


<p>Software</p> <ul style="list-style-type: none"> Algorithmen Hochsprache Betriebssystem Programmiersprache Übersetzer <p>Hardware</p> <ul style="list-style-type: none"> Mikroprozessoren Digitale Logik Transistoren, Verkabelung Kosten, Support, Kollisions <p>A.V.D.</p> <p>Computer-technik</p> <p>Digitalechnik</p> <p>Halbleitertechnik</p> <p>Anzahl der Ziffern in Basis x: x: Basis z: Zahl</p> <p>$m = \lfloor \log_x(z) \rfloor + 1$</p>	<p>Definition Datenstruktur</p> <ul style="list-style-type: none"> logische Anordnung von Datenobjekten die Informationen repräsentieren den Zugriff auf die Informationen über Operationen auf Daten ermöglichen die Informationen verwalten <p>Hauptbestandteile:</p> <ul style="list-style-type: none"> Datenobjekte Operationen auf den Objekten 	<p>Verifikation:</p> <p>$\{VOR\} \wedge \{NACH\}$</p> <p>- gilt $\{VOR\}$ nicht, so ist Aussage immer wahr</p> <p>Bsp: $\{x = 0\} \wedge \{x = x + 1\} \wedge \{x = 1\}$ ist wahr</p> <p>$\{VOR\} \wedge \{MITTE\}$</p> <p>$\{MITTE\} \wedge \{NACH\}$</p> <p>Invariante z.B. $while(i < n) \{$ $i = i + 1;$ $sum = sum + 1;$ $\}$</p>				
<p>Größte Zahl aus n Ziffern: $Z_{max} = x^n - 1$</p> <p>Primitive Datentypen:</p> <ul style="list-style-type: none"> 8 Bit: unignrad. Leer 16 Bit: unignrad. Short int 32 Bit: unignrad. Long int 64 Bit: unignrad. Long long <p>2er-Komplement:</p> <p>$-x_2 = 2^n - x$</p> <p>Bsp: $-5_2 = 2^4 - 5 = 16 - 5 = 11 = 1011_2$</p>	<p>Einfach verkettete Liste:</p>  <p>Doppelt verkettete Liste:</p>  <p>Vorteile:</p> <ul style="list-style-type: none"> Durchlaufen in beide Richtungen möglich einfügen / löschen einfacher <p>Nachteile:</p> <ul style="list-style-type: none"> Verständlicher Speicher Zeigerverwaltung komplizierter 	<p>$\rightarrow P = (sum = i * (i + 1) / 2 \wedge i \leq n)$</p> <p>Test auf Korrektheit:</p> <ol style="list-style-type: none"> Eröffnungseintritt Eröffnungsdurchlauf Austritt <p>Validation:</p> <ul style="list-style-type: none"> White-Box Black-Box Regressions-Test Integrations-Test <p>Landau-Symbole:</p> <p>$\Theta: 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$</p> <p>$O: 0 \leq f(n) \leq c \cdot g(n)$</p> <p>Bsp: $f(n) = 60n + 12$ Behauptung: $f(n) = O(n)$</p> <p>$0 \leq 60n + 12 \leq cn$</p> <p>$0 \leq 60 + \frac{12}{n} \leq c$</p> <p>z.B. $n_0 = 12 \rightarrow c = 61 \checkmark$</p>				
<p>Exaktkommazahlen:</p> <table border="1" data-bbox="127 929 406 1008"> <tr> <td>ganze. Anteil</td> <td>gebr. Anteil</td> </tr> <tr> <td></td> <td>Komma</td> </tr> </table> <p>Nachteile:</p> <ul style="list-style-type: none"> weniger große Zahlen darstellbar hohe Genauigkeit der Nachkomm. <p>Fließkommazahlen:</p> <p>Vorzeichen, Mantisse, Exponent</p> <p>Problem z.B.: zw. $1,23 \cdot 10^4$ und $1,24 \cdot 10^4$ keine Zahl darstellbar</p> <p>$z = (-1)^s \cdot (1, m) \cdot 2^{e - bias}$</p> <p>$e = \lfloor \log_2(z) \rfloor + 127 (bias)$</p> <p>$m = \left(\frac{ z }{2^{e - bias}} - 1 \right) \cdot 2^{23}$</p>	ganze. Anteil	gebr. Anteil		Komma	<p>Beurteilung von Algorithmen:</p> <ul style="list-style-type: none"> Elementarer Verarbeitungsschritt Sequenz. Elem. Verarbeitungssch. nacheinander bedingter Verarbeitungsschritt Wiederholung <p>Abstrakte Datentypen:</p> <ul style="list-style-type: none"> Stack: <ul style="list-style-type: none"> last in - first out (LIFO) Implementation z.B. als sequentielle Liste, verkettete Liste Queue: <ul style="list-style-type: none"> first in - first out (FIFO) Implementation z.B. als sequentielle Liste, verkettete Liste, zwei Stacks Problem: Länderschliff \rightarrow rekursive sequentielle sequentielle Priority Queue: <ul style="list-style-type: none"> Größenform des Elements mit minimalem (bzw. max.) Schlüssel Implementation: <ul style="list-style-type: none"> mit sortierten Feldern: insert $O(n)$ mit unsortierten Feldern: extractMin $O(n)$ als MinHeap: $O(\log n)$ (insert) hiermit lassen sich Sortieralgorithmen implementieren 	<p>Algorithmen-Entwurf:</p> <ul style="list-style-type: none"> Verfeinerung Divide and Conquer <ul style="list-style-type: none"> große Aufgaben in mehrere kleine Teilaufgaben zerlegen (\rightarrow rekursiv) Greedy (naive Implementation: $O(n^2)$) <ul style="list-style-type: none"> in jedem Schritt den bestmöglichen Schritt ohne Berücksichtigung zukünftiger Schritte wählen Voraussetzung: feste Menge von Eingabewerten, Lösungen lassen sich schrittweise durch Hinzufügen von Eingabewerten aufbauen Backtracking <ul style="list-style-type: none"> systematische Suchtechnik, um vorgegebenen Lösungsraum vollständig abzuarbeiten meist $O(2^n)$ oder schlimmer Dynamisches Programmieren <ul style="list-style-type: none"> stark Rekursion vom kleinsten Problem aufwärts berechnen Zwischenergebnisse werden in Tabellen gespeichert
ganze. Anteil	gebr. Anteil					
	Komma					
<p>Logik: NOR: \downarrow NAND: \uparrow</p> <ul style="list-style-type: none"> Konjunktion (UND): \wedge Disjunktion (ODER): \vee Negation: \neg Implikation: $a \vee \bar{b}$ Äquivalenz: $(a \wedge b) \vee (\bar{a} \wedge \bar{b})$ 	<p>ASCII:</p> <ul style="list-style-type: none"> 7 Bit verschiedene Erweiterungen auf 8 Bit <p>Unicode:</p> <ul style="list-style-type: none"> 16 Bit <p>UTF-8:</p> <ul style="list-style-type: none"> 1-6 Bytes MultiByte - Codierung von Unicode 	<p>Definition: Abstrakter Datentyp</p> <p>Mathematisches Modell für bestimmte Datenstrukturen mit vergleichbarem Verhalten</p> <p>Definition: Algorithmen:</p> <p>Verfahren mit einer präzisen, endlichen Beschreibung unter Verwendung effektiver, elementarer Verarbeitungsschritte</p>				
<p>Unicode:</p> <ul style="list-style-type: none"> 16 Bit <p>UTF-8:</p> <ul style="list-style-type: none"> 1-6 Bytes MultiByte - Codierung von Unicode <p>Europäische Skizze:</p> <p>Alle gängigen Sprachen sind äquivalent!</p> <p>Bsp: \bullet Java \bullet C# \bullet JavaScript \bullet C \bullet Eurodiagramme \bullet C++</p> <p>Strings in Arrays mit dyn. Länge mit Pointer:</p> <ul style="list-style-type: none"> Speicher muss über malloc/free angefordert bzw. freigegeben werden 	<p>Nachteile der Korrektheit von Algorithmen:</p> <ul style="list-style-type: none"> Verifikation: formaler mathem. Beweis Validation: systematisches Testen 	<p>FFT: Fourier-Transformation Setzt Signal in Frequenzbereich über</p> <p>$1, \ln(n), \log_2(n), (\log_2(n))!, n^2, (\log_2(n))^{\log_2(n)}$</p> <p>$e^n, n!, n^n, 2^{2^n}$</p>				

Sortierverfahren

- Insertion Sort
 - entfernt einer unsortierten Liste ein beliebiges Element und fñhrt es an richtiger Stelle ein
 - best case: $O(n)$ • worst case: $O(n^2)$
 - in-place • stabil
- Selection Sort: $O(n^2)$
 - entfernt einer unsortierten Liste das passende Element
 - in-place • nicht stabil
- Merge Sort: $O(n \log n)$
 - divide and conquer (rekursiv)
 - extra Speicher nötig
 - (nicht) stabil
- Quick Sort
 - divide and conquer (rekursiv)
 - mit Pivot-Element
 - im Mittel: $O(n \log n)$
 - worst case: $O(n^2)$
 - in-place • nicht stabil
- Heap Sort: $O(n \log n)$
 - Idee: Heap erstellen; extract Min wiederholen bis Heap leer ist
 - in-place • nicht stabil
 - minHeap sortiert in absteigender Reihenfolge

Graphen

② → ①
Knoten 1 ist adjacent zu Knoten 2

② — ①
Knoten 1 ist adjacent zu Knoten 2
Knoten 2 ist adjacent zu Knoten 1

Gerichteter Graph:

- stark zusammenhängend, falls jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- (Ungerichteter Graph: → zusammenh.)

Adjazenzmatrix: $O(|V|^2)$

- bei ungerichteten Graph symmetrisch
- sinnvoll, wenn Graph locker vollständig ist

Adjazenzliste:

- erste Liste enthält alle Knoten
- für jeden Knoten alle durch eine Kante zu erreichende Knoten speichern

Operation	Adjazenzmatrix	Adj.-Liste
Kante einfügen	$O(1)$	$O(V)$
Kante löschen	$O(1)$	$O(V)$
Knoten einfügen	$O(V ^2)$	$O(1)$
Knoten löschen	$O(V ^2)$	$O(V + E)$

Bäume:

- Anzahl der Knoten von vollständigen Binärbaum mit Höhe k : $2^{k+1} - 1$
- Anzahl der Blätter: 2^k

DFS = Tiefensuche:

- Pre-order: Wurzel → linker TB → r. TB
- In-order: linker TB → Wurzel → r. TB
- Post-order: linker TB → r. TB → Wurzel

BFS = Breitensuche:

- besuche Wurzel → alle Knoten aktueller Ebene
- Anwendung: z.B. Suchtree

Kürzeste Pfade

- Dijkstra: $O(|E| \log |V|)$ (mit Heap)
 - nur positive Kantengewichte
 - Implementierung: Priority Queue
 - Greedy
- A*:
 - nur positive Kantengewichte
 - besucht zuerst Knoten, die wahrscheinlich schneller ans Ziel führen
- Bellman-Ford: $O(|V| \cdot |E|)$
 - auch negative Kantengewichte
- Floyd-Warshall: $O(|V|^3)$
 - kürzeste Pfade zwischen allen Knotenpaaren

Matrizen-Multiplikation: $O(n^3)$

- Strassen-Algorithmus: $O(n^{2,807})$
 - Divide and conquer
 - weniger numerisch stabil
 - n muss 2er-Potenz sein
 - mehr Speicher und benötigt
- Coppersmith-Winograd-Str.: $O(n^{2,376})$
 - mit heutigen Computern nicht praktikabel

Invertierbare Matrizen:

- $\det(A) \neq 0$; Kern(A) = $\{0\}$; Rang(A) = n

Orthogonale Matrizen:

- Spalten von A bilden eine Orthonormalbasis
- $A^T = A^{-1} \Leftrightarrow A^T A = I \Leftrightarrow \det(A) = \pm 1$

Matrizen-Zerlegungen

- Cholesky: $\sim \frac{1}{3} n^3$ FLOPS
 - A symmetrisch ($A^T = A$) und positiv definit
 - \exists untere Δ -Matrix mit strikt positiven Diagonalelementen: $A = LL^T$
 - Lösung von $Ax = b$: $LL^T x = b$
 - löse $Lz = b$ → löse $L^T x = z$
- LUP: $\sim \frac{2}{3} n^3$ FLOPS
 - A invertierbar, eine obere Dreiecksmatrix U
 - \exists untere Einheitsdreiecksmatrix L und eine Permutationsmatrix P: $PA = LU$
 - Lösung von $Ax = b$: $PAx = Pb \Leftrightarrow Ux = Pb$
 - löse $Lz = Pb$ → löse $Ux = z$

Such- Algorithmen

- Lineare Suche: $O(n)$
- Binäre Suche: $O(\log n)$
 - vorsortiertes Feld
 - Suche mittels Divide and Conquer (Rekursiv)
- Binärer Suchbaum
 - best case: $O(\log n)$
 - worst case: $O(n)$
 - löschen von Knoten: $O(n)$ (Kahn)

Balancierte Suchbäume

- AVL-Baum:
 - Höhenunterschied von linkem und rechten Teilbaum ist maximal 1
 - reparieren der AVL-Bedingung mittels Rotation und Doppelrotation

Matrizen-Zerlegungen

- QR: $\sim \frac{2}{3} n^3$ FLOPS
 - A invertierbar
 - \exists orthogonale Matrix Q ($Q^{-1} = Q^T$) und eine obere Δ -Matrix R mit pos. Diagonaleinträgen: $A = QR$
 - Lösung von $Ax = b$: $Rx = Q^T b$
- SVD: $2mn^2 + 2n^3$ FLOPS
 - $A \in \mathbb{R}^{m \times n}$
 - \exists Matrix U mit $U^T U = I_n$, eine orthogonale Matrix V, sowie eine diagonale Matrix $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_n)$ mit nicht-negativen Diagonaleinträgen in monoton fallender Reihenfolge $A = U \Sigma V^T$
 - Lösung von $Ax = b$ (wenn alle $\sigma_i \neq 0$): $x = A^{-1} b = (U \Sigma V^T)^{-1} b = V \Sigma^{-1} U^T b$

Suchen in Zeichenketten

- Breite-Force: $O(n \cdot m)$
- KMP: $O(n + m)$
 - verfeinert Breite-Force durch Berechnung bereits gelesener Infos bei einem Mismatch (next-Table)

Graph- Algorithmen

- Ziefensuche (DFS): $O(|V| + |E|)$
 - erzeugt Spannwald bzw. Spannbaum falls zusammenhängend
 - erlaubt Markierung spez. Kanten:
 - Rückkanten, Vorwärtskanten, Brück-K.
 - Implementierung: Stack, rekursiv
 - Anwendungen: Test auf Zyklenfreiheit, Zusammenhang
- Breitensuche (BFS): $O(|V| + |E|)$
 - erzeugt Spannbaum (kein Spannwald)
 - Implementierung: Stack, Queue
 - Berechnen von Anzahl der Kantens, alle Knoten in Zusammenhangskomponente besuchen

Beste Lösung: $\min \|Ax - y\|_2$ ($Ax = y$ überbestimmt)

- gegeben: Datenreihe mit m Datenpunkten
- $$\begin{pmatrix} f(x_1) \\ \vdots \\ f(x_m) \end{pmatrix} = \begin{pmatrix} f_1(x_1) & \dots & f_n(x_1) \\ \vdots & & \vdots \\ f_1(x_m) & \dots & f_n(x_m) \end{pmatrix} \cdot \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$$
- A (Normalenmatrix) A c Lösung falls nicht überbestimmt
- Pseudoinverse: $A^+ = (A^T A)^{-1} \cdot A^T$
- falls A invertierbar: $A^+ = A^{-1}$ ($AC = y$)
- Lösung: $c = A^+ y$ $A^T A \cdot c = A^T y$ Normalengl.

Heap:

- best vollständiger Binärbaum
- kein abstrakter Datentyp
- MinHeap: Vaterknoten \leq Sohn
- minHeapify: $O(\log n)$
- Knoten ablesen kann, bis Min-Heap-Eigenschaft wiederhergestellt ist
- Heap erzeugen: $O(n)$
 - Binärbaum erzeugen
 - min Heapify für alle Knoten anwenden

Minimales Spannbaum

- A. von Kruskal: $O(|E| \log |V|)$
 - Greedy
- A. von Prim: $O(|E| \log |V|)$
 - Greedy
 - Implementierung: Priority Queue