Übungsblatt: Algorithmen

Übungsaufgaben zur Digitalisierung und Programmierung

Prof. Dr. Nicolas Meseth

Zusammenfassung

Algorithmen sind ein fundamentales Konzept der Informatik und der digitalen Problemlösung. In diesem Kapitel befassen wir uns mit den grundlegenden Aspekten von Algorithmen und beantworten dabei zentrale Fragen:

- Was ist ein Algorithmus und wie grenzt er sich von einem Computerprogramm ab?
- Wie könne wir Algorithmen darstellen?
- Welche verschiedenen Arten von Algorithmen existieren und durch welche Beispiele lassen sie sich veranschaulichen?
- Wie können wir Algorithmen systematisch und präzise formulieren?
- Welche Herausforderungen begegnen uns beim Entwurf von Algorithmen?
- Nach welchen Kriterien bewerten wir die Effizienz und Eignung verschiedener Algorithmen für spezifische Problemstellungen?

Was ist ein Algorithmus?

Herkunft des Begriffs

Der Begriff *Algorithmus* hat eine interessante geschichtliche Herkunft. Er leitet sich vom Namen eines persischen Mathematikers des 9. Jahrhunderts ab: *Abu Dscha^{c}far Muhammad ibn Mūsā al-Chwārizmī*. Dieser Gelehrte schrieb ein Buch über Rechenverfahren mit den damals neuen indisch-arabischen Ziffern. In lateinischer Übersetzung des Werks wurde aus seinem Namen "Algoritmi", woraus sich das Wort *Algorismus* und schließlich *Algorithmus* entwickelte. Ursprünglich bezeichnete *Algorismus* tatsächlich die schriftlichen Rechenverfahren (Addition, Subtraktion etc.) mit arabischen Ziffern. Heutzutage verstehen wir unter einem Algorithmus aber jedes Verfahren, das nach festen Regeln abläuft – nicht nur Rechenverfahren. Die historische Anekdote zeigt jedoch, dass Algorithmen seit jeher mit systematischem **Problemlösen** und **Rechenvorschriften** verbunden sind.

Heute bezeichnet ein **Algorithmus** eine präzise, endliche Abfolge von Anweisungen, die ein bestimmtes Problem lösen oder eine Aufgabe erfüllen sollen. Im Alltag begegnen uns Algorithmen ständig, oft, ohne dass wir es merken: beim Kochen, bei der Wegbeschreibung oder beim Aufbau eines IKEA-Regals.

Algorithmen und Programme

Ein wichtiger Aspekt von Algorithmen ist ihre Universalität: Sie sind nicht an Computer gebunden. Ein Algorithmus ist im Kern eine strukturierte Anleitung zur Problemlösung, unabhängig davon, wer oder was diese Anleitung ausführt. Diese Flexibilität zeigt sich besonders deutlich in unserem Alltag, wo wir ständig algorithmische Anleitungen befolgen - sei es beim Aufbau eines Möbelstücks oder beim Kochen nach einem Rezept. Bei diesen Tätigkeiten führen wir Menschen die algorithmischen Schritte aus, ganz ohne Beteiligung eines Computers:

- Kochen: Ein Rezept ist ein Algorithmus für die Zubereitung eines Gerichts.
- Wegbeschreibung: Eine Schritt-für-Schritt-Anleitung, um von Punkt A nach Punkt B zu gelangen.
- Bastelanleitung: Die Anweisungen, um ein Modellflugzeug zusammenzubauen.

Viele Algorithmen können von Computern ausgeführt werden. Dafür ist jedoch eine Übersetzung in eine maschinenverständliche Form notwendig. Diese Übersetzung erfolgt durch das Programmieren, wobei wir den Algorithmus in einer Programmiersprache formulieren. Um die Beziehung zwischen Algorithmen und Computerprogrammen besser zu verstehen, ist es hilfreich, drei zentrale Begriffe zu unterscheiden:

- **Algorithmus**: Die abstrakte Beschreibung einer Lösungsmethode in Form einer präzisen, endlichen Sequenz von individuellen Anweisungen. Ein Algorithmus ist unabhängig von der konkreten Umsetzung und kann sowohl von Menschen als auch von Maschinen ausgeführt werden.
- **Programm**: Die konkrete Implementation eines oder mehrerer Algorithmen in einer Programmiersprache. Das Programm übersetzt die abstrakten Anweisungen des Algorithmus in eine Form, die ein Computer verstehen und ausführen kann.
- **Prozess**: Die tatsächliche Ausführung eines Programms durch einen Computer. Dabei werden die programmierten Anweisungen Schritt für Schritt abgearbeitet, um das gewünschte Ergebnis zu erzielen.

Im Folgenden konzentrieren wir uns zunächst auf das Konzept des Algorithmus an sich. Die praktische Implementierung in Form von Programmen werden wir später am Beispiel der Programmiersprache Python kennenlernen. Um Algorithmen jedoch bereits jetzt systematisch beschreiben und analysieren zu können, benötigen wir geeignete Darstellungsformen und Notationen.

Wie stellen wir Algorithmen dar?

Natürliche Sprache

Die natürliche Sprache bietet eine intuitive Möglichkeit, Algorithmen zu beschreiben. Ein klassisches Beispiel hierfür sind Kochrezepte, die als informelle algorithmische Beschreibungen verstanden werden können. Diese Art der Darstellung folgt keinen festgelegten Regeln - jeder Autor kann die Anweisungen nach eigenem Ermessen formulieren. Entscheidend ist dabei nur, dass andere Menschen die Beschreibung *lesen* und *verstehen* können.

Für die professionelle Informatik ist diese informelle Darstellungsform jedoch nur bedingt geeignet. Während handschriftliche oder natürlichsprachliche Notizen für erste Entwürfe und Skizzen durchaus nützlich sein können, erfordern die präzise Dokumentation und der fachliche Austausch über Algorithmen eine formellere Notation.

Die natürliche Sprache weist für die präzise Beschreibung von Algorithmen drei wesentliche Einschränkungen auf:

Erstens ist sie inhärent mehrdeutig: Ein und derselbe Satz kann je nach Kontext und Interpretation unterschiedliche Bedeutungen haben. Dies steht im Widerspruch zu der Eindeutigkeit, die Algorithmen erfordern. Das Wort “Bank” ist ein klassisches Beispiel für Mehrdeutigkeit im Deutschen. Es kann sich auf eine Sitzgelegenheit zum Ausruhen oder auf ein Finanzinstitut beziehen. Der Satz “Gehe zur Bank” könnte also zwei völlig unterschiedliche Bedeutungen haben - soll Geld abgehoben oder sich hingesetzt werden?

Zweitens setzt die natürliche Sprache oft implizites Wissen und Erfahrung voraus. Ein Kochrezept kann beispielsweise die Anweisung “Backe den Kuchen, bis er goldbraun ist” enthalten. Für einen Algorithmus ist diese Anweisung problematisch, da sie keine messbaren Kriterien enthält und stattdessen menschliches Urteilsvermögen voraussetzt. Ein Algorithmus benötigt exakte Angaben, wie zum Beispiel eine Zeitdauer und eine Temperatur. Eine bessere Version im Sinne eines Algorithmus wäre demnach: “Backe den Kuchen bei 180° C im vorgeheizten Ofen für 80 Minuten.”.

Drittens neigt die natürliche Sprache zu Weitschweifigkeit und Redundanz. Dies erschwert die präzise und effiziente Kommunikation algorithmischer Abläufe. Um diese Herausforderungen zu bewältigen, haben sich in der Informatik zwei formellere und präzisere Darstellungsformen etabliert: Pseudocode und Flussdiagramme.

Pseudocode

Pseudocode ist eine strukturierte, programmiersprachenähnliche Notation zur Beschreibung von Algorithmen. Er ist eine Art Zwischenlösung zwischen natürlicher Sprache und Programmiersprache und kombiniert Elemente der natürlichen Sprache mit grundlegenden Programmierkonzepten wie Schleifen, Bedingungen und Funktionen. Der Vorteil des Pseudocodes liegt

in seiner Präzision und Klarheit, ohne dabei an die strengen syntaktischen Regeln einer echten Programmiersprache gebunden zu sein.

Ein wichtiges Merkmal des Pseudocodes ist seine Flexibilität: Er kann je nach Bedarf formeller oder informeller gestaltet werden, solange die grundlegende Logik und Struktur des Algorithmus klar erkennbar bleiben. Dabei werden häufig standardisierte Schlüsselwörter wie “IF”, “THEN”, “WHILE” oder “REPEAT” verwendet, die die algorithmische Struktur verdeutlichen.

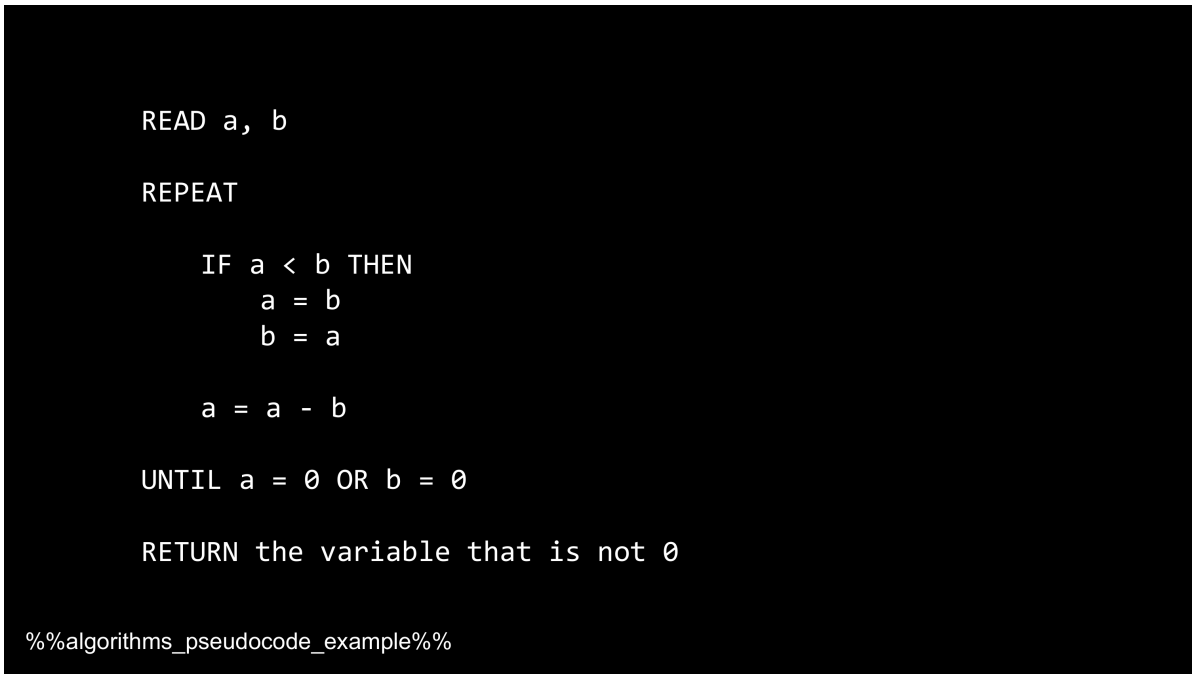


Abbildung 1: Der Euklid’sche Algorithmus als Pseudocode.

Flussdiagramme

Flussdiagramme bieten eine visuelle Darstellung von Algorithmen durch standardisierte grafische Symbole und Verbindungslinien. Diese Notation ist besonders hilfreich, um den Ablauf eines Algorithmus und die logischen Verzweigungen auf einen Blick zu erfassen. Abbildung 2 zeigt die wichtigsten Elemente eines Flussdiagramms: Start- und Endpunkte, Ein- und Ausgaben (Parallelogramm), Anweisungen (Rechteck), Entscheidungen (Rauten), Wiederholungen (Sechseck).

Dazu kommen Verbindungspfeile, die den Kontrollfluss anzeigen. Diese sind im Beispiel in Abbildung 3 zu sehen.

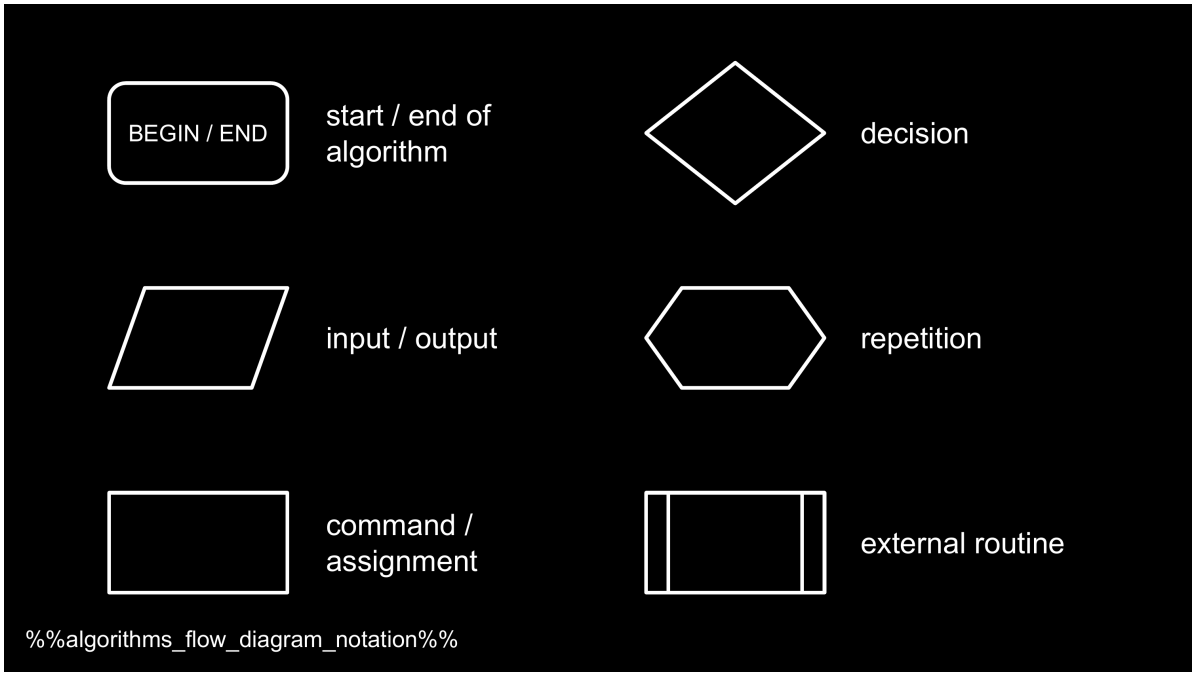


Abbildung 2: Die Notationselemente des Flussdiagramms.

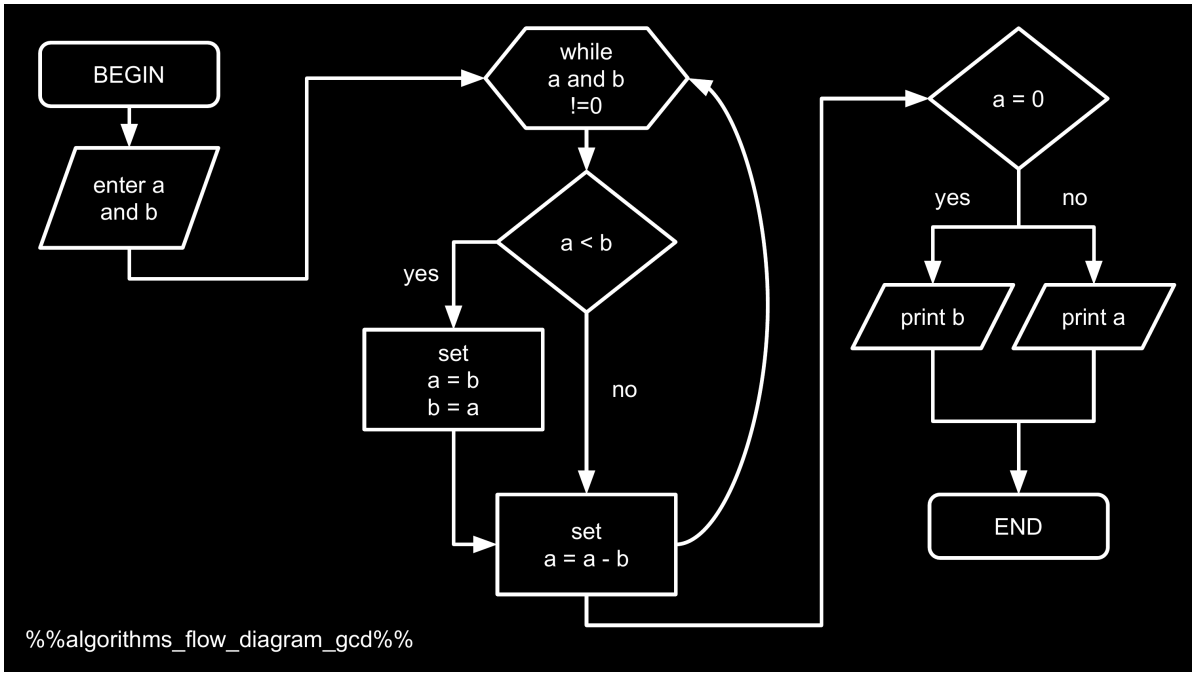


Abbildung 3: Der Euklid'sche Algorithmus als Flussdiagramm.

Eine Alternative zu Flussdiagrammen sind die Struktogramme, die auch Nassi-Shneiderman-Diagramme genannt werden. Diese sind insbesondere im akademischen Bereich poulär, werden hier aber nicht weiter betrachtet.

Programmcode

Die praktische Umsetzung eines Algorithmus erfolgt typischerweise als Computerprogramm, das in einer spezifischen Programmiersprache wie Python, Java oder C++ geschrieben wird. Der resultierende **Code** stellt eine maschinenausführbare Repräsentation des Algorithmus dar. Dabei ist zu beachten, dass der Code strikt an die Syntax und Regeln der jeweiligen Programmiersprache gebunden ist. Derselbe Algorithmus kann in verschiedenen Programmiersprachen implementiert werden, wobei die konkreten Programme sehr unterschiedlich aussehen können. Diese Unterscheidung führt zu einer wichtigen konzeptionellen Trennung, die wir oben bereits gesehen haben: Der Algorithmus beschreibt die abstrakte Lösungsstrategie, während das Programm die spezifische technische Realisierung in einer bestimmten Programmiersprache darstellt.

Welche Arten von Algorithmen gibt es?

Algorithmen können nach ihrer grundlegenden Vorgehensweise oder ihrem Anwendungsbereich in verschiedene Kategorien eingeteilt werden. Jede Kategorie repräsentiert einen spezifischen Problemlösungsansatz:

- **Mathematische Algorithmen**: Berechnen oder approximieren Werte
- **Suchalgorithmen**: Finden bestimmte Elemente in einer Datenmenge
- **Sortieralgorithmen**: Ordnen Daten nach bestimmten Kriterien
- **Optimierungsalgorithmen**: Finden die bestmögliche Lösung für ein Problem
- **Graphenalgorithmen**: Arbeiten mit vernetzten Strukturen
- **Stochastische Algorithmen**: Verwenden Zufallselemente, um ein Problem zu lösen
- **Maschinelle Lernalgorithmen**: Erkennen Muster und treffen Vorhersagen

Diese Kategorien sind weder vollständig noch strikt voneinander getrennt. Viele Algorithmen lassen sich mehreren Kategorien zuordnen. Ein anschauliches Beispiel hierfür ist der **Dijkstra-Algorithmus**, der die kürzeste Route zwischen zwei Punkten findet. Er ist sowohl ein Graphenalgorithmus, da er auf vernetzten Strukturen arbeitet, als auch ein Optimierungsalgorithmus, da er die optimale (kürzeste) Route ermittelt.

Im folgenden beleuchten wir ein oder mehr Beispiele für jeder der genannten Klassen.

Mathematische Algorithmen

Größter gemeinsamer Teiler (GGT)

Der Algorithmus zur Berechnung des größten gemeinsamen Teilers (GGT) ist ein klassisches Beispiel für einen eleganten mathematischen Algorithmus. Er wurde vom griechischen Mathematiker Euklid um 300 v. Chr. in seinem Werk "Die Elemente" beschrieben und demonstriert eindrucksvoll die zeitlose Natur algorithmischen Denkens.

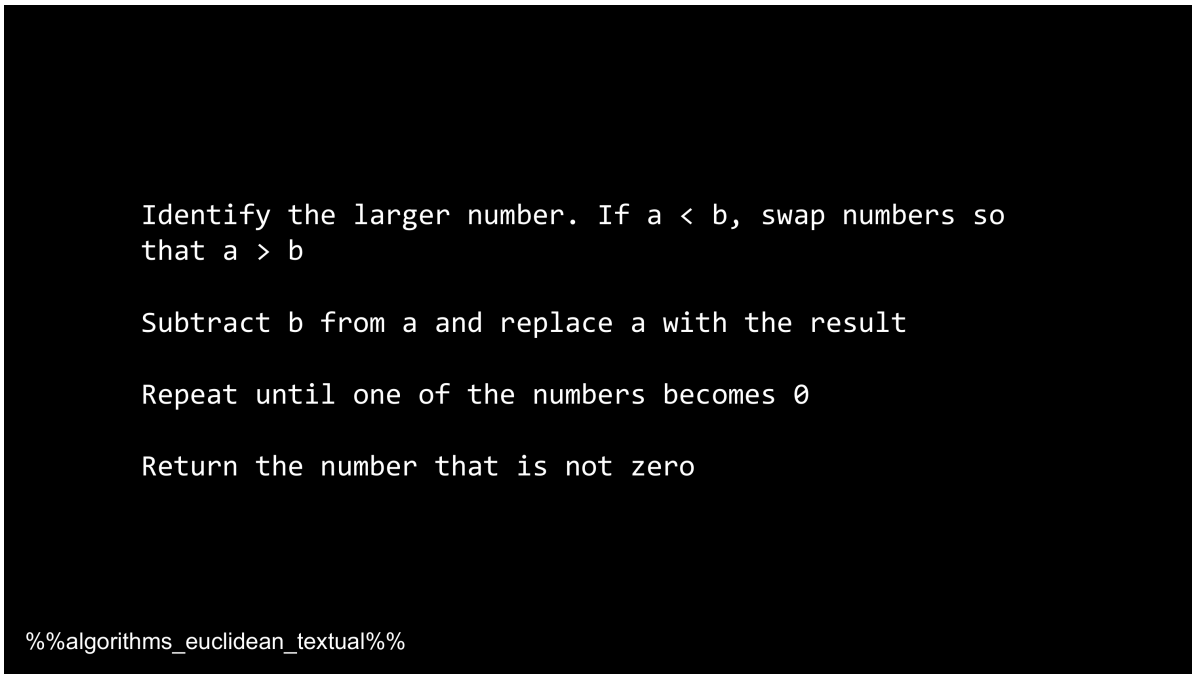


Abbildung 4: Vorgehen des Algorithmus nach Euklid.

Das Verfahren basiert auf einem einfachen, aber genialen Prinzip: Der GGT zweier Zahlen ist identisch mit dem GGT der kleineren Zahl und der Differenz beider Zahlen (Abbildung 4). Zum Beispiel haben die Zahlen 48 und 18 den gleichen GGT wie 18 und 30 (48-18). Durch wiederholtes Anwenden dieser Regel wird der GGT systematisch ermittelt. Die Eleganz dieses Verfahrens liegt in seiner Einfachheit und mathematischen Präzision - Eigenschaften, die auch heute noch moderne Algorithmen auszeichnen.

Abbildung 5 zeigt die Schritte des Euklidschen Algorithmus für das obige Zahlenbeispiel.

Babylonisches Wurzelziehen

Das Babylonische Verfahren zur Berechnung der Quadratwurzel ist ein weiteres Beispiel für einen mathematischen Algorithmus. Während der Euklid'sche Algorithmus exakte Ergebnisse liefert, zeigt das Babylonische Verfahren eine andere wichtige Eigenschaft mathematischer

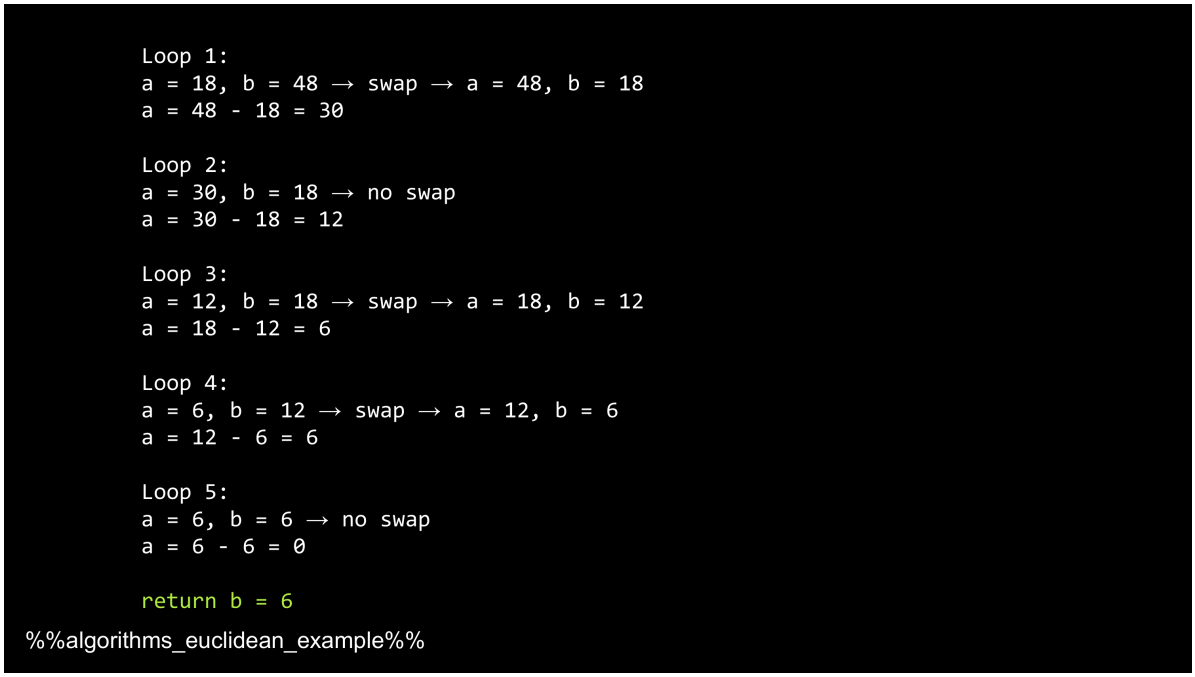


Abbildung 5: Beispiel für die Anwendung des Algorithmus nach Euklid.

Algorithmen: **die schrittweise Annäherung an einen Zielwert**. Der Algorithmus *approximiert* die Quadratwurzel durch wiederholte Verfeinerung der Schätzung und *konvergiert* dabei gegen den tatsächlichen Wert. Diese Methode demonstriert, wie auch ohne exakte Berechnung präzise Ergebnisse erzielt werden können.

Ein anschauliches Beispiel für das Babylonische Verfahren ist in Abbildung 6 dargestellt. Der Algorithmus berechnet die Quadratwurzel einer Eingabezahl durch geometrische Annäherung: Wenn wir die Eingabezahl x als Flächeninhalt eines Rechtecks interpretieren, suchen wir die Seitenlängen eines Quadrats mit der gleichen Fläche.

Das Verfahren nähert sich diesem Wert schrittweise an, indem es die Seitenlängen A und B eines Rechtecks iterativ anpasst. Nehmen wir an, wir wollen die Quadratwurzel aus $x = 16$ ziehen. Zu Beginn setzen wir $A = 1$ und berechnen B so, dass das Produkt der Seitenlängen die gesuchte Fläche ergibt: $B = \frac{x}{A} = 16$. Dieser Ausgangszustand ist in Abbildung 6 dargestellt.

Nach der Festlegung der Startwerte beginnt der eigentliche iterative Prozess der Annäherung. In jedem Durchlauf werden die Werte für A und B nach spezifischen Formeln neu berechnet, wodurch sie sich schrittweise einander annähern:

$$A = \frac{A + B}{2}$$

Und damit die Gesamtfläche der gesuchten Zahl x entspricht, setzen wir B wie folgt:

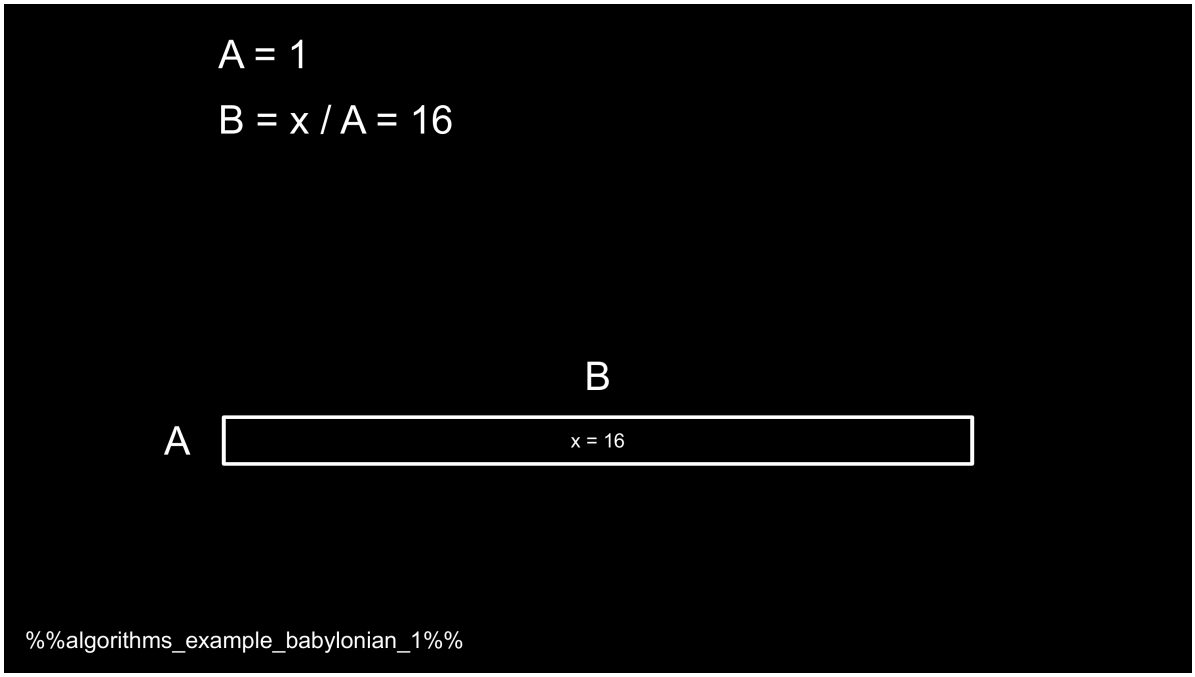


Abbildung 6: Ausgangszustand im Babylonischen Wurzelziehen.

$$B = \frac{x}{A}$$

Wenden wir diese Formeln auf unsere Ausgangswerte an, ergibt sich für den zweiten Iterationsschritt das in Abbildung 7 visualisierte Rechteck. Hier berechnen wir zunächst die neue Länge A:

$$A = \frac{1 + 16}{2} = 8.5$$

Und daraus folgt für B:

$$B = \frac{16}{8.5} \approx 1.88$$

Verglichen mit dem Ausgangszustand haben sich die Werte A und B bereits angenähert, aber sie weichen noch deutlich voneinander ab. Um eine bessere Approximation zu erreichen, führen wir eine weitere Iteration durch und berechnen die neuen Werte für A und B nach dem etablierten Schema:

$$A = \frac{8.5 + 1.88}{2} \approx 5.19$$

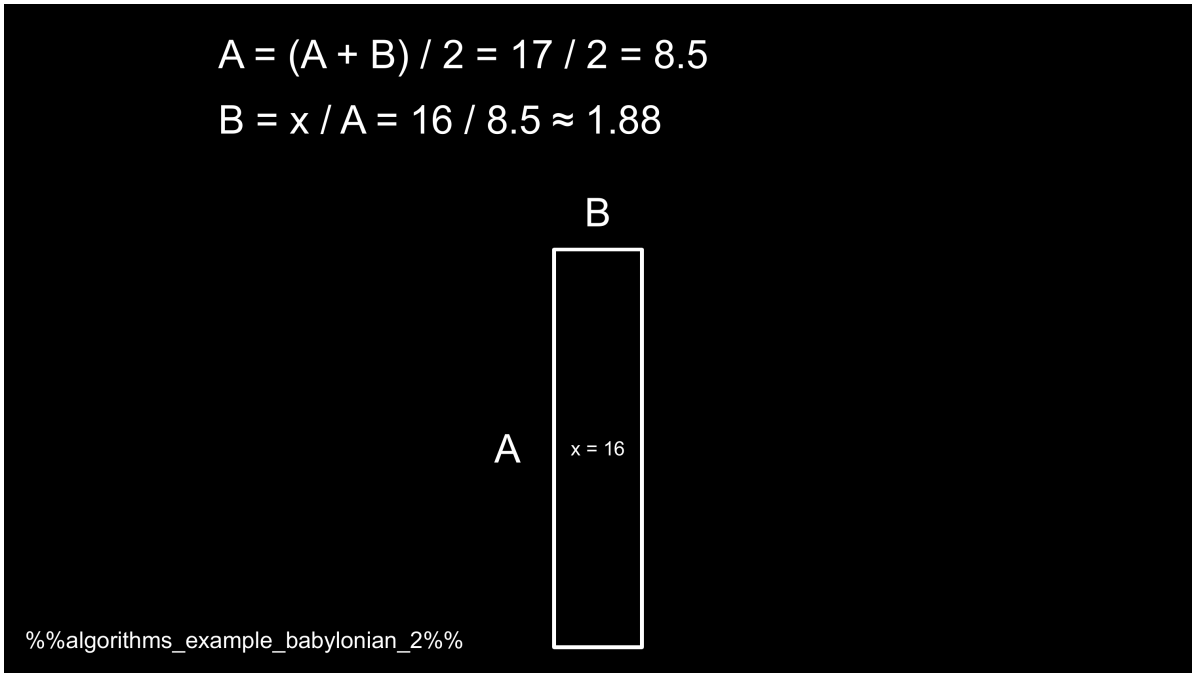


Abbildung 7: Zustand des Babylonischen Verfahrens nach dem ersten Schritt.

Und für die andere Kante:

$$B = \frac{16}{5.19} \approx 3.08$$

Die Werte haben sich weiter angenähert. Da wir wissen, dass beide Kanten am Ende die Länge 4 haben sollen, ist eine weitere Iteration erforderlich. Nach der dritten Berechnung, wie in Abbildung 9 dargestellt, nähern sich beide Kanten bereits sehr präzise dem Zielwert von 4 an.

Nach einer weiteren Iteration konvergiert der Algorithmus zum gesuchten Wert: Beide Seiten A und B erreichen durch Rundung den Wert 4. Damit haben wir das gesuchte Quadrat erfolgreich konstruiert, und der Algorithmus kann beendet werden.

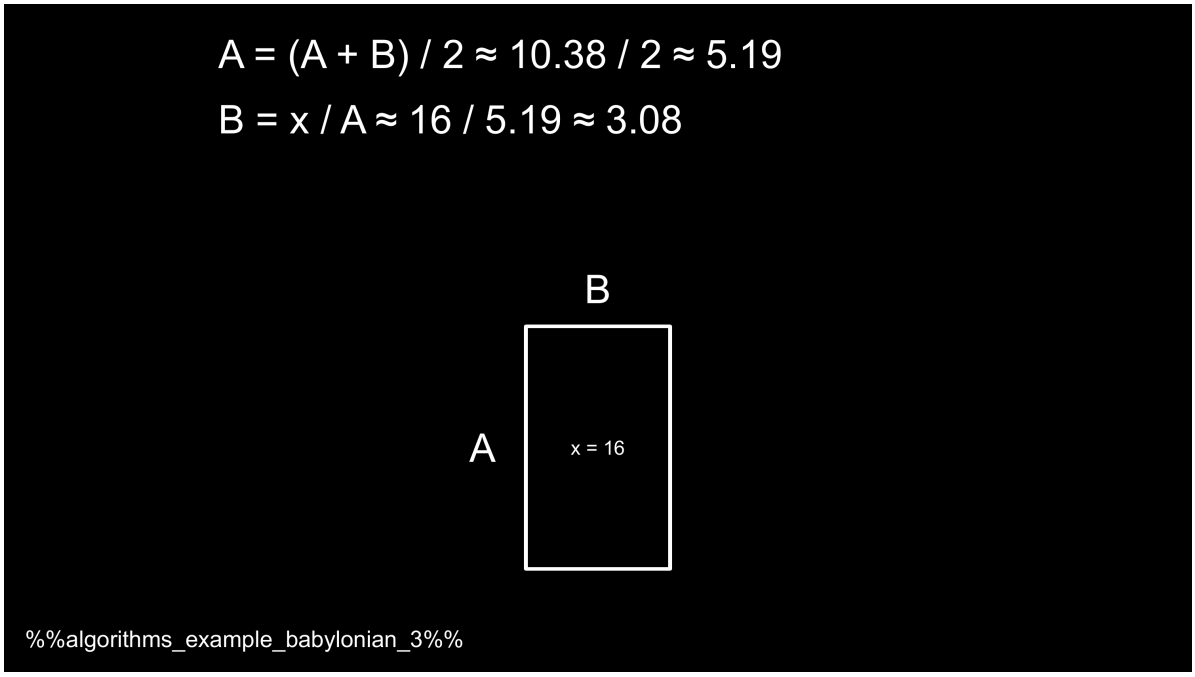


Abbildung 8: Zustand des Babylonischen Verfahrens nach dem zweiten Schritt.

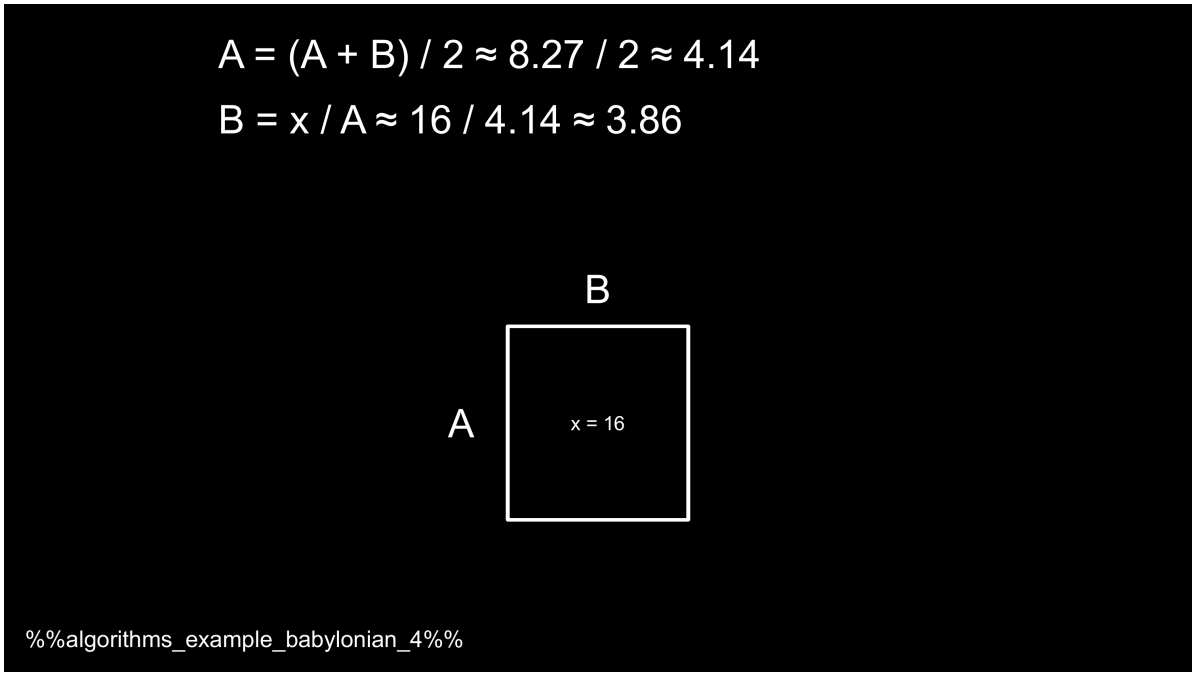
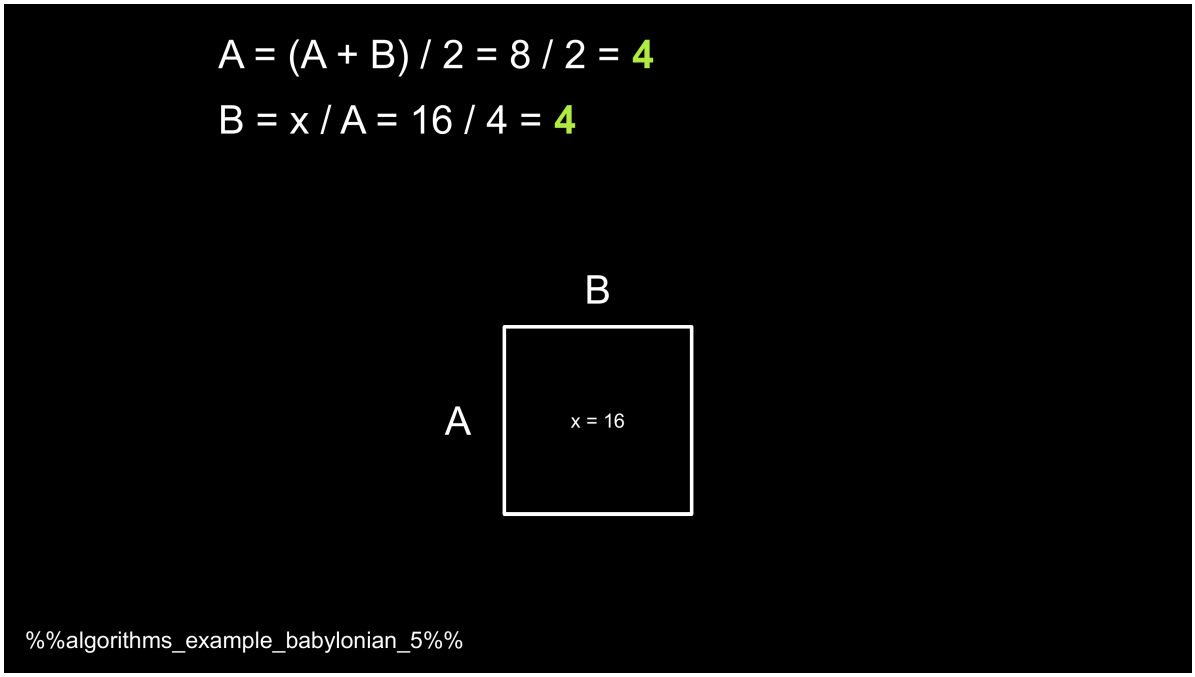


Abbildung 9: Zustand des Babylonischen Verfahrens nach dem dritten Schritt.



In diesem Beispiel konvergiert der Algorithmus bereits nach wenigen Iterationen zum korrekten Ergebnis. Die Geschwindigkeit der Konvergenz hängt jedoch von zwei wesentlichen Faktoren ab:

1. Der Eingabezahl, aus der die Wurzel gezogen werden soll
2. Der Wahl der Startwerte für A und B

Je nach Konstellation dieser Faktoren kann die Annäherung deutlich mehr Iterationen benötigen. Zudem erreichen die beiden Seiten in vielen Fällen keine exakte Gleichheit, sondern nähern sich lediglich bis auf eine beliebig kleine Differenz an. Dies wirft eine wichtige praktische Frage auf: Wann ist die Approximation der Quadratwurzel präzise genug, um den Algorithmus zu beenden? Dazu später in Kapitel mehr.

Ein pragmatisches Abbruchkriterium für das Babylonische Verfahren ist das Erreichen einer vordefinierten Genauigkeit. Nach jedem Iterationsschritt wird die Differenz zwischen den Werten A und B berechnet. Ist diese Differenz kleiner als ein festgelegter Schwellenwert, wird der Algorithmus beendet. Der vollständige Ablauf des Verfahrens ist in Abbildung 10 dargestellt.

Das Flussdiagramm beginnt auf der linken Seite mit einer Eingabeprüfung: Die Zahl x wird eingelesen und auf Positivität getestet, da die Quadratwurzel aus negativen Zahlen nicht definiert ist. Eine Verzweigung überprüft die Bedingung $x > 0$ und leitet den Algorithmus entsprechend weiter. Bei positiver Eingabe werden die Startwerte für A und B initialisiert. Anschließend beginnt eine WHILE-Schleife, die solange durchlaufen wird, bis die Differenz zwischen A und B

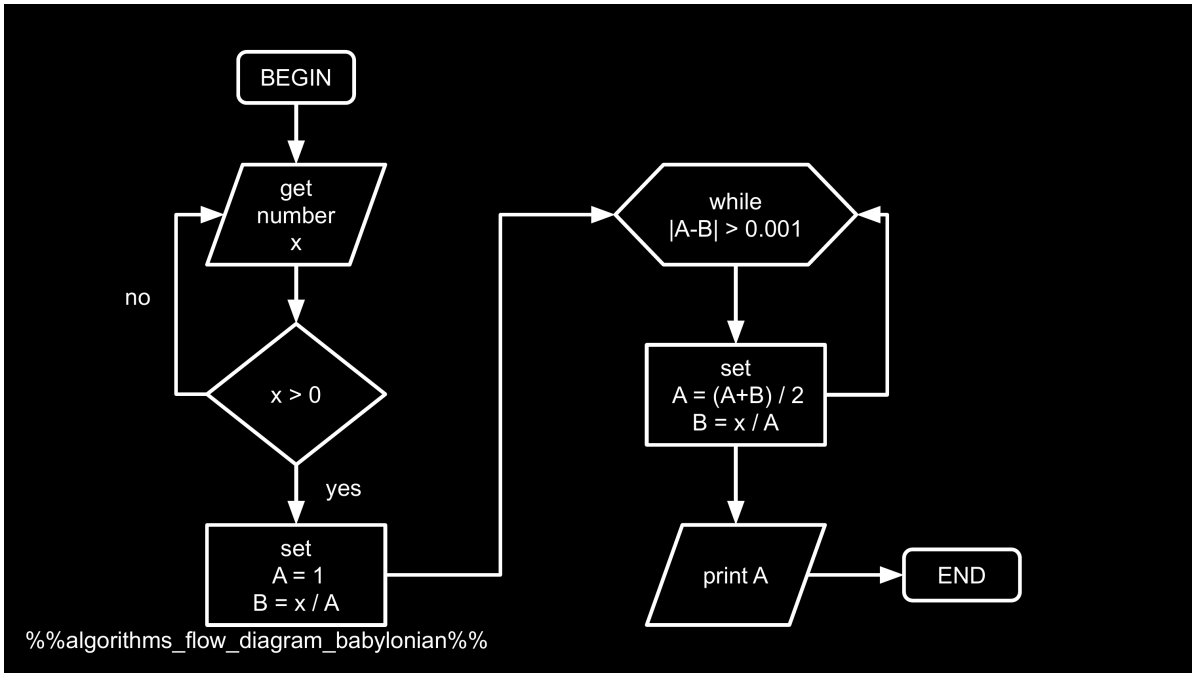


Abbildung 10: Das Babylonische Verfahren als Flussdiagramm.

einen definierten Schwellenwert unterschreitet. Nach erfolgreicher Konvergenz wird der finale Wert von A als Ergebnis ausgegeben.

Eine weitere wichtige Frage bei der Entwicklung von Algorithmen betrifft deren Terminierung: Wir müssen nicht nur festlegen, wann ein Algorithmus beendet werden soll, sondern auch sicherstellen, dass er überhaupt ein Ende erreicht. Am Beispiel des Babylonischen Verfahrens lässt sich dies gut veranschaulichen: Die entscheidende Frage ist, ob die Werte A und B tatsächlich konvergieren, also sich einander systematisch annähern.

In diesem Fall können wir die Frage positiv beantworten. Die Berechnungsformel, die in jedem Schritt den neuen Wert für A als Durchschnitt aus A und B bestimmt, garantiert eine stetige Annäherung der beiden Werte. Diese mathematische Eigenschaft sichert die Terminierung des Algorithmus. Bei anderen Algorithmen muss die Terminierung jedoch sorgfältig analysiert werden, um unendliche Ausführungen zu vermeiden. Mehr dazu in Kapitel .

Suchalgorithmen

Nach der Betrachtung mathematischer Algorithmen wenden wir uns nun den Suchalgorithmen zu - einer fundamentalen Kategorie von Algorithmen, die ein essentielles Element der Informatikausbildung darstellt. Diese Algorithmen bilden die Grundlage für effiziente Datenzugriffe und sind damit von großer praktischer Bedeutung.

Lineare Suche

Die lineare Suche ist der einfachste Suchalgorithmus und folgt einem intuitiven Ansatz: Sie durchläuft eine Liste sequentiell von Anfang bis Ende, bis das gesuchte Element gefunden wird. Obwohl dieser Algorithmus konzeptionell sehr einfach ist, hat er den Nachteil einer relativ hohen Laufzeit, da im schlechtesten Fall die gesamte Liste durchsucht werden muss. Dennoch ist die lineare Suche für unsortierte Listen oder kleine Datenmengen eine praktikable Lösung.

Binäre Suche

Die binäre Suche ist ein hocheffizienter Algorithmus, der auf dem "Teile und Herrsche"-Prinzip (Divide & Conquer) basiert. Eine wichtige Voraussetzung für ihre Anwendung ist, dass die zu durchsuchende Liste sortiert vorliegt. Der Algorithmus arbeitet systematisch, indem er in jedem Schritt die Liste in der Mitte teilt und anhand eines Vergleichs entscheidet, ob sich das gesuchte Element im linken oder rechten Teilbereich befindet. Durch diese Halbierung des Suchbereichs in jedem Schritt erreicht die binäre Suche eine bemerkenswerte Effizienz.

Diese Strategie, die wir bereits im vorigen **?@sec-divide-and-conquer** als *Divide-and- Conquer*-Ansatz kennengelernt haben, ermöglicht es der binären Suche, selbst in großen Datensätzen ein gesuchtes Element mit minimaler Anzahl von Vergleichsoperationen zu finden.

Suche in Bäumen

Die Suche in Baumstrukturen stellt eine weitere wichtige Variante der Suchalgorithmen dar. Binäre Suchbäume ermöglichen durch ihre hierarchische Struktur eine besonders effiziente Suche, da in jedem Knoten eine binäre Entscheidung getroffen wird, die den Suchbereich systematisch einschränkt. Diese Baumstrukturen kombinieren die Vorteile der binären Suche mit einer dynamischen Datenorganisation, die Einfüge- und Löschoperationen effizient unterstützt.

Sortieralgorithmen

Sortieralgorithmen bilden eine weitere fundamentale Kategorie von Algorithmen, die sich mit der systematischen Anordnung von Datenelementen in einer bestimmten Reihenfolge befassen. Die Bedeutung dieser Algorithmen liegt nicht nur in ihrer direkten Anwendung zur Sortierung von Daten, sondern auch in ihrer Rolle als Grundlage für effizientere Such- und Analyseverfahren. Im Folgenden betrachten wir drei klassische Sortieralgorithmen, die sich in ihrer Herangehensweise und Effizienz deutlich unterscheiden.

Bubblesort

Bubblesort ist ein einfacher Sortieralgorithmus, der paarweise benachbarte Elemente vergleicht und bei Bedarf vertauscht. Dieser Prozess wird solange wiederholt, bis keine Vertauschungen mehr notwendig sind und die Liste sortiert ist. Obwohl Bubblesort aufgrund seiner einfachen Implementierung und Verständlichkeit oft zu Lehrzwecken verwendet wird, ist er für große Datenmengen wegen seiner quadratischen Laufzeitkomplexität wenig effizient.

Selctionsort

Selectionsort ist ein weiterer elementarer Sortieralgorithmus, der die Liste schrittweise sortiert, indem er wiederholt das kleinste Element im unsortierten Teil der Liste findet und es an die korrekte Position im sortierten Teil verschiebt. Ähnlich wie Bubblesort ist auch dieser Algorithmus konzeptionell einfach zu verstehen, weist jedoch ebenfalls eine quadratische Laufzeitkomplexität auf. Der Vorteil gegenüber Bubblesort liegt in der geringeren Anzahl tatsächlicher Vertauschungsoperationen, da Elemente nur dann verschoben werden, wenn sie tatsächlich an ihre finale Position gebracht werden.

Mergesort

Mergesort ist ein effizienter Sortieralgorithmus, der auf dem Divide-and-Conquer-Prinzip basiert. Er teilt die zu sortierende Liste rekursiv in kleinere Teilsequenzen, bis diese nur noch ein Element enthalten, und führt diese dann schrittweise in sortierter Reihenfolge wieder zusammen. Im Gegensatz zu Bubblesort und Selectionsort erreicht Mergesort durch seine rekursive Strategie eine deutlich bessere Laufzeitkomplexität, was ihn besonders für große Datenmengen attraktiv macht.

Optimierungsalgorithmen

Optimierungsalgorithmen sind eine wichtige Klasse von Algorithmen, die darauf abzielen, die bestmögliche Lösung für ein gegebenes Problem zu finden. Diese Algorithmen sind besonders relevant in praktischen Anwendungen, wo optimale oder nahezu optimale Lösungen für komplexe Probleme gefunden werden müssen. Typische Beispiele finden sich in der Logistik, der Produktionsplanung oder der Ressourcenallokation. Die verschiedenen Ansätze zur Optimierung unterscheiden sich dabei hauptsächlich in ihrer Herangehensweise und Effizienz.

Brute-Force

Der Brute-Force-Ansatz ist die einfachste, aber auch rechenintensivste Methode der Optimierung. Dabei werden systematisch alle möglichen Lösungen durchprobiert, um die beste zu finden. Obwohl dieser Ansatz für kleine Probleme praktikabel sein kann, wird er bei wachsender

Problemgröße schnell ineffizient, da die Anzahl der zu prüfenden Kombinationen exponentiell steigt.

Random-Sampling

Random-Sampling ist ein stochastischer Optimierungsansatz, der zufällig ausgewählte Lösungen aus dem Suchraum evaluiert. Im Gegensatz zur erschöpfenden Suche des Brute-Force-Verfahrens werden hier nur Stichproben untersucht, was den Rechenaufwand deutlich reduziert. Dieser Ansatz eignet sich besonders für große Suchräume, in denen eine vollständige Enumeration nicht praktikabel ist, kann aber nicht garantieren, das globale Optimum zu finden.

Evolutionäre Algorithmen

Evolutionäre Algorithmen sind Optimierungsverfahren, die auf den Prinzipien der biologischen Evolution basieren. Sie operieren mit einer Population von Lösungskandidaten und verbessern diese durch drei zentrale Mechanismen: Mutation (zufällige Veränderungen), Rekombination (Kombination erfolgreicher Lösungen) und Selektion (Auswahl der besten Varianten). Der iterative Prozess aus Variation und Auswahl führt schrittweise zu besseren Lösungen.

Diese Algorithmen sind besonders effektiv bei komplexen Optimierungsproblemen, für die keine exakten Lösungsmethoden existieren oder diese zu rechenintensiv wären. Da die Mutation der Lösungskandidaten auf Zufallsprozessen basiert, gehören evolutionäre Algorithmen zur Klasse der stochastischen Verfahren.

Bayes'sche Optimierung

Die Bayes'sche Optimierung ist ein effizienter Ansatz zur Optimierung von komplexen Funktionen, der auf dem Bayes'schen Theorem basiert. Anders als klassische Optimierungsverfahren nutzt dieser Algorithmus probabilistische Modelle, um die Zielfunktion zu approximieren und vielversprechende Bereiche des Suchraums zu identifizieren. Besonders nützlich ist dieser Ansatz bei rechenintensiven Problemen, bei denen jede Auswertung der Zielfunktion zeit- oder kostenaufwendig ist.

Graphenalgorithmen

Dijkstra-Algorithmus

Der Dijkstra-Algorithmus, entwickelt 1956 von Edsger W. Dijkstra, ist ein fundamentaler Algorithmus zur Berechnung kürzester Pfade in gewichteten Graphen. Seine Funktionsweise basiert auf dem Prinzip der schrittweisen Optimierung: Ausgehend von einem Startknoten berechnet er systematisch die kürzesten Wege zu allen anderen Knoten im Graphen. Aufgrund seiner

Effizienz und Zuverlässigkeit findet der Algorithmus heute breite praktische Anwendung, insbesondere in der Routenplanung, Netzwerkanalyse und Logistik.

Das Traveling Salesman-Problem

Das Traveling Salesman-Problem (TSP) gehört zu den bekanntesten Optimierungsproblemen der Graphentheorie. Die Aufgabenstellung ist dabei einfach zu verstehen: Ein Handelsreisender muss eine Route planen, die alle vorgegebenen Städte genau einmal besucht und am Ende zum Ausgangspunkt zurückführt. Das Ziel ist es, die kürzeste mögliche Route zu finden. Trotz dieser scheinbar einfachen Formulierung ist das TSP ein NP-schweres Problem - das bedeutet, dass bisher kein Algorithmus gefunden wurde, der für große Städtemengen in angemessener Zeit die optimale Lösung garantiert berechnen kann.

Stochastische Algorithmen

Annäherung der Kreiszahl π

Die Monte-Carlo-Methode ist ein anschauliches Beispiel für stochastische Algorithmen. Sie nutzt Zufallszahlen zur Annäherung mathematischer Werte, wie etwa der Kreiszahl . Das Verfahren platziert dabei zufällig Punkte in einem Quadrat mit einbeschriebenem Kreis. Das Verhältnis der Punkte innerhalb des Kreises zur Gesamtanzahl der Punkte ermöglicht eine Approximation von . Ein wichtiges Merkmal dieser stochastischen Methode ist, dass die Genauigkeit der Berechnung mit der Anzahl der generierten Punkte steigt.

Maschinelle Lernalgorithmen

Maschinelles Lernen hat sich in den letzten Jahren zu einem der wichtigsten Bereiche der Algorithmenentwicklung entwickelt. Diese Algorithmenklasse bildet die technische Grundlage für viele praktische Anwendungen der Künstlichen Intelligenz (KI). Maschinelle Lernalgorithmen zeichnen sich dadurch aus, dass sie aus vorhandenen Daten Muster erkennen und auf dieser Basis Vorhersagen oder Entscheidungen treffen können.

Welche wichtigen algorithmischen Denkmuster gibt es?

Sequenzen

Die Sequenz ist das grundlegendste Muster in der Algorithmik. Sie beschreibt eine geordnete Abfolge von Anweisungen, die nacheinander ausgeführt werden. Wie bei einer Wegbeschreibung folgt dabei ein Schritt dem anderen in einer festgelegten Reihenfolge. Die korrekte und vollständige Ausführung aller Schritte ist entscheidend - das Auslassen oder Vertauschen von Anweisungen führt in der Regel nicht zum gewünschten Ergebnis.

Sowohl Menschen als auch Computer verarbeiten Anweisungen standardmäßig sequentiell - also Schritt für Schritt von oben nach unten. Diese intuitive Vorgehensweise bildet die Grundlage für das Verständnis von Algorithmen. Allerdings können Algorithmen auch komplexere Strukturen enthalten, die von diesem linearen Ablaufmuster abweichen und alternative Ausführungspfade ermöglichen.

Verzweigungen

Verzweigungen stellen eine grundlegende Form der nicht-linearen Ausführung eines Algorithmus dar. Sie ermöglichen es, dass der Algorithmus basierend auf bestimmten Bedingungen unterschiedliche Ausführungspfade einschlägt. Eine Verzweigung führt also je nach erfüllter oder nicht erfüllter Bedingung zu verschiedenen Anweisungsfolgen. Ein alltägliches Beispiel findet sich in Kochrezepten: “Wenn der Teig zu flüssig ist, füge mehr Mehl hinzu”. Die Anweisung, mehr Mehl hinzuzufügen, wird nur dann ausgeführt, wenn die Bedingung “Teig ist zu flüssig” zutrifft. Ist der Teig bereits von idealer Konsistenz, wird diese Anweisung übersprungen.

Iterationen

Ein Beispiel für ein iteratives, schrittweise Vorgehen ist das Babylonische Wurzelziehen aus Kapitel .

Kapselung

Kapselung beschreibt das Prinzip, komplexe Algorithmen in kleinere, überschaubare Einheiten zu zerlegen. Diese Modularisierung erhöht nicht nur die Lesbarkeit und Wartbarkeit des Algorithmus, sondern ermöglicht auch die Wiederverwendung von Teilfunktionen in anderen Kontexten. Ein klassisches Beispiel ist die Auslagerung häufig benötigter Berechnungen in separate Funktionen, die dann an verschiedenen Stellen aufgerufen werden können.

Rekursion

Rekursion ist ein mächtiges algorithmisches Konzept, bei dem sich ein Algorithmus selbst aufruft, um ein Problem zu lösen. Dieses Prinzip eignet sich besonders gut für Probleme, die sich in kleinere, gleichartige Teilprobleme zerlegen lassen. Ein klassisches Beispiel ist die Berechnung der Fakultät einer Zahl, bei der sich die Lösung aus der Multiplikation mit der Fakultät der nächstkleineren Zahl ergibt. Aber auch die binäre Suche kann rekursiv implementiert werden, indem der Algorithmus den zu durchsuchenden Bereich in der Mitte teilt und sich selbst mit der relevanten Hälfte aufruft. Diese rekursive Struktur macht den Algorithmus nicht nur elegant, sondern auch besonders effizient.

Was sind Herausforderungen bei der Fomulierung von Algorithmen?

Haltekriterium

Ein wichtiges Kriterium bei der Formulierung von Algorithmen ist die Bestimmung eines geeigneten Haltekriteriums. Das Haltekriterium definiert die Bedingung, unter der ein Algorithmus seine Ausführung beendet und ein Ergebnis zurückliefert. Beim Babylonischen Wurzelziehen beispielsweise ist das Haltekriterium erreicht, wenn die Differenz zwischen zwei aufeinanderfolgenden Näherungswerten einen bestimmten Schwellenwert unterschreitet.

Die Bestimmung eines geeigneten Haltekriteriums stellt für manche Algorithmen eine besondere Herausforderung dar. Dies zeigt sich besonders deutlich bei Optimierungsalgorithmen, wo die Frage nach dem optimalen Zeitpunkt für die Beendigung des Algorithmus nicht trivial ist. Da das theoretische Optimum in der Regel unbekannt ist, lässt sich schwer einschätzen, wie nah die aktuelle Lösung bereits am bestmöglichen Ergebnis liegt. Eine pragmatische Lösung für dieses Problem besteht in der Festlegung eines Optimierungsbudgets, das die maximale Anzahl der Durchläufe definiert, die der Algorithmus ausführen darf.

Endlosschleifen

Endlosschleifen stellen eine kritische Herausforderung bei der Entwicklung von Algorithmen dar. Sie entstehen, wenn ein Algorithmus das definierte Haltekriterium nie erreicht und stattdessen kontinuierlich dieselben Anweisungen wiederholt. Ein typisches Beispiel ist eine While-Schleife, deren Bedingung permanent wahr bleibt - der Algorithmus verbleibt dann in einem unendlichen Ausführungszyklus. Ohne geeignete Fehlerbehandlung führt dies meist zu einem Programmabsturz, da Systemressourcen wie Arbeitsspeicher oder Prozessorzeit erschöpft werden.

Beurteilung des Ergebnisses

Die Beurteilung der Qualität eines algorithmischen Ergebnisses ist nicht immer eindeutig und hängt stark vom jeweiligen Anwendungsfall ab. Bei numerischen Berechnungen lässt sich die Genauigkeit oft durch den Vergleich mit bekannten Referenzwerten oder theoretischen Grenzen bestimmen. Bei Optimierungsproblemen hingegen ist die Bewertung komplexer, da das theoretische Optimum häufig unbekannt ist und die Qualität der Lösung von verschiedenen, teils konkurrierenden Kriterien abhängt.

Erklärbarkeit des Ergebnisses

Die Erklärbarkeit algorithmischer Entscheidungen spielt eine zentrale Rolle in der modernen Informatik, besonders in kritischen Bereichen wie der Medizin oder Rechtsprechung. Dabei stellt die mangelnde Transparenz komplexer Algorithmen eine fundamentale Herausforderung dar: Der Prozess von der Eingabe bis zur Entscheidungsfindung ist oft nicht direkt nachvollziehbar. Dieses Problem manifestiert sich besonders deutlich bei Deep Learning Modellen, deren mehrschichtige Strukturen und komplexe Berechnungen eine intuitive Interpretation der Entscheidungswege erschweren. Die Entwicklung von Methoden zur besseren Erklärbarkeit dieser "Black Box"-Systeme ist daher ein aktives Forschungsfeld, das darauf abzielt, die Akzeptanz und Vertrauenswürdigkeit algorithmischer Entscheidungen zu erhöhen.

Gibt es bessere und schlechtere Algorithmen?

Komplexität

Die Komplexität eines Algorithmus beschreibt, wie sich sein Ressourcenbedarf (meist Zeit und Speicher) in Abhängigkeit von der Eingabegröße entwickelt. Diese mathematische Charakterisierung ermöglicht einen objektiven Vergleich verschiedener algorithmischer Lösungen für dasselbe Problem. Besonders wichtig ist dabei die asymptotische Komplexität, die das Verhalten des Algorithmus für große Eingabemengen beschreibt.

Verständlichkeit

Die Verständlichkeit eines Algorithmus ist ein weiteres wichtiges Qualitätsmerkmal. Ein gut strukturierter und dokumentierter Algorithmus erleichtert nicht nur die Wartung und Weiterentwicklung, sondern reduziert auch die Wahrscheinlichkeit von Fehlern bei der Implementierung. Dabei spielt die Wahl aussagekräftiger Bezeichner und eine klare Dokumentation der einzelnen Schritte eine zentrale Rolle.

Übungsaufgaben

1. Woher stammt der Begriff “Algorithmus”?
2. Definiere, was ein Algorithmus ist, und gib drei Beispiele für Algorithmen aus dem Alltag, die keinen direkten Bezug zu Computern haben.
3. Beobachte eine Woche lang Algorithmen in deinem Alltag (z.B. Navigationssysteme, Empfehlungssysteme, Suchmaschinen). Dokumentiere deren Eingaben, Verarbeitung und Ausgaben.
4. Welche grundlegenden Ansätze zur Klassifizierung von Algorithmen gibt es?
5. Erläutere, was mit der Komplexität eines Algorithmus gemeint ist. Warum ist die Komplexität eines Algorithmus wichtig? Wie wird sie angegeben?
6. Welche Komplexitätsklassen kennst du? Bringe sie in eine Reihenfolge von der geringsten zur höchsten Komplexität.
7. Vergleiche die Laufzeitkomplexität von linearer und binärer Suche anhand eines konkreten Beispiels mit einer Million sortierten Zahlen!
8. Berechne den größten gemeinsamen Teiler der Zaheln 56 und 98 mithilfe des euklidischen Algorithmus! Dokumentiere jeden Schritt!
9. Wir haben exemplarisch für einen Algorithmus die babylonische Methode zur Approximation einer Quadratwurzel kennengelernt. Beantworte die nachfolgenden Fragen in diesem Kontext:
    a. Berechne die Quadratwurzel von 25 mit der babylonischen Methode und dokumentiere jeden Schritt! Wähle einen sinnvollen Startwert!
    b. Vergleiche die Ergebnisse der babylonischen Methode nach 3, 5 und 7 Iterationen mit dem exakten Wert der Quadratwurzel.
    c. Erkläre die Funktionsweise des babylonischen Algorithmus zur Berechnung der Quadratwurzel. Verwende dazu visuelle Hilfsmittel. Warum konvergiert der Algorithmus gegen den exakten Wert der Quadratwurzel?
10. Erläutere die Monte-Carlo-Methode zur Schätzung von π und erkläre, wie man mithilfe von Zufallszahlen eine Annäherung an π erreichen kann.
11. Erkläre den Unterschied zwischen einem stochastischen und einem deterministischen Algorithmus anhand eines selbst gewählten Beispiels.
12. Finde weitere Probleme, die sich durch Monte-Carlo-Simulationen lösen lassen. Weshalb sind manche dieser Probleme mit anderen Methoden nicht lösbar?
13. Du hast zwei Sanduhren: Eine läuft 4 Minuten, die andere 7 Minuten. Wie kannst du damit genau 9 Minuten messen? Entwickle einen präzisen Algorithmus für das Problem!

14. Du hast drei Gefäße mit 3, 5 und 8 Litern Fassungsvermögen. Das 8-Liter-Gefäß ist voll, die anderen sind leer. Entwickle einen Algorithmus, um genau 4 Liter abzumessen!

15. Entwickle einen Algorithmus, der prüft ob eine eingegebene Zahl eine Primzahl ist. Stelle den Algorithmus als Flussdiagramm dar!

16. Entwirf einen Algorithmus für den einfachen Getränkeautomaten aus dem vorigen Kapitel. Schreibe den Algorithmus als Pseudocode und als Flussdiagramm auf!

17. Recherchiere einen Fall, bei dem ein algorithmisches System zu problematischen Entscheidungen geführt hat. Analysiere die Ursachen und schlage Möglichkeiten zur Verbesserung!

18. Diskutiert die ethischen Implikationen von nicht-erklärbaren algorithmischen Entscheidungen in kritischen Bereichen wie Medizin oder Rechtsprechung! In welchen Bereichen könnte eine fehlende Erklärbarkeit noch kritisch sein?