Analyse von Algorithmen

Algorithmen und Datenstrukturen 1
VU 186.813, 4h, 6 ECTS, SS 2016
Letzte Änderung: 17. März 2016

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: $\wedge, \vee, \neg$, ...
- 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 schwer (oder 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.

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 und Wertebereich die nicht-negativen ganzen Zahlen sind.

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).
- 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 $n_0 = 1$ 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.
- Daraus 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 $c > 0$, $n_0 > 0$ 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 $c' > 0$, $n_0 > 0$ 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 $n \geq \max(n_0, n_0') = n_0^*$. □

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 Funktionen.

Eigenschaft: Additivität

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

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

Scharfe Schranken: Daraus 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 $f + g = \Theta(f)$. $\square$

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 $n \geq \max(n_0, n'_0) = n^*_0$. □

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: Daraus 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).
- Analog können wir zeigen, dass $f(n)$ in $\Omega(n^d)$ ist und daher gilt dann: $f(n) = \Theta(n^d)$. $\square$

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, $\log n = O(n^\varepsilon)$.

- *log wächst asymptotisch langsamer als jede Polynomfunktion*

Asymptotische Schranken für einige gebräuchliche Funktionen

Exponentiell: Für alle Konstanten $r > 1$ und $d > 0$ gilt, $n^d = O(r^n)$.

- *Jede Exponentialfunktion wächst asymptotisch schneller als jede Polynomfunktion*

Hinweis: Für unterschiedliche Konstanten $r > s > 1$ gilt (im Gegensatz zu den Basen bei den Logarithmen) nicht $r^n = \Theta(s^n)$.

Beweis: $r^n = \Theta(s^n)$ erfordert, dass $r^n \leq c \cdot s^n$ für eine Konstante $c > 0$ gilt. Umgeformt ergibt das $(r/s)^n \leq c$, was aber nicht sein kann, da der Ausdruck $(r/s)^n$ gegen Unendlich geht (da wir ja $r > s > 1$ 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 $0 < \varepsilon < 1$.
- $c > 1$
- $k > 3$

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 $=$ oder $\neq$ (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 Mann ist frei und kann noch einen Antrag machen ...).

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:

- Gib alle Zweierpotenzen kleiner n aus.
- Liegt zum Beispiel in $\Theta(\log_2(n))$.

```
i ← 1
while i < n
    Gib i aus
    i ← i · 2
```

Laufzeiten einiger gebräuchlicher Funktionen

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 Programmkonstruktion)

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

Lineares Wachstum: $O(n)$

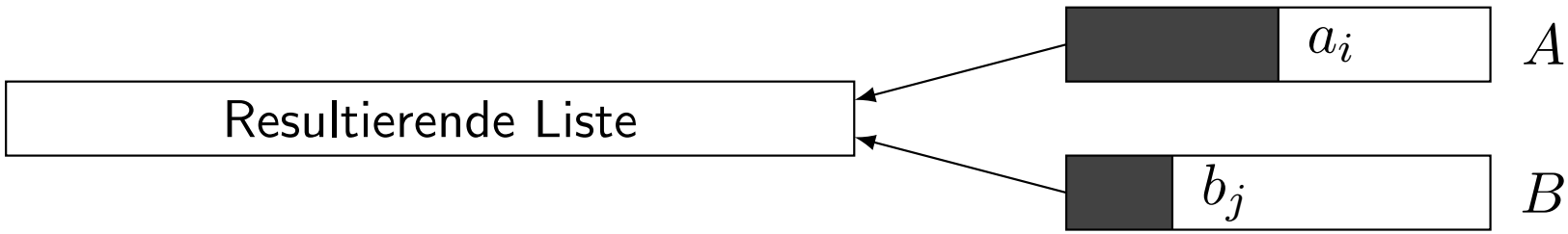
Lineares Wachstum: Laufzeit ist proportional zur Eingabegröße.

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 sortierte Liste.




```
i ← 1, j ← 1
while beide Listen sind nicht leer
    if a_i ≤ 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 nicht leeren 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. Bubble Sort (siehe VU Programmkonstruktion) 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

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.

```
foreach i ∈ {1, 2, ..., n}
    foreach j ∈ {1, 2, ..., 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

Stabile Menge (*independent set*) der Größe k: Existieren in einem Graphen mit n Knoten k Knoten, bei denen zwischen jeweils zwei Knoten keine Kante existiert?

- k *ist eine Konstante*

$O(n^k)$ Lösung: Bestimme alle Teilmengen von jeweils k Knoten.

```
foreach Teilmenge S von jeweils k Knoten
    Überprüfe ob S eine stabile Menge ist
    if S ist eine stabile Menge
        Gib aus, dass S eine stabile Menge ist
```

- Überprüfen ob S eine stabile Menge ist benötigt $O(k^2)$ Schritte.
- Anzahl der Teilmengen mit k Elementen
  $\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!} = O(n^k)$
  (da k eine Konstante ist, ist auch $k!$ eine Konstante).
- Insgesamt ist der Aufwand also $O(k^2 n^k)$.
- Da wir den konstanten Faktor k^2 weglassen können, erhalten wir $O(n^k)$.

Exponentielles Wachstum

Stabile Menge: Wie groß ist das maximale k für eine stabile Menge in einem gegebenen Graphen?

$O(n^2 2^n)$ Lösung: Überprüfe alle Teilmengen.

```
S* ← ∅
foreach Teilmenge S von Knoten
    Überprüfe ob S eine stabile Menge ist
    if S ist die größte bisher überprüfte stabile Menge
        Aktualisiere S* ← S
```

Laufzeiten am Beispiel von Stable Matching

Gale–Shapley-Algorithmus (Wiederholung)

```
Kennzeichne jede Person als frei
while ein Mann ist frei und kann noch einen Antrag machen
    Wähle solch einen Mann m aus
    w ist erste Frau in der Präferenzliste von m,
    der m noch keinen Antrag gemacht hat
    if w ist frei
        Kennzeichne m und w als einander zugeordnet
    elseif w bevorzugt m gegenüber ihrem aktuellen Partner m'
        Kennzeichne m und w als einander zugeordnet und m' als frei
    else
        w weist m zurück
```

Implementierung von Stable Matching

Ausgangssituation: Wir wissen, dass der Gale-Shapley-Algorithmus ein Stable Matching zwischen n Männern und n Frauen 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:

- Jeder Mann und jede Frau hat eine Präferenzliste aller Personen des anderen Geschlechts. 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:

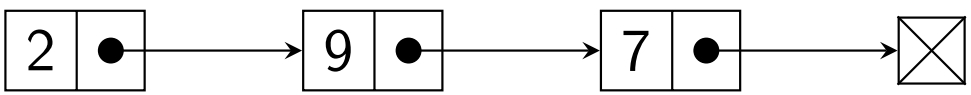
0	1	2	3	4	5	6	7
0	7	2	6	0	3	4	5

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 Programmkonstruktion.

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 Männer und n Frauen.
- Männer und Frauen werden jeweils mit einer Nummer $0, .., n-1$ assoziiert.
- Damit kann man ein Array anlegen, dessen Indexwerte sich aus diesen Nummern ergeben.

Präferenzlisten:

- Jeder Mann und jede Frau hat eine Präferenzliste in Form eines Arrays.
- `ManPref[`m`,`i`]` ist die Frau mit Index i in der Liste von Mann m (diese Frau hat bei Mann m den Rang $i+1$).
- `WomanPref[`w`,`i`]` ist der Mann mit Index i in der Liste von Frau w (dieser Mann hat bei Frau w den Rang $i+1$).

Platzbedarf: Es gibt $2n$ Personen und jede hat ein Array der Länge n, daher $O(n^2)$ (wie bei Laufzeit asymptotisch abgeschätzt).

Grundlegende Datenstruktur

Beispiel allgemein:

	1.	2.	3.
Xaver	Anna	Berta	Caroline
Yannis	Berta	Anna	Caroline
Ziggy	Anna	Berta	Caroline

Mapping für Array `ManPref`: Xaver $= 0$, Yannis $= 1$, Ziggy $= 2$, Anna $= 0$, Berta $= 1$, Caroline $= 2$

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

Schritte in der Iteration

Erster Schritt: Einen freien Mann finden.

Lösung:

- Verkettete Liste L mit allen freien Männern. Zu Beginn enthält L alle Männer.
- Das erste Element m wird ausgewählt (konstanter Aufwand).
- m wird aus der Liste gelöscht und möglicherweise ein m' (wenn ein anderer Mann m' 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 einen Mann jene Frau mit dem höchsten Rang finden, der er noch keinen Antrag gemacht hat.

Lösung:

- Dazu wird ein Array `Next` benutzt, das für jeden Mann die Position (Index in einem `ManPref`-Array) der nächsten auszuwählenden Frau angibt.
- Zunächst wird für jeden Mann m das Array mit `Next[`m`]` $= 0$ initialisiert.
- Wenn ein Mann m einen Antrag machen möchte, dann wählt er die Frau w aus mit $w =$ `ManPref[`m`, Next[`m`]]`.
- Wird der Antrag gemacht, dann wird `Next[`m`]` 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 Frau w müssen wird entscheiden, ob w schon zugewiesen wurde und wer der zugewiesene Partner ist.

Lösung:

- Wir verwenden ein Array `Current` der Länge n.
- `Current[`w`]` ist der Partner von w.
- Hat w 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.
- Dieser Schritt kann daher auch in konstanter Zeit durchgeführt werden.

Schritte in der Iteration

Vierter Schritt: Für eine Frau w und zwei Männer m und m' müssen wir entscheiden, ob m oder m' von w bevorzugt wird.

Lösung:

- Dazu wird am Anfang ein $n \times n$ Array `Ranking` erstellt.
- `Ranking`$[w, m]$ enthält den Rang von Mann m in der sortierten Reihenfolge von w.
- Für jede Frau 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`$[w, m]$ und `Ranking`$[w, m']$ verglichen werden.
- Damit ist auch dieser Schritt in konstanter Zeit ausführbar.

Erstellung des Arrays Ranking

Für jede Frau (Zeile): Erzeuge eine inverse Liste der Präferenzliste der Männer.

Beispiel: Anna (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

Anna bevorzugt Mann 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,WomanPref[i,j]] ← j
```

Schritte in der Iteration

Laufzeit: Alle vier Schritte 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 ManPref und WomanPref

```
for i ← 0 bis n − 1
    for j ← 0 bis n − 1
        Ranking[i,WomanPref[i,j]] ← j
for i ← 0 bis n − 1
    Next[i] ← 0
    Current[i] ← -1
L ← Liste aller Männer
while L ist nicht leer
    m ← erstes Element aus L, lösche erstes Element aus L
    w ← ManPref[m,Next[m]]
    m′ ← Current[w]
    if m′ = -1
        Current[w] ← m
    elseif Ranking[w,m] < Ranking[w,m′]
        Current[w] ← m
        Füge m′ in L an erster Stelle ein
    else
        Füge m in L an erster Stelle ein
    Next[m] ← Next[m] + 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.

Sortierproblem

Gegeben: Folge von Datensätzen $s_1, s_2, \ldots, s_n$ mit den Schlüsseln $k_1, k_2, \ldots, k_n$, auf denen eine Ordnungsrelation „$\leq$“ definiert ist.

Gesucht: Permutation $\pi : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ der Zahlen von 1 bis n, sodass die Umordnung der Datensätze gemäß π die Schlüssel in aufsteigende Reihenfolge bringt:
$k_{\pi(1)} \leq k_{\pi(2)} \leq \ldots \leq k_{\pi(n)}$.

Rahmenbedingungen und Annahmen

Internes Sortieren: Alle Daten sind im Hauptspeicher.

Datenstruktur: Array mit n Elementen, $A[0], \ldots, A[n-1]$

Praktische Implementierung: Trennung von Schlüssel und Informationen.

- Die Schlüssel sind ansprechbar durch $A[i].\mathit{key}$.
- Das Informationsfeld ist ansprechbar durch $A[i].\mathit{info}$ und kann sehr groß sein.
- Nach dem Sortieren gilt:

$$A[0].\mathit{key} \leq A[1].\mathit{key} \leq \ldots \leq A[n-1].\mathit{key}.$$

Dieser Foliensatz: Aus Gründen der Einfachheit werden wir zunächst nicht zwischen Schlüssel und Informationsfeld unterscheiden. Wir gehen immer von einem Array mit n Elementen (Schlüsseln) aus.

Elementare Sortierverfahren

Bubble Sort:

- Aus VU Programmkonstruktion 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 (*Selection sort*)
- Sortieren durch Einfügen (*Insertion sort*)

Selection-Sort (Sortieren durch Minimumsuche)

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

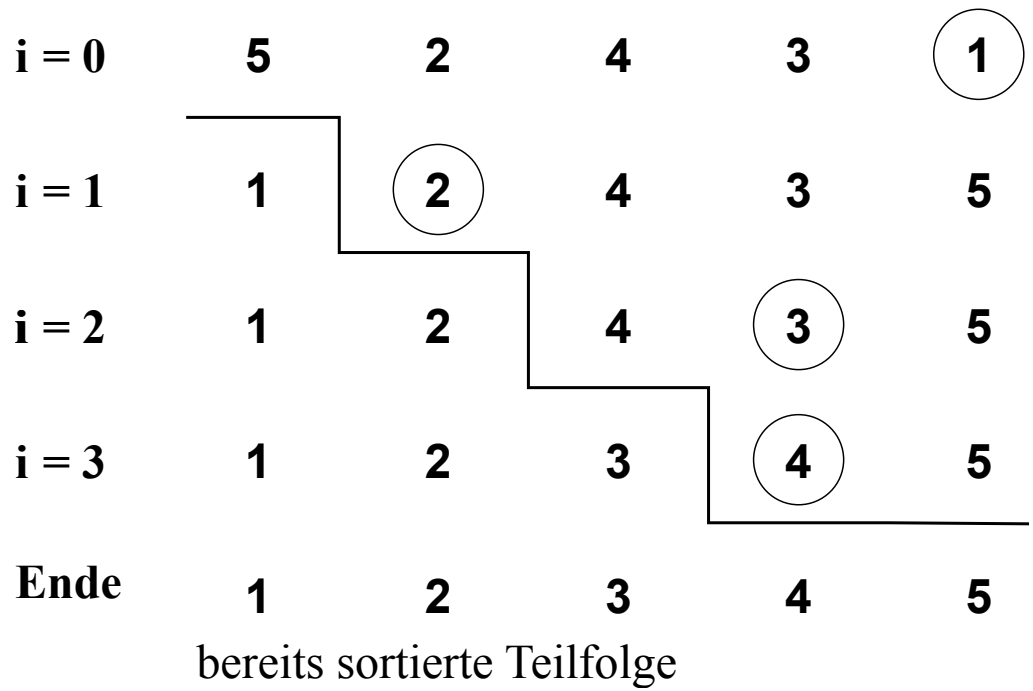
Beispiel: Implementierung mit einem Array A mit n Elementen.

```
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]
```

Selection-Sort: Beispiel

Beispiel:

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



Selection-Sort: 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. $\Theta(n)$

Insertion-Sort (Sortieren durch Einfügen)

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

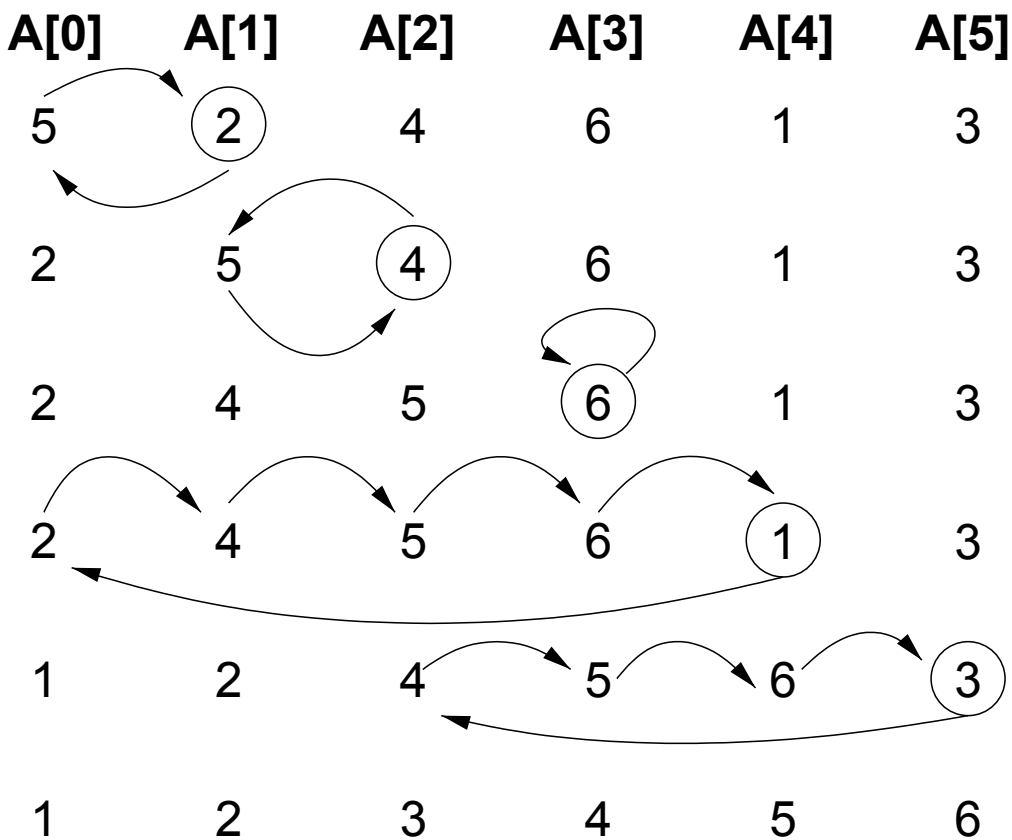
Beispiel: Implementierung mit einem Array A mit n Elementen.

```
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
```

Insertion-Sort: Beispiel

Beispiel:

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



Insertion-Sort: 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 Insertion-Sort 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 *Männern und* n *Frauen*

Polynomialzeit:

> Es existieren Konstanten $c > 0$ und $d > 0$, sodass für jede Eingabe der Größe n die Laufzeit höchstens cn^d Schritte umfasst.

Definition: Ein Algorithmus läuft in Polynomialzeit, wenn er die obige Eigenschaft erfüllt.

Erwünschte Skalierungseigenschaft: Wenn sich die Größe der Eingabe verdoppelt, dann sollte sich die Laufzeit nur um einen konstanten Faktor C erhöhen (d.h. $C = 2^d$).

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.