Algorithmen 1 – Sommer 2024

1 Lernziele

Die Vorlesung sollte dich dazu befähigen eigene Algorithmen zu entwickeln. Dafür ist es essentiell, dass du grundlegende Datenstrukturen und Algorithmen kennst und verstanden hast.

Darüber hinaus erfordert das Entwickeln eigener Algorithmen viel Übung. Das liegt insbesondere daran, dass man ein Gefühl dafür bekommen muss, welche algorithmischen Ideen sich auch tatsächlich im Detail umsetzen lassen. Zudem muss man bei der Ideenfindung immer auch schon die Korrektheit und die Laufzeit des Algorithmus mit bedenken. Die in der Vorlesung behandelten Algorithmen und Datenstrukturen liefern Beispiele für diesen Prozess der Entwicklung von Algorithmen. Insbesondere haben wir daran exemplarisch einige allgemeine Methoden und Techniken zur Entwicklung von effizienten Algorithmen kennengelernt.

1.1 Grundlegende Datenstrukturen

Ein essentieller Bestandteil der meisten Algorithmen ist eine geschickte Verwaltung der Daten, sodass häufig verwendete Operationen effizient ausgeführt werden können. Da auch verschiedene Algorithmen oft doch sehr ähnliche Operationen benötigen, kapseln wir die geschickte Verwaltung der Daten in Datenstrukturen, die dann in verschiedenen Algorithmen universell eingesetzt werden können.

In der Vorlesung haben wir an verschiedenen Stellen diskutiert, welche Datenstrukturen sich für die aktuelle Situation (nicht) eignen. Beispiele dafür sind Insertion Sort, beim Telefonnummernverzeichnis, beim Best-Fit Bin-Packing oder beim Algorithmus von Prim. VL 5, 7, 12, 16

Um in jeder Situation die geeignete Datenstruktur auswählen und ggf. den eigenen Bedürfnissen anpassen zu können, solltest du die folgenden Datenstrukturen kennen und ihre Funktionsweise verstanden haben.

- dynamische Arrays VL 3
- Listen VL 4
- binäre Heaps VL 11
- (a, b)-Bäume VL 12, 13
- Hashmaps VL 7
- Union–Find VL 17
- Graphdatenstrukturen (insbesondere Adjazenzliste) VL 8

1.2 Grundlegende Algorithmen

Gewisse grundlegende algorithmische Probleme trifft man immer wieder an, manchmal als Teil eines anderen Problems. Diese grundlegenden Probleme und ihre algorithmischen Lösungen zu kennen, hilft bei der Auswahl geeigneter Algorithmen. Ein tieferes Verständnis der Algorithmen sorgt darüber hinaus dafür, dass man die Algorithmen der gegebenen Situation anpassen oder auf Basis ähnlicher Methoden neue Algorithmen entwerfen kann.

Die folgenden Algorithmen solltest du kennen und verstanden haben.

- binäre Suche — VL 4
- Sortieralgorithmen
  - Mergesort — VL 5
  - Quicksort — VL 5
  - Bucketsort — VL 6
  - Radixsort — VL 6
- kürzeste Wege in Graphen
  - Breitensuche (BFS): SSSP, ungewichtet — VL 8
  - Dijkstras Algorithmus: SSSP, nicht-negative Gewichte — VL 9
  - Bellman–Ford: SSSP, negative Gewichte — VL 10
  - Floyd–Warshall: APSP — VL 10
- Tiefensuche (DFS) auf gerichteten und ungerichteten Graphen — VL 14, 15
- Berechnung minimaler Spannbäume: Prim und Kruskal — VL 16

1.3 Allgemeine Methoden und Techniken

Die Entwicklung eigener Algorithmen beinhaltet die Formalisierung des in natürlicher Sprache gegebenen Problems, den Algorithmenentwurf auf hohem Abstraktionsniveau, die Detailumsetzung in (Pseudo-)Code, den Beweis der Korrektheit, sowie die Laufzeitabschätzung.

1.3.1 Formalisierung

Oft sind Probleme in natürlicher Sprache gegeben und müssen erst in eine formale Problemstellung übersetzt werden. Das war vor allem regelmäßig Teil der Übungsaufgaben. — VL 15, 18

1.3.2 Algorithmenentwurf

Im Prinzip haben wir in jeder Vorlesung Algorithmen entworfen. Neben dem „scharf nachdenken und eine gute Idee haben“ haben wir da insbesondere folgende Techniken kennengelernt.

- Teile und Herrsche — VL 2, 5
- Greedy-Algorithmen — VL 16

- dynamische Programmierung — VL 10, 18

Die folgenden Aspekte sind keine Techniken, die so allgemein verbreitet sind, dass sie einen eigenen Namen haben. Dennoch stellen sie Tricks und Denkweisen dar, die in vielen Situationen hilfreich sind.

- Datenstrukturen: etwas Unordnung erlauben → deutlich schnellere Modifikation der Datenstruktur auf kosten etwas langsamerer Anfragen — VL 11, 12, 17
- Lazy Evaluation — VL 11
- nutze Zahlen, die eigentlich Teil der zu verarbeitenden Daten sind, zur Adressierung (also als Index in ein Array) — VL 6, 7

1.3.3 Umsetzung im Detail

Wir haben uns regelmäßig damit auseinander gesetzt, die algorithmische Idee im Detail auszuarbeiten und in Pseudocode zu übersetzten. Besonders ausführlich mit Zwischenschritten haben wir das im Kontext der binären Suche, der Breitensuche und von $(2, 3)$-Bäumen gesehen. — VL 4, 8, 13

1.3.4 Korrektheitsbeweis

Für Beweise gibt es kein Kochrezept. Einen Beweis zu führen fordert Kreativität und mathematisches Verständnis. Dennoch haben wir einige Techniken und Tricks kennengelernt, die hier regelmäßig zum Einsatz kommen.

- Invarianten — VL 4, 5, 9, 12, 13, 17
- Induktion — VL 5, 8, 10, 18
- Widerspruchsbeweis und minimales Gegenbeispiel — VL 9
- Austauschargument (Greedy) — VL 16

1.3.5 Laufzeitanalyse

Wie bei Korrektheitsbeweisen gibt es auch hier kein Kochrezept, aber doch einige Werkzeuge, Techniken und Erkenntnisse, die immer wieder zum Einsatz kommen.

- Asymptotische Laufzeit: O-Notation — VL 1
- Rekurrenzen analysieren via Rekursionsbaum — VL 2
- Asymptotik von exponentiellen Summen — VL 2
- amortisierte Analyse — VL 3
- erwartete Laufzeit bei randomisierten Algorithmen — VL 5, 7
- Adversary Argument — VL 7
- Summe der Knotengrade eines Graphen: $\Theta(m)$ — VL 8
- Vereinfachen des Algorithmus vor der Analyse zu einem anderen Algorithmus mit äquivalenter Laufzeit — VL 17

2 Literatur

Die Vorlesung hat inhaltlich große Überschneidung mit vielen Lehrbüchern, wobei es bei jedem Thema natürlich verschiedene Herangehensweisen gibt. Im Folgenden ist für jede Vorlesung aufgelistet, an welchem Buch sie sich am ehesten orientiert. Wenn du also etwas nochmal detaillierter nachlesen möchtest, dann ist das vermutlich der beste Startpunkt. Für alternative Sichtweisen lohnt aber sicher auch ein Blick in die jeweils anderen Bücher zum selben Thema.

Die Bücher sind zum Teil frei oder aus dem Uninetz als PDF abrufbar [2, 3, 4]. Alle anderen Bücher sollten in der Bibliothek verfügbar sein.

[1] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009. URL: `https://mitpress.mit.edu/books/introduction-algorithms-third-edition`.

[2] Jeff Erickson. *Algorithms*. 1st ed. 2019. URL: `http://algorithms.wtf/#book`.

[3] Jeff Erickson. *Lecture Notes – Director's Cut*. URL: `http://algorithms.wtf/#notes`.

[4] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures – The Basic Toolbox*. Springer, 2008. URL: `https://link.springer.com/book/10.1007/978-3-540-77978-0`.

[5] Tim Roughgarden. *Algorithms Illuminated*. 1st ed. Soundlikeyourself Publishing, 2021. URL: `https://www.algorithmsilluminated.org/`.

Vorlesung 1 VL 1

Die O-Notation wird in quasi jedem algorithmischen Textbuch eingeführt. Siehe beispielsweise [4, Kapitel 2.1], [5, Kapitel 2] oder [1, Kapitel 3].

Vorlesung 2 VL 2

Im Kern geht es in der Vorlesung um das Auflösen von (einfachen) Rekurrenzen durch Analyse des Rekurrenzbaums. Auch wenn die Vorlesung nicht auf Basis dessen Entstanden ist, kommt das dem Inhalt in [2, Kapitel 1.7] recht nahe.

Ansonsten gab es hier noch zwei (nicht so relevante) Nebenschauplätze: Karatsubas Algorithmus und das Mastertheorem. Karatsubas Algorithmus hat hier nur als Beispiel gedient, ist aber ansonsten kein besonders essentieller Algorithmus. Wenn du trotzdem mehr dazu lesen möchtest, dann wirst du in [5, Kapitel 1.2] oder [4, Kapitel 1] fündig. Das Mastertheorem ist eine Aussage, die im Prinzip direkt aus der Analyse der Rekurrenzbäume herausfällt. Das Theorem an sich ist nicht besonders spannend oder nützlich; zu wissen, wo es herkommt aber schon. Mehr zum Mastertheorem zum Beispiel in [5, Kapitel 4] oder [4, Kapitel 2.6.2].

Vorlesung 3 VL 3

Unbeschränkte (dynamisch wachsende) Arrays findet ihr in [4, Kapitel 3.2].

Der Teil zur amortisierten Analyse orientiert sich an keinem Buch im Speziellen. Abschnitte zur amortisierten Analyse findest du aber beispielsweise in [3, Kapitel 9], [4, Kapitel 3.3] oder [1, Kapitel 17].

Vorlesung 4 VL 4

Listen werden ausführlich in [4, Kapitel 3.1] behandelt. Die binäre Suche findest du zum Beispiel in [4, Kapitel 2.5].

Vorlesung 5 VL 5

Mergesort wird zum Beispiel in [4, Kapitel 5.2] oder [2, Kapitel 1.4] behandelt. Eine ausführliche Behandlung von Quicksort, inklusive Laufzeitanalyse für zufällige Pivot-Wahl, findest du in [5, Kapitel 5]. Einen Beweis der unteren Schranke für vergleichsbasiertes Sortieren findest du unter anderem in [5, Kapitel 5.6].

Vorlesung 6 VL 6

Bucketsort und (LSD) Radixsort werden unter anderem in [4, Kapitel 5.6] behandelt. Die Folien sind hier aber deutlich ausführlicher.

Vorlesung 7 VL 7

Alles Wissenswerte zu Hashing kannst du im Detail nochmal in [5, Kapitel 12] oder [4, Kapitel 4] nachlesen.

Vorlesung 8 VL 8

Die Vorlesung zu Breitensuche (BFS) orientiert sich an keinem Buch im Speziellen. Die BFS wird unter anderem in [5, Kapitel 8.2], [2, Kapitel 8.4] oder [4, Kapitel 9.1] behandelt. Wie man damit die Zusammenhangskomponenten eines Graphen bekommt steht in [5, Kapitel 8.3].

Vorlesung 9 VL 9

Die Vorlesung zu Dijkstras Algorithmus orientiert sich an keinem Buch im Speziellen. Eine Behandlung des Themas findest du aber zum Beispiel in [5, Kapitel 9], [2, Kapitel 8.6] oder [4, Kapitel 10.3].

Vorlesung 10 VL 10

Der Bellman–Ford sowie der Floyd–Warshall Algorithmus wird in [5, Kapitel 18] behandelt. Weitere Informationen zu Bellman–Ford gibt es in [2, Kapitel 8.7]. Eine deutlich ausführlichere Behandlung des APSP Problems findest du in [2, Kapitel 9].

Vorlesung 11 VL 11

Die Vorlesung zu binären Heaps orientiert sich an keinem Buch im Speziellen. Das Thema wird aber beispielsweise in [5, Kapitel 10] und in [4, Kapitel 6] behandelt.

Vorlesung 12 und 13 VL 12, 13

Sortierte Folgen und insbesondere (a, b)-Bäumen werden in [4, Kapitel 7] behandelt.

Vorlesung 14 VL 14

Die Vorlesung zum Finden von Brücken orientiert sich an keinem Buch im Speziellen. Die DFS wird aber zum Beispiel in [5, Kapitel 8.4] behandelt.

Vorlesung 15 VL 15

Einen Aufschrieb zur Topologischen Sortierung mittels DFS findest du in [5, Kapitel 8.5].

Vorlesung 16 VL 16

Die Algorithmen von Prim und Kruskal findest du zum Beispiel in [5, Kapitel 15].

Vorlesung 17 VL 17

Die Union–Find Datenstruktur findest du in [3, Kapitel 11] unter dem alternativen Namen *Disjoint Sets*. Das Kapitel beinhaltet auch einen Beweis für die amortisierte Laufzeit von $O(\log^* n)$ pro Operation, der dem in der Vorlesung ähnlich ist.

Vorlesung 18 VL 18

Für dynamische Programme ist [2, Kapitel 3] empfehlenswert.

3 Glossar

A

(a, b)-Baum Ein Suchbaum, der das Konzept von sortierten Folgen algorithmisch effizient umsetzt. Ein gewurzelter Baum heißt (a, b)-Baum, wenn er die folgenden zwei Eigenschaften hat. Erstens, jeder innere Knoten (ohne Wurzel) hat mindestens a und höchstens b Kinder. Zweitens, jedes Blatt hat die selbe Tiefe. Die Datenstruktur erlaubt das Einfügen, Suchen und Löschen jeweils in Zeit $\Theta(\log n)$. VL 12

Adjazenzliste Eine Graphdatenstruktur. Erlaubt es insbesondere effizient über die Nachbarn eines Knotens zu iterieren. VL 8

Adjazenzmatrix Eine Graphdatenstruktur. Ermöglicht es insbesondere effizient zu testen, ob zwei Knoten benachbart sind. VL 8

Adversary Argument Wir wollen bei Algorithmen meist den Worst Case über alle möglichen Eingaben analysieren. Daher macht es Sinn sich vorzustellen, dass ein:e Gegner:in (Adversary) den Algorithmus kennt und auf Basis dessen die ungünstigste Eingabe wählt. VL 7

Aggregationsmethode Eine Methode für die amortisierte Analyse von Algorithmen. VL 3

amortisierte Analyse Bei Datenstrukturen kann es vorkommen, dass es zwar einzelne teure Operationen gibt, jede Folge von Operationen aber im Schnitt pro Operation günstige Kosten verursacht. Um dem Rechnung zu tragen, spezifiziert man *amortisierte Kosten* für die einzelnen Operationen. Nutzt man eine solche Datenstruktur in einem Algorithmus, so kann es sein, dass die *tatsächlichen Kosten* einzelner Operationen höher sind als ihre amortisierten Kosten. Da sich das in Summe über die ganze Folge an Operationen aber immer ausgleicht, ist es korrekt für die Gesamtlaufzeit des Algorithmus mit den amortisierten Kosten zu rechnen. VL 3

Damit das so funktioniert muss man beweisen, dass für jede beliebige Folge von Operationen, die Summe der tatsächlichen Kosten höchstens der Summe der amortisierten Kosten entspricht. Zeigt man diese Ungleichung direkt, indem man die Summen ausrechnet, so nennt man das die *Aggregationsmethode*. Diese Methode ist konzeptuell schön einfach, diese Summen explizit auszurechnen ist aber manchmal nicht so leicht; insbesondere, wenn man unterschiedliche Operationen hat, die in beliebiger Reihenfolge auftreten können.

Bei der *Charging-Methode* ist die Idee Kosten-Token von den teuren zu den günstigen Operationen zu verschieben. Man muss dann beweisen, dass am Ende keine Operation zu viele Token hat.

Bei der *Konto-Methode* legt man für jede Operation fest, wie viele Kosten-Token sie aufnehmen bzw. abgeben möchte. Etwas anders formuliert gibt es ein Konto und Operationen können in das Konto einzahlen (sie nehmen zusätzliche Kosten-Token auf) oder vom Konto abheben (sie geben Kosten-Token ab). Die amortisierten Kosten einer Operation sind dann die tatsächlichen Kosten plus die Einzahlung (Abhebung ist negative Einzahlung). Damit das korrekt ist, darf nie mehr abgehoben werden als vorher eingezahlt wurde. Wir müssen also zeigen, dass der Kontostand nie negativ wird.

Bei der *Potential-Methode* legt man nicht die Einzahlung der einzelnen Operationen, sondern den Kontostand in Abhängigkeit vom Zustand der Datenstruktur fest. Den Kontostand für eine Datenstruktur A bezeichnet man auch als *Potential* $\Phi(A)$. Wenn A_{vor} und A_{nach} die Datenstruktur vor bzw. nach einer Operation ist, so hat diese Operation also $\Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}})$ eingezahlt und damit haben wir amortisierte Kosten = tatsächliche Kosten + $\Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}})$. Um die Potentialmethode zu verwenden, muss man nur Φ definieren, zeigen dass Φ nie negativ wird und die amortisierten Kosten jeder Operation entsprechend dieser Formel ablesen.

APSP Das All-Pair Shortest Path Problem. Gegeben einen (gewichten, gerichteten) Graphen $G = (V, E)$, berechne die Distanz $\text{dist}(s, t)$ für alle $s, t \in V$. VL 10

Array Folge konsekutiver Speicherzellen. Mittels Index in konstanter Zeit adressierbar. Beim Erstellen des Arrays muss die Anzahl der gewünschten Speicherzellen festgelegt werden. Siehe auch: dynamisches Array VL 3

Asymptotik Wachstumsverhalten einer Funktion $f(n)$ für $n \to \infty$ unter Ignorierung konstanter Faktoren und Terme tieferer Ordnung; siehe auch O-Notation. VL 1

Austauschargument Eine Beweistechnik für die Korrektheit von Greedy Algorithmen, die in jedem Schritt gierig ein Element x wählen. Für jeden dieser Schritte muss man zeigen, dass diese Wahl nicht falsch war. Dazu beweist man, dass jede VL 16

Lösung ohne x in eine Lösung mit x umgebaut werden kann, ohne dabei die Lösung schlechter zu machen.

Average Case Erwartete Laufzeit, wenn die Laufzeit eine Zufallsvariable ist. Das kann der Fall sein, wenn der Algorithmus zufällige Entscheidungen trifft oder wenn man annimmt, dass die Eingabe zufällig gewählt ist. VL 5, 7

B

Baum Ein *Baum* ist ein zusammenhängender und kreisfreier Graph. Bäume werden oft gewurzelt betrachtet. VL 2, 8

Baum, Blatt Ein Knoten eines Baums heißt *Blatt*, wenn er Grad 1 hat. In einem gewurzelten Baum sind die Blätter die Knoten ohne Kinder.

Baum, Elter Sei (u, v) eine Kante in einem gewurzelten Baum. Dann ist u der *Elter* von v. Man kann leicht einsehen, dass jeder Knoten außer der Wurzel genau einen Elter hat (die Wurzel hat keinen Elter).

Baum, Geschwister Seien u und v zwei unterschiedliche Knoten in einem gewurzelten Baum. Dann sind u und v *Geschwister*, wenn sie den gleichen Elter haben. VL 12

Baum, gewurzelt Ein Baum zusammen mit einem ausgezeichneten Knoten, den wir Wurzel nennen.

Um zusätzliche Begriffe wie Kind oder Elter definieren zu können ist es hilfreich über gewurzelte Bäume als gerichtete Graphen nachzudenken. Dazu richten wir jede Kante so, dass sie von der Wurzel weg zeigt. Etwas formaler, seien $u, v \in V$ zwei adjazente Knoten Baumes. Falls u auf dem Pfad von der Wurzel zu v liegt, so enthält der Baum die gerichtete Kante (u, v). Andernfalls enthält er die gerichtete Kante (v, u).

Beachte: Manchmal ist es auch hilfreich die Kanten in genau die umgekehrte Richtung gerichtet zu betrachten. Für die Definitionen in diesem Dokument verwenden wir aber die oben definierte Konvention.

Baum, Höhe Die Höhe eines gewurzelten Baum ist die Länge des längsten Pfads von der Wurzel zu einem Blatt.

Baum, innerer Knoten Ein Knoten in einem Baum heißt *innerer Knoten*, wenn er kein Blatt ist. Bei gewurzelten Bäumen wird manchmal zusätzlich die Wurzel ausgenommen. VL 12

Baum, Kind Sei (u, v) eine Kante in einem gewurzelten Baum. Dann ist v ein *Kind* von u.

Baum, Nachfolger In einem gewurzelten Baum ist v ein *Nachfolger* von u, wenn es einen gerichteten Pfad von u nach v gibt.

Baum, Tiefe Die *Tiefe* eines Knotens in einem gewurzelten Baum ist seine Distanz zur Wurzel. Beachte: Die Höhe des Baumes ist die maximale Tiefe der Knoten.

Baum, Vorgänger In einem gewurzelten Baum ist u ein *Vorgänger* von v, wenn es einen gerichteten Pfad von u nach v gibt.

Bellman–Ford Algorithmus Ein Algorithmus zur Berechnung von kürzesten Pfaden (SSSP Problem). In jeder Iteration relaxiert der Algorithmus jede Kante. Man kann zeigen, dass man so die korrekten Distanzen nach $n-1$ Iterationen kennt. Damit erhält man eine Laufzeit von $\Theta(n \cdot m)$. Der Algorithmus funktioniert auch bei negativen Kantengewichten. Gibt es negative Kreise, so kann man das mit dem Algorithmus feststellen. VL 10

binäre Suche Algorithmus zum Finden eines Elements in einem sortierten Array in $\Theta(\log n)$ Zeit. Ist das Element nicht enthalten, so kann auch der Vorgänger oder Nachfolger in der sortierten Folge zurückgegeben werden. VL 4

binärer Heap Implementierung einer Priority-Queue, die `push` und `popMin` in $\Theta(\log n)$ erlaubt. VL 11

Breitensuche (BFS) Algorithmus der einen Graphen in $\Theta(n + m)$ traversiert. Die BFS geht dabei lagenweise vor und kann genutzt werden um kürzeste Pfade zu berechnen. VL 8

Bucketsort Algorithmus zum Sortieren von n ganzen Zahlen der Größe m in Zeit $\Theta(n + m)$. Bucketsort ist ein stabilies Sortierverfahren. VL 6

C

Chaining Strategie zur Auflösung von Kollisionen bei Hashtabellen. VL 7

Charging-Methode Eine Methode für die amortisierte Analyse von Algorithmen. VL 3

D

DAG Ein gerichteter Graph der *azyklisch* ist, also keinen gerichteten Kreis enthält. Der Name ist eine Abkürzung für *directed acyclic graph*.

Dijkstras Algorithmus Ein Algorithmus zur Berechnung von kürzesten Pfaden (SSSP Problem) auf Graphen mit nicht-negativen Kantengewichten. Der Algorithmus exploriert in jedem Schritt den Knoten mit minimalem Distanzwert. Dazu werden die Distanzwerte in einer Priority-Queue verwaltet. Abhängig von der genauen Implementierung der Priority-Queue läuft Dijkstras Algorithmus in $\Theta((n + m) \log n)$ oder $\Theta(n \log n + m)$ Zeit. VL 9, 11

dynamisches Array Erweiterung (statischer) Arrays um die Komforfunktionen `pushBack` und `popBack`. Das Array wächst beim Einfügen mittels `pushBack` automatisch mit. Wie bei Arrays können Einträge mittels Index in konstanter Zeit adressieren (wahlfreier Zugriff). Die Operationen `pushBack` und `popBack` haben (amortisiert) konstante Laufzeit. VL 3

dynamisches Programm Eine algorithmische Technik, bei der man für ein Problem Teilprobleme spezifiziert und eine Rekurrenz dafür aufstellt, wie sich Lösungen für größere Teilprobleme aus Lösungen von kleinen Teilproblemen ergeben. Um nicht das selbe Teilproblem mehrfach zu lösen berechnet man die Teillösungen dann iterativ und speichert die Zwischenergebnisse („Tabelle ausfüllen“). VL 10, 18

E

explorieren Ein Begriff, der bei der Berechnung von kürzesten Pfaden (SSSP Problem) vorkommt. Einen Knoten zu explorieren bedeutet alle seine inzidenten Kanten zu relaxieren. VL 9

exponentielle Summe Eine Summe der Form $\sum_i c^i$ für eine Konstante $c \in \mathbb{R}^+$. Aus der Konvergenz der geometrischen Reihe erhält man, dass eine solche Summe asymptotisch immer durch ihren größten Summanden dominiert wird, außer wenn $c = 1$ (in dem Fall ist jeder Summand gleich). Für $0 < a \leq b$ gilt also VL 2

$$\sum_{i=a}^{b} c^i \in \begin{cases} \Theta(c^a) & \text{wenn } c < 1, \\ \Theta(c^b) & \text{wenn } c > 1, \\ \Theta(b-a) & \text{wenn } c = 1. \end{cases}$$

Meist tauchen solche Summen bei Laufzeiten auf, die von einer Eingabegröße n abhängen. In diesem Fall ist c konstant (unabhängig von n), a und b können aber von n abhängen.

F

Floyd–Warshall Algorithmus Ein Algorithmus zur Berechnung von kürzesten Pfaden zwischen allen Knotenpaaren (APSP Problem). Es ist ein dynamisches Programm über die Menge der Knoten, die als Zwischenkonten auf den Pfaden auftauchen dürfen. Der Algorithmus hat Laufzeit $\Theta(n^3)$. VL 10

G

Graph Ein Graph $G = (V, E)$ besteht aus einer Knotenmenge V und Kantenmenge E, die eine binäre Relation auf V modelliert. Typischerweise bezeichnen wir die Anzahl Knoten und Kanten mit $n = |V|$ und $m = |E|$. VL 8

Es gibt verschiedene Arten von Graphen, insbesondere ungerichtete und gerichtete. Wenn nicht anders angegeben nehmen wir implizit an, dass der betrachtete Graph ungerichtet und einfach ist. Die hier eingeführten Begrifflichkeiten übertragen sich meist kanonisch auf gerichtete Graphen unter Ignorierung der Kantenrichtung. Manchmal gibt es aber auch zusätzliche davon abweichende Definitionen für gerichtete Graphen.

Graph, adjazent Zwei Knoten $u, v \in V$ heißen *adjazent*, wenn $\{u, v\} \in E$. VL 8

Graph, Brücke Sei $G = (V, E)$ ein Graph. Eine Kante $e = \{v, u\}$ heißt *Brücke*, wenn u und v in $G - e$ in verschiedenen Zusammenhangskomponenten liegen. Dabei ist $G - e$ eine Kurzschreibweise für den Graphen $(V, E \setminus \{e\})$. VL 14

Graph, Distanz Die *Distanz* $\text{dist}(s, t)$ zwischen zwei Knoten $s, t \in V$ ist die Länge des kürzesten Pfades mit Startknoten s und Endknoten t. Auf gerichteten Graphen interessiert man sich hier meist für gerichtete Pfade. VL 8

Falls es keinen Pfad von s nach t gibt, so ist es meist am sinnvollsten formal $\text{dist}(s, t) = \infty$ zu definieren.

Graph, einfach Falls die selbe Kante $\{u, v\}$ mehrfach in E vorkommt (E ist also eine Multimenge), so nennen wir $\{u, v\}$ eine *Mehrfachkante*. Eine Kante der Form $\{v, v\}$ (dazu müssen wir erlauben, dass die Kanten selbst Multimengen sind) heißt *Schleife*. Ein Graph ohne Mehrfachkanten und ohne Schleifen heißt *einfach*. Einen nicht-einfachen Graphen nennt man auch *Multigraph*. VL 8

Wenn nicht explizit anders angegeben, dann nehmen wir an, dass ein gegebener Graph einfach ist.

Graph, gerichtet In einem gerichteten Graphen ist $E \subseteq V \times V$. Eine Kante ist also ein Tupel (u, v) mit $u, v \in V$. Wir sagen dann auch, dass (u, v) für u eine *ausgehende* und für v eine *eingehende* Kante ist. VL 8

Graph, inzident Für eine Kante $e = \{u, v\} \in E$ sagen wir, dass u und v *inzident* zu e sind. VL 8

Graph, Knotengrad Der *Grad* eines Knotens v ist die Größe $|N(v)|$ seiner Nachbarschaft. VL 8

In gerichteten Graphen unterscheidet man auch zwischen *Eingangs-* und *Ausgangsgrad*. Der Eingangsgrad zählt die eingehenden und der Ausgangsgrad nur die ausgehenden Kanten.

Graph, Kreis Eine Knotenfolge $K = \langle v_1, \ldots, v_\ell \rangle$ ist ein *Kreis*, wenn K ein Pfad ist und zusätzlich $\{v_\ell, v_1\} \in E$. In gerichteten Graphen sprechen wir von einem *gerichteten Kreis* oder auch *Zyklus*, wenn K ein gerichteter Pfad ist und $(v_\ell, v_1) \in E$. VL 8

Die *Länge* des Kreises K ist ℓ. Ein Kreis der Länge 3 ist ein *Dreieck*.

Graph, Nachbarschaft Die Nachbarschaft eines Knotens v ist die Menge zu v adjazenter Knoten. Wir bezeichnen diese Menge auch mit $N(v)$. VL 8

Graph, Pfad Ein *Pfad* in einem Graphen ist eine Knotenfolge $\langle v_0, \ldots, v_\ell \rangle$ mit $\{v_{i-1}, v_i\} \in E$ für alle $i \in \{1, \ldots, \ell\}$. Die *Länge* des Pfades ist ℓ. Die Knoten v_0 und v_ℓ sind die *Start-* und *Endknoten* des Pfades. Sie werden auch oft mit $s = v_0$ und $t = v_\ell$ bezeichnet. In gerichteten Graphen ist $\langle v_0, \ldots, v_\ell \rangle$ ein *gerichteter Pfad*, wenn $(v_{i-1}, v_i) \in E$. VL 8

Hat man zu dem Graphen noch Kantengewichte $\text{len}\colon E \to \mathbb{Z}$ gegeben, dann ist $\sum_{i=1}^{k} \text{len}(\{v_{i-1}, v_i\})$ die *Länge* des Pfades $\langle v_0, \ldots, v_\ell \rangle$. VL 9

Graph, Quelle Ein Knoten in einem gerichteten Graphen, der keine eingehende Kante hat.

Graph, Senke Ein Knoten in einem gerichteten Graphen, der keine ausgehende Kante hat.

Graph, ungerichtet In einem ungerichteten Graphen ist $E \subseteq \binom{V}{2}$. Eine Kante ist also eine Menge $\{u, v\}$ bestehend aus zwei Knoten $u, v \in V$. VL 8

Graph, Zusammenhang Für zwei Knoten $u, v \in V$ sagen wir, dass v von u *erreichbar* ist, wenn es einen Pfad von u nach v gibt. Beachte, dass die Erreichbarkeit eine Äquivalenzrelation auf der Knotenmenge ist. Die Äquivalenzklassen dieser Relation sind die *Zusammenhangskomponenten* (kurz: Komponenten) des Graphen. VL 8

Ein Graph heißt *zusammenhängend*, wenn er nur aus einer Komponente besteht. Da unzusammenhängende Graphen aus algorithmischer Sicht meist nicht besonders spannend sind (man kann mit einer BFS die Komponenten finden und sie dann getrennt behandeln), nehmen wir oft implizit an, dass ein gegebener Graph zusammenhängend ist.

Nebenbemerkung: Betrachtet man bei gerichteten Graphen Erreichbarkeit bezüglich gerichteter Pfade so kann man das Konzept der starken Zusammenhangskomponenten erhalten (nicht Stoff der Vorlesung).

Greedy Algorithmus Ein *Greedy Algorithmus* ist ein Algorithmus, der in jedem Schritt die zu diesem Zeitpunkt scheinbar beste Entscheidung trifft. Oft liefern Greedy Algorithmen nicht die korrekte (bzw. nicht die beste) Lösung. Wenn doch, dann kann man das typischerweise mit einem Austauschargument beweisen. VL 16

H

Hashtabelle Eine Datenstruktur die Schlüssel–Wert Paare verwaltet, indem der Schlüssel mittels einer Hashfunktion auf eine ausreichend kleine natürlich Zahl (Hash) abgebildet wird. Dieser Hash kann dann als Index in ein Array genutzt werden. Solange es nicht zu viele Kollisionen gibt (also unterschiedliche Schlüssel mit dem selben Hash), kann so mithilfe des Schlüssel schnell auf den zugehörigen Wert zugegriffen werden. Bei Wahl einer geeigneten Hashfunktion erhält man erwartet konstante Laufzeit für Einfüge-, Such- und Löschoperationen. VL 7

I

Invariante Eine Eigenschaft die bei der Ausführung eines Algorithmus erhalten bleibt. Eine geschickt gewählte Invariante kann dabei helfen die Korrektheit eines Algorithmus zu beweisen. VL 4, 5, 9, 12, 13, 17

J

K

Konto-Methode Eine Methode für die amortisierte Analyse von Algorithmen. VL 3

Kruskals Algorithmus Ein Algorithmus zur Berechnung eines minimalen Spannbaums (MST, minimum spanning tree). Der Algorithmus sortiert die Kanten nach Gewicht (nicht-absteigend) und wählt Kanten in dieser Reihenfolge, wobei Kanten übersprungen werden, die einen Kreis schließen. Um schnell zu überprüfen, ob eine Kante einen Kreis schließen würde, verwalten wir die Zusammenhangskomponenten bezüglich bereits gewählter Kanten in einer Union–Find Datenstruktur. Die Laufzeit ist $\Theta(m \log n)$ (dominiert durch das Sortieren der Kanten). VL 16

L

Lage Lagen betrachtet man oft in gewurzelten Bäumen oder auch allgemein in beliebigen Graphen. Sei s die Wurzel eines Baumes oder allgemeiner ein beliebiger aber fester Knoten in einem Graphen. Ein Knoten $v \in V$ ist in *Lage* i (bezüglich s), wenn $\text{dist}(s, v) = i$. VL 2, 8

Landau-Notation Siehe O-Notation.

Lazy Evaluation Ein Trick um Änderungs- und Löschoperationen in Datenstrukturen effizient umzusetzen. Dabei werden die zu entfernenden Elemente nur als gelöscht markiert, ohne sie tatsächlich zu löschen. Entsprechend muss man später damit umgehen, dass die Datenstruktur einem ggf. eigentlich gelöschte Elemente liefert. VL 11

lexikographische Ordnung Ordnung nach mehreren Kriterien absteigender Wichtigkeit, wie beispielsweise bei einem Medaillenspiegel. Etwas formaler, sei M eine geordnete Menge und seien $a, b \in M^k$ mit $a = (a_1, \ldots, a_k)$ und $b = (b_1, \ldots, b_k)$ zwei unterschiedliche Vektoren. Sei i der kleinste Index, sodass $a_i \neq b_i$. Dann ist $a < b$ wenn $a_i < b_i$ und $a > b$ wenn $a_i > b_i$. Die lexikographisch Ordnung ist also eine Ordnung auf M^k. VL 6

Liste Einfache verzeigerte Datenstruktur, die eine Folge von Elementen speichert. Hat man einen Zeiger auf ein Listenelement, so kann man an dieser Stelle schnell einfügen oder löschen. Auch kompliziertere Umbauoperationen wie `splice` sind möglich. Listen erlauben *keinen* wahlfreien Zugriff mittels Index. VL 4

M

Master-Theorem Ein Theorem mit einem beeindruckenden Namen zum Auflösen von Rekurrenzen. VL 2

Mergesort Vergleichsbasierter Sortieralgorithmus basierend auf dem Teile und Herrsche Prinzip mit Laufzeit $\Theta(n \log n)$. VL 5

minimales Gegenbeispiel Beweistechnik basierend auf einem Widerspruchsbeweis. Bei Widerspruchsbeweisen nimmt man an, die zu zeigende Aussage gilt nicht und führt diese Annahme zum Widerspruch. Dabei ist es oft hilfreich, sich ein Gegenbeispiel für die zu zeigende Aussage (das aufgrund der Annahme existiert) in die Hand zu nehmen. Es kommt dann regelmäßig vor, dass man sich von Gegenbeispiel zu Gegenbeispiel hangeln muss, bis man einen tatsächlichen Widerspruch gefunden VL 9

hat. Dieses Problem kann man oft lösen, indem man ein Gegenbeispiel annimmt, das in einer bestimmten Weise minimal ist.

N

O

***O*-Notation** Die *O-Notation*, auch *Laundau-Notation*, ist ein Werkzeug zur Untersuchung der Asymptotik von Funktionen. Sie ignoriert konstante Faktoren und Terme niedriger Ordnung. Die O-Notation ist ein zentrales Werkzeug für die Laufzeitanalyse von Algorithmen. VL 1

P

Pfadkompression Eine Strategie, die bei einer Union–Find Datenstruktur beim Ausführen von `find` alle auf dem Pfad zur Wurzel betrachteten Knoten anschließend an die Wurzel hängt. Das verringert die Tiefe dieser Knoten und damit die Laufzeit zukünftiger `find`s.

Potential-Methode Eine Methode für die amortisierte Analyse von Algorithmen. VL 3

Prims Algorithmus Ein Algorithmus zur Berechnung eines minimalen Spannbaums (MST, minimum spanning tree). Der Algorithmus startet mit einem einzelnem Knoten von dem aus der Spannbaum Schritt für Schritt wächst, wobei in jedem Schritt eine möglichst kleine Kante hinzugefügt wird. Die nächste Kante kann dabei effizient bestimmt werden, indem man die noch unbetrachteten Knoten in einer Priority-Queue verwaltet. VL 16

Der resultierende Algorithmus hat sehr große Ähnlichkeit zu Dijkstras Algorithmus und läuft, abhängig von der genauen Implementierung der Priority-Queue, in $\Theta((n+m)\log n)$ oder $\Theta(n \log n + m)$ Zeit.

Priority-Queue Eine Datenstruktur, die Elemente zusammen mit einer Priorität verwaltet, sodass man zu jedem Zeitpunkt das Element minimaler Priorität effizient entfernen kann. Konkret werden die Operationen `push`, `popMin` und manchmal auch `decPrio` unterstützt. `push` fügt ein Element mit einer gegebenen Priorität ein. `popMin` gibt das Element mit kleinster Priorität zurück und löscht es aus der Datenstruktur. `decPrio` verringert die Priorität eines Elements (ähnlich wie beim Löschen in einer Liste muss man dazu meist einen Zeiger auf den entsprechenden Knoten in der Datenstruktur kennen). Eine mögliche Implementierung ist der binäre Heap. VL 11

Q

Queue Datenstruktur, die Elemente nach dem FIFO-Prinzip (First In – First Out) verwaltet. Sie unterstützt die Operationen `pushBack` und `popFront`. Nicht zu verwechseln mit einer Priority-Queue. VL 4

Quicksort Vergleichsbasierter Sortieralgorithmus basierend auf dem Teile und Herrsche Prinzip mit erwarteter Laufzeit $\Theta(n \log n)$. VL 5

R

Radixsort Ein Sortieralgorithmus der n Zahlen polynomieller Größe in $\Theta(n)$ sortieren kann. Konkret haben wir die Variante LSD (least significant digit) kennen gelernt. VL 6

Rekurrenz Eine rekursiv definierte Funktion. Im Kontext der Vorlesung beschrieben diese Funktionen meist die Laufzeit rekursiver Algorithmen. Beispiel: Für einen Algorithmus sei $T(n)$ die Laufzeit für eine Instanz der Größe n mit VL 2

$$T(n) = \begin{cases} \Theta(1) & \text{wenn } n = 1, \\ 3 \cdot T(\frac{n}{2}) + \Theta(n) & \text{wenn } n > 1. \end{cases}$$

Um Rekurrenzen aufzulösen (also eine Formel in geschlossener Form zu finden) haben wir den Rekursionsbaum analysiert.

Rekursionsbaum Ein rekursiver Algorithmus (sowie die dazugehörige Rekurrenz) definieren auf natürliche Art und Weise eine gewurzelte Baumstruktur. Startend bei einem Knoten mit Größe n beschriebt die Rekurrenz wie viele Kinder welcher Größe dieser Knoten hat. Der Basisfall (typischerweise für kleine n) definiert die Blätter des Baumes. Die obige Beispiel-Rekurrenz beschreibt also einen Baum bei dem jeder Knoten drei Kinder hat und jedes Kind halb so groß ist, wie der Elter. VL 2

Um eine Rekurrenz mit der Baumsichtweise aufzulösen haben wir meist zunächst die folgenden Fragen beantwortet: Wie viele Knoten befinden sich auf Lage i des Baumes? Wie groß ist das n für Knoten der Lage i? Wie viel Zeit kostet ein Knoten in Lage i?

relaxieren Ein Begriff, der bei der Berechnung von kürzesten Pfaden (SSSP Problem) vorkommt. Dabei hat man für jeden Knoten v eine Schätzung $d[v]$ für die Distanz $\text{dist}(s, v)$ vom Startknoten s zu v gespeichert. Eine Kante (u, v) zu relaxieren bedeute, dass man überprüft ob $d[v] > d[u] + \text{len}(u, v)$ und falls ja, dann wird $d[v] = d[u] + \text{len}(u, v)$ gesetzt. VL 9, 10

Dabei kann man einsehen, dass das Relaxieren einer Kante die Invariante $\text{dist}(s, v) \leq d[v]$ erhält. Die Schätzung $d[v]$ ist also eine obere Schranke für die tatsächliche Distanz.

S

Schnitt Sei $G = (V, E)$ ein Graph. Ein *Schnitt* ist eine Zerlegung von G in zwei nicht-leere Teilmengen S und $T = V \setminus S$. Eine Kante $\{s, t\} \in E$ mit $s \in S$ und $t \in T$ heißt *Schnittkante*. VL 16

sortierte Folge Eine *sortierte Folge* ist eine Datenstruktur, die vergleichbare Elemente sortiert vorhält. Dabei möchte man schnell einfügen, suchen und löschen können. Häufig werden sortierte Folgen als Suchbäume umgesetzt. Wir haben konkret die (a, b)-Bäume kennen gelernt. VL 12

Im Kontext von sortierten Folgen betrachtet man meist Schlüssel–Wert Paare. Das heißt, der Wert ist das Objekt, für das wir uns eigentlich interessiert, es wird aber nach dem Schlüssel sortiert. In dem Fall kann die sortierte Folge als Abbildung von Schlüsseln zu Werten gesehen werden. Entsprechend sieht das Interface oft so aus,

dass man die folgenden Operationen hat. `set`(k, v) um den Wert für den Schlüssels k auf v zu setzen. `find`(k) um den Wert zu bekommen, der für den Schlüssel k gesetzt wurde. `remove`(k) um das Element zu entfernen, das zum Schlüssel k gehört.

Spannbaum Sei $G = (V, E)$ ein Graph. Ein *Spannbaum* von G ist ein Baum $T = (V, E_T)$ mit $E_T \subseteq E$. Beachte, dass G und T die selbe Knotenmenge haben. VL 16

stabilies Sortierverfahren Ein Sortierverfahren heißt *stabil*, wenn es die relative Ordnung von gleichen Elementen in der Eingabe erhält. Formal, sei A eine Menge mit einer schwachen Ordnung $\leq$. Wendet man ein stabiles Sortierverfahren auf die Folge $\langle a_1, \ldots, a_n \rangle$ mit $a_i \in A$ an, so gilt für alle $1 \leq i < j \leq n$ mit $a_i = a_j$, dass a_i in der Ausgabe vor a_j steht. VL 6

SSSP Das Single-Source Shortest Path Problem. Gegeben einen (gewichten, gerichteten) Graphen $G = (V, E)$ und einen Startknoten $s \in V$, berechne die Distanz $\text{dist}(s, t)$ für alle $t \in V$. VL 9

Stack Datenstruktur, die Elemente nach dem LIFO-Prinzip (Last In – First Out) verwaltet. Sie unterstützt die Operationen `pushBack` und `popBack`. VL 4

T

Teile und Herrsche Eine algorithmische Technik, bei der man ein großes Problem in mehrere kleinere disjunkte Teilprobleme zerlegt, Lösungen für die Teilprobleme bestimmt und diese Lösungen kombiniert, um eine Lösung für das große Problem zu erhalten. VL 2, 5

Tiefensuche (DFS) Algorithmus der einen Graphen in $\Theta(n + m)$ traversiert. Die DFS läuft dabei möglichst lange von Knoten zu Knoten, bis alle Nachbarn des aktuellen Knotens schon besucht wurden. Dann läuft die Suche auf dem selben Weg, den sie gekommen ist, so weit zurück, bis es wieder einen unbesuchten Nachbarn gibt. VL 14, 15

topologische Sortierung Sei $G = (V, E)$ ein gerichteter Graph. Eine *topologische Sortierung* ist eine totale Ordnung $\prec$ auf V, sodass $(u, v) \in E$ impliziert dass $u \prec v$. Eine topologische Sortierung existiert genau dann, wenn G ein DAG ist. VL 15

U

Union by Rank Eine Umsetzung der `union` Operation in einer Union–Find Datenstruktur. Dabei wird immer die Wurzel des Baums mit geringerem Rang als Kind von der Wurzel des Baumes mit höherem Rang eingefügt. Haben beide Wurzeln den selben Rang, so wird der Gleichstand beliebig aufgelöst und der Rang der Wurzel des resultierenden Baumes wird um 1 erhöht. Jeder Knoten startet mit Rang 0.

Union–Find Eine Datenstruktur, die eine Menge von disjunkten Mengen verwaltet; daher manchmal auch als *disjoint sets* bezeichnet. Dabei hat man zwei Operationen `union` und `find` zur Verfügung. Für zwei Elemente a und b vereinigt `union`(a, b) die Mengen, die a und b enthalten. Für ein Element a liefert `find`(a) einen eindeutigen

Vertreter der Menge, die a enthält. Das heißt, zwei Elemente a und b sind in der selben Menge genau dann wenn $\texttt{find}(a) = \texttt{find}(b)$.

Umgesetzt haben wir das mit einer Waldrepräsentation, in der jeder Baum eine der Mengen darstellt. Mit Union by Rank und Pfadkompression erhält man eine amortisierte Laufzeit von $\Theta(\alpha(n))$ für jede Operation. Dabei ist $\alpha(n)$ die inverse Ackermannfunktion. Wir haben eine obere Schranke von $O(\log^* n)$ pro Operation (amortisiert) bewiesen.

V

W

Worst Case Betrachtung der für den Algorithmus ungünstigsten Eingabe. Bei Laufzeitanalysen nehmen wir meist implizit den Worst Case an. Das heißt, eine Aussage der Form „der Algorithmus benötigt $\Omega(n^2)$ Schritte“ bedeutet, dass es eine Eingabe gibt, sodass die Laufzeit mindestens quadratisch ist.

X

Y

Z