

- Naiver Algorithmus: Einwickeln.
- Beginne bei beliebigem Punkt, der garantiert zur konvexen Hülle gehören muss, z.B. dem ganz unten (wenn mehrere, dann ganz unten links).
- Schwenke eine Gerade durch diesen Punkt so lange, bis sie auf einen anderen Punkt trifft. Mache an diesem Punkt weiter (Richtung beibehalten).
- Laufzeit $\mathcal{O}(n^2)$.

4.16.6 Konvexe Hülle mit Graham-Scan

- Verbinde Punkt zu einem einfachen geschlossenen Pfad ($\mathcal{O}(nlogn)$).
- Beginne bei einem Punkt, der garantiert zur konvexen Hülle gehören muss.
- Füge immer den nächsten Punkt zur konvexen Hülle hinzu. Wenn man dabei nach links bzw. nach rechts gehen muss, entferne den vorherigen Punkt (evtl. sogar mehrere).
- Laufzeit für diesen Teil linear.
- Innere Elimination: Finde zum Beispiel 4 Punkte, die auf der konvexen Hülle liegen müssen. Alle Punkte innerhalb dieses Vierecks kann man streichen. Danach normalen Algorithmus starten,

4.16.7 Scanline: Geometrischer Schnitt (Manhattan)

- Sortiere Punkte nach y-Koordinaten.
- Horizontale Durchsuchungslinie verläuft von unten nach oben: horizontale Linien sind Intervalle, vertikale Linien sind Punkte.
- Wenn eine vertikale Linie beginnt, füge x-Koordinate zu binärem Suchbaum hinzu, wenn sie endet: lösche den Knoten.

- Bei horizontaler Linie: führe Bereichssuche (Start- und Endpunkt des Intervalls) durch.
- Laufzeit $\mathcal{O}(nlogn + I)$, wenn I die Anzahl der Schnitte (wg. Bereichssuche) ist.

4.16.8 Problem des nächsten Paares

- Sortiere Knoten nach x-Koordinate.
- Teile in Hälften: entweder ist das Paar links, rechts (rekursiv) oder ein Punkt links und einer rechts.
- Punkt links und rechts: *min* ist Abstand des kleinsten Paares in der linken und rechten Hälfte.
- Betrachte von nun an nur Paare, die maximal um *min* von der Trennlinie entfernt sind.
- Sortiere übrige Punkte nach y-Koordinate. Gehe jeden Punkt durch und berechne Abstand zu Punkten, deren Abstand maximal *min* ist (Sortierte Liste).
- Sortieren nach der y-Koordinate ist mit Mergesort effizient, da es die gleiche Divide-and-Conquer Struktur wie das aktuelle Problem aufweist. In einem Schritt sortieren und minimales Paar finden.
- Laufzeit wie bei Mergesort $\mathcal{O}(nlogn)$.

4.17 Kreise zeichnen: Bresenham-Algorithmus

- Symmetrien ausnutzen: Es muss nur $\frac{1}{8}$ berechnet werden, die Koordinaten ergeben sich: $(x, y), (y, x), (-y, x), (x, -y), (-x, -y), (-y, -x), (y, -x), (-x, y)$.
- Im $\frac{1}{8}$ Kreissegment (rechts oben): entweder gehe nach rechts oder nach rechts und nach unten, weil Steigung immer zwische 0 und -1.
- Schaue, ob der Punkt noch im Kreis ist. $F(x, y) = x^2 + y^2 - R^2$, betrachte dabei nicht (x, y), sondern $(x + 1, y - \frac{1}{2})$ (unteres Ende des Pixels noch im Kreis).
- Weitere Optimierungen: Berechnung von F: Keine Potenzen (Quadrat), keine Multiplikationen, nur Additionen mit iterativem Verfahren.

4.18 Gauß-Seidel-Iteration: LGS näherungsweise lösen

- Näherungsweise die Lösung eines linearen Gleichungssystems lösen.
- Gaußsches Eliminationsverfahren ist anfällig für Rundungsfehler.
- $Ax = b$

- Für jede k-te Gleichung und Variable: $x_k^{(m+1)} = \frac{1}{a_{k,k}}(b_k - \sum_{i=1}^{k-1} a_{k,i} x_i^{(m+1)} - \sum_{i=k+1}^{n} a_{k,i} x_i^{(m)})$, im $m+1$ Iterationsschritt.
- Bei unendlich vielen Iterationen ergibt sich eine exakte Lösung.
- Verfahren konvergiert sicher bei Diagonaldominanz ($|a_{i,i}| \geq \sum_{j \neq i} |a_{i,j}|$, für alle i).
- Vgl. Beispiel Fussballspieler in einer Reihe, Anwendung bei physikalischen Simulationen, Wärmeverteilung, PageRank.

4.19 DP: Levenshtein-Distanz

- Transformiere Zeichenfolge $x \to y$ mit den Operationen *Einfügen*, *Löschen*, *Ersetzen* und den entsprechenden Kosten 2, 2, 3 pro Operation.
- Mache $x^{(m)} \to y^{(n-1)}$ und hänge letztes Zeichen $y^{(n)}$ an.
- Mache $x^{(m-1)} \to y^{(n)}$ und streiche letztes Zeichen $x^{(m)}$.
- Mache $x^{(m-1)} \to y^{(n-1)}$ und ersetze letztes Zeichen $x^{(m)}$ durch $y^{(n)}$.
- Operationen beeinflussen sich nicht gegenseitig.
- Baue $(m+1 \times (n+1)$-Matrix auf, in $\mathcal{O}(mn)$.

4.20 Bin-Packing (Online-Algorithmus)

- Eine (anfangs unbekannte) Folge $G_1, G_2, \ldots, G_n$ von Gegenständen soll ihn möglichst wenige Kisten der Größe V verpackt werden.
- **Next-Fit:** Wenn G_i in Kiste j nicht Platz hat, erstelle Kiste $j+1$.
- **First-Fit:** Stecke G_i in die erste Kiste j, die noch genügend Platz hat.
- Beide nicht optimal. First-Fit: 1,7-kompetitiv, bester denkbarer Algorithmus ist $\frac{4}{3}$-kompetitiv.

4.21 Bin-Packing (Offline-Algorithmus)

- NP-vollständiges Entscheidungsproblem: Können die n Objekte so auf die k Behälter der Größe b verteilt werden, dass keiner überläuft?
- $\frac{3}{2}$-Approximationsalgorithmus: First-Fit-Decreasing.

1. Sortiere Gegenstände nach absteigendem Gewicht.
2. Füge Objekte der Reihe nach ein, sodass ein Objekt in den ersten Behälter gelegt wird, der noch Platz hat.

4.22 Das Rucksackproblem (0-1-Knapsack)

- Eines von Karp's 21 NP-vollständigen Problemen.
- Pseudopolynomielle Lösung mit DP: T: Gewichtsschranke, n: Anzahl der Objekte.
- $R[T][n] = \max\{R[T][n-1]; R[T - Gewicht(n)][n-1] + Wert(n)\}$.
- Entscheidung: entweder Gegenstand nicht nehmen oder Gegenstand nehmen.
- Problem darstellbar als Integer-LOP.
- Lösung mit Branch-and-bound: Bilde LP-Relaxation (Problem ohne Ganzzahligkeitsbedingung), bilde Quotient Wert pro Gewicht und nehme die besten Objekte so weit wie möglich ganz, das nächste anteilig. Verzweige: Nimm das Objekt oder nimm es nicht (Hinweis zu Branch-and-bound: Obere und untere Schranke, wann muss ein Problem nicht weiter ausgelotet werden?).

4.23 Algorithmus für das Erfüllbarkeitsproblem (SATISFIABILITY)

- Gegeben eine aussagenlogische Formel in konjunktiver Normalform (KNF), finde eine gültige Belegung, bzw. gibt es eine Belegung?
- Brute-Force: 2^n Möglichkeiten.
- **Davis-Putnam-Verfahren:** PROCEDURE SPLIT(E) wird auf Formel in KNF angewendet.
- Suche nächste freie Variable und splitte: $x_i = 0$ und $x_i = 1$.
- Alle Klauseln mit $\neg x_i$ bzw. x_i werden gelöscht, da erfüllt. Literale mit x_i bzw. $\neg x_i$ werden gelöscht, da die Klausel dadurch nicht erfüllt wird.
- Wenn eine leere Klausel () auftritt, höre auf (nicht mehr erfüllbar).
- Wenn keine Klauseln mehr übrig sind, ist die Formel erfüllt.
- Im Worst-Case exponentieller Zeitbedarf.

4.24 Maximum Subarray Problem

- **Problem:** Finde das Subarray, das - wenn man die Einzelwerte aufsummiert - die größte Summe hat.
- Teile das Array in zwei gleich große Hälften *links* und *rechts*.
- Entweder ist das maximale Subarray im linken bzw. rechten Teil (Rekursion!) oder es überschreitet die Grenze.

- Überschreitendes Subarray: kann in linearer Zeit gefunden werden: gehe von der Mitte bis zur Grenze des linken bzw. rechten Teils und speichere Maximalwert.
- Laufzeit $\mathcal{O}(n \log n)$.

4.25 Counting-Sort: Sortieren durch Zählen

- Sortieren in linearer Zeit.
- Es sollen Zahlen von 1 bis k sortiert werden (k bekannt).
- Erzeuge int-Array B mit k Slots.
- Wenn Zahl i gefunden wird, erhöhe $B[i]$ um eins.
- Verändere das Zählarray B: bis zu welchem Index sind die (vorderen) Zahlen kleiner/gleich dieser Zahl.
- Gehe alle n Elemente $A[i]$ rückwärts durch und schreibe in das neue Array an die Stelle $B[A[i]]$ die Zahl $A[i]$. Verringere $B[A[i]]$ dann um eins.
- Speicherplatzbedarf hängt von k ab.

4.26 Radix-Sort: Sortieren durch Zählen

- Basiert auf Counting-Sort.
- Sortiere zuerst nach der Stelle ganz rechts, dann immer weiter nach links.
- Funktioniert, weil Counting-Sort ein stabiles Sortierverfahren ist.
- Speicherplatzbedarf ist geringer als bei Counting-Sort.
- Lineare Laufzeit, wenn die maximale Länge der Zahlen (Ziffern) von Anfang an feststeht.

4.27 Längste gemeinsame Sequenz (DP)

- Betrachte 2 Strings, was ist die längste Subsequenz, wenn 0 oder mehr Zeichen ausgelassen werden dürfen.
- Zum Beispiel: ABC ist eine Subsequenz von BAAABXXCA.
- Lösung mit DP: $c[i, j]$: Länge der längsten Sequenz mit Strings 1 i und String 2 j Zeichen lang.
- $c[i, j] = 0$, wenn $i = 0$ oder $j = 0$.
- $c[i, j] = c[i - 1, j - 1] + 1$, wenn $a_i = b_j$.

- $c[i, j] = \max\{c[i-1, j], c[i, j-1]\}$, sonst.
- Baue Tabelle zeilenweise auf.

4.28 Activity-Selection-Problem

- Gegeben eine Liste mit Aktivitäten mit Start- und Endzeitpunkt. Finde möglichst viele kompatible Aktivitäten.
- Sortiere Aktivitäten aufsteigend nach Endzeitpunkt.
- Gehe alle Aktivitäten durch, wenn Startzeitpunkt größer/gleich de, letzten (hinzugefügten) Endzeitpunkt ist, füge die Aktivität hinzu (Greedy-Algorithmus).
- Erste Aktivität ist immer dabei.
- Laufzeit wegen Sortierung bei $\mathcal{O}(n \log n)$.

4.29 Der Simplex-Algorithmus

- Mache Ungleichungen zu Gleichungen (Schlupfvariablen).
- Baue das Simplex-Tableau aus den Gleichungen auf (inkl. Zielfunktion).
- Gehe durch geeignete Umformungen von Basislösung zu Basislösung.
- Pivotspalte: Spalte mit kleinstem Eintrag in der Zielfunktion.
- Bilde Quotienten aus Konstanten ganz rechts und dem entsprechenden Eintrag in der Pivotspalte, für alle Zeilen.
- Wähle Zeile mit dem kleinsten nicht-negativen Quotienten.
- Dividiere Pivotzeile durch das Pivotelement.
- Addiere geeignete Vielfache der Pivotzeile zu jeder anderen Zeile, sodass die Pivotspalte überall Null wird (außer aktuelle Zeile).
- Wenn alle Werte in der Zielfunktion positiv sind, fertig (?)
- Nicht-Basisvariablen gleich Null setzen, die anderen ergeben sich aus dem Tableau.

4.30 Das Partition-Problem (NP-vollständig)

- **Problem:** Gibt es eine Teilmenge von X, sodass sich die Elemente zu j aufsummieren?
- Pseudopolynomielle Lösung (DP): Baue Tabelle (i, j) auf, wobei $(i, j) = 1$ genau dann wenn es eine Auswahl unter den ersten i Elementen gibt, die sich zu j aufsummieren.
- Betrachte dazu $(i-1, j)$ und $(i-1, j-x_i)$.
- Erster Fall: Schon vorher ergibt sich eine Teilmenge mit j.
- Zweiter Fall: Vorher ergibt sich eine Teilmenge mit $j-x_i$ und das aktuelle Element hat x_i.

5 Theoretische Informatik

5.1 Landau-Notation für asymptotisches Wachstum

- $f \in \mathcal{O}(g)$**:** Wachstum von f ist kleiner oder gleich dem Wachstum von g.
- $f \in \mathcal{O}(g) \Leftrightarrow \exists c > 0.\exists x_0.\forall x > x_0 : |f(x)| \leq c \cdot |g(x)|$**:** Wachstum ist beschränkt durch g.
- $f \in o(g) \Leftrightarrow \exists c > 0.\exists x_0.\forall x > x_0 : |f(x)| < c \cdot |g(x)|$**:** Wachstum ist kleiner als g.
- $f \in \Omega(g) \Leftrightarrow \exists c > 0.\exists x_0.\forall x > x_0 : |f(x)| \geq c \cdot |g(x)|$**:** Wachstum ist mindestens g.
- $f \in \omega(g) \Leftrightarrow \exists c > 0.\exists x_0.\forall x > x_0 : |f(x)| > c \cdot |g(x)|$**:** Wachstum ist größer als g.
- $f \in \Theta(g) \Leftrightarrow f \in \mathcal{O}(g) \wedge f \in \Omega(g)$**:** Wachstum entspricht g.

5.2 Endlicher Automat

- Darstellbar als 5-Tupel $(Q, \Sigma, Q_0, F, \delta)$, wobei Q die Zustandsmenge, Σ das Eingabealphabet, Q_0 der Startzustand, F die Menge der Endzustände und δ die Übergangsfunktion sind.
- Endliche Automaten akzeptieren die regulären Ausdrücke.
- Bei DFA: $\delta : Q \times \Sigma \to Q$.
- Bei NFA: $\delta : Q^* \times \Sigma \to Q^*$.
- Umwandlung von NFA zu DFA mit Potenzmengenkonstruktuion (evtl. exponentieller Aufblaseffekt).
- Anwendung von NFAs: Pattern Matching.

- ϵ-NFA: spontane Zustandsübergänge möglich.
- ϵ-NFA zu DFA: Potenzmengenkonstruktion mit ϵ-Hülle für jeden Zustand: die Menge der Zustände, die mit ϵ erreichbar sind.
- Umwandlung Regulärer Ausdruck zu Endlicher Automat: Stückweise Ersetzung, mit möglicherweise vielen ϵ-Übergängen (schreibe auf die Übergangspfeile komplexe Ausdrücke und zerstückle diese dann weiter).

5.3 Endliche Automaten mit Ausgabe

- **Moore-Modell:** Ausgabe hängt nur vom aktuellen Zustand q ab.
- **Mealy-Modell:** Es gibt eine Übergangsfunktion, die jeder Zustand-Eingabe-Kombination ein Ausgabesymbol zuweist: $Q \times \Sigma \to \Omega$.

5.4 Minimierung von DFAs

1. Erstelle Tabelle mit allen Zustandspaaren.
2. Markiere die Paare (x, y), bei denen $x \in F \wedge y \notin F$ (unterscheidbare Zustände).
3. Für jedes unmarkierte Paar (x, y) und jedes $a \in \Sigma$ teste, ob $(\delta(x, a), \delta(y, a))$ markiert ist: wenn ja, markiere auch (x, y).
4. Wiederhole Schritt 3, bis sich nichts mehr ändert.
5. Nicht-markierte Paare können kollabiert werden.

5.5 Nicht-deterministischer Kellerautomat (NPDA)

- $M = (Q, \Sigma, F, \Gamma, q_0, Z_0, \delta)$, wobei Q die Zustandsmenge, Σ das Eingabealphabet, F die Menge der Endzustände, Γ das Kelleralphabet, q_0 der Startzustand, Z_0 das Anfangssymbol im Keller und δ die Übergangsfunktion sind.
- (q, w, γ) ist eine Konfiguration des Kellerautomaten, wobei q der aktuelle Zustand, w das restliche Eingabewort und γ der Kellerinhalt sind.
- Wortakzeptanz bei Finalzustand $\Leftrightarrow (q_0, w, Z_0) \to_M^* (f, \epsilon, \gamma)$ mit $f \in F, \gamma \in \Gamma^*$.
- Wortakzeptanz bei leerem Keller $\Leftrightarrow (q_0, w, Z_0) \to_M^* (q, \epsilon, \epsilon)$ mit $q \in Q$.
- **Überlegung:** Konstruiere NPDA für die Sprache $\{ww^R\}$, wobei w^R *w rückwärts* ist.

5.6 Grammatiken

- Grammatiken und Automaten sind endliche Darstellungsformen der i.A. unendlichen Sprachen.
- $G = (V, \Sigma, P, S)$, wobei V die Menge der Variablen (Nicht-Terminalzeichen), Σ die Menge der Terminalzeichen, P die Menge der Produktionen und S die Startvariable sind, mit $P \subseteq ((V \cup \Sigma)^+ \times (V \cup \Sigma)^*)$.
- $x \to_G y$ genau dann wenn es in G eine Ableitung von x nach y gibt (entsprechend $x \to_G^* y$).

5.7 Die Chomsky-Hierarchie

- **Typ-0-Grammatiken:** keine Einschränkungen, Phrasenstrukturgrammatik, wird von Turingmaschinen akzeptiert.
- **Typ-1-Grammatiken:** Typ-0-Grammatik mit $|w_1| \leq |w_2|$ für alle w_1, w_2 mit $w_1 \to w_2$, kontextsensitive Grammatik, wird von linear beschränkten Automaten akzeptiert.
- **Typ-2-Grammatiken:** Typ-1-Grammatik mit $w_1 \in V$ falls $w_1 \to w_2$, kontextfreie Grammatik, wird von Kellerautomaten akzeptiert.
- **Typ-3-Grammatiken:** Typ-2-Grammatik mit $w_2 \in (\Sigma \cup \Sigma V)$, reguläre Grammatik, wird von endlichen Automaten akzeptiert.

5.8 Das Wortproblem

- Für Typ-0-Sprachen nicht entscheidbar (vgl. Halteproblem Turingmaschine).
- Für Typ-1-Sprachen NP-hart.
- Für Typ-2-Sprachen mit CYK-Algorithmus in kubischer Zeit entscheidbar.
- Für Typ-3-Sprachen in linearer Zeit entscheidbar.

5.9 Entscheidungsprobleme vs. Optimierungsprobleme

- **Entscheidungsprobleme:** Ja/Nein, z.B. *Gibt es einen hamiltonschen Weg in G?*.
- **Optimierungsproblem:** Ausgabe einer optimalen Lösung, z.B. *Welche TSP-Tour ist in G optimal?*.
- Optimierungsprobleme können in Entscheidungsprobleme umgewandelt werden, z.B. *Gibt es eine Rundreise, die höchstens 1000 Einheiten kostet?*.
- Optimierungsprobleme sind mindestens so schwer wie Optimierungsprobleme.

- Für Komplexitätsanalysen, Theoretische Informatik, NP-Theorie sind Entscheidungsprobleme interessant.

5.10 Komplexitätsklassen P und NP

- **P:** Lösungsalgorithmus in polynomieller Laufzeit.
- **NP:** Verifikation eines Lösungsvorschlages in polynomieller Zeit (oder: kann von einer nicht-deterministischen Turingmaschine in Polynomialzeit gelöst werden).
- Offensichtlich gilt $P \subseteq NP$, allerdings ist $P = NP$ unbekannt.
- **Satz von Cook:** SAT ist NP-vollständig (NP-hart und selbst in NP). Wenn dieses Problem in P liegt, dann **alle** Probleme in NP.
- NP-Härte: Ein Problem ist NP-hart (NP-schwer), wenn jedes andere Problem aus NP in polynomieller Zeit auf dieses Problem reduziert werden kann. Praktisches Verfahren: Reduziere ein bekanntes NP-vollständiges Problem auf das aktuelle Problem (Transitivität der Reduktion).
- Es gibt auch NP-harte Probleme, die nicht in NP liegen, z.B. Halteproblem für Turingmaschinen (da nicht entscheidbar).
- Funktioniert nur für Entscheidungsprobleme.

5.11 Einige NP-vollständige Probleme

- Hamiltonscher Weg: Jeder Knoten soll in einem Graph genau einmal besucht werden.
- TSP (Traveling Salesman Problem): Finde Tour mit minimalen Kosten.
- Stundenplanproblem
- SAT (Satisfiability)
- Knapsack: Finde Gegenstände mit maximalem Wert und Gewicht unter w (pseudopolynomieller Algorithmus mit DP).
- bin packing (Kastenproblem): Gegeben Kästen der Kapazität L, wie viele werden für bestimmte Gegenstände benötigt?
- Teilsummenproblem: Teile eine Menge von Gegenständen so in zwei Teilmengen auf, dass beide Teilmengen das gleiche Gewicht haben.

5.12 Karp's 21 NP-vollständige Probleme

- **SATISFIABILITY:** z.B. in konjunktiver Normalform.
- **CLIQUE**.
- **SET PACKING:** Gegeben n Teilmengen $S_i \subseteq U$. Gibt es $k \leq n$ paarweise disjunkte Teilmengen?
- **0-1-INTEGER-PROGRAMMING**.
- **VERTEX COVER:** Minimale Menge von Knoten, sodass jede Kante adjazent zu einem Knoten ist.
- **SET COVERING:** Gegeben n Teilmengen, wähle möglichst wenige aus, sodass alle Elemente aus dem Universum enthalten sind.
- **FEEDBACK VETEX SET:** Gegeben ein Graph G (gerichtet/ungerichtet), finde eine minimale Menge $X \subseteq V$, sodass $G = (V - X, E)$ keinen Zyklus enthält.
- **FEEDBACK ARC SET:** Gegeben ein gerichteter Graph, finde eine minimale Menge $X \subseteq E$, sodass $G = (V, E - X)$ ein DAG (directed acyclic graph) ist.
- **DIRECTED/UNDIRECTED HAMILTONIAN CIRCUIT**.
- **3-SAT:** Genau 3 Literale pro Klausel.
- **CHROMATIC NUMBER**.
- **CLIQUE COVER:** auch PARTITION INTO CLIQUES. Teile Knotenmenge in k Teile, die allesamt Cliquen sind.
- **EXACT COVER:** Gegeben n Teilmengen $S \subseteq P(X)$, finde eine Auswahl von Mengen S, sodass X genau einmal in den Mengen vorkommt.
- **HITTING SET**.
- **STEINER TREE:** Wie MST, nur dass neue Knoten (steiner vertices) und Kanten (steiner edges) hinzugefügt werden dürfen, um die Länge zu minimieren.
- **3-DIMENSIONAL-MATCHING:** Wie bipartites Matching, nur mit 3 Mengen und Tripel.
- **0-1-KNAPSACK**.
- **JOB SEQUENCING**.
- **MAXIMUM CUT**.
- **PARTITION:** Kann eine Menge X so in $Y \subseteq X$ und $Z \subseteq X$ zerlegt werden, dass beide Mengen gleich groß sind (Summe der Elemente in den Mengen)?

5.13 Church-Turing-These

- **Die Menge der Turin-berechenbaren Funktionen ist genau die Klasse der intuitiv (vom Mensch) berechenbaren Funktionen.**
- Es gibt kein Rechnermodell, das mächtiger als die Turingmaschine ist.
- Gleich mächtige Modelle (Beispiele): Turingmaschine (egal wie viele Bänder), Random Access Machine, Typ-0-Grammatiken, Lambda-Kalkül, rekursive Funktionen, die meisten Programmiersprachen, Quantencomputer.

5.14 LOOP-Berechenbarkeit

- **Syntaktische Komponenten:** Variablen, Konstanten (natürliche Zahlen), Trennsymbole (; :=), Operationszeichen, Schlüsselwörter (LOOP, DO, END).
- Wenn $x_i := x_j + c$ und $x_1 := x_j - c$ Programme (P_i) sind, dann auch $P_1; P_2$ und *LOOP* x_i *DO* P *END* (führt eine Anweisung echt x_i mal aus).
- LOOP-Programme terminieren immer (totale Funktionen).
- *IF x=0* kann simuliert werden.
- Nicht alle intuitiv berechnenbaren oder totalen Funktionen sind LOOP-berechenbar (z.B. Ackermannfunktion)
- LOOP-Programme entsprichen den primitiv-rekursiven Funktionen.

5.15 WHILE-Berechenbarkeit

- Erweiterung der LOOP-Syntax: *WHILE* $x_i \neq 0$ *DO* P *END*.
- Wenn f an der Stelle x undefiniert ist, terminiert das Programm nicht.
- WHILE-Programme und Turingmaschinen können sich gegenseitig simulieren.
- Umwandlung in GOTO-Programm möglich (und umgekehrt).
- Syntax von **GOTO-Programmen:** $P ::= M_1{:}A_1; \ldots; M_k{:}A_k$, $A ::= x_i := x_j + c \mid x_i := x_j - c \mid$ *GOTO* $M_j \mid$ *IF* $x_i = c$ *THEN GOTO* $M_j \mid$ *STOP*.

5.16 Primitiv rekursive Funktionen

- Echte Teilmenge der intuitiv berechenbaren Funktionen (z.B. Ackermannfunktion nicht primitiv-rekursiv).
- Primitiv rekursive Funktionen entsprechen den LOOP-Programmen.

1. **Nullstellenfunktion:** $O^k(n_1, \ldots, n_k) = 0$.

2. **Projektion:** $\pi_i^k(n_1, \ldots, n_k) = n_i$.
3. **Nachfolgerfunktion:** $\nu(n) = n + 1$.
4. **Komposition:** $C[g, h_1, \ldots, h_m](n_1, \ldots, n_k) = g(h_1(n_1, \ldots, n_k), \ldots, h_m(n_1, \ldots, n_k))$.
5. **Primitive Rekursion:** $g : \mathbb{N}^{k-1} \to \mathbb{N}$, $h : \mathbb{N}^{k+1} \to \mathbb{N}$, $R[g, h](n_1, \ldots, n_k) := g(n_2, \ldots, n_k)$ wenn $x_1 = 0$ und $R[g, h] := h(R[g, h](n_1 - 1, n_2, \ldots, n_k), n_2, \ldots, n_k$ sonst.

- **Beispiele:** Addition, Multiplikation, Potenz, Vorgängerfunktion, min, max.

5.17 EA-Turingmaschinen

- Kommunikation eines Prozesses mit seinem Umfeld.
- TM mit Eingabe- und Ausgabeband.
- Befindet sich unter dem Lesekopf ein Leerzeichen, darf er nicht weiter bewegt werden (auf dem Leseband).
- Inhalt des Eingabandes ist Teil der Vorbedingung des Schreibbefehls.
- EA-Maschine kann unendlich lange laufen.
- Grundzustände markieren normales, nicht abbrechendes Verhalten (z.B. Betriebssystem akzeptiert im Grundzustand immer wieder Kommandos).
- Wenn sich eine EA-Maschine immer wieder in einem Grundzustand befindet, arbeitet sie regulär.

5.18 Reduktion: Entscheidungsprobleme und Optimierungsprobleme (CLIQUE)

- Optimierungsproblem zu Entscheidungsproblem: Berechne CLIQUE, zähle Knoten und gib JA/NEIN für gegebene Schranke k aus.
- Entscheidungsproblem zu Zahlvariante (ermittle k): Binäre Suche mit Entscheidungsproblem, logarithmische viele Schritte.
- Entscheidungsproblem zu Optimierungsproblem: Berechne Zahlvariante. Entferne Knoten 1, wenn k dadruch sinkt, gehört der Knoten zur CLIQUE, sonst nicht. Fahre mit dem nächsten Knoten fort.
- Das Optimierungsproblem ist im Gegensatz zum Entscheidungsproblem nur polynomiell schwieriger.

5.19 Komplexitätsklasse für Parallelrechner

- Betrachte PRAMs (Parallel Random Access Machine) mit mehreren Prozessoren/Akkumulatoren und gemeinsamen Speicher.
- Klasse NC (Nick's Class) enthält Sprachen, die in $\mathcal{O}((\log n)^k)$, k konstant, auf polynomiell (?) vielen Prozessoren entschieden werden können.
- NC ist Teilmenge von P und wahrscheinlich ist NC ungleich P, sonst wären alle Probleme in P effizient parallelisierbar.
- NC-Reduktion zum Nachweis von P-Vollständigkeit.

5.20 Pumping Lemma für reguläre Sprachen

- Wie zeigt man, dass eine Sprache nicht regulär ist?
- Endlicher Automat mit n Zuständen: wenn es ein Wort mit $\geq n$ Zeichen gibt, das vom Automat akzeptiert wird, muss es eine Schleife geben, die man *aufpumpen* kann.
- Für jede reguläre Sprache L gibt es eine Zahl n. Ein Wort $|z| \geq n$ kann man dann auf bestimmte Weise in uvw zerlegen, mit $|uv| \leq n$, $v \neq \epsilon$, sodass uv^iw, i beliebig, auch in L ist.
- Das klappt mit manchen nicht-regulären Sprachen jedoch auch.
- Trivialbeispiel: a^nb^n.

5.21 RAM - Random Access Machine (Registermaschine)

- Berechnungsmodell äquivalent zur Turingmaschine (können sich gegenseitig polynomiell simulieren).
- Besteht aus unendlich vielen Speicherzellen, die beliebig große Zahlen aufnehmen können, einem Register (Akkumulator), einem durchnummerierten Programm und dem Befehlszähler.
- Gültige Befehle: LOAD, LOAD INDIRECT, STORE äquivalent, ADD, SUB, JMP (GOTO), JMP IF ZERO, HALT, evtl. mehr.
- Sehr nahe an richtigen realen Rechnern.
- RAM mit mehr Registern ist gleich mächtig.

5.22 Polynomielle Reduktion: Färbung zu SAT

- SAT ist mindestens so schwer wie Graphenfärbung.
- Betrachte 3-Färbung.
- Transformation: $(\forall v_1 \in V) r_i \vee g_i \vee b_i$.
- $(\forall (v_i, v_j) \in E) \neg(r_i \wedge r_j) \wedge \neg(g_i \wedge g_j) \wedge \neg(b_i \wedge b_j)$.
- Erste Forderung: jeder Knoten hat eine Farbe. Zweite Forderung: zwei benachbarte Knoten haben nicht die gleiche Farbe.
- Transformation in Polynomialzeit.
- Es besteht sogar eine Beziehung zwischen den beiden Problemen in die andere Richtung.

5.23 Polynomielle Reduktion: Hamilton zu TSP

- **Problem:** Hamilton ist NP-vollständig, zeige, dass TSP auch NP-vollständig ist.
- Polynomielle Transformation: Verwende die gleichen Knoten. Wenn im Hamilton-Graph eine Verbindung zwischen zwei Knoten existiert, gewichte die Kante mit 1, ansonsten mit 2.
- Es existiert genau dann ein Hamilton-Zyklus, wenn das TSP eine Tour der Länge $|V|$ besitzt.
- TSP ist NP-vollständig.

5.24 Wie funktioniert der Satz von Cook?

- Benutze eine nichtdeterministische Turingmaschine.
- Stelle eine vollständige mathematische Definition der NTM auf.
- Beschreibung jedes Merkmals der NTM (wie werden Anweisungen ausgeführt, etc.) als aussagenlogische Formel.
- Jedes Problem in NP kann als Programm der NTM kodiert werden.
- Lösung der aussagenlogischen Formel ist eine Lösung des NP-Problems.

5.25 Fleißiger Biber - Busy Beaver

- Wie viele Einsen kann eine Turingmaschine auf das Band schreiben, ohne dass sie in eine Endlosschleife gerät, wenn sie n Zustände hat.
- $\Sigma(n)$ ist die Anzahl der Einsen.
- Die Funktion ist i.A. nicht algorithmisch berechenbar (Widerspruchsbeweis).
- Für $n = 1$ bis $n = 4$ noch bekannt, darüber hinaus nur Schätzungen.

5.26 Funktionsweise nichtdeterministischer Turingmaschinen

- Man stelle sich vor, das Band der NTM habe 3 Teile: Rateteil, Eingabeteil, Arbeitsteil.
- Auf dem Eingabeteil befindet sich die Eingabe, z.B. eine Partition-Instanz.
- Jetzt rät die TM auf *magische* Art und Weise eine Lösung.
- Mit dem Arbeitsband versucht die TM, die Lösung zu verifizieren: rechne Zahlen zusammen, multipliziere, ...
- Wenn das in polynomieller Zeit geht, ist das Problem in NP enthalten.

6 Verschiedenes

6.1 Darstellung von reellen Zahlen

- Es handelt sich fast immer um Näherungen.
- Festkommazahl: $10001,1000000$ (immer 8 Nachkommastellen).
- Gleitkommazhal: $a = (-1)^{VZ} \cdot m \cdot b^E$, wobei VZ das Vorzeichenbit, m die Mantisse, b die Basis (in der Regel 2) und E der Exponent sind.
- b^E verschiebt nur das Komma, z.B. im Dezimalsystem: $123,456 = 1,23456 \cdot 10^2 = 0,123456 \cdot 10^3$, oder im Binärsystem: $101,011 = 1,01011 \cdot 2^2 = 0,101011 \cdot 2^3$.
- Mantisse: $m = z,zz \ldots z$ (p Ziffern, eine davon vor dem Komma).
- Erste Ziffer der Mantisse (vor dem Komma) muss 1 sein, weil 0 nicht erlaubt.
- Datentyp single: $p = 23$, $E = 8$, $VZ = 1$, Datentyp double: $p = 52$, $E = 11$.

6.2 Programmiersprachen

6.2.1 Programmiersprache: Fortan

- 1957 von John Backus bei IBM entwickelt.
- Compiler macht Optimierungen: fast so schnell wie Assembler.
- Anfangs nur GOTO-Anweisungen, seit Fortran 77 Prozeduren etc.
- Seit Fortran 2003 objektorientiert.
- Gute numerische Mathematik: Bibliothek für komplexe Zahlen, Potenz-Operator von Anfang an, usw.
- Beliebt wegen der großen Anzahl an Bibliotheken.
- Imperative Programmiersprache.

6.2.2 Programmiersprache: LISP

- 1958 entwickelt von John McCarthy am MIT (basiert auf Lambda-Kalkül).
- LISP: List Processing Language.
- Grunddatenstrukturen: Einzelwerte (Skalarwerte) und Listen (beliebig verschachtelt).
- Programmanweisungen sind Listen, z.B. $(+\ 2\ 2)$, $(defun\ square\ (x)\ (*\ x\ x))$.
- Keine Unterscheidung zwischen Programm und Daten.
- Automatisiertes Speichermanagement, Compiler und Interpreter: schnelle Entwicklung von Programmen.
- Funktionale Programmiersprache.

6.2.3 Programmiersprache: ALGOL60

- Entwickelt bis 1960 von der ACM (Association for Computing Machinery): Algorithmic Language.
- Prozeduale Programmiersprache für wissenschaftliche, numerische Zwecke.
- Meilenstein unter den Programmiersprachen: saubere Definition der Sprache, unabhängig von der Implementierung; klarer, einfacher Sprachkern.
- Rekursive Prozeduren sind möglich.
- Im akademischen Bereich wichtig, in der Praxis weniger.

6.2.4 Programmiersprache: BASIC

- 1964: BASIC = Beginner's all purpose symbolic instruction code.
- Grundprinzipien einfach zu erlernen, universell einsetzbar, Erweiterbarkeit, Hardware- und Betriebssystemunabhängigkeit.
- Basic-Compiler war erstes Produkt von Microsoft.
- Minimalistische Sprache mit vielen Einschränkungen (Kontrollstrukturen fehlen, nur GOTO, wenig Datentypen, z.B. nur Arrays mit maximaler Größe).

6.2.5 Programmiersprache: Prolog

- Deskriptive Programmiersprache, 1972 von französischem Informatiker entwickelt.
- Programme sind Regeln und Fakten, z.B. $A \Rightarrow B$.
- Positives Resultat bedeutet, dass das Ergebnis ableitbar ist (es wurde bewiesen).
- **Beispiel:** mann(adam), vater(tobias, frank) sind Fakten (Datenbank). Die Anfrage mann(adam) liefert *wahr* zurück.
- System antwortet immer entweder mit *Ja* oder *Nein*.
- Eingebautes Backtracking, Listen, Rekursion.

6.2.6 Programmiersprachen: Paradigmen

- **Imperativ (Turingmaschine):** Folgen von Befehlen, sehr maschinennah, Veränderungen von Zuständen (im Speicher). Beispiele: Algol, Basic, C, Fortran, Pascal.
- **Funktional (Lambda-Kalkül):** Programme als Funktionen, die Eingabewerte in Ausgabewerte transformieren. Beispiele: LISP, Scheme, Haskell.
- **Pradikativ, Logik (Prädikatenkalkül):** Programme als System aus Regeln und Fakten. Beispiel: Prolog.
- **Objektorientiert:** Programme bestehen aus Objekten: Polymorphismus (z.B. Überladung für Multiplikation von nat., rellen, komplexen Zahlen), Kapselung (Prozeduren zu einer Einheit), Vererbbarkeit.

6.3 Kryptographie

- **Ziele:** Vertraulichkeit (Daten vor Anderen geheim halten), Integrität (Schutz der Daten vor Manipulation), Authentizität (Sicherstellung der Identität des Absenders).
- Symmetrische / Assymmetrische Verfahren (public key)

- **Kerckhoff's Prinzip:** Die Sicherheit eines Kryptosystems darf nicht von der Geheimhaltung des Algorithmus abhängen.

6.3.1 Klassische Verfahren

- **Caesar-Chiffre:** Jeden Buchstaben um eine konstante Zahl (im Alphabet) verschieben (nur 25 Schlüssel möglich).
- **Vigenere-Verschlüsselung:** Shift-Betrag wird bei jedem Zeichen geändert (am Ende des Schlüssels wieder von vorne).
- Beide mit Häufigkeitsverteilung von Buchstaben zu knacken (Vigenere-Chiffre: überprüfen, für welche Schlüssellänge sich eine passende Häufigkeitsverteilung ergibt, dann wie Caesar-Chiffre).

6.3.2 One-Time-Pad (Einmalblock)

- Einziges nachweislich sicheres Verfahren
- Wie Vigenere-Chiffre nur ist die Schlüssellänge mindestens so groß wie der Text.
- Anforderungen an den Schlüssel: geheim, unvorhersagbar zufällig, nur einmal verwenden.
- Statt Byte-Addition auch XOR-Verknüpfung o.Ä. möglich.

6.3.3 Symmetrische Verfahren

- Meistens Blockverschlüsselung (es werden z.B. immer 8 Byte verschlüsselt).
- Meistens rundenbasiert: Ergebnis der letzen Runde ist Eingabe der nächsten Runde (typischerweise 8-30 Runden).
- Nur 1 Schlüssel (mindestens 128 Bit).
- Entschlüsselung: In umgekehrter Reihenfolge, Anwendung der inversen Funktion.
- Siehe auch Feistel-Chiffre (Blowfish, DES, MARS, Twofish).

6.3.4 Teilen von Geheimnissen

- Das Codewort zum Entschlüsseln eines Textes soll unter n Personen aufgeteilt werden, sodass m zusammen den Text ermitteln können.
- **Beispiel:** Es genügen 2 Personen: Gerade im 2D-Raum für jede Person. Schnittpunkt ist der Schlüssel, dieser kann sich aus 2 Geraden eindeutig ermitteln lassen.
- **Beispiel:** Es genügen 3 Personen: Ebene im 3D-Raum für jede Person. Schnittpunkt dreier Ebenen ist ein eindeutiger Punkt.

- **Beispiel:** Es genügen m Personen: höherdimensionaler Raum analog.

6.3.5 Einwegfunktionen

- $f(x)$ einfach und $f^{-1}(x)$ schwer zu berechnen.
- Falltür/Hintertür: Mit einer Zusatzinformation lässt sich auch $f^{-1}(x)$ leicht berechnen.
- Trivialbeispiel: Briefkasten. Brief einwerfen ist einfach. Brief herausholen ist schwer. Falltür ist der Schlüssel.
- **Anwendung:** RSA-Kryptographie: modulare/s Exponentitation und Logarithmieren, Hash-Funktionen (z.B. MD5, SHA-1), Primfaktorzerlegung vs. Multiplikation von Primfaktoren.

6.4 Das quadratische Zuweisungsproblem

- Gegeben n Objekte und n Orte mit Fluss f_{ij} zwischen den Objekten und Distanz d_{ij}, minimiere $\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{a=1}^{n}\sum_{b=1}^{n} f_{ij}d_{ab}x_{ia}x_{jb}$, wobei $x_{ia} = 1$, wenn Objekt i auf dem Platz a ist.
- Nebenbedingungen $\sum_{i=1}^{n} x_{ki} = 1$ für alle k, d.h. ein Objekt immer nur an einem Ort. Außerdem analog an jedem Ort nur ein Objekt. Minimiere den Zielfunktionswert.
- Lösung als ganzzahliges LOP (NP-schwer, vgl. Branch-and-Bound Simplex).
- Lösung mit Metaheuristik (z.B. Tabu-Suche).
- **Anwendung:** Platzierung von Dienstorten, Schaltern, Bauteile (Schaltplan), Tastatur-Tasten.

6.5 Metaheuristik Tabu-Suche: Beispiel Permutation

- Permutation P mit Optimum $f(p)$ ist minimal/maximal und $p \in P$.
- Nachbarschaft $N(p)$: Vertauschen zweier Elemente, Element verschieben.
- Kandidatenliste: Besuche nur die Nachbarn mit bestem $f(p)$ und prüfe ggf. nur einen Teil von $N(p)$.
- Züge verbieten: Liste mit verbotenen (bereits besuchten) Situationen (viel Speicher, ggf. mit Hashtabelle).
- Liste mit $x < |N(p)|$ verbotenen Zügen, wobei Liste m^{-1} enthält, wenn ein Zug m angewandt wurde (inverser Zug), oder besser: speichere, dass i an p_i und j an p_j war.

6.6 Metaheuristik Simulierte Abkühlung

- Beachte eine einzige Lösung i.
- Erstelle Nachbarn $N(i)$ und berechne dessen Energieniveau.
- Bei Energieabsenkung akzeptiere neuen Zustand sofort, sonst akzeptiere mit Wahrscheinlichkeit $e^{-\frac{\Delta E}{T}}$.
- Führe Aktionen mehrmals hintereinander aus, senke dann Temperatur ab und wiederhole bis System gefroren.
- Evtl. Funktion mit Boltzmann-Konstante (vgl. Physik).

6.7 Genetische Algorithmen

- Initialisiere zuerst Population
- Fitness Level eines Individuums ist umso höher, je niedriger die Zielfunktion ist (bei Minimum).
- Auswahl für Reproduktion ist wahrscheinlicher, je höher die Fitness.
- Crossover (paarweise) zwischen Individuen, es entstehen neue Individuen.
- Anwendung von (zufälliger) Mutation auf die neuen Individuen.
- Anzahl der Individuen immer konstant halten (z.B. alte Individuen ersetzen oder schlechte Individuen ersetzen).

6.8 Bildformate

6.8.1 Windows Bitmap

- Dateiendung .bmp oder .dib (selten).
- Praktisch nur Version 3 relevant (kein Alpha-Kanal), Version 4 und 5 sehr selten und mit Alpha-Kanal.
- Farbtiefen 1, 4, 8, 16, 24, 32 bpp.
- Komprimierung mit RLE (Lauflängenkodierung) oder gar nicht.
- Sehr großer Speicherverbrauch, aber simples und effizientes Verfahren.

6.8.2 Graphics Interchange Format

- Bis zu 256 Farben, Endung .GIF.
- Mehrere Einzelbilder in einer Datei möglich: Animation.
- Effiziente LZW Kompression (Patentprobleme), kleine Bilder fürs WWW.
- Farbtabelle mit 256 frei wählbaren Farbwerten (aus $256^3 = 16,7$ Millionen Farben).
- Eine Farbe stellt Transparenz dar (kein Alpha-Kanal).
- Interlacing: Es werden zuerst jede 8., dann jede 4., dann 2. und dann die übrigen Zeilen gespeichert: Voransicht, wenn das Bild noch nicht komplett übertragen wurde.
- PNG: Bessere Kompression, mehr Farben, keine Patentprobleme.

6.8.3 Portable Network Graphics

- Endung .PNG, Rastergrafik mit verlustfreier Kompression, Ersatz für GIF (keine Patentbeschränkungen).
- Keine Animation vorgesehen, dafür das Format .MNG.
- Farbtiefen 24- oder 48-Bit.
- Transparenzinformation für einzelne Farbe (wie GIF) oder Alpha-Kanal (besser).
- Vorfilter für Kompression: Angabe von Differenzfarbwerten benachbarter Pixel (oft Folgen gleicher Werte, pro Zeile verschiedene Filter möglich, d.h. andere Bezugspixel für Differenzwerte).
- Kompression mit Deflate (ZIP, verlustlos), andere Verfahren theoretisch möglich.
- **Vergleich:** Farbverlauf mit PNG 13-mal kleiner als mit GIF.

6.8.4 Joint Photographic Experts Group (JPEG)

- YCbCr-Farbmodell: Helligkeit, Blau, Rot, wobei Helligkeit genauer gespeichert wird, da das menschl. Auge dafür empfindlicher ist (50 Prozent Dateneinsparung, ohne spürbaren Qualitätsverlust, Tiefpassfiltertung, Unterabtastung von CbCr, also in geringerer Auflösung)
- Aufteilung des Bildes in 8x8-Blöcke, dann diskrete Kosinustransformation. Weitere komplizierte Operationen.
- Zum Schluss Huffman-Kodierung.

6.8.5 Bildkompression

- **Verlustfrei:** Umsortierung, dann Wörterbücher (Folgen), Lauflängenkodierung, LZW, Huffman, Deflate (PNG); nur bei ähnlichen Flächen oder Farbverläufen (PNG) gut.
- **Verlustbehaftet:** möglichst nicht sichbar für das menschliche Auge, Quantisierung bei JPEG: Darstellung von Amplituden oder Ortsfrequenzen, werden abgerundet und kleine Kooeffizienten verschwinden (weniger Bits).
- **Fraktale Bildkompression:** Selbstähnlichkeit laut Chaostheorie, suche ähnliche Bereiche im Bild und speichere nur Positionen (Codebook), aufwendig: neuronale Netze.

6.8.6 Tagged Image File Format

- Mehrere Bilder pro Datei möglich.
- Verschiedene Farbräume, Kompressionsalgorithmen (z.B. LZW, RLE).
- Bildpunkte sind beliebig viele Einzelwerte (Samples), normal 1 Byte, auch Alpha-Kanal.
- Speicherung von Streifen (Pixelzeilen) oder Kacheln: Ausschnitte können im RAM gehalten werden.
- Größter Nachteil: hohe Komplexität durch viele mögliche TIFF-Optionen.
- Streaming nicht immer möglich: ganze Datei laden (schlecht fürs WWW).
- Max. 4GB Dateigröße manchmal zu klein (Astronomie).

6.9 Suchmaschinen: PageRank

- Allgemein bei Suchmaschinen: Prüfe Vorkommen der Suchwörter: Gewichtung nach Übereinstimmung in Überschriften, Suchwörter nah beeinander, Häufigkeit der Suchwörter.
- Google PageRank: Wichtigkeit einer Seite ist Anzahl der Links auf eine Seite bzw. Anzahl der Besuche eines zufälligen Besuchers.
- Darstellung als Graph: Internetseiten sind Knoten, gerichtete Kante von A nach B, wenn es einen Link von A nach B gibt.
- Simuliere zufälligen Besucher, der irgendeine Internetseite (URL) eintippt, in diesem Beispiel mit $0,2$ gewichtet.

- **Beispiel:** Knoten C hat 3 Links, von denen einer zu Knoten B führt, Knoten D hat einen Link der zu B führt, ebenso Knoten E. Insgesamt gibt es 6 Internetseiten. Dann ist die Wichtigkeit von Knoten B $b = 0,8 \cdot (\frac{1}{3}c + d + e) + 0,2 \cdot \frac{1}{6}$.
- Stelle Gleichungen für alle Knoten auf: Lineares Gleichungssystem.
- Löse das Gleichungssystem z.B. mit Gauß-Seidel-Verfahren (exaktes Verfahren zu langsam): Setze Variablen zuerst beliebig, dann rechne die Gleichungen nacheinander aus, so lange bis sich nichts mehr ändert.

6.10 Compiler

1. Frontend (Analysephase): **Lexikalische Analyse:** Aufteilen in Token, z.B. Schlüsselwörter, Zahlen, Operatoren; **Syntaktische Analyse:** Umwandlung in Syntaxbaum, Fehler wenn Code nicht in Grammatik; **Semantische Analyse:** z.B. Prüfung von Datentypen bei Zuweisung (attributierter Syntaxbaum).
2. Backend (Synthesephase): **Zwischencodeerzeugung** (z.B. bei mehreren Plattformen); **Programmoptimierungen** (Schleifen vermeiden etc.); **Codegenerierung:** Aus Zwischencode oder direkt aus dem Syntaxbaum, Linken mit Bibliotheken.

6.11 Programmoptimierungen bei Compilern

- **Einsparung von Maschinenbefehlen:** z.B: Werte in Registern behalten, wenn sie bald wieder benötigt werden.
- **Statische Formelauswertung:** z.B. $2\pi r = 6,283r$.
- Unerreichbaren Code entfernen.
- Unbenutze Variablen entfernen.
- **Schleifenoptimierung:** Variablen in Registern, Schleifen zusammenfassen oder auflösen, Laufvariable herunterzählen statt hochzählen (jump-not-zero ist effizienter).
- Verwendung schnellerer Anweisungen, z.B. Shiften statt Multiplikation mit 2.
- Weglassen von Laufzeitprüfungen wenn möglich.
- Verzögern oder vorziehen von Anweisungen: Manche Anweisungen können parallel ausgeführt werden.

6.12 EAN/ISBN-13-Codes

- **Beispiel:** 978-0-362-03293-3.
- Erste 7 Ziffern: internationale Lokationsnummer (Land und Hersteller).
- 5 Ziffern Artikelnummer: wird vom Hersteller bestimmt.
- 1 Prüfziffer: $s = x_1 + x_3 + \ldots + 3(x_2 + x_4 + \ldots)$. $x_1 3$ wird so gewählt, dass $s + x_1 3$ durch 10 teilbar ist.
- 1 falsche Ziffer wird immer erkannt, 1 Zifferndreher nicht immer.
- Beispiel fehlerkorrigierender Code: 000, 001, 010, 100 ist 0 und 111, 110, 101, 001 ist 1, korrekt wenn nur 1 Bit falsch ist.

6.13 Mehrheitsbestimmung mit dem Majority-Algorithmus

- **Einfacher Algorithmus:** Strichliste, Problem: Liste mit Namen durchgehen, Datenschutz.
- **Majority-Algorithmus:** Stapel S anfangs leer, alle Stimmzettel durchgehen: falls Stapel leer oben drauf, falls anderes Element als auf dem Stapel oben, entferne erstes Element, falss gleichen Element wie auf dem Stapel oben, lege Element auf den Stapel.
- Der Stapel enthält also immer nur gleiche Elemente.
- Wenn der Stapel zum Schluss leer ist, gibt es keine Mehrheit.
- Gehe zum Schluss nochmal alle Elemente durch und prüfe, ob es wirklich $\frac{n}{2}$ viele Mehrheitselemente gibt.

6.14 Pseudo-Zufallsgenerator

- Iterative Rechenvorschrift für Zahlen $[0; m[$.
- $x_{i+1} = (a \cdot x_i + c) \mod m$.
- Möglicherweise werden nicht alle Zahlen im Wertebereich erreicht.
- Echter Zufallsgenerator nur durch physikalische Phänomene: Münzwurf, radioaktiver Zerfall.

6.15 Spieltheorie: Streichholzspiel

- n Streichhölzer auf dem Tisch, 2 Spieler ziehen abwechselnd.
- Nehme 1, 2 oder 3 Hölzer, wer das letzte Holz nimmt, hat verloren.
- DP: Starte mit 1 Holz: sicher verloren, 2 Hölzer: sicher gewonnen, 3 Hölzer: es gibt eine Gewinnstrategie.
- Gegner und Spieler handeln intelligent. Spieler hat eine Gewinnstrategie gdw. der Gegner keine Gewinnstrategie hat.
- Eigentlich kein Algorithmus notwendig, es ergibt sich eine regelmäßige Folge (Gewinnen oder Verlieren).

6.16 Faires Teilen (rekursiv)

- Teile X unter n Personen auf: platziere n Striche (jede Person) zum Teilen.
- Teile zwischen erstem und zweitem Strich: 1. Person bekommt etwas mehr als gewünscht, andere Personen teilen sich etwas mehr als gewünscht.
- Fahre rekursiv fort, bei 2 Personen: Einer teilt, der Andere wählt.

6.17 Kleinster umschließender Kreis (probabilistisch)

- Es werden maximal (!) 3 Punkte für die Platzierung des Kreises und dessen Radius benötigt.
- Angenommen es gibt einen effizienten Algorithmus für 13 Punkte.
- Wähle 13 Punkte zufällig, bilde kleinsten Kreis.
- Für alle nicht enthaltenen Punkte: gewichte beim nächsten Mal doppelt.
- Warum geht das? Wenn Punkte außerhalb sind, wurde mindestens einer der 3 Punkte nicht gewählt. Erhöhe dessen Wahrscheinlichkeit.

6.18 Online-Algorithmus

- Zu Beginn sind nicht alle Eingabedaten vorhanden.
- Algorithmus muss schon vorher Entscheidungen treffen.
- **Beispiel:** Suche eines kürzesten Weges in einem umbekannten Graphen, Seitenfehlerproblem.
- **k-Kompetivität:** Lösung ist niemals schlechter als das k-fache Optimum.

- Strategien beim Seitenwechselproblem: Least-Recently-Use (LRU, 2-komp.), First-In-First-Out (FIFO, 2-komp.), Most-Recently-Used (MRU, schlecht).

6.19 Boolsche Funktionen

- Darstellung einer boolschen Funktion mit Tabelle.
- Jede boolsche Funktion kann mit AND und NOT ausgedrückt werden (vollständiges Operatorensystem/Basis).
- **Beispiel:**

x	0	0	0	0	1	1	1	1
y	0	0	1	1	0	0	1	1
z	0	1	0	1	0	1	0	1
F	0	1	0	0	1	1	1	1

- $F(x, y, z) = \neg(\neg x \wedge \neg y \wedge \neg z) \vee \neg(\neg x \wedge y \wedge \neg z) \vee \neg(\neg x \wedge y \wedge z)$.
- Weitere vollständige Basen: NAND, NOR (jeweils einzeln). Beweis: Bilde AND, OR, NOT mit NAND nach.
- Konjunktive Normalform.
- Minimierung mit Karnaugh-Veith-Diagrammen.

6.20 Gödelscher Unvollständigkeitssatz

- Hilbertprogramm: Formalisierung der Mathematik. In einem neuen Kalkül sollten alle Aussagen bewiesen oder widerlegt werden können.
- Jedes hinreichend mächtige (Peano-Axiome) formale System ist entweder widersprüchlich oder unvollständig.
- Beweis durch Diagonalisierung.

6.21 Kolmogoroff-Theorie

- Zufälligkeit einer endlichen Zeichenfolge ergibt sich aus dem kleinsten Programm, das sie erzeugt.
- $r \leq$ (Länge des Programm + Länge Eingabe (Bit)) / Länge Ausgabe (Zeichen).
- Kolmogoroff-Komplexität ist nicht berechenbar.

6.22 Graphen schön zeichnen

- Knoten im Kreis anordnen.
- Knoten vertauschen (simulierte Abkühlung).
- Knoten, die adjazent sind, ziehen sich an, andere stoßen sich ab, mehrere Iterationen.

6.23 Mandelbrot-Menge

- Popularisiert 1980 von Benoit Mandelbrot.
- Fraktal, Selbstähnlichkeit.
- Definition über Rekursion: Mandelbrotmenge enthält komplexe Zahlen c, für die $z_{n+1} = z_n^2 + c$ mit $z_0 = 0$ beschränkt bleibt, d.h. nicht über alle Grenzen wächst.
- Spiegelsymmetrie zur RE-Achse.
- Zusammenhängend (keine Inseln).

6.24 OSI-Schichtenmodell

- OSI = Open Systems Interconnection Reference Model von der ISO mit 7 Schichten (1983 standardisiert).
- Jede Schicht stellt Dienste bereit, wobei darunterliegende Schichten verwendet werden dürfen.

1. Physical Layer: Bitübertragung.
2. Data Link Layer: Sicherungsschicht.
3. Network Layer: Vermittlung.
4. Transport Layer: Transport.
5. Session Layer: Sitzungen.
6. Presentation Layer: Darstellung.
7. Application Layer: Anwendungen.

6.24.1 Physical Layer (OSI-Schicht 1)

- Mechanische, elektrische Hilfsmittel für physikalische Verbindungen: Übertragung von Bits via elektrischen Signalen, optischen Signalen, elektromagnetischen Wellen usw.
- **Probleme:** Wie werden Bits auf einem Koax-Kabel bzw. via Funk übertragen, wo nur Spannungspulse/elektromagnetische Wellen verfügbar sind?
- Elemente der Schicht 1 (Beispiele): Modem (Modulator, Demodulator: Aufmodulierung eines digitalen Signals auf eine Trägerfrequenz) , Hub (verstärkt und entrauscht das Signal und schickt es an alle Geräte), Repeater, T-Stück, Abschlusswiderstand.

6.24.2 Data Link Layer (OSI-Schicht 2)

- **Problem:** Gewährleistung einer sicheren Übertragung.
- Aufteilen des Bitdatenstroms, Hinzufügen von Folgenummern und Prüfsummen.
- Empfänger kann falsche Blöcke erneut anfordern (Quittungs- und Wiederholmechanismus).
- Aufteilung des Layers 2 in 2 Unterschichten durch IEEE: Media Access Control (MAC) für Zugriffskontrolle auf gemeinsam genutze Medien, Logical Link Control (LLC) für Fehlerkorrektur.
- Hardware auf Schicht 2: Bridge, Switch.

6.24.3 Media Access Control (OSI-Schicht 2a)

- **Problem:** Gemeinsamer Zugriff verschiedener Geräte auf ein gemeinsames Übertragungsmedium, Datenkollision und -Verlust.
- Kontrollierter Zugriff ohne Kollisionen (collision avoidance): Es wird festgelegt, wer sprechen darf (Schüler-Lehrer), z.B. mit Token oder speziellem Kanal.
- Konkurrierender Zugriff: Es gibt Kollisionen und Regeln zum Behandeln dieser (vgl. Telefongespräch: wenn beide reden, hören beide auf und warten eine gewisse Zeit).
- Switch: Aufteilung des Netzes in Kollisionsdomänen anhand der MAC-Adresse.

6.24.4 Network Layer (OSI-Schicht 3)

- **Problem:** Keine direkte Punkt-zu-Punkt-Verbindung, Weiterleitung über mehrere Knoten notwendig.
- Weitergeleitete Pakete gelangen nicht in höhere Schichten.
- Wichtige Aufgaben: Aktualisierung von Routingtabellen, Übersetzung zwischen verschiedenen Protokollen, Fragmentierung.
- Hardware: Router, Layer-3-Switch.
- Protokolle: IP, IPsec, ICMP.

6.24.5 Internet Protocol v4 Adressen (OSI-Schicht 3)

- 32-bit-Adressen, also nur max. 4,2 Mrd. Adressen.
- Aufteilung der Adresse in Netz- und Geräteteil.
- Befindet sich eine Adresse nicht im eigenen Netz, wird ein Router befragt.
- **Beispiel:** IP: 192.168.0.23, Subnetzmaske: 255.255.255.0, Bits der Subnetzmaske: $1^8.1^8.1^8.0^8$.
- Wenn ein Bit in der Subnetzmaske 1 ist, gehört diese Stelle zur Netzadresse.
- Alternative Darstellung: 192.168.0.23/24, bedeutet, dass die ersten 24 Bits in der Subnetzmaske 1 sind.
- Broadcast-Adressen: 192.168.0.255/24 (ermöglichte DoS Attacken), 255.255.255.255 geht immer ins eigene Subnetz.
- 192.168.0.0 ist das Netz selbst.

6.24.6 Internet Protocol v4 (OSI-Schicht 3)

- Maximale Paketlänge $2^16 - 1 = 65535$ Bytes, in der Regel aber durch Schicht 1 oder 2 beschränkt (MTU bei Ethernet 1518).
- Keine Unterscheidung zwischen Endgeräten (Hosts) und Routern (Vermittungsgeräte).
- Gesamtheit aller Router bilden das Internet.
- Router weist in der Routingtabelle jeder Netzadresse ein Zielnetzwerk zu.
- Jedes Paket wird einzeln geroutet (Ausfallsicherheit): Pakete können doppelt ankommen, verschiedene Wege nehme und fragmentiert werden.
- Zu wenige IPv4-Adressen: Entwicklung von IPv6.

6.24.7 IPv4 mit Ethernet

- Eigene 48-bit-Adressen im Ethernet (MAC-Adressen), vom Hersteller festgelegt.
- ARP (Address Resolution Protocol): Auflösung IP zu MAC, wird im ARP-Cache gespeichert.
- Bei unbekannten IPs: ARP-Request via Broadcast, Empfänger antwortet (ARP-Reply), ARP-Attacken im lokalen Subnetz möglich.
- Ehternet: OSI Schicht 1 und 2, für kabelgebundenes Datennetz.
- Festlegung von Kabeltypen, Stecker, Übertragungsschicht (Bitsignale auf den Kabeln), Collision Detetction (z.B. bei Koax).

6.24.8 Transport Layer (OSI-Schicht 4)

- Vermeidung von Staus (congestion avoidance).
- Beispiel TCP (Transmission Control Protocol): Datenverlust erkennen und beheben, Übertragung in beide Richtungen, keine Netzüberlastung.
- Nach außen sind nur die Sockets sichtbar.
- Mit Ports werden die einzelnen *Prozesse* am PC identifiziert.
- TCP-Verbindung ist durch Quell- und Ziel-IP und -Port definiert.

6.25 Mooresches Gesetz

- Formuliert 1965 von Gordon Moore.
- Komplexität (Anzahl der Schaltkreiskomponenten) verdoppelt sich alle 2 Jahre (18-24 Monate).
- Laut Moore (2007) ist das Gesetz noch bis 2017-2022 gültig, laut einem Intel-Mitarbeiter bis 2029.
- Nur eine empirische Beobachtung.
- Bezieht sich auf das Kostenoptimum, also geringe Kosten pro Schaltkreis.
- Bedeutet nicht zwangsläufig Verdoppelung der Leistung.

6.26 Binärzahlen

- Binärzahl zu Dezimalzahl: $010111_2 = 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 23_{10}$.
- Dezimalzahl zu Binärzahl: Ermittle Zweierpotenzen:
- $163_{10} \div 2 = 81R1$.
- $81_{10} \div 2 = 40R1$.
- $40_{10} \div 2 = 20R0$.
- $20_{10} \div 2 = 10R0$.
- $10_{10} \div 2 = 5R0$.
- $5_{10} \div 2 = 2R1$.
- $2_{10} \div 2 = 1R0$.
- $1_{10} \div 2 = 0R1$.
- $163_{10} = 10100011_2$.

6.27 Darstellung von ganzen Zahlen

- Vorzeichenlos: einfach als Binärzahl.
- Mit Vorzeichen: Zweierkomplement.
- **Beispiel:** 8-Bit-Zahlen.
- $00000000_2 = 0_{10}$.
- $01111111_2 = +127_{10}$.
- $10000000_2 = -128_{10}$.
- $11111111_2 = -1_{10}$.
- Erstes Bit zeigt Vorzeichen an.
- Überlaufbit: XOR der Überträge an höchster und zweihöchster Stelle.
- Es sind keine neuen Schaltkreise für Vorzeichenzahlen notwendig (anders als bei trivialer Kodierung).

6.28 Oktal- und Hexadezimalzahlen

- Binärzahl zu Oktalzahl: $1010011100101_2 = 123456_8$.
- Fasse immer je 3 Bits zusammen.
- Bei Hexadezimalzahlen analog: $1010011100101110_2 = A72E_{16}$.
- Verwendung von Oktalzahlen: Zugriffrechte unter LInux, leichtere Lesbarkeit von Bitfolgen.
- BCD (Binary Coded Decimal): Jede Ziffer einer Dezimalzahl wird binär kodiert (4 Bits pro Ziffer), manche (wenige) Prozessoren haben hardwareunterstütze Dezimalzahlarithmetik.

6.29 CRC - Cyclic Redundancy Check

- Bitfolge als Polynom, z.B. $10011101 = x^7 + x^4 + x^3 + x^2 + 1$.
- Generatorpolynom wird zur Berechnung der Prüfsumme benötigt.
- Polynomdivision: Polynom / Generatorpolynom (Modulo 2).
- Wenn das Generatorpolynom Grad n hat, werden an das Polynom zuvor n Nullen angehängt.
- Rest der Polynomdivision ersetzt dann die angehängten Nullen.
- Erneuter CRC mit neuem Polynom muss als Rest Null ergeben.
- CRC nur für die Erkennung von (zufälligen) Fehlern geeignet, nicht als kryptographische Hashfunktion.

6.30 Fehlerkorrigierender Code: Reed-Muller-Code

- NASA-Mission: Bilder vom Mars.
- 32 Graustufen pro Pixel, 5 Bits/Pixel.
- 32 Codes der Länge 32 Bits, wobei jeder Code einer Farbe entspricht.
- Codes unterscheiden sich untereinander um je 16 Bits (Stellen).
- Wenn 7 Bitfehler auftreten, kann der Pixelwert noch genau zugeordnet werden, 15 Bitfehler können maximal erkannt werden.
- Konstruktion über Hadamard-Matrix, rekursiv.

6.31 Interpolation mit Splines

- **Problem:** Gegeben n Kontrollpunkte, finde eine Kurve, die durch die n Punkte geht (ohne Knicke, etc.).
- Problem bei Polynominterpolation: Kurve kann stark oszillieren.
- Splines mit Grad k: Finde $n-1$ solche Polynome vom Grad k, sodass die Kontrollpunkte verbunden werden und die Kurven an den Punkten die gleiche Steigung (+evtl. Krümmung) haben.
- LGS lösen.
- Anwendung kubischer Splines: Achterbahnen, Schienen bei Zügen.

6.32 Algorithmus von Waltz

- 3D-Zeichnungen erkennen.
- Untersuche Grenzen (Kanten) von Flächen: verschiedene Möglichkeiten: Pfeil, Gabel, L, T, K.
- Beschrifte die Kanten: Grenze, konvexe Kante, konkave Kante.
- Es sind nicht alle möglichen Beschriftungen *physikalisch* möglich.

1. Erstelle Liste mit allen möglichen Beschriftungen bei jeder Kreuzung.
2. Betrachte Nachbarkreuzungen und streiche inkompatible Fälle.
3. Entferne alle jetzt unmöglichen Fälle bei den Nachbarn, usw.

6.33 Fehlererkennender Code: Gray-Code

- Benachbarte Codewörter unterscheiden sich nur durch 1 Dualzahl (Hamming-Distanz 1).
- **Verwendung:** Übertragung von digitalen Signalen auf Analogleitungen, z.B. Temperatursensor mit 5 (Bit) Kanälen, die sich ggf. nicht zeitgleich ändern. Dann gibt es ggf. mehrere Zwischenzustände, welcher ist echt?
- Rekursiver Generator: $n = 1$: 0, 1. Verfahren: Setze 0 davor und behalte alten Code bei, setze 1 davor und invertiere (Reihenfolge) alten Code. $n = 2$: 00, 01, 11, 10. $n = 3$: 000, 001, 011, 010, 110, 111, 101, 100.
- Gray-Code hat auch zyklische Hamming-Distanz 1.

6.34 Newton-Verfahren (Iteration)

- **Problem:** Finde die Nullstellen einer beliebigen (i.A. nicht-linearen) Funktion.
- Beginne bei beliebigem Kurvenpunkt, berechne Schnitt der Tangente mit der x-Achse. Mache an dieser x-Position weiter.
- Verfahren konvergiert nicht immer, man muss die Suche ausreichend nahe an der Nullstelle beginnen, dann verdoppelt sich die Anzahl der gültigen Stellen in jedem Schritt.
- **Größtes Problem:** Ableitung der Funktion wird benötigt.

6.35 Primzahltest

- Sieb des Eratosthenes: exponentielle Laufzeit.
- Kleiner Satz von Fermat: für alle Primzahlen $a^{p-1} \equiv 1(modp), \forall a < p$, nicht hinreichend!
- Pseudoprimzahlen bestehen den Test (Carmichael Primzahlen).
- Miller-Rabin-Test: probabilistischer Algorithmus, arbeitet mit Zufallszahlen, Wahrscheinlichkeit, dass das Ergebnis falsch ist, ist kleiner als $\frac{1}{4}$. Algorithmus mehrfach hintereinander ausführen!
- 2002: AKS-Primzahltest: deterministisch und in Polynomialzeit: PRIMES $\in$ P.

6.36 Motivation: Neuronale Netze

- Anzahl der Recheneinheiten: 10^{11} Neuronen, 10^9 Transistoren.
- Art der Berechnung: parallel, in der Regel seriell.
- Schaltzeit: $10^{-3}s$, $10^{-9}s$.
- Schaltvorgänge (tatsächlich/theoretisch): $10^{12}Hz$ / $10^{13}Hz$, $10^{10}Hz$ / $10^{18}Hz$.
- Das Gehirn arbeitet effizienter und parallel.
- Ziele: Lernfähigkeit, Generalisierungsfähigkeit, Fehlertoleranz (z.B. Alkohol vernichtet Neuronen, hat aber keinen großen Einfluss auf Leistungsfähigkeit).
- 100-Schritt-Regel: Mensch braucht ca. 0,1 Sekunden zum Erkennen einer Person (ca. 100 Takte). Computer kann in 100 seriellen Schritten so gut wie gar nichts machen. Parallelisierung notwendig!

6.37 Bereichssuche (mehrdimensional)

- Beispiel 2D-Punkte.
- Erste Variante: Teile das Koordinatensystem in $n \times n$ Sektoren auf und speichere Liste in jedem Sektor.
- 2D-Binärbaum: Wurzel: Verzweige in y-Richtung (links kleiner, rechts größer), nächste Ebene in x-Richtung, dann wieder y-Richtung.
- Ziel: Es muss keine vollständige Suche in $\mathcal{O}(n)$ ausgeführt werden.
- Problemstellung: große mehrdimensionale Datenbank (z.B. Personen mit Alter, Beruf, ...) und viele Anfragen.

7 TO-DO: Fehlende Themen

- Pattern Matching mit Boyer-Moore-Horspool, Knuth-Morris-Pratt, Endlicher Automat.
- Fehlerkorrigierende Codes: Hamming-Code.
- Shannon-Hartley-Gesetz.
- Karnaugh-Diagramme: Minimierung von boolschen Ausdrücken.
- Neuronale Netze.
- Gaußsches Eliminationsverfahren.
- Wie funktioniert ein Computer eigentlich?
- Themen die ich vorher schon gut kannte und deshalb ausgelassen habe.

8 Interessante Bücher und Quellen

- Vöcking et al.: Taschenbuch der Algorithmen. Berlin, Heidelberg: Springer, 2008.
- Domschke, Drexl: Einführung in Operations Research. 7. Auflage. Berlin, Heidelberg: Springer, 2007.
- Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms. 3. Auflage. Cambridge, London: The MIT Press, 2009.
- Schönig, Uwe: Theoretische Informatik - kurz gefasst. 5. Auflage. Heidelberg: Spektrum Akademischer Verlag, 2008.

- Ziegenbalg, Jochen: Algorithmen. Von Hammurapi bis Gödel. Heidelberg, Berlin, Oxford: Spektrum Akademischer Verlag GmbH, 1996.
- Heineman, Pollice, Selkow: Algorithms in a Nutshell. A Desktop Quick Reference. O'Reilly, 2009.
- Selke, Gisbert: Kryptographie. Verfahren, Ziele, Einsatzmöglichkeiten. 1. Auflage. O'Reilly, 2000.
- Skiena, Steven: The Algorithm Design Manual. New York, Berlin, Heidelberg: Springer-Verlag, 1998.
- Sedgewick, Robert: Algorithmen. 2. Auflage. München: Pearson Education Deutschland GmbH, 2002.
- Dewdney, A. K.: The New Turing Omnibus. 66 Excursions in Computer Science. New York: Henry Holt, 2001.