

2. Analyse von Algorithmen

Einleitung

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 $\circ \in \{+, -, *, /, mod, \dots\}$
- Logische Operationen: $\wedge, \vee, \neg, \dots$
- Vergleichsoperatoren (z.B. bei Verzweigungen): `if a \circ b else ... mit`
 $\circ \in \{<, \leq, =, \neq, >, \geq\}$

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

Schlüsselwörter:

- Schlüsselwörter werden **fett** und *hervorgehoben* dargestellt (z.B. **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** $! \text{finished}$...).

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 ...).

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!

Funktionen:

- Bei ausgewählten wichtigen Algorithmen werden die Funktionsnamen an den Anfang gestellt (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 freigegeben wird (*garbage collection*).
- Es wird daher nie explizit die Freigabe von Speicher angegeben.

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

Gesamter Aufwand $T(n)$:

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

Hierbei stellen c_1, c_2, c_3, c_4, c_5 konstante Kosten für verschiedene primitive Operationen dar. Die Laufzeit $T(n)$ hängt also nicht nur von der Eingabegröße n ab, sondern auch von dem Wert der Variablen j .

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

$$\begin{aligned} T(n) &= (c_2 + c_3) \cdot n + c_4 \cdot 0 + (c_1 + c_5) \\ &= (c_2 + c_3) \cdot n + (c_1 + c_5) \\ &= a_1 n + b_1 \quad \text{für Konstanten } a_1 \text{ und } b_1 \end{aligned}$$

Im besten Fall ist der Wert von j so, dass er die Laufzeit minimiert. In diesem Beispiel wird angenommen, dass $j = 0$ den geringsten Einfluss auf die Gesamlaufzeit hat. Dadurch reduziert sich der Term $c_4 \cdot j$ auf null, und die Laufzeit ist eine lineare Funktion von n ($a_1 n + b_1$).

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

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

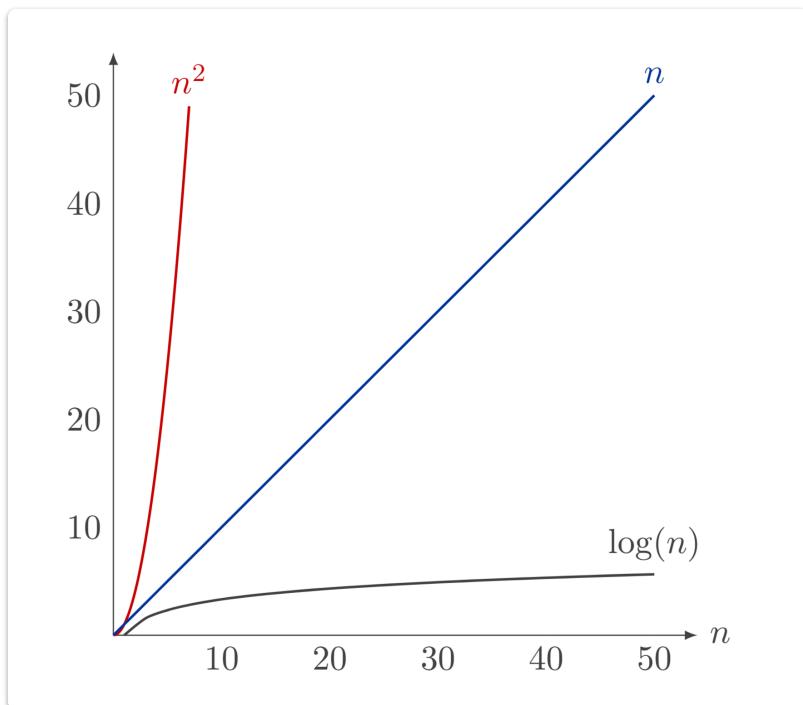
Im schlechtesten Fall ist der Wert von j so, dass er die Laufzeit maximiert. Hier wird angenommen, dass $j = n$ den größten Einfluss hat. Dadurch wird der Term $c_4 \cdot j$ zu $c_4 \cdot n$, und die Laufzeit bleibt eine lineare Funktion von n ($a_2 n + b_2$), wobei die Konstante a_2 möglicherweise größer ist als a_1 .

Zusammenfassend lässt sich sagen: Dieses Beispiel verdeutlicht, dass die genaue Laufzeit eines Algorithmus von der Eingabegröße (n) und möglicherweise von anderen Faktoren (hier j) abhängen kann. Die Best-Case- und Worst-Case-Analysen betrachten die extremen Werte dieser Faktoren, um die minimal und maximal mögliche Laufzeit in Abhängigkeit von der Eingabegröße zu bestimmen. In beiden betrachteten Fällen ist die Laufzeit linear in Bezug auf n , was bedeutet, dass die Laufzeit proportional zur Eingabegröße wächst.

Asymptotisches Wachstum

- 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.
 - Fokus auf Laufzeit $T(n)$ bei **großer** Eingabegröße n .
 - Unterschiede bei **kleinen** n oft irrelevant.
 - Ziel: Vergleich von Algorithmen **unabhängig** von Hardware.
 - Betrachtung des Wachstums von $T(n)$ für $n \rightarrow \infty$.
 - Vernachlässigung **konstanter Faktoren**.
 - Betrachtung des **dominannten Terms** in $T(n)$.
 - Eingabegröße n ist eine **nicht-negative ganze Zahl**.
-

Asymptotisches Wachstum bekannter Funktionen



Grundidee: Schranken

■ 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 :

für $n > n_0$ gilt: $c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot f(n)$

■ Schreibweise:

$$T(n) \in \Omega(g(n)), \quad T(n) \in O(f(n))$$

Obere Schranke (O)

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
- $T(n) = O(f(n))$ (**Achtung:** Hier nicht als Gleichheit zu verstehen!)

Intuitive Erklärung: Die obere Schranke $O(f(n))$ gibt uns eine Vorstellung davon, wie schnell die Laufzeit eines Algorithmus im schlimmsten Fall wächst. Sie sagt aus, dass die Laufzeit für genügend große Eingaben niemals schneller wächst als ein konstantes Vielfaches von $f(n)$.

Beispiel für 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

Untere Schranke (Ω)

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:

$$T(n) \geq c \cdot f(n)$$

Intuitive Erklärung: Die untere Schranke $\Omega(f(n))$ gibt uns eine Vorstellung davon, wie langsam die Laufzeit eines Algorithmus im besten Fall mindestens wächst. Sie sagt aus, dass die Laufzeit für genügend große Eingaben niemals langsamer wächst als ein konstantes Vielfaches von $f(n)$.

Beispiel für 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

Scharfe Schranke (Θ)

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.

Intuitive Erklärung: Die scharfe Schranke $\Theta(f(n))$ beschreibt das genaue asymptotische Wachstumsverhalten der Laufzeit eines Algorithmus. Sie bedeutet, dass die Laufzeit für genügend große Eingaben weder wesentlich schneller noch wesentlich langsamer wächst als ein konstantes Vielfaches von $f(n)$. Die Laufzeit ist in diesem Fall "eingeklemmt" zwischen zwei konstanten Vielfachen von $f(n)$.

Beispiel für 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

Keine Schranke

Hier geht es darum wie man zeigt, dass etwas nicht in einer gewissen Schranke ist, das macht man mit einem Beweis durch Widerspruch:

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:

$$\begin{aligned} 32n^2 + 17n + 5 &\leq c \\ 32n^2 &\leq c - 17 \\ n^2 &\leq \frac{c-17}{32} \\ n &\leq \sqrt{\frac{c-17}{32}} \end{aligned}$$

- 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.

Nicht nur Laufzeit: Wir verwenden die asymptotische Notation nicht nur für die Laufzeit von Algorithmen, sondern auch für die Abschätzung verschiedener anderer Größen, z.B.:

- Speicherbedarf
- Rückgabewert von Funktionen
- Anzahl gewisser Operationen (wie Vertauschung von Elementen beim Sortieren)

Untere und obere Schranken

- **Definitionen:**

- Falls $f = O(g)$, dann gilt $g = \Omega(f)$.
- $f = \Omega(g)$ genau dann, wenn $g = O(f)$.
- $f = \Theta(g)$ genau dann, wenn $f = \Omega(g)$ und $f = O(g)$ gilt. Äquivalent dazu: $f = \Theta(g)$ genau dann, wenn $g = \Theta(f)$.

- **Beweis (Auszug für $f = O(g) \implies g = \Omega(f)$):**

- Annahme: $f = O(g)$.
- Daraus folgt: Es existieren Konstanten $c > 0$ und $n_0 > 0$, sodass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot g(n)$.
- Wir wählen als neue Konstante $c' = 1/c$. Da $c > 0$, ist auch $c' > 0$.
- Für alle $n \geq n_0$ gilt somit: $g(n) \geq \frac{1}{c} \cdot f(n) = c' \cdot f(n)$.
- Also gilt $g = \Omega(f)$.

Eigenschaft: Additivität

- **Obere Schranke:** 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 (Obere Schranke):**
 - Da $f = O(h)$, existieren Konstanten $c > 0$ und $n_0 > 0$, sodass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot h(n)$.
 - Da $g = O(h)$, existieren Konstanten $c' > 0$ und $n'_0 > 0$, sodass für alle $n \geq n'_0$ gilt: $g(n) \leq c' \cdot h(n)$.
 - Wir wählen $n''_0 = \max(n_0, n'_0)$. Für alle $n \geq n''_0$ gelten beide Ungleichungen.
 - Daraus ergibt sich: $f(n) + g(n) \leq c \cdot h(n) + c' \cdot h(n) = (c + c') \cdot h(n)$.
 - Somit gilt $f(n) + g(n) \leq c'' \cdot h(n)$ für alle $n \geq n''_0$, wobei $c'' = c + c'$ eine positive Konstante ist.
 - Also ist $f + g = O(h)$.

Beispiel

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.

Untere und scharfe Schranke

- **Untere Schranken:** Mit ähnlichen Argumenten wie für die obere Schranke 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 $f + g = \Theta(h)$.

Anwendung

- **Eine Anwendung:** Wenn $g = O(f)$, dann gilt $f + g = \Theta(f)$.
- **Beweis:**
 - Da laut Annahme $g = O(f)$ und klarerweise $f = O(f)$ (mit $c = 1$ und beliebigem $n_0 \geq 0$), folgt mittels der Additivitätseigenschaft für obere Schranken: $f + g = O(f)$.
 - Es gilt aber auch $f + g = \Omega(f)$, da für alle $n \geq 0$ gilt: $f(n) + g(n) \geq f(n)$ (da Funktionswerte asymptotisch nicht negativ sind). Wir können also $c = 1$ und $n_0 = 0$ wählen.
 - Daraus folgt $f + g = \Theta(f)$, da $f + g = O(f)$ und $f + g = \Omega(f)$.

Summenbeispiel nochmal:

Hier betrachten wir das Beispiel vom Anfang ([Summenbeispiel](#)) nochmal mit neu erworbenen Wissen an:

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 $T(n) = a_2n + b_2$ 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 Gesamlaufzeit $\Theta(n)$.

Transitivität

- **Obere Schranken:** Wenn $f = O(g)$ und $g = O(h)$, dann gilt $f = O(h)$.
- **Beweis:**
 - Da $f = O(g)$, existieren Konstanten $c > 0$ und $n_0 > 0$, sodass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot g(n)$.
 - Da $g = O(h)$, existieren Konstanten $c' > 0$ und $n'_0 > 0$, sodass für alle $n \geq n'_0$ gilt: $g(n) \leq c' \cdot h(n)$.
 - Wir wählen $n''_0 = \max(n_0, n'_0)$ und $c'' = c \cdot c'$. Beide sind positive Konstanten.
 - Damit ergibt sich für alle $n \geq n''_0$:
$$f(n) \leq c \cdot g(n) \leq c \cdot (c' \cdot h(n)) = (c \cdot c') \cdot h(n) = c'' \cdot h(n).$$
 - Also gilt $f(n) \leq c'' \cdot h(n)$ für alle $n \geq n''_0$, was bedeutet, dass $f = O(h)$.
- **Beispiel:**
 - $f(n) = n$, $g(n) = n^2$, $h(n) = n^3$
 - Es gilt $f(n) = O(g(n))$ (da $n \leq 1 \cdot n^2$ für $n \geq 1$) und $g(n) = O(h(n))$ (da $n^2 \leq 1 \cdot n^3$ für $n \geq 1$).
 - Folglich gilt auch $f(n) = O(h(n))$ (da $n \leq 1 \cdot n^3$ für $n \geq 1$).
- **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): Es existieren $c_1 = 1 > 0$ und $c_2 = 1 > 0$ sowie ein $n_0 \geq 0$ (z.B. $n_0 = 0$), sodass für alle $n \geq n_0$ gilt: $c_1 \cdot f(n) \leq f(n) \leq c_2 \cdot f(n)$.
 - Wenn $f = \Theta(g)$ und $g = \Theta(h)$, dann gilt $f = \Theta(h)$ (Transitivität).

Da die Relation Θ reflexiv, symmetrisch und transitiv ist, handelt es sich um eine Äquivalenzrelation.

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 asymptotischen Wachstums.
- Wir schreiben dann $f \ll g$.

Alternative Sichtweise:

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

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

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

Beispiel: Keine Dominanz

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

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

$$\lim_{n \rightarrow \infty} n^b/n^a = \lim_{n \rightarrow \infty} n^{b-a} = 0.$$

Asymptotische Schranken von gebräuchlichen Funktionen

Polynome:

Polynome: $f(n) = a_0 + a_1n + \dots + a_dn^d$ 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.

Logarithmen:

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*

Exponentiell:

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

Verhältnisse:

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

$$1 \ll \log n \ll \sqrt[k]{n} \ll n \ll n \log n \ll n^{1+\epsilon} \ll n^2 \ll n^3 \ll \dots \ll c^n \ll n! \ll n^n$$

- **Hinweise:**

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

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

```
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
```

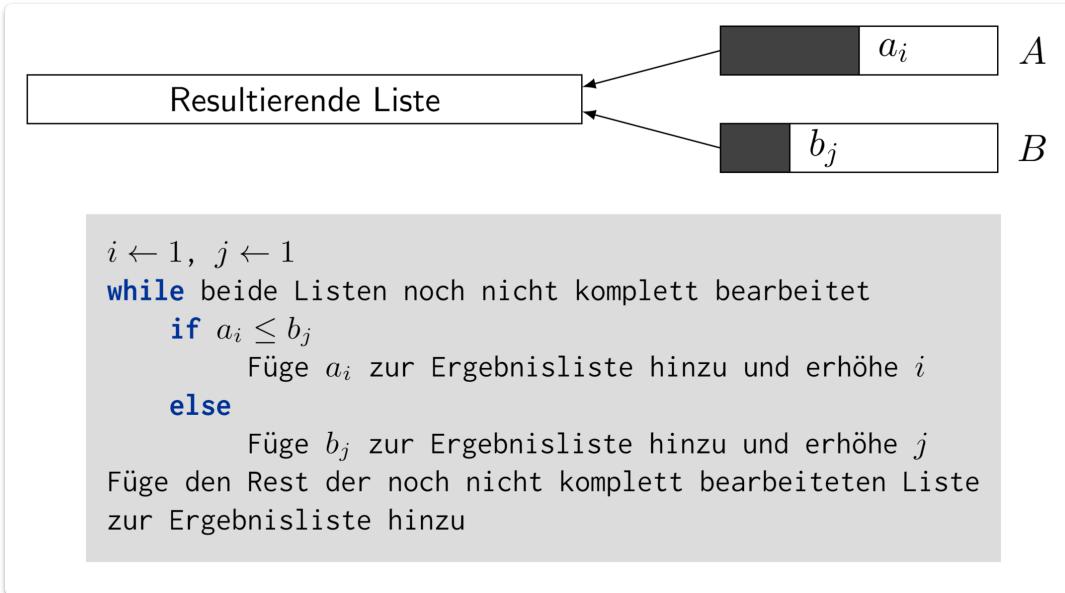
Lineares Wachstum: $O(n)$

- **Lineares Wachstum:** Laufzeit ist proportional zur Eingabegröße.
- **Maximumsuche:** Ermittle das Maximum von n Zahlen a_1, a_2, \dots, a_n .

```
 $max \leftarrow a_1$ 
for  $i \leftarrow 2$  bis  $n$ 
    if  $a_i > max$ 
         $max \leftarrow a_i$ 
```

Die Maximumsuche hat eine lineare Zeitkomplexität, da jedes Element des Arrays einmal betrachtet wird. Die Anzahl der Operationen wächst direkt mit der Anzahl der Elemente n .

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



Der Merge-Prozess hat eine **lineare Zeitkomplexität**, da im schlimmsten Fall jedes Element der beiden Eingabelisten einmal in die Ergebnisliste eingefügt wird. Die Anzahl der Operationen ist proportional zur Summe der Längen der beiden Listen, also $O(n + m)$. Wenn beide Listen die Größe n haben, ist die Komplexität $O(2n) = O(n)$.

$O(n \log n)$ 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 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.
- **Beispiel - Größtes leeres Intervall:** Es seien n Zeitpunkte x_1, \dots, 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.
 - **Sortieren:** Das Sortieren der n Zeitpunkte benötigt typischerweise $O(n \log n)$ Zeit (z.B. mit Mergesort).
 - **Durchlaufen und Differenz bilden:** Das anschließende Durchlaufen der sortierten Liste und das Berechnen der Differenzen zwischen aufeinanderfolgenden Zeitpunkten benötigt $O(n)$ Zeit.
 - **Gesamlaufzeit:** Die Gesamlaufzeit ist somit $O(n \log n) + O(n) = O(n \log n)$.

Quadratisches Wachstum: $O(n^2)$

- **Quadratisches Wachstum:** Betrachte alle Paare von Elementen.
- **Dichtestes Punktpaar:** Gegeben sei eine Liste von n Punkten in einer Ebene $(x_1, y_1), \dots, (x_n, y_n)$ 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

- **Ablauf des Algorithmus:**

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

- **Summe:** $(n - 1) + (n - 2) + \dots + 1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \Theta(n^2)$.
- **Anmerkung:** $\Omega(n^2)$ erscheint unvermeidbar, aber es gibt für dieses Problem einen Algorithmus, der effizienter als der offensichtliche ist (z.B. ein $O(n \log n)$ Algorithmus basierend auf Divide and Conquer).

Kubisches Wachstum: $O(n^3)$

- **Kubisches Wachstum:** Zähle alle Dreiergruppen von Elementen.
- **Disjunkte Mengen:** Gegeben seien n Mengen S_1, \dots, S_n , wobei jede Menge eine Teilmenge von $\{1, 2, \dots, n\}$ 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

```

Die Komplexität ergibt sich, da wir $O(n^2)$ Paare von Mengen betrachten und für jedes Paar im schlimmsten Fall $O(n)$ Elemente vergleichen müssen (wenn die Mengen bis zu Größe n

haben können).

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

Die vier verschachtelten Schleifen iterieren über alle möglichen Kombinationen von vier Indizes i, j, k, l . Im schlimmsten Fall werden alle $O(n^4)$ Kombinationen überprüft.

- **Hinweis:** Dieses Problem kann man in der Regel effizienter lösen (oftmals mit dynamischer Programmierung oder anderen Techniken).

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 im schlimmsten Fall untersucht werden. Daher die exponentielle Laufzeit von $O(2^n)$.

Laufzeiten am Beispiel von Stable Matching

Gale–Shapley–Algorithmus

Hier nochmal der Algorithmus der in [Stable Matching Problem](#) schon vorgekommen ist:

```

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

```

Wir wollen uns jetzt anschauen, wie man so etwas bestenfalls implementiert.

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

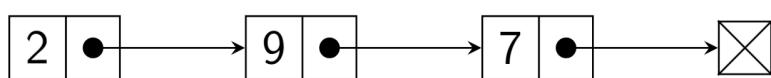
- **Array:**
 - Ist eine statische Datenstruktur mit im Speicher sequentiell abgelegten Elementen.
 - Alle Elemente haben den gleichen Typ.
 - Zugriff über Index in konstanter Zeit möglich ($O(1)$).
- **Beispiel:**

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

- **Zweidimensionale Arrays:** Zweidimensionale (oder mehrdimensionale) Arrays können als Array von Arrays realisiert werden.

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	$\Theta(n)$	$\Theta(1)$

Genaueres zu Arrays und Listen [Liste](#) und [Array](#)

Grundlegende Datenstruktur

- **Vereinfachung:**
 - Es gibt sowohl n Kinder und n Familien.
 - Kinder und Familien werden jeweils mit einer Nummer ($0, \dots, n - 1$) 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.
 - $SPräf[s, i]$ ist die Familie mit Index i in der Liste von Kind s (diese Familie hat bei Kind s den Rang $i + 1$).
 - $FPräf[f, i]$ 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).

Beispiel

- **Beispiel allgemein:**

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

- **Mapping für Array SPräf:** Xavier = 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 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.

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 $\text{Next}[s] = 0$ initialisiert. (Aufwand proportional zu n)
- Wenn ein Kind s eine Familie auswählen möchte, dann wählt es die Familie $f = \text{SPref}[s, \text{Next}[s]]$ aus.
- Wird eine Familie ausgewählt, dann wird $\text{Next}[s]$ um 1 erhöht (unabhängig vom Ergebnis).
- Alle Operationen benötigen konstante Zeit und daher kann dieser Schritt in konstanter Zeit ausgeführt werden.

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 .
- $\text{Current}[f]$ 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.

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.
- $\text{Ranking}[f, s]$ 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 $\text{Ranking}[f, s]$ und $\text{Ranking}[f, s']$ 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 Familien.

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 $\underbrace{\text{Ranking}[0, 2]}_1 < \underbrace{\text{Ranking}[0, 5]}_6$

```

for  $i \leftarrow 0$  bis  $n - 1$ 
  for  $j \leftarrow 0$  bis  $n - 1$ 
     $\text{Ranking}[i, \text{FPref}[i, j]] \leftarrow j$ 
  
```

Laufzeit und Implementierung

- **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 (des Gale-Shapley-Algorithmus, vermutlich) lassen sich in konstanter Zeit ausführen. Daher liegt die Gesamlaufzeit 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  $i \leftarrow 0$  bis  $n - 1$ 
    for  $j \leftarrow 0$  bis  $n - 1$ 
         $\text{Ranking}[i, \text{FPref}[i, j]] \leftarrow j$ 
for  $i \leftarrow 0$  bis  $n - 1$ 
     $\text{Next}[i] \leftarrow 0$ 
     $\text{Current}[i] \leftarrow -1$ 
 $\text{SFree} \leftarrow$  Liste aller Kinder
while  $\text{SFree}$  ist nicht leer
     $s \leftarrow$  erstes Element aus  $\text{SFree}$ , lösche erstes Element aus  $\text{SFree}$ 
     $f \leftarrow \text{SPref}[s, \text{Next}[s]]$ 
     $s' \leftarrow \text{Current}[f]$ 
    if  $s' = -1$ 
         $\text{Current}[f] \leftarrow s$ 
    elseif  $\text{Ranking}[f, s] < \text{Ranking}[f, s']$ 
         $\text{Current}[f] \leftarrow s$ 
        Füge  $s'$  in  $\text{SFree}$  an erster Stelle ein
    else
        Füge  $s$  in  $\text{SFree}$  an erster Stelle ein
     $\text{Next}[s] \leftarrow \text{Next}[s] + 1$ 

```

Sortieren als praktisches Beispiel

Sortieren: Gegeben sind n Elemente die aufsteigend angeordnet werden sollen.

- **Sortiere eine Liste von Namen**
- Organisiere eine MP3-Bibliothek
- Zeige Google PageRank Resultate an
- Liste RSS-Feedelemente in umgekehrt chronologischer Reihenfolge auf
- Finde den Median
- Finde das nächste Paar
- Binäre Suche

Rahmenbedingungen und Annahmen

- **Internes Sortieren:** Alle Daten sind im Hauptspeicher.
- **Datenstruktur:** Array mit n Elementen, $A[0], \dots, A[n - 1]$, 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 sollen. 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 Minimumssuche (Selectionsort)
 - Sortieren durch Einfügen (Insertionsort)

Selectionsort (Sortieren durch Minimumssuche)

- **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.

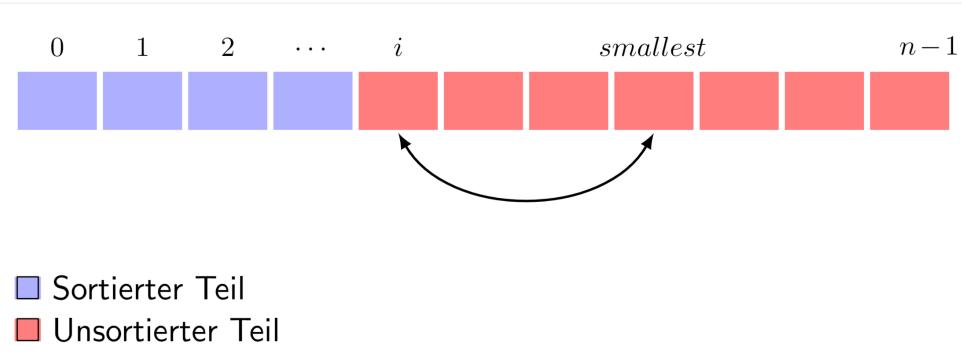
Selectionsort(A):

```

for  $i \leftarrow 0$  bis  $n - 2$ 
     $smallest \leftarrow i$ 
    for  $j \leftarrow i + 1$  bis  $n - 1$ 
        if  $A[j] < A[smallest]$ 
             $smallest \leftarrow j$ 

```

Vertausche $A[i]$ mit $A[smallest]$



Beispiel

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

$i = 0$	5	2	4	3	1
$i = 1$	1	(2)	4	3	5
$i = 2$	1	2	4	(3)	5
$i = 3$	1	2	3	(4)	5
Ende	1	2	3	4	5

Analyse

- **Laufzeit:** Die Laufzeit liegt immer (Best/Average/Worst-Case) in $\Theta(n^2)$.
- **Analyse:**
 - Die innere Schleife wird beim ersten Durchlauf der äußeren Schleife $n - 1$ -mal ausgeführt.
 - Im nächsten Durchlauf $n - 2$, dann $n - 3$ -mal usw.
 - $(n - 1) + (n - 2) + (n - 3) + \dots + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \Theta(n^2)$.

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

Insertionsort (Sortieren durch Einfügen)

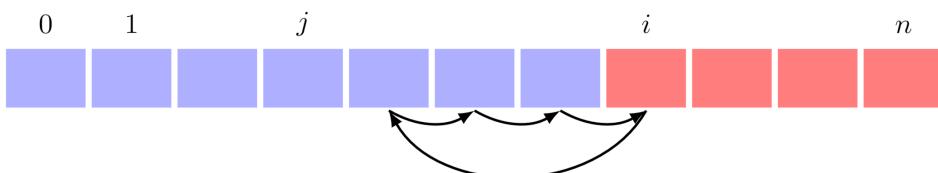
- **Insertionsort:** Insertionsort entnimmt der unsortierten Teilfolge ein Element und fügt es an der richtigen Stelle in die (anfangs leere) sortierte Teilfolge ein.
- **Beispiel:** Implementierung mit einem Array A mit n Elementen.

Insertionsort(A):

```

for  $i \leftarrow 1$  bis  $n - 1$ 
     $key \leftarrow A[i]$ ,  $j \leftarrow i - 1$ 
    while  $j \geq 0$  und  $A[j] > key$ 
         $A[j + 1] \leftarrow A[j]$ 
         $j \leftarrow j - 1$ 
     $A[j + 1] \leftarrow key$ 

```



- Sortierter Teil
- Unsortierter Teil

Beispiel

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

5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	
1	2	3	4	5	6

Analyse

- **Laufzeit:**

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

$$1 + 2 + 3 + \dots + (n-2) + (n-1) = \frac{(n-1)n}{2} = \Theta(n^2)$$

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, ist die Anzahl der Vertauschungen im Worst- und Average-Case ebenfalls $\Theta(n^2)$. Im Best-Case (sortiertes Array) sind es 0 Vertauschungen, also $\Theta(1)$.

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

- **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	4 s
$n =$	30	< 1 s	< 1 s	< 1 s	< 1 s	18 min	10^{25} Jahre
$n =$	50	< 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	sehr lange
$n =$	1,000	< 1 s	< 1 s	1 s	18 min	sehr lange	sehr lange
$n =$	10,000	< 1 s	< 1 s	2 min	12 Tage	sehr lange	sehr lange
$n =$	100,000	< 1 s	2 s	3 Stunden	32 Jahre	sehr lange	sehr lange
$n =$	1,000,000	1 s	20 s	12 Tage	31,710 Jahre	sehr lange	sehr lange

Die Tabelle verdeutlicht den enormen Unterschied zwischen polynomiellen und exponentiellen Laufzeiten. Während Algorithmen mit polynomieller Laufzeit für relativ große Eingaben oft noch praktikabel sind, werden Algorithmen mit exponentieller Laufzeit schnell unbrauchbar, da die Laufzeit mit der Eingabegröße extrem stark ansteigt.

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 und 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.
- **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.

Mehr dazu: [9. Polynominalzeitreduktionen](#)