Analyse von Algorithmen

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 1. März 2023
Vorlesungsfolien

Effizient berechenbare Probleme

„For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing.“

Francis Sullivan

Analyse von Algorithmen

Maße für die Effizienz von Algorithmen:

- Laufzeit
- Speicherplatz
- Besondere charakterisierende Parameter (problemabhängig)

Beispiel für problemabhängige Parameter: Sortieren

- Anzahl der Vergleichsoperationen
- Anzahl der Bewegungen der Datensätze

Laufzeit: Maschinenmodell

Laufzeit: Für die Betrachtung müssen wir eine Maschine festlegen, auf der wir rechnen.

RAM (*Random Access Machine*):

- Es gibt genau einen Prozessor
- Alle Daten liegen im Hauptspeicher
- Die Speicherzugriffe dauern alle gleich lang

Laufzeit: Primitive Operationen

Zeit für eine Operation: Wenn nichts Genaueres spezifiziert wird, dann zählen wir die primitiven Operationen eines Algorithmus als jeweils eine Zeiteinheit.

Primitive Operationen:

- Zuweisungen: $a \leftarrow b$
- Arithmetische Befehle: $a \leftarrow b \circ c$ mit $\circ \in \{+, -, \cdot, /, \mod, \ldots\}$
- Logische Operationen: $\wedge$, $\vee$, $\neg, \ldots$
- Vergleichsoperatoren (z.B. bei Verzweigungen): **if** $a \diamond b$ ... **else** ... mit $\diamond \in \{<, \leq, =, \neq, >, \geq\}$

Wir vernachlässigen:

- Indexrechnung
- Typ der Operanden
- Länge der Operanden

Laufzeit eines Algorithmus

Laufzeit:

- Ist die Anzahl der vom Algorithmus ausgeführten primitiven Operationen.
- Die Laufzeit $T(n)$ wird als Funktion der Eingabegröße n beschrieben.

Beispiel: Sortieren von 1000 Zahlen dauert länger als das Sortieren von 10 Zahlen.

Instanz: Eine Eingabe wird auch als Instanz (*instance*) des Problems, das durch den Algorithmus gelöst wird, bezeichnet.

Laufzeitanalyse

Worst-Case-Laufzeit: Ist die größtmögliche Laufzeit eines Algorithmus bei einer Eingabe mit Größe n.

- Erfasst allgemein die Effizienz in der Praxis.
- Pessimistische Sichtweise, eine Alternative ist aber schwer zu finden.

Average-Case-Laufzeit: Ist die durchschnittliche Laufzeit eines Algorithmus über alle möglichen gültigen Eingaben mit Größe n.

- Es ist allerdings schwer (und oft unmöglich) reale Instanzen durch zufällige Verteilungen exakt zu modellieren.
- Algorithmen, die an bestimmte Eingabeverteilungen angepasst werden, können für andere Eingaben schlechte Ergebnisse liefern.

Best-Case-Laufzeit: Ist die Laufzeit eines Algorithmus bei der bestmöglichen Eingabe mit Größe n.

- Untere Schranke, die nur in Spezialfällen erreicht wird.

Wichtig: In allen drei Fällen wird die Laufzeit als Funktion der Eingabegrösse n angegeben.

Konventionen für Pseudocode

Konventionen für Pseudocode

Schlüsselwörter:

- Schlüsselwörter werden fett und hervorgehoben dargestellt.
- Beispiele dafür sind **while**, **for**, **if**.

Zuweisung: Mit $\leftarrow$ (z.B. i $\leftarrow$ 1).

Vergleich: Mit $<$, $\leq$, $=$, $\neq$, $>$, $\geq$ (z.B. **if** x $=$ 10 ...)

Negation: Mit ! (z.B. **if** !finished ...)

Bedingungen:

- Bedingungen werden nicht geklammert, wenn die Auswertung aus dem Kontext ersichtlich ist.
- Komplexe Bedingungen werden mit ganzen Sätzen beschrieben (z.B. **while** ein Kind ist frei und kann noch eine Gastfamilie auswählen ...).

Konventionen für Pseudocode

Blockstruktur:

- Es werden keine Klammern verwendet.
- Alles auf der gleichen Einrückungsebene gehört zum selben Block.
- Wenn notwendig, werden mehrere Anweisungen durch Beistrich getrennt in eine Zeile geschrieben.

Bei Übungen und bei der Prüfung:

- Sie können Zuweisungen oder Vergleiche auch sprachlich beschreiben.
- Es wird empfohlen, zusätzliche Klammern bei Blöcken zu verwenden. Die Zugehörigkeit einer Anweisung zu einem bestimmten Block muss jedenfalls erkennbar sein.

Wichtigster Punkt: Struktur und Ablauf des Algorithmus müssen erkennbar sein!

Konventionen für Pseudocode

Funktionen:

- Bei ausgewählten wichtigen Algorithmen werden die Funktionsnamen am Anfang (mit etwaigen Parametern) angegeben (z.B. BFS(s): ....).
- Arrays werden immer „per Referenz“ übergeben (d.h. Änderungen am Arrayinhalt in einer Funktion sind auch außerhalb der Funktion sichtbar)
- Wenn weitere Parameter per Referenz übergeben werden, dann wird das explizit angegeben.

Speicher:

- Wir gehen davon aus, dass nicht benötigter Speicher automatisch frei gegeben wird (*garbage collection*).
- Es wird daher nie explizit die Freigabe von Speicher angegeben.

Konventionen für Pseudocode: Beispiel

Beispiel:

- Array A mit n Elementen gegeben.
- Ermittle die Summe aller Elemente, die größer als x sind und gib die Summe aus.

```
Sum(A, x):
sum ← 0
for i ← 0 bis n − 1
    if A[i] > x
        sum ← sum + A[i]
Gib sum aus
```

Anweisung	Konstante Kosten	Wie oft ausgeführt?
$sum \leftarrow 0$	c_1	1
for $i \leftarrow 0$ bis $n-1$	c_2	n
if $A[i] > x$	c_3	n
$sum \leftarrow sum + A[i]$	c_4	j mit $0 \leq j \leq n$
Gib sum aus	c_5	1

Konventionen für Pseudocode: Beispiel

Gesamter Aufwand $T(n)$:

$$\begin{aligned} T(n) &= c_1 \cdot 1 + c_2 \cdot n + c_3 \cdot n + c_4 \cdot j + c_5 \cdot 1 \\ &= (c_2 + c_3) \cdot n + c_4 \cdot j + (c_1 + c_5) \end{aligned}$$

Best-Case: $j = 0$, d.h.

$$\begin{aligned} T(n) &= (c_2 + c_3) \cdot n + (c_1 + c_5) \\ &= a_1 n + b_1 \quad \text{für Konstanten } a_1 \text{ und } b_1 \end{aligned}$$

Worst-Case: $j = n$, d.h.

$$\begin{aligned} T(n) &= (c_2 + c_3) \cdot n + c_4 \cdot n + (c_1 + c_5) \\ &= (c_2 + c_3 + c_4) \cdot n + (c_1 + c_5) \\ &= a_2 n + b_2 \quad \text{für Konstanten } a_2 \text{ und } b_2 \end{aligned}$$

Asymptotisches Wachstum

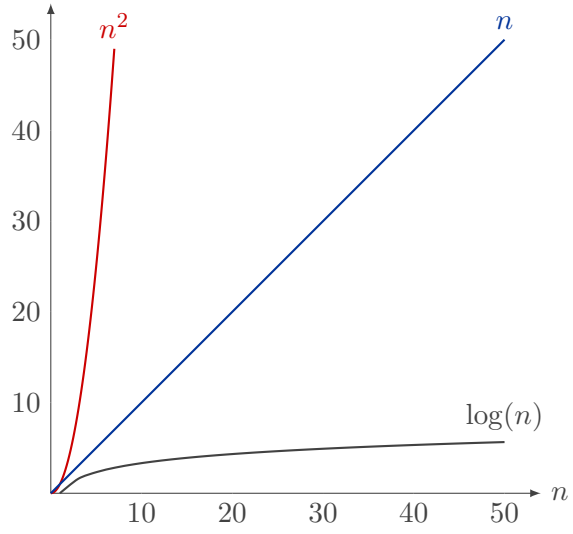
Problematik bei der Analyse von Laufzeiten

- Die Laufzeit wird als Funktion $T(n)$ in Abhängigkeit von der Eingabegröße n angegeben.
- Eine exakte Berechnung der Laufzeit ist meist aber äußerst aufwendig bis praktisch unmöglich, auch wenn ein stark vereinfachtes Maschinenmodell angenommen wird.
- Wir wollen vor allem Algorithmen unabhängig von konkreten Maschinenparametern vergleichen können.

Problematik bei der Analyse von Laufzeiten

- Wir sind vor allem an der Laufzeit großer Instanzen interessiert, da hier der Einfluss der Eingabegröße am deutlichsten ist. Bei kleinen Instanzen sind Unterschiede oft vernachlässigbar.
  - $\rightarrow$ Wir betrachten das asymptotische Wachstum der Laufzeit in Abhängigkeit von der Eingabegröße.
  - $\rightarrow$ Skalierung um einen konstanten Faktor soll dabei keine Rolle spielen.
- Im Folgenden betrachten wir Funktionen deren Definitionsbereich die nicht-negativen ganzen Zahlen sind.

Asymthotisches Wachstum bekannter Funktionen



Schematische Grundidee

- Schranken:
$$g(n) \leq T(n) \leq f(n)$$
- Ignoriere konstante Faktoren:
$$c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot f(n)$$
- Ignoriere kleine n:
$$\text{für } n > n_0 \text{ gilt: } c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot f(n)$$
- Schreibweise:
$$T(n) \in \Omega(g(n)), \quad T(n) \in O(f(n))$$

Asymptotisches Wachstum: Obere Schranke

Allgemein: Sei $f(n)$ eine Funktion. Dann bezeichnet $O(f(n))$ die Menge aller Funktionen, die asymptotisch durch $c \cdot f(n)$ (c ist eine positive Konstante) von oben beschränkt wird.

Definition: Eine Funktion $T(n)$ ist in $O(f(n))$ wenn Konstanten $c > 0$ und $n_0 > 0$ existieren, sodass für alle $n \geq n_0$ gilt: $T(n) \leq c \cdot f(n)$.

Notation: Eigentlich müsste man $T(n) \in O(f(n))$ schreiben. In der Praxis schreibt man aber

- $T(n)$ ist in $O(f(n))$ oder kürzer
- $T(n) = O(f(n))$ (hier aber nicht als Gleichheit zu verstehen!)

Asymptotisches Wachstum: Obere Schranke

Beispiel: $T(n) = 32n^2 + 17n + 5$. Es ist zu zeigen, dass $T(n)$ in $O(n^2)$ ist.

Beweis:

- Wir zeigen, dass es Konstanten $c > 0$ und $n_0 > 0$ gibt, sodass $32n^2 + 17n + 5 \leq c \cdot n^2$ für alle $n \geq n_0$.
- Wir dividieren die Ungleichung durch n^2 und erhalten: $32 + \frac{17}{n} + \frac{5}{n^2} \leq c$.
- $32 + \frac{17}{n} + \frac{5}{n^2} \leq c$ gilt z.B. für alle $n \geq 18$, wenn $c \geq 34$ (dann sind die beiden Brüche jeweils < 1) oder für alle $n \geq 35$, wenn $c \geq 33$.
- Also können wir z.B. $n_0 = 18$ und $c = 34$ wählen.
- Daher können wir schreiben: $T(n) = O(n^2)$. $\square$

Asymptotisches Wachstum: Untere Schranke

Allgemein: Sei $f(n)$ eine Funktion. Dann bezeichnet $\Omega(f(n))$ die Menge aller Funktionen, die asymptotisch durch $c \cdot f(n)$ (c ist eine positive Konstante) von unten beschränkt wird.

Definition: Eine Funktion $T(n)$ ist in $\Omega(f(n))$ wenn Konstanten $c > 0$ und $n_0 > 0$ existieren, sodass für alle $n \geq n_0$ gilt: $T(n) \geq c \cdot f(n)$.

Asymptotisches Wachstum: Untere Schranke

Beispiel: $T(n) = 32n^2 + 17n + 5$. Es ist zu zeigen, dass $T(n)$ in $\Omega(n^2)$ ist.

Beweis:

- Wir zeigen, dass es Konstanten $c > 0$ und $n_0 > 0$ gibt, sodass $32n^2 + 17n + 5 \geq c \cdot n^2$ für alle $n \geq n_0$.
- Wir dividieren die Ungleichung durch n^2 und erhalten: $32 + \frac{17}{n} + \frac{5}{n^2} \geq c$.
- $32 + \frac{17}{n} + \frac{5}{n^2} \geq c$ gilt z.B. für alle $n \geq 1$, wenn $c \leq 32$.
- Wir wählen daher z.B. $n_0 = 1$ und $c = 32$.
- Daher können wir schreiben: $T(n) = \Omega(n^2)$. $\square$

Asymptotisches Wachstum: Scharfe Schranke

Allgemein: Sei $f(n)$ eine Funktion. Dann bezeichnet $\Theta(f(n))$ die Menge aller Funktionen, die asymptotisch gleich großes Wachstum wie $c \cdot f(n)$ besitzen (c ist eine positive Konstante).

Definition: Eine Funktion $T(n)$ ist in $\Theta(f(n))$ wenn $T(n)$ sowohl in $O(f(n))$ als auch in $\Omega(f(n))$ ist.

Asymptotisches Wachstum: Scharfe Schranke

Beispiel: $T(n) = 32n^2 + 17n + 5$. Es ist zu zeigen, dass $T(n)$ in $\Theta(n^2)$ ist.

Beweis:

- Wir haben auf den vorherigen Folien schon gezeigt, dass $T(n)$ in $O(n^2)$ und in $\Omega(n^2)$ ist.
- Aus den beiden Aussagen für obere und untere Schranken folgt, dass $T(n)$ in $\Theta(n^2)$ ist. $\square$

Asymptotisches Wachstum: Scharfe Schranke

Beispiel: $T(n) = 32n^2 + 17n + 5$. Es ist zu zeigen, dass $T(n)$ in $\Theta(n^2)$ ist. Man kann aber auch obere und untere Schranke zugleich in einem Beweis zeigen.

Beweis: Obere und untere Schranke zugleich

- Wir zeigen, dass es Konstanten $c_1 > 0$, $c_2 > 0$ und $n_0 > 0$ gibt, sodass $c_1 \cdot n^2 \leq 32n^2 + 17n + 5 \leq c_2 \cdot n^2$ für alle $n \geq n_0$.
- Wird dividieren die Ungleichung durch n^2 und erhalten: $c_1 \leq 32 + \frac{17}{n} + \frac{5}{n^2} \leq c_2$.
- Das gilt für $c_1 = 32$, $c_2 = 34$ und $n_0 = 18$ (Maximum von $n_0 = 1$ für untere und $n_0' = 18$ für obere Schranke).
- Daher können wir schreiben: $T(n) = \Theta(n^2)$. $\square$

Weitere Beispiele

Allgemein: Man kann jetzt auf ähnliche Weise wie auf den vorherigen Folien für $T(n) = 32n^2 + 17n + 5$ zeigen, dass $T(n)$ z.B. in $O(n^3)$ und $\Omega(n)$ ist.

Hinweis:

- Man versucht normalerweise Schranken zu finden, die möglichst nahe bei $T(n)$ liegen.
- Z. B. bringt es für das obige Beispiel wenig, wenn man zeigt, dass $T(n)$ in $O(n^{10})$ liegt.

Keine Schranken

Beispiel: $T(n) = 32n^2 + 17n + 5$, $T(n)$ ist nicht in $O(n)$.

Beweis: (durch Widerspruch)

- Angenommen, es gibt Konstanten $c > 0$ und $n_0 > 0$, sodass $32n^2 + 17n + 5 \leq c \cdot n$ für alle $n \geq n_0$. Wir formen um:

$$\begin{aligned} 32n + 17 + \frac{5}{n} &\leq c \\ 32n + \frac{5}{n} &\leq c - 17 \\ n + \frac{5}{32n} &\leq \frac{c-17}{32} \\ n \leq \frac{c-17}{32} - \frac{5}{32n} &\leq \frac{c-17}{32} \end{aligned}$$

- Das ist aber falsch für $n > \frac{c-17}{32}$. Widerspruch zur Annahme.
- Daher können wir nicht schreiben: $T(n) = O(n)$. $\square$

Weitere Beispiele: Ähnlich lässt sich z.B. zeigen, dass $T(n)$ nicht in $\Omega(n^3)$, $\Theta(n)$, oder $\Theta(n^3)$ ist.

Richtige Anwendung

Sinnlose Aussage: Jeder vergleichsbasierte Sortieralgorithmus für n Elemente benötigt zumindest $O(n \log n)$ Vergleiche.

- Es sollte Ω für die untere Schranke benutzt werden.

Untere und obere Schranken

Es gilt: Falls $f = O(g)$, dann $g = \Omega(f)$.

Beweis:

- Wir nehmen an $f = O(g)$.
- Daraus folgt, es gibt $c > 0$ und $n_0 > 0$, sodass für alle $n \geq n_0$ gilt, $f(n) \leq c \cdot g(n)$.
- Wir wählen als neue Konstante $c' = 1/c$, und folgern:
- Es gibt $c' > 0$ und $n_0 > 0$, sodass für alle $n \geq n_0$ gilt, $g(n) \geq c' \cdot f(n)$.
- Also ist $g = \Omega(f)$. $\square$

Analog gilt: Falls $f = \Omega(g)$, dann $g = O(f)$.

Insgesamt gilt: $f = \Omega(g)$ genau dann, wenn $g = O(f)$.

Weiters: $f = \Theta(g)$ gilt genau dann, wenn $g = \Theta(f)$ gilt.

Eigenschaft: Additivität

Obere Schranken: Wenn $f = O(h)$ und $g = O(h)$, dann gilt f+g $= O(h)$.

- $f + g$ *bezeichnet die Funktion, die definiert ist, durch* $(f + g)(n) = f(n) + g(n)$.

Beweis:

- Für Konstanten $c > 0$ und $n_0 > 0$ gilt für alle $n \geq n_0$: $f(n) \leq c \cdot h(n)$.
- Für andere Konstanten $c' > 0$ und $n'_0 > 0$ gilt für alle $n \geq n'_0$: $g(n) \leq c' \cdot h(n)$.
- Jetzt wählen wir $n^*_0 = \max(n_0, n'_0)$ und $c^* = c + c'$.
- Daraus ergibt sich: $f(n) + g(n) \leq c \cdot h(n) + c' \cdot h(n)$ und somit $f(n) + g(n) \leq (c + c') \cdot h(n) = c^* \cdot h(n)$ für alle $n \geq \max(n_0, n'_0) = n^*_0$. $\square$

Eigenschaft: Additivität

Beispiel:

- $f(n) = 2n + 3$ $(= O(n^2))$, $g(n) = 5n^2$ $(= O(n^2))$
- $f(n) + g(n) = 5n^2 + 2n + 3$ $(= O(n^2))$

Weiteres Beispiel:

- Angenommen, ein Algorithmus besteht aus zwei Teilen A und B, die hintereinander ausgeführt werden.
- Die Ausführung von A benötigt $O(n^2)$ Zeit.
- Die Ausführung von B benötigt $O(n^3)$ Zeit.
- Der gesamte Algorithmus benötigt dann $O(n^3)$ Zeit.

Hinweis: Das gilt auch für die Summe von mehreren (aber konstant vielen) Funktionen.

Eigenschaft: Additivität

Untere Schranken: Mit ähnlichen Argumenten wie auf der vorherigen Folie kann man zeigen:

Wenn $f = \Omega(h)$ oder $g = \Omega(h)$, dann gilt $f + g = \Omega(h)$.

Scharfe Schranken: Aus den beiden Aussagen für obere und untere Schranken folgt:

Wenn $f = \Theta(h)$ und $g = \Theta(h)$, dann gilt $f + g = \Theta(h)$.

Eigenschaft: Additivität

Eine Anwendung: Wenn $g = O(f)$, dann gilt $f + g = \Theta(f)$.

Beweis:

- Da laut Annahme $g = O(f)$ und klarerweise $f = O(f)$, folgt mittels Additivitätseigenschaft $f + g = O(f)$.
- Es gilt aber auch $f + g = \Omega(f)$, da für alle $n \geq 0$ gilt: $f(n) + g(n) \geq f(n)$.
- Daraus folgt $f + g = \Theta(f)$. $\square$

Summenbeispiel nochmals betrachtet

Beispiel:

- Array A mit n Elementen gegeben.
- Ermittle die Summe aller Elemente, die größer als x sind und gib die Summe aus.

```
Sum(A, x):
sum ← 0
for i ← 0 bis n − 1
    if A[i] > x
        sum ← sum + A[i]
Gib sum aus
```

- Wir haben eine Laufzeit von $T(n) = a_2 n + b_2$ für Konstanten a_2 und b_2 bestimmt. Es gilt also $T(n) = \Theta(n)$.
- Diese asymptotische Abschätzung können wir auch einfacher erreichen:
- Die Schleife wird n mal ausgeführt, der Aufwand im Schleifenkörper sowie die Initialisierung sind konstant $\Theta(1)$. Es ergibt sich die Gesamtlaufzeit $\Theta(n)$.

Eigenschaft: Transitivität

Obere Schranken: Wenn $f = O(g)$ und $g = O(h)$, dann gilt $f = O(h)$.

Beweis:

- Für Konstanten $c > 0$ und $n_0 > 0$ gilt für alle $n \geq n_0$: $f(n) \leq c \cdot g(n)$.
- Für Konstanten $c' > 0$ und $n_0' > 0$ gilt für alle $n \geq n_0'$: $g(n) \leq c' \cdot h(n)$.
- Jetzt wählen wir $n_0^* = \max(n_0, n_0')$ und $c^* = c \cdot c'$.
- Damit ergibt sich: $f(n) \leq c \cdot g(n) \leq c \cdot c' \cdot h(n)$ und somit $f(n) \leq c \cdot c' \cdot h(n) = c^* \cdot h(n)$ für alle $n \geq \max(n_0, n_0') = n_0^*$. $\square$

Beispiel:

- $f(n) = n$, $g(n) = n^2$, $h(n) = n^3$
- $f(n) = O(g(n))$ und $g(n) = O(h(n))$, dann gilt auch $f(n) = O(h(n))$

Eigenschaft: Transitivität

Untere Schranken: Mit ähnlichen Argumenten wie für obere Schranken kann man zeigen:

Wenn $f = \Omega(g)$ und $g = \Omega(h)$, dann gilt $f = \Omega(h)$.

Scharfe Schranken: Aus den beiden Aussagen für obere und untere Schranken folgt:

Wenn $f = \Theta(g)$ und $g = \Theta(h)$, dann gilt $f = \Theta(h)$.

Asymptotische Äquivalenz

Äquivalenz: Die binäre Relation $f = \Theta(g)$ zwischen Funktionen bildet eine Äquivalenzrelation.

Beweis:
Wir haben schon gezeigt:

- $f = \Theta(g)$ gilt genau dann wenn $g = \Theta(f)$ gilt (Symmetrie).
- $f = \Theta(f)$ (Reflexivität).
- Wenn $f = \Theta(g)$ und $g = \Theta(h)$, dann gilt $f = \Theta(h)$ (Transitivität). $\square$

Asymptotische Dominanz

Dominanz von Funktionen:

- f wird von g dominiert, wenn $f = O(g)$ aber nicht $g = O(f)$ gilt.
- f und g gehören nicht zur gleichen Äquivalenzklasse bezüglich asymptotischem Wachstums.
- Wir schreiben dann $f \ll g$.

Alternative Sichtweise

Dominanz: g dominiert f, wenn $\lim_{n\to\infty} f(n)/g(n) = 0$.

Beispiel: Funktion $g(n)$ dominiert $f(n)$

- $g(n) = n^3$ und $f(n) = n^2$
- $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} n^2/n^3 = \lim_{n\to\infty} 1/n = 0$

Beispiel: Keine Dominanz

- $g(n) = 2n^2$ und $f(n) = n^2$
- $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} n^2/2n^2 = \lim_{n\to\infty} 1/2 \neq 0$

Alternative Sichtweise

Hinweis:

- Wir gehen meist von stetigen Funktionen aus.
- Es gibt Paare von Funktionen f und g, sodass weder f die Funktion g dominiert noch g die Funktion f dominiert (z.B. bei nicht stetigen Funktionen).

Allgemein gilt für Polynome: n^a dominiert n^b, wenn $a > b$, da
$\lim_{n\to\infty} n^b/n^a = \lim_{n\to\infty} n^{b-a} = 0$.

Asymptotische Schranken für einige gebräuchliche Funktionen

Polynome: $f(n) = a_0 + a_1 n + \cdots + a_d n^d$ ist in $\Theta(n^d)$ wenn $a_d > 0$.

Beweis:

- Für jeden Koeffizienten a_j für $j < d$ gilt, dass $a_j n^j \leq |a_j| n^d$ für alle $n \geq 1$.
- Daher ist jeder Term in $O(n^d)$.
- Da $f(n)$ die Summe von einer konstanten Anzahl an Funktionen ist, ist auch $f(n)$ in $O(n^d)$ (wegen Additivitätseigenschaft).
- Da ein Summand in $f(n)$ in $\Omega(n^d)$ ist, ist $f(n)$ in $\Omega(n^d)$ (wegen Additivitätseigenschaft) und daher: $f(n) = \Theta(n^d)$. $\square$

Polynomialzeit: Laufzeit liegt in $O(n^d)$ für eine Konstante d, wobei d unabhängig von der Eingabegröße n ist.

Asymptotische Schranken für einige gebräuchliche Funktionen

Logarithmen: Für Konstanten a, $b > 0$ gilt $\Theta(\log_a n) = \Theta(\log_b n)$.

Beweis: Wir berücksichtigen folgende Identität

$$\log_a n = \frac{\log_b n}{\log_b a}$$

$\log_b a$ ist aber eine Konstante und daraus folgt, dass $\log_a n = \Theta(\log_b n)$. $\square$

Hinweis: Daher braucht bei asymptotischen Angaben die Basis von Logarithmen nicht angegeben werden.

Vergleich zu Polynomfunktionen: Für jede Konstante $\varepsilon > 0$ gilt, $\log n \ll n^{\varepsilon}$.

- *log wächst asymptotisch langsamer als jede Polynomfunktion*

Asymptotische Schranken für einige gebräuchliche Funktionen

Exponentiell: Für alle Konstanten $c > 1$ und $d > 0$ gilt, $n^d \ll c^n$).

- *Jede Exponentialfunktion wächst asymptotisch schneller als jede Polynomfunktion*

Hinweis: Für Konstanten $1 < c_1 < c_2$ gilt (im Gegensatz zu den Basen bei den Logarithmen) $c_1^n \ll c_2^n$.

Beweis: $c_1^n = O(c_2^n)$ ist klar. Angenommen $c_2^n = O(c_1^n)$. Daraus würde folgen, dass $c_2^n \leq c \cdot c_1^n$ für eine Konstante $c > 0$ gilt. Umgeformt ergibt das $(c_2/c_1)^n \leq c$, was aber nicht sein kann, da der Ausdruck $(c_2/c_1)^n$ gegen Unendlich geht (da wir ja $1 < c_1 < c_2$ angenommen haben). $\square$

Asymptotische Schranken: Verhältnisse

Verhältnisse: Ordnung der Dominanz von Funktion $f(n)$ für $n \geq 0$ ($a \ll b$ bedeutet b dominiert a)

$$1 \ll \log n \ll \sqrt{n} \ll n \ll n \log n \ll n^{1+\varepsilon} \ll n^2 \ll n^3 \ll n^k \ll c^n \ll n! \ll n^n$$

Hinweise:

- 1 bezeichnet die konstante Funktion $f(n) = 1$.
- $n^{1+\varepsilon}$ für $0 < \varepsilon < 1$.
- $c > 1$
- $k > 3$

Laufzeiten einiger gebräuchlicher Funktionen

Konstantes Wachstum: $O(1)$

Konstantes Wachstum: Unabhängig von der Eingabegröße n ist der Aufwand für eine Operation konstant.

Hinweis: Konstant bedeutet hier nicht, dass die Operation auf unterschiedlichen Systemen gleich viel Aufwand benötigt. Vielmehr wird sich der Aufwand auf unterschiedlichen Systemen sehr wohl unterscheiden. Auf einem System wird aber der Aufwand für die Operation unabhängig von n gleich bleiben.

Beispiele:

- Addition zweier Zahlen
- Zugriff auf das i-te Element in einem Array der Größe n.

Logarithmisches Wachstum: $O(\log n)$

Logarithmisches Wachstum: Wenn z.B. die Eingabegröße in jedem Schritt halbiert wird.

Binäre Suche: Binäre Suche nach einem gegebenen Wert in einem aufsteigend sortierten Array A (siehe VU Einführung in die Programmierung 1)

```
Ermittle den mittleren Index s im Array A
if A[s] = gesuchter Wert
    Gib aus, dass Wert gefunden wurde
elseif gesuchter Wert ist kleiner als A[s]
    Wende binäre Suche auf das Teilarray links von s an
elseif gesuchter Wert ist größer als A[s]
    Wende binäre Suche auf das Teilarray rechts von s an
```

Lineares Wachstum: $O(n)$

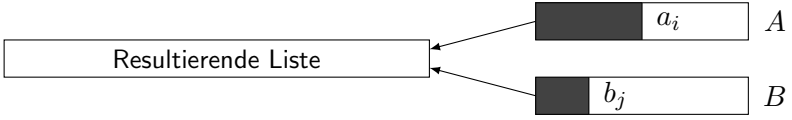
Lineares Wachstum: Laufzeit ist proportional zur Eingabegröße.

Maximumsuche: Ermittle das Maximum von n Zahlen $a_1, \ldots, a_n$.

```
max ← a_1
for i ← 2 bis n
    if a_i > max
        max ← a_i
```

Lineares Wachstum: $O(n)$

Merge: Verschmelze zwei sortierte Listen $A = a_1, a_2, \ldots, a_n$ und $B = b_1, b_2, \ldots, b_n$ zu einer sortierten Liste.



$i \leftarrow 1,\ j \leftarrow 1$
while beide Listen noch nicht komplett bearbeitet
    if $a_i \leq b_j$
        Füge a_i zur Ergebnisliste hinzu und erhöhe i
    else
        Füge b_j zur Ergebnisliste hinzu und erhöhe j
Füge den Rest der noch nicht komplett bearbeiteten Liste zur Ergebnisliste hinzu

$O(n \log n)$ Wachstum

$O(n \log n)$ Laufzeit: Tritt z.B. bei „teile und herrsche“(*Divide-and-Conquer*)-Algorithmen auf.

- *Wird auch als leicht überlinear bezeichnet*

Sortieren:

- Naives Sortieren wie z.B. Bubblesort (siehe VU Einführung in die Programmierung 1) hat eine Laufzeit von $O(n^2)$.
- Schnelleres Sortieren ist möglich. Mergesort ist ein Sortieralgorithmus, der garantiert nur $O(n \log n)$ Vergleiche durchführt. Wird im Laufe dieser Vorlesung noch besprochen.

$O(n \log n)$ Wachstum

Beispiel - Größtes leeres Intervall: Es seien n Zeitpunkte $x_1, \ldots, x_n$ gegeben, zu denen Kopien einer Datei am Server abgelegt werden. Wie groß ist das größte Intervall, in dem keine Kopien am Server ankommen?

$O(n \log n)$ Lösung: Sortiere die Zeitpunkte. Durchlaufe die Liste in sortierter Reihenfolge und berechne das größte Intervall zwischen zwei aufeinanderfolgenden Zeitpunkten.

Quadratisches Wachstum: $O(n^2)$

Quadratisches Wachstum: Betrachte alle Paare von Elementen.

Dichtestes Punktpaar: Gegeben sei eine Liste von n Punkten in einer Ebene $(x_1, y_1), \ldots, (x_n, y_n)$ und es sollen die zwei am dichtesten beieinander liegenden Punkte gefunden werden.

$O(n^2)$ Lösung: Überprüfe alle Paare von Punkten.

```
min ← (x_1 − x_2)^2 + (y_1 − y_2)^2
for i ← 1 bis n − 1
    for j ← i + 1 bis n
        d ← (x_i − x_j)^2 + (y_i − y_j)^2
        if d < min
            min ← d
```

- Es muss nicht die Wurzel gezogen werden

Quadratisches Wachstum: $O(n^2)$

Quadratisches Wachstum: Ablauf des Algorithmus:

Wert für i	Werte für j	Anzahl der Durchläufe
1	$2 \cdots$ n	$n-1$
2	$3 \cdots$ n	$n-2$
$\cdots$	$\cdots$	$\cdots$
$n-1$	n	1

Summe: $(n-1)+(n-2)+\cdots+2+1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \Theta(n^2)$

Anmerkung: $\Omega(n^2)$ erscheint unvermeidbar, aber es gibt für dieses Problem einen Algorithmus, der effizienter als der offensichtliche ist.

Kubisches Wachstum: $O(n^3)$

Kubisches Wachstum: Zähle alle Dreiergruppen von Elementen.

Disjunkte Mengen: Gegeben seien n Mengen $S_1, \ldots, S_n$, wobei jede Menge eine Teilmenge von $\{1, 2, \ldots, n\}$ ist. Existiert ein Paar von Mengen, das disjunkt ist?

$O(n^3)$ Lösung: Bestimme für jedes Paar von Mengen, ob es disjunkt ist.

```
for i ← 1 bis n − 1
    for j ← i + 1 bis n
        foreach Element p von S_i
            Bestimme ob p auch in S_j vorhanden ist
        if kein Element von S_i ist in S_j vorhanden
            Gib aus, dass S_i und S_j disjunkt sind
```

Polynomielle Laufzeit: $O(n^k)$ Wachstum

Teilsummenproblem mit k Zahlen: Gegeben seien n verschiedene ganze Zahlen in einem Array A. Gibt es genau k Zahlen ($k < n$), sodass die Summe dieser Zahlen einer gegebenen Zahl x entspricht?

Beispiel: $O(n^4)$ Lösung für $k = 4$ mit Ausgabe

```
for i ← 0 bis n − 4
    for j ← i + 1 bis n − 3
        for k ← j + 1 bis n − 2
            for l ← k + 1 bis n − 1
                if (A[i] + A[j] + A[k] + A[l]) = x
                    Gib A[i], A[j], A[k] und A[l] aus
```

Hinweis: Dieses Problem kann man effizienter lösen!

Exponentielles Wachstum: $O(c^n)$

Teilsummenproblem (Subset Sum): Gegeben sei eine Menge von ganzen Zahlen. Gibt es eine Untermenge dieser Zahlen, deren Elementsumme einer vorgegebenen Zahl x entspricht?

Lösung:

```
foreach Teilmenge S von Zahlen
    Überprüfe, ob die Summe der Elemente in S gleich x ist
```

Exponentielles Wachstum: Eine endliche Menge mit n Elementen hat genau 2^n Teilmengen und jede dieser Teilmengen muss untersucht werden.

Laufzeiten am Beispiel von Stable Matching

Gale–Shapley-Algorithmus (Wiederholung)

```
Kennzeichne jede Familie/jedes Kind als frei
while ein Kind ist frei und kann noch eine Familie auswählen
    Wähle solch ein Kind s aus
    f ist erste Familie in der Präferenzliste von s,
    die s noch nicht ausgewählt hat
    if f ist frei
        Kennzeichne s und f als einander zugeordnet
    elseif f bevorzugt s gegenüber ihrem aktuellen Partner s'
        Kennzeichne s und f als einander zugeordnet und s' als frei
    else
        f weist s zurück
```

Implementierung von Stable Matching

Ausgangssituation: Wir wissen, dass der Gale-Shapley-Algorithmus ein Stable Matching zwischen n Kindern und n Familien mit $\leq n^2$ Iterationen findet.

Ziel: Wir möchten zeigen, dass der gesamte Algorithmus mit einer Laufzeit in $O(n^2)$ implementiert werden kann.

Überlegungen zur Datenstruktur:

- Jedes Kind und jede Familie hat eine Präferenzliste aller Mitglieder der anderen Gruppe. Wie repräsentiert man so eine Rangfolge?
- Außerdem muss in jedem Schritt das aktuelle Matching gespeichert werden.
- Wir werden zeigen, dass dafür Arrays und Listen ausreichen.

Array

Array:

- Ist eine statische Datenstruktur mit im Speicher sequentiell abgelegten Elementen.
- Alle Elemente haben den gleichen Typ.
- Zugriff über Index in konstanter Zeit möglich.

Beispiel:

0	1	2	3	4	5	6	7
0	7	2	6	0	3	4	5

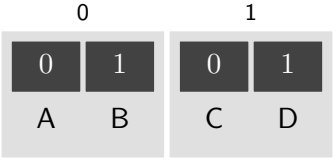
Zweidimesionale Arrays

Zweidimesionale Arrays: Zweidimesionale (oder mehr dimensionale) Arrays können als Array von Arrays realisiert werden.

Zweidimesionales Array:

	0	1
0	A	B
1	C	D

Array von Arrays:

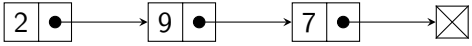


Verkettete Liste

Einfach verkettete Liste:

- Ist eine dynamische Datenstruktur.
- Speicherung von miteinander in Beziehung stehenden Knoten (durch Zeiger auf nachfolgende Knoten).
- Anzahl der Knoten muss im Vorhinein nicht bekannt sein.

Beispiel:



Implementierungsdetails: Siehe VU Einführung in die Programmierung 2.

Arrays und Listen

Typische Operationen: Worst-Case-Komplexität von Operationen auf einem unsortierten Array und einer unsortierten einfach verketteten Liste.

Tabelle: Vergleich von Operationen auf Arrays und auf Listen.

Operation	**Array**	**Liste**
Beliebiges Element suchen	$\Theta(n)$	$\Theta(n)$
Auf Element mit Index i zugreifen	$\Theta(1)$	$\Theta(n)$
Element am Anfang einfügen	$\Theta(n)$	$\Theta(1)$

Grundlegende Datenstruktur

Vereinfachung:

- Es gibt sowohl n Kinder und n Familien.
- Kinder und Familien werden jeweils mit einer Nummer $0, .., n-1$ assoziiert.
- Damit kann man ein Array anlegen, dessen Indexwerte sich aus diesen Nummern ergeben.

Präferenzlisten:

- Jedes Kind und jede Familie hat eine Präferenzliste in Form eines Arrays.
- `FPref[`s`,`i`]` ist die Familie mit Index i in der Liste von Kind s (diese Familie hat bei Kind s den Rang $i+1$).
- `SPref[`f`,`i`]` ist das Kind mit Index i in der Liste der Familie f (dieses Kind hat bei Familie f den Rang $i+1$).

Platzbedarf: Es gibt n Kinder und n Familien, beide haben je ein Array der Länge n, daher $\Theta(n^2)$ (wie bei Laufzeit asymptotisch abgeschätzt).

Grundlegende Datenstruktur

Beispiel allgemein:

	1.	2.	3.
Xaver	Abel	Boole	Church
Yvonne	Boole	Abel	Church
Zola	Abel	Boole	Church

Mapping für Array `SPref`: Xaver $= 0$, Yvonne $= 1$, Zola $= 2$, Abel $= 0$, Boole $= 1$, Church $= 2$

	0	1	2
0	0	1	2
1	1	0	2
2	0	1	2

Schritte in der Iteration

Erster Schritt: Ein freies Kind finden.

Lösung:

- Verkettete Liste `SFree` mit allen freien Kindern. Zu Beginn enthält `SFree` alle Kinder. (Aufwand der Initialisierung $\Theta(n)$)
- Das erste Element s wird ausgewählt (konstanter Aufwand).
- s wird aus der Liste gelöscht und möglicherweise ein s' (wenn ein anderes Kind s' wieder frei wird) in die Liste an der ersten Stelle eingefügt.
- Dieser Schritt kann in konstanter Zeit ausgeführt werden.

Schritte in der Iteration

Zweiter Schritt: Man muss für ein Kind jene Familie mit dem höchsten Rang finden, die es noch nicht ausgewählt hat.

Lösung:

- Dazu wird ein Array `Next` benutzt, das für jedes Kind die Position (Index in einem `SPref`-Array) der nächsten auszuwählenden Familie angibt.
- Zunächst wird für jedes Kind s das Array mit `Next[`s`]` $= 0$ initialisiert. (Aufwand proportional zu n)
- Wenn ein Kind s eine Familie auswählen möchte, dann wählt es die Familie $f =$ `SPref[`s`, Next[`s`]]` aus.
- Wird eine Familie ausgewählt, dann wird `Next[`s`]` um 1 erhöht (unabhängig vom Ergebnis).
- Alle Operationen benötigen konstante Zeit und daher kann dieser Schritt in konstanter Zeit ausgeführt werden.

Schritte in der Iteration

Dritter Schritt: Für eine Familie f müssen wir entscheiden, ob f schon zugewiesen wurde und wer das zugewiesene Kind ist.

Lösung:

- Wir verwenden ein Array `Current` der Länge n.
- `Current[`f`]` ist der Partner von f.
- Hat f keinen Partner, dann wird das durch eine spezielle Zahl (z.B. -1) angezeigt.
- Am Anfang werden alle Einträge des Arrays auf die spezielle Zahl gesetzt. (Aufwand proportional zu n)
- Dieser Schritt kann daher auch in konstanter Zeit durchgeführt werden.

Schritte in der Iteration

Vierter Schritt: Für eine Familie f und zwei Kinder s und s' müssen wir entscheiden, ob s oder s' von f bevorzugt wird.

Lösung:

- Dazu wird am Anfang ein $n \times n$ Array `Ranking` erstellt.
- `Ranking`$[f, s]$ enthält den Rang von Kind s in der sortierten Reihenfolge von f.
- Für jede Familie muss daher nur die Präferenzliste einmal durchlaufen werden.
- Der Aufwand dafür ist proportional zu n^2, aber die Erstellung von `Ranking` wird vor(!) dem eigentlichen Algorithmus ausgeführt.
- Bei einer Iteration des Algorithmus müssen daher nur mehr die zwei Einträge `Ranking`$[f, s]$ und `Ranking`$[f, s']$ verglichen werden.
- Damit ist auch dieser Schritt in konstanter Zeit ausführbar.

Erstellung des Arrays Ranking

Für jede Familie (Zeile): Erzeuge eine inverse Liste der Präferenzliste der Kinder.

Beispiel: Abel (hat Nummer 0)

Präferenz	0	1	2	3	4	5	6	7
0	7	2	6	0	3	4	5	1

Ranking	0	1	2	3	4	5	6	7
0	3	7	1	4	5	6	2	0

Abel bevorzugt Kind 2 gegenüber 5 da $\underbrace{\texttt{Ranking[0,2]}}_{1} < \underbrace{\texttt{Ranking[0,5]}}_{6}$

```
for i ← 0 bis n − 1
    for j ← 0 bis n − 1
        Ranking[i,FPref[i,j]] ← j
```

Schritte in der Iteration

Laufzeit: In der Initialisierungsphase werden zwei $\Theta(n^2)$ Arrays erstellt. Die Laufzeit dieser Phase liegt in $O(n^2)$. Alle vier Schritte der Iteration lassen sich in konstanter Zeit ausführen. Daher liegt die Laufzeit für den Algorithmus in $O(n^2)$.

Implementierung: Vom abstrakten Pseudocode zu einer Beschreibung in genauerem Pseudocode ist aber noch Einiges an Arbeit zu leisten!

Pseudocode mit Arrays

Gegeben: Arrays SPref und FPref

```
for i ← 0 bis n − 1
    for j ← 0 bis n − 1
        Ranking[i,FPref[i,j]] ← j
for i ← 0 bis n − 1
    Next[i] ← 0
    Current[i] ← -1
SFree ← Liste aller Kinder
while SFree ist nicht leer
    s ← erstes Element aus SFree, lösche erstes Element aus SFree
    f ← SPref[s,Next[s]]
    s' ← Current[f]
    if s' = -1
        Current[f] ← s
    elseif Ranking[f,s] < Ranking[f,s']
        Current[f] ← s
        Füge s' in SFree an erster Stelle ein
    else
        Füge s in SFree an erster Stelle ein
    Next[s] ← Next[s] + 1
```

Sortieren als praktisches Beispiel

Sortieren

Sortieren: Gegeben seien n Elemente, die aufsteigend angeordnet werden sollen.

Anwendungen:

- Sortiere eine Liste von Namen
- Organisiere eine MP3-Bibliothek
- Zeige Google PageRank Resultate an
- Liste RSS-Feedelemente in umgekehrt chronologischer Reihenfolge auf

Offensichtliche Anwendungen

- Finde den Median
- Finde das näheste Paar
- Binäre Suche

Probleme werden leichter lösbar, wenn die Elemente in sortierter Reihenfolge vorliegen.

Rahmenbedingungen und Annahmen

Internes Sortieren: Alle Daten sind im Hauptspeicher.

Datenstruktur: Array mit n Elementen, $A[0], \ldots, A[n-1]$, auf denen eine Ordnungsrelation „$\leq$“ definiert ist.

Hinweis: Aus Gründen der Einfachheit gehen wir in den folgenden Beispielen von einem Array mit n Werten aus, die sortiert werden. In der Praxis können bei diesen Werten noch zusätzliche Informationen vorhanden sein, d.h. die Werte dienen als Schlüssel zum Auffinden von Informationen.

Elementare Sortierverfahren

Bubblesort:

- Aus VU Einführung in die Programmierung 1 bekannt.
- Laufzeit liegt im Worst- und Average-Case in $\Theta(n^2)$. Im Best-Case kann die Laufzeit bei optimierten Implementierungen in $\Theta(n)$ liegen.

Weitere Beispiele für elementare Verfahren:

- Sortieren durch Minimumsuche (*Selectionsort*)
- Sortieren durch Einfügen (*Insertionsort*)

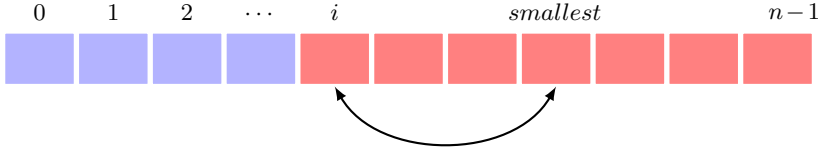
Selectionsort (Sortieren durch Minimumsuche)

Selectionsort: Selectionsort sucht in jeder Iteration das kleinste Element in der noch unsortierten Teilfolge, entnimmt es und fügt es dann an die sortierte Teilfolge an.

Beispiel: Implementierung mit einem Array A mit n Elementen.

```
Selectionsort(A):
for i ← 0 bis n − 2
    smallest ← i
    for j ← i + 1 bis n − 1
        if A[j] < A[smallest]
            smallest ← j
    Vertausche A[i] mit A[smallest]
```

Selectionsort: Schematische Darstellung

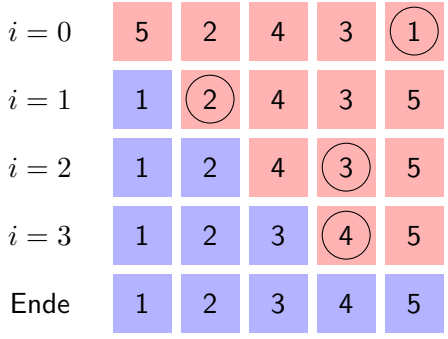


Sortierter Teil
Unsortierter Teil

Selectionsort: Beispiel

Beispiel:

- Ausgangssequenz: 5, 2, 4, 3, 1.
- Minimum im jeweiligen Durchlauf eingekreist.
- Die sortierte Teilfolge wird von links her aufgebaut.



Selectionsort: Analyse

Laufzeit: Die Laufzeit liegt immer (Best/Average/Worst-Case) in $\Theta(n^2)$.

Analyse:

- Innere Schleife wird beim ersten Durchlauf der äußeren Schleife $n-1$-mal ausgeführt.
- Im nächsten Durchlauf $n-2$, dann $n-3$-mal usw.
- $(n-1)+(n-2)+(n-3)+\cdots+2+1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \Theta(n^2)$

Anzahl der Vertauschungen: Eine Vertauschung pro äußerem Schleifendurchlauf, d.h. $\Theta(n)$

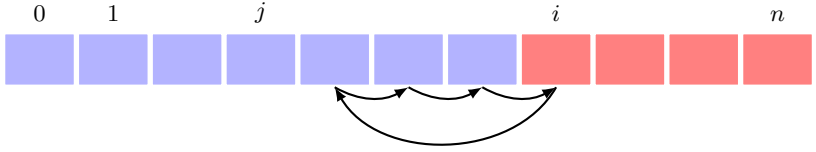
Insertionsort (Sortieren durch Einfügen)

Insertionsort: Insertionsort entnimmt der unsortierten Teilfolge ein Element und fügt es an richtiger Stelle in die (anfangs leere) sortierte Teilfolge ein.

Beispiel: Implementierung mit einem Array A mit n Elementen.

```
Insertionsort(A):
for i ← 1 bis n − 1
    key ← A[i], j ← i − 1
    while j ≥ 0 und A[j] > key
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← key
```

Insertionsort: Schematische Darstellung

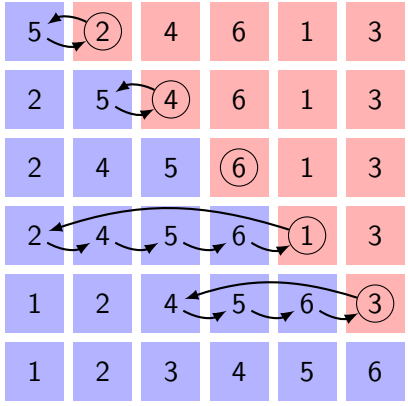


Sortierter Teil
Unsortierter Teil

Insertionsort: Beispiel

Beispiel:

- Ausgangssequenz: 5, 2, 4, 6, 1, 3.
- Jede Zeile zeigt das Einordnen des aktuellen Elements in die sortierte Teilfolge.



Insertionsort: Analyse

Laufzeit:

- Im Best-Case ist das Array schon sortiert und die Laufzeit liegt in $\Theta(n)$ (while-Schleife wird nie durchlaufen).
- Im Worst-Case muss die innere Schleife i-mal ausgeführt werden, d.h. die Laufzeit ist die Summe von

$$1 + 2 + 3 + \cdots + (n-2) + (n-1)$$

  und damit liegt die Laufzeit wieder in $\Theta(n^2)$.
- Es kann gezeigt werden, dass im Average-Case die Laufzeit auch in $\Theta(n^2)$ liegt. Der Beweis ist kompliziert. Er beruht auf der Idee, dass die innere Schleife im Mittel $\frac{i}{2}$-Mal ausgeführt wird.

Anzahl der Vertauschungen: Wie oben, da Insertionsort in jedem Schritt der inneren Schleife Vertauschungen vornehmen muss.

Sortieren: Ausblick

Elementare Sortierverfahren: Die Laufzeit liegt im Worst- und Average-Case immer in $\Theta(n^2)$.

Frage: Kann man im Worst- und Average-Case schneller sortieren?

Antwort: Ja. Die Erklärung folgt im Kapitel über Divide-and-Conquer-Algorithmen.

Polynomialzeit

Polynomialzeit

Brute-Force-Methode: Für viele nicht triviale Probleme gibt es einen einfachen Algorithmus, der jeden möglichen Fall überprüft.

- In der Praxis häufig zu zeitaufwendig.
  - $n!$ *Möglichkeiten für Stable-Matching mit* n *Kindern und* n *Familien*

Polynomialzeit:

> Es existiert eine Konstante $d \geq 1$, sodass die Laufzeit in $O(n^d)$ liegt.

Erklärung: Ein Algorithmus läuft in Polynomialzeit, wenn die Laufzeit höchstens polynomiell mit der Größe der Eingabe n des Problems wächst.

Warum das wichtig ist

Tabelle: Laufzeiten (aufgerundet) von Algorithmen mit unterschiedlichem Laufzeitverhalten für steigende Eingabegrößen auf einem Prozessor, der eine Million primitive Operationen pro Sekunde ausführen kann. Wenn die Laufzeit 10^{25} Jahre überschreitet, dann wird das als *sehr lange* angeführt.

		n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n =$	10	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
$n =$	30	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 min	10^{25} Jahre
$n =$	50	< 1 s	< 1 s	< 1 s	< 1 s	11 min	36 Jahre	sehr lange
$n =$	100	< 1 s	< 1 s	< 1 s	1 s	$12,892$ Jahre	10^{17} Jahre	sehr lange
$n =$	$1,000$	< 1 s	< 1 s	1 s	18 min	sehr lange	sehr lange	sehr lange
$n =$	$10,000$	< 1 s	< 1 s	2 min	12 Tage	sehr lange	sehr lange	sehr lange
$n =$	$100,000$	< 1 s	2 s	3 Stunden	32 Jahre	sehr lange	sehr lange	sehr lange
$n =$	$1,000,000$	1 s	20 s	12 Tage	$31,710$ Jahre	sehr lange	sehr lange	sehr lange

Worst-Case Polynomialzeit

Definition: Wir nennen einen Algorithmus effizient, wenn seine Laufzeit polynomiell in der Eingabegröße ist.

Cobham–Edmonds Annahme: Effiziente Lösbarkeit mit Lösbarkeit in Polynomialzeit gleichzusetzen, geht auf Alan Cobham and Jack Edmonds zurück, die das in den 1960er-Jahren vorgeschlagen haben.

Rechtfertigung:

- In der Praxis haben polynomielle Algorithmen meist kleine Konstanten und kleine Exponenten (man kann natürlich pathologische Fälle konstruieren ...).
- Das Überwinden der exponentiellen Schranke von Brute-Force-Algorithmen legt meist eine wichtige Struktur des Problems offen.
- Diese Cobham–Edmonds-Annahme hat sich weitgehend durchgesetzt und die Informatikforschung der letzten 50 Jahre geprägt.

Edmonds 1963, Cobham 1965

PATHS, TREES, AND FLOWERS

JACK EDMONDS

1. Introduction. A *graph* G for purposes here is a finite set of elements called *vertices* and a finite set of elements called *edges* such that each edge *meets* exactly two vertices, called the *end-points* of the edge. An edge is said to *join* its end-points.

A *matching* in G is a subset of its edges such that no two meet the same vertex. We describe an efficient algorithm for finding in a given graph a matching of maximum cardinality. This problem was posed and partly solved by C. Berge; see Sections 3.7 and 3.8.

THE INTRINSIC COMPUTATIONAL DIFFICULTY OF FUNCTIONS

ALAN COBHAM
I.B.M. Research Center, Yorktown Heights, N. Y., U.S.A.

The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why? I grant I have put the first of these questions rather loosely; nevertheless, I think the answer, ought to be: *yes*. It is the second, which asks for a justification of this answer which provides the challenge.

Worst-Case Polynomialzeit

So effizient wie möglich!

- Wenn wir ein Problem in Polynomialzeit lösen können, wollen wir natürlich einen Algorithmus mit möglichst kleiner polynomieller Laufzeit finden.
- Es bedarf oft viel Aufwand, z.B. eine Laufzeit von $O(n^3)$ auf $O(n^2)$ zu reduzieren, oder von $O(n^2)$ auf $O(n \log n)$.
- In den nächsten Abschnitten werden wir verschiedene algorithmische Probleme betrachten und möglichst effiziente Algorithmen zu ihrer Lösung kennenlernen.