Grundlagen der Programmierung

Ressourcen:

Zeit und Platz ♦ Asymptotik ♦ Brute Force

Algorithmisches Denken: Vom Problem zur Lösung

1. Identifizieren des Problems
2. Formulieren des Problems
3. Entwurf des Algorithmus
4. Implementierung des Algorithmus
5. Anwendung des Algorithmus

→ Problemlösung

Entwurf des Algorithmus

- **Wie** werden die Eingabedaten in die Ausgabedaten überführt? (*Folge von Anweisungen*)

- **Korrekt?** (für alle Eingabedaten)
- **Terminiert?** (für alle Eingabedaten)
- **Effizient?** (für alle Eingabedaten)

Effizienz - Motivation

- Es existieren immer mehrere Algorithmen für dasselbe Problem. → *Welchen benutzen?*
- **Effizienz** beachten → *Wie viele Ressourcen werden benötigt?*
  - **Zeit** (z.B. Antwortzeiten)
  - **Platz** (Speicherplatz)
    → *Stackframes beanspruchen Speicherplatz*
  - Energie, Anzahl der Anweisungen, ...
- Kann von der Implementierung abhängen (*z.B. Datentypen*)

Rolle der Datenrepräsentation - Beispiel

Name: Komplementärgraph

Eingabe: ungerichteter Graph G=(V,E) als Adjazenzmatrix A

Ausgabe: Komplementärgraph von G als Adjazenzmatrix

für i ← 1 **bis** |V|
    für j ← 1 **bis** |V|
        A[i,j] ← 1 – A[i,j]
gib A **aus**

Probieren Sie, diese Aufgabe zu lösen, wenn G durch die Mengen seiner Knoten und Kanten gegeben ist ...

→ *Bei der Implementierung Datentypen beachten!*

Zeitkomplexität

- Kann von verwendeten Datentypen abhängen
- Kann von Hardware, Netzwerk, aktueller Belastung etc. abhängen
  → Zeitmessung gibt wenig Aufschluss über Algorithmus!
  → *Wie also messen?!*
- Definition der Zeitkomplexität unabhängig von
  - Hardware
  - Situation während der Programmausführung
  - Besonderheiten einer bestimmten Programmiersprache

Zeitkomplexität

- Hängt von der Größe der Eingabe ab
  - Lesen der gesamten Eingabe
  - Laufzeit des Algorithmus wächst i.A. mit Eingabegröße
  - *Beispiel: Durchsuchen einer Liste* **`L`** *abhängig von* **`len(L)`**

- Zuordnung der Anzahl der auszuführenden Operationen zu jeder Eingabegröße

- Zeitkomplexität ist Funktion

  ***t*: Eingabegröße $\longrightarrow$ Anzahl von Operationen**

Eingabegröße messen

- **Liste:** Länge der Liste
  - ggf. Größe der gespeicherten Werte berücksichtigen

- **Zahl *x*:**
  - Zahl *x* oder
  - Anzahl der Bits zur Repräsentation von *x:* $\lfloor \log_2 x \rfloor + 1$

- **Graph** mit *n* Knoten und *m* Kanten: $m + n$
  - ggf. nur *n* oder nur *m*

➢ Besonderheiten des Algorithmus berücksichtigen

Operationen zählen

- „elementare“ Operationen (*Was ist elementar?*)
  - *z.B. arithmetische oder boolesche Operation, Vergleich, Zugriff auf Listenelement, Funktionsaufruf, return, ...*
- <u>Annahme</u>:
  - „elementare“ Operationen <u>unabhängig von Eingabegröße</u>
  - einheitliche, feste Zeit zur Ausführung der Operation (*auch wenn wenig realistisch*)
  - kann durch Verfeinerung realistischer/exakter werden
  - Annahme gerechtfertigt, solange einheitlich gezählt wird (*Erinnerung: Es geht um den <u>Vergleich</u> von Algorithmen*)

Auswahl der Eingabedaten

- Es gibt verschiedene Eingaben derselben Größe.
- Drei mögliche Szenarien:

1. **Best Case**
   Wähle für jede Eingabegröße eine Eingabe,
   die zu den wenigsten Schritten führt.
2. **Worst Case**
   Wähle für jede Eingabegröße eine Eingabe,
   die zu den meisten Schritten führt.
3. **Average Case**
   Durchschnitt über alle Eingaben derselben Größe

1. Beispiel: Durchschnitt

Eingabe: Liste *L* mit Zahlen

Ausgabe: arithmetisches Mittel der Listenelemente

1 ...... $k \leftarrow 0$
2 ...... $n \leftarrow |L|$
n mal ...... für $i \leftarrow 1$ bis n
  3 .... $k \leftarrow k + L[i]$
2...... $k \leftarrow k / n$
1...... gib k aus

Eingabegröße: |*L*| = *n*

t*(*n*) = 6 + 3*n (*grob*)

oder genauer ***t*(*n*) = 6 + 5*n***

Hier:
worst case = best case
= average case

2. Beispiel: Komplementärgraph

Name: Komplementärgraph

Eingabe: ungerichteter Graph $G=(V,E)$ als Adjazenzmatrix A

Ausgabe: Komplementärgraph von G als Adjazenzmatrix

n mal…… **für** i ← 1 **bis** |V|
  n mal…… **für** j ← 1 **bis** |V|
      3 ………… A[i,j] ← 1 – A[i,j]
  1…… **gib** A **aus**

Eingabegröße: $|V|+|E| = n+m$

$t(n,m) = 1 + n \cdot n \cdot 3 = 1 + 3n^2$
(*grob*)

Auch hier:
worst case = best case
= average case

3. Beispiel: Abstand von Knoten

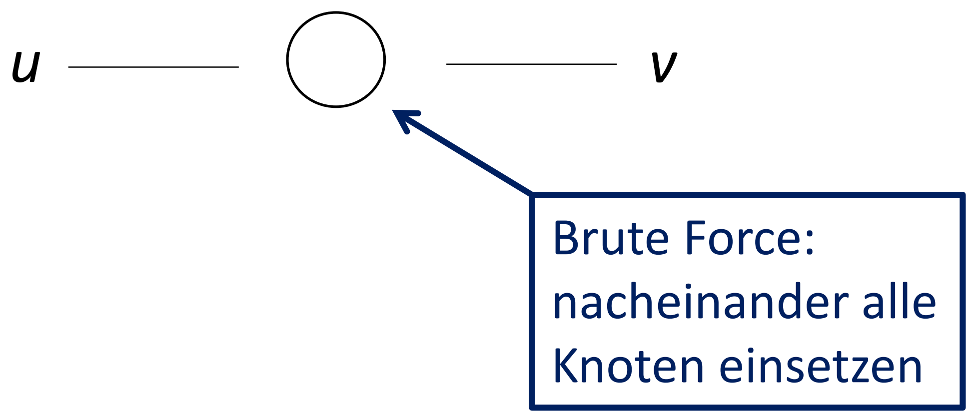
Name: Abstand von Knoten (Brute Force)
Eingabe: ungerichteter, schlingenfreier Graph $G = (V,E)$,
$u, v \in V, u \neq v$
Ausgabe: $D(u,v)$

für $k \longleftarrow 1$ bis $|V| - 1$
  falls Pfad der Länge k von u nach v existiert
    gib k aus

→ *<u>Verfeinerung</u>: Algorithmus, der für zwei Knoten u und v und eine positive ganze Zahl k feststellt, ob ein Pfad der Länge k von u nach v existiert*

Erinnerung: Algorithmische Idee (BF)

1. Idee für $k = 1$: $\{u,v\} \in E$?
2. Idee für $k = 2$:



3. Idee für beliebiges k:
Probieren aller <u>Teilmengen</u> von V der Größe k – 1
und aller <u>Anordnungen</u> von deren Elementen

Abstand: Brute Force Algorithmus

```
k ⟵ 1
solange k < |V|
|    u_0 ⟵ u
|    u_k ⟵ v
|    für jede Teilmenge V' ⊆ V mit k – 1 Elementen
|    |    für jede Permutation u_1,u_2, ..., u_{k-1} ihrer Elemente
|    |    |    istPfad ⟵ 1
|    |    |    für j ⟵ 0 bis k – 1
|    |    |    |    falls {u_j,u_{j+1}} ∉ E
|    |    |    |    |    istPfad ⟵ 0
|    |    |    falls istPfad = 1
|    |    |    |    gib k aus
|    |    |    |    STOP
|    k ⟵ k + 1
gib ∞ aus
```

Vorüberlegungen Brute Force Effizienz

Name: Abstand von Knoten (Brute Force)
Eingabe: ungerichteter, schlingenfreier Graph $G = (V,E)$,
$u, v \in V, u \neq v$
Ausgabe: $D(u,v)$

Eingabegröße: n Knoten und m Kanten

Repräsentation: als Adjazenzmatrix

Worst Case: wenn u und v nicht verbunden ($D(u,v) = \infty$)

Abstand: Brute Force Algorithmus

$$t(n,m) = 2 + 4(n\text{-}1) + \sum_{k=1}^{n-1}\left[\binom{n}{k-1}(k\text{-}1)!\,(2 + 2k)\right]$$

```
      1 ...... k ⟵ 1
n-1 mal ...... solangе k < |V|
           1 ........... u_0 ⟵ u
           1 ........... u_k ⟵ v
 (n über k−1) mal ...... für jede Teilmenge V' ⊆ V mit k – 1 Elementen
              (k-1)! mal ...... für jede Permutation u_1,u_2, ..., u_{k-1} ihrer Elemente
                            1 ................ istPfad ⟵ 1
                        k mal ................ für j ⟵ 0 bis k – 1
                                  1 .................... falls {u_j,u_{j+1}} ∉ E
                                        1 ...................... istPfad ⟵ 0
                            1 ................ falls istPfad = 1
                                  1 .................... gib k aus
                                                         STOP
           2 ........... k ⟵ k + 1
      1 ...... gib ∞ aus
```

Zeitkomplexität (*worst case*) im Vergleich

n	$6 + 3n$	$6 + 5n$	$1 + 3n^2$	BF Alg.
2	12	16	13	10
5	21	31	76	812
10	36	56	301	> 50.000.000
100	306	506	30.001	$> 10^{160}$

- Entscheidend ist, wie schnell die Anzahl der Operation bei wachsender Eingabegröße anwächst.
- Vergleich mit Standardfunktionen (*linear*, *quadratisch*, ...)

Wachstum des BF Algorithmus

- Algorithmus untersucht für jedes k von 1 bis n jede Teilmenge $V' \subseteq V$ mit $k - 1$ Elementen

  → untersucht 2^{n-1} Teilmengen

  → **untere Schranke**: 2^{n-1} Operationen (**exponentielle Fkt.**)

  $2^{99} > 10^{29}$ $\qquad$ $2^{999} > 10^{300}$

- *typisches Vorgehen:*
  Suchen Standardfunktion, die das Wachstum von $t(n)$ von unten begrenzt.
  → *Aussage über Praktikabilität des Algorithmus*

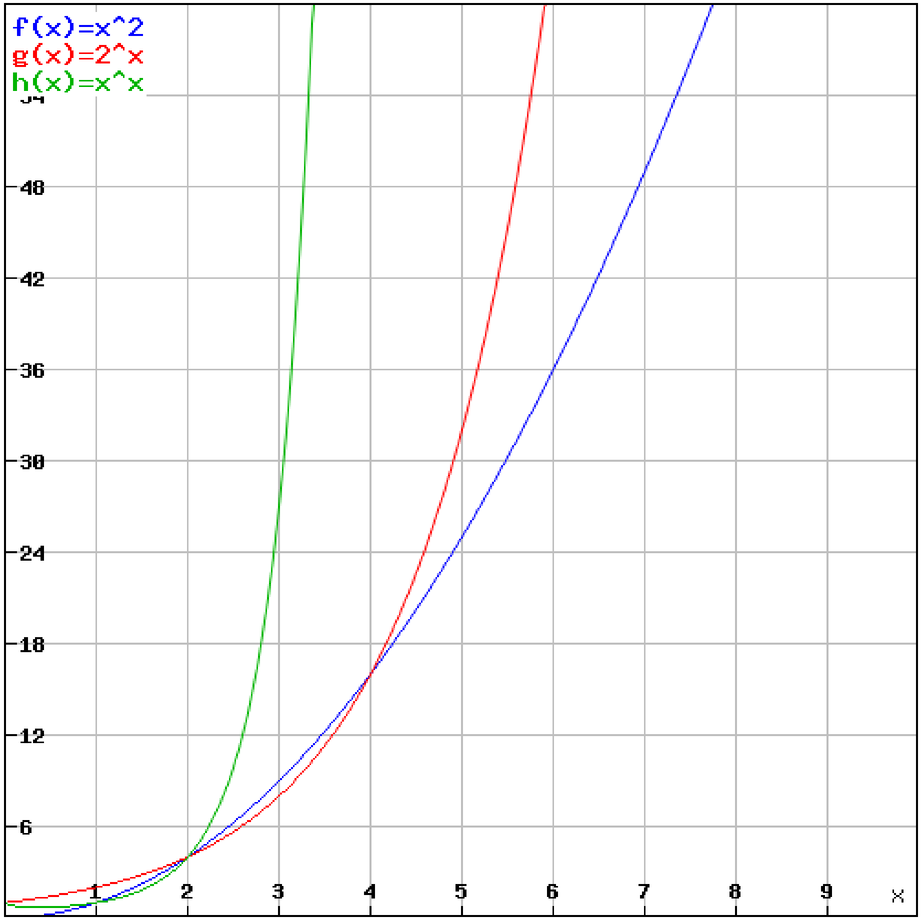
Rechenzeiten

Annahme: Prozessor mit 100 GFLOPS

n	5	10	50	100
$t(n) = n^2$	0,00000025 s	0,000001 s	0,000025 s	0,0001 s
$t(n) = n^5$	0,00003125 s	0,001 s	3,125 s	ca. 2 min
$t(n) = 2^n$	0,00000032 s	0,00001024 s	ca. 130 Tage	ca. 10^{15} Jahre
$t(n) = n^n$	0,00003125 s	ca. 2 min	> 10^{69} Jahre	

- Man kann von kleinen Eingabewerten absehen.
- Man kann von konstanten Faktoren absehen.
  (wichtig ist nur die Qualität *linear, quadratisch, kubisch, ..., exponentiell, ...*)

Wachstum von Funktionen



Konstante Faktoren „verschieben“ nur den Punkt, ab dem die schneller wachsende Funktion größere Werte besitzt.

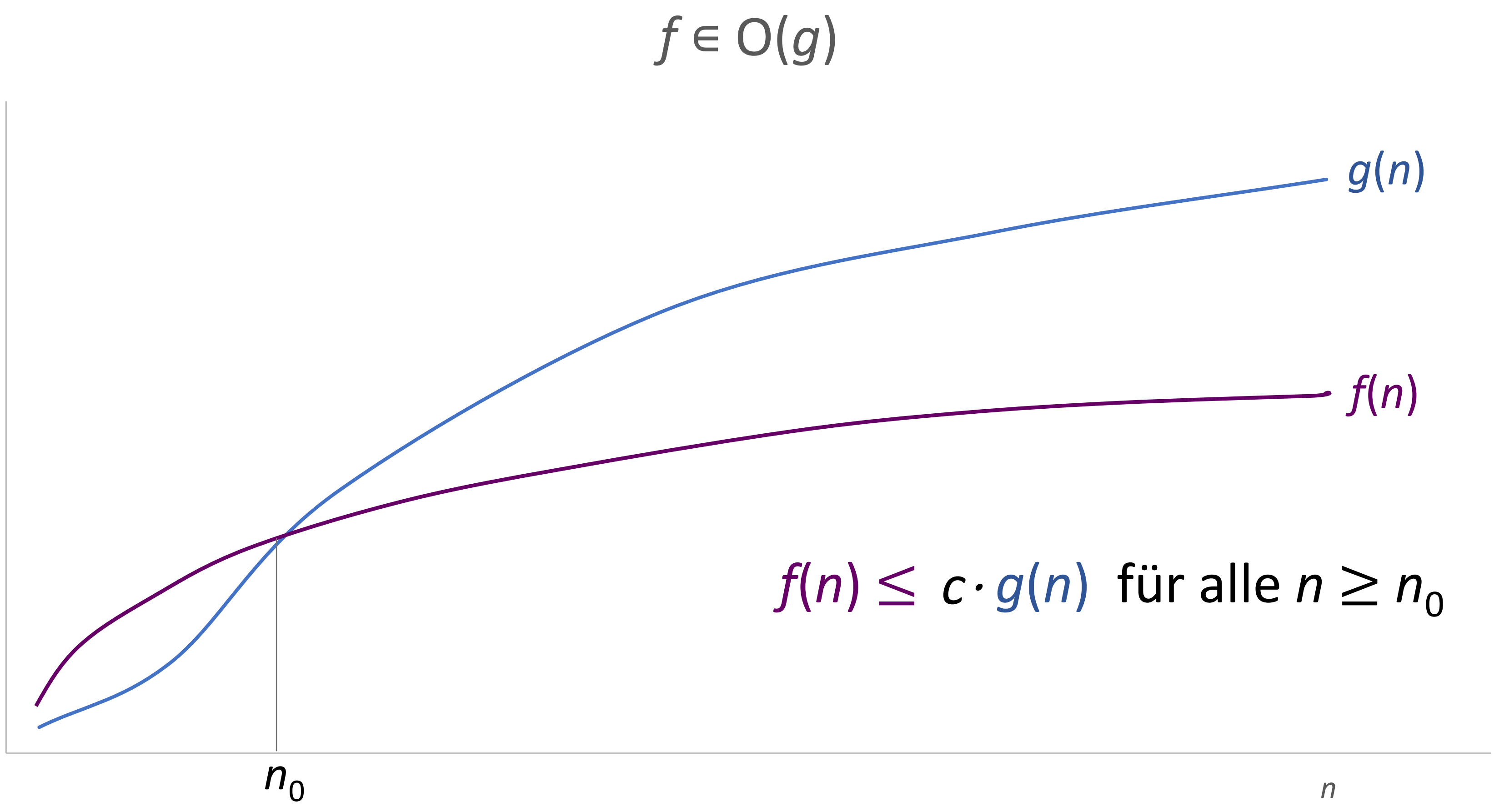
Beispiel:

$f(n) = 1000\, n$

$g(n) = n^2$

$\rightarrow$ ab $n \geq 1000$ gilt $g(n) \geq f(n)$

Vergleich von Funktionen



Vergleich von Funktionen

- ***f*** wächst *asymptotisch* höchstens so schnell wie ***g***, wenn es zwei Konstanten $n_0 \geq 0$ und $c > 0$ gibt, so dass $\boldsymbol{f(n) \leq c \cdot g(n)}$ für alle $\boldsymbol{n \geq n_0}$ gilt. $\boldsymbol{f(n) \in O(g(n))}$
  - **Für den Vergleich zweier Algorithmen:**
    Algorithmus mit $t(n) = f(n)$ ist (asymptotisch) mindestens *so effizient* wie Algorithmus mit $t(n) = g(n)$
  - **Für Abschätzungen durch Standardfunktionen:**
    $1000\,n \in O(n^2)$
    $\boldsymbol{f(n) \in O(g(n))}$ → ***g*** ist *obere Schranke* von ***f***

Vergleich von Funktionen

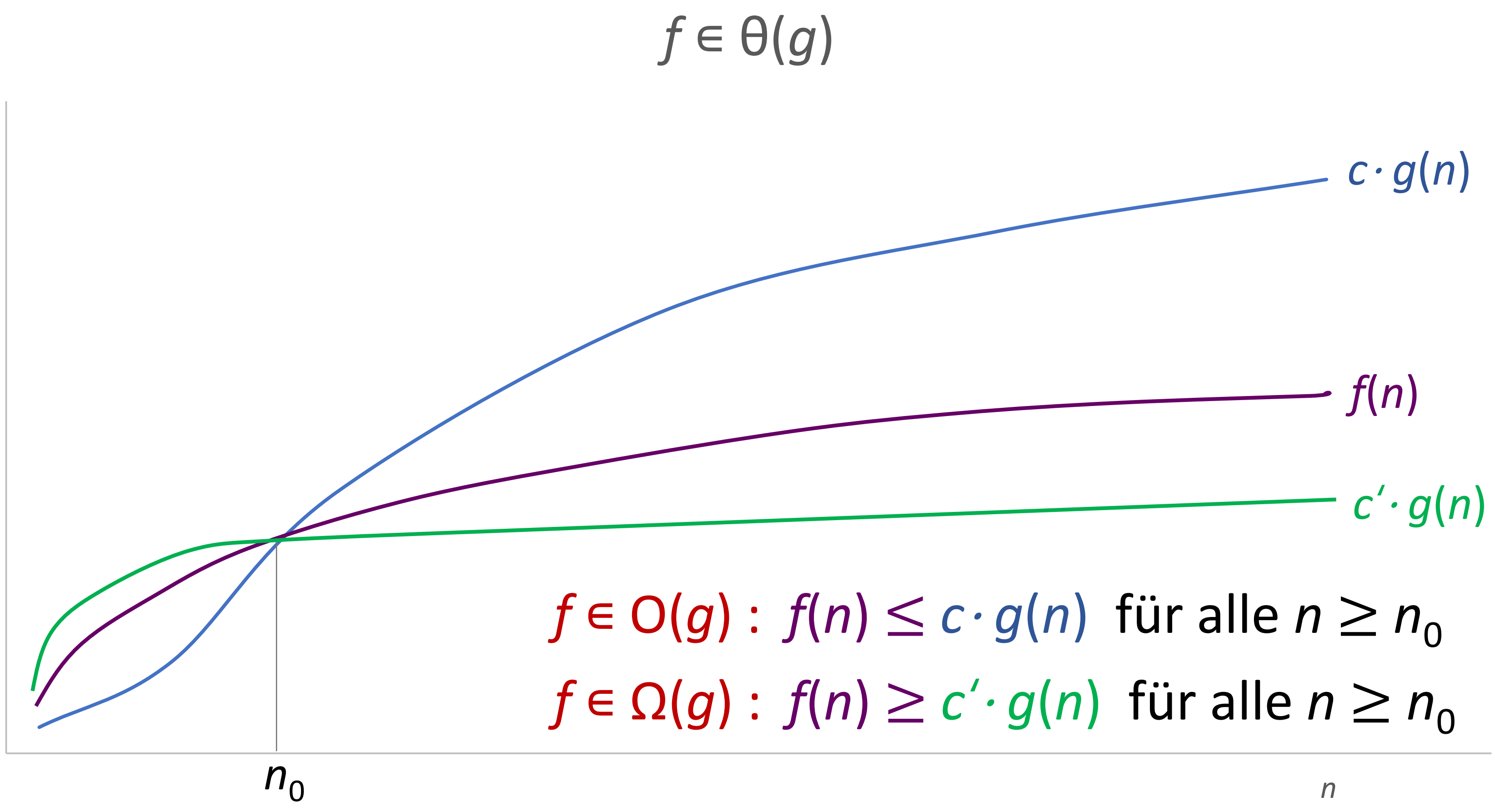
- ***f*** wächst *asymptotisch* mindestens so schnell wie ***g***, wenn es zwei Konstanten $n_0 \geq 0$ und $c > 0$ gibt, so dass $\boldsymbol{f(n) \geq c \cdot g(n)}$ für alle $\boldsymbol{n \geq n_0}$ gilt. $\boldsymbol{f(n) \in \Omega\,(g(n))}$

- ***g*** wächst *asymptotisch* genau so schnell wie ***f***, wenn $\boldsymbol{f(n) \in O(g(n))}$ und $\boldsymbol{f(n) \in \Omega\,(g(n))}$**.** $\boldsymbol{f(n) \in \theta\,(g(n))}$

- $\boldsymbol{f \in O(g)}$ **–** ***g*** **obere Schranke von** ***f***
  $\boldsymbol{f \in \Omega(g)}$ **–** ***g*** **untere Schranke von** ***f***
  $\boldsymbol{f \in \theta(g)}$ **–** ***g*** **wächst (asymptotisch) wie** ***f***

Beispiel: BF Algorithmus: $t(n) \in \Omega(2^n)$ (→ *ineffizient*)

Vergleich von Funktionen



Bestimmen von Vergleichsrelationen

- Suchen Schranken, die möglichst „eng“ sind
- Nachweis von Relationen:

$$f(n) = 800 + 23n + 3n^2$$
$$f(n) \in O(n^2)$$

$$800 + 23n + 3n^2 \leq 800n^2 + 23n^2 + 3n^2 = 826n^2$$

$$\boxed{n \geq 1}$$

Mit $n_0 = 1$, $c = 826$, $g(n) = n^2$ gilt also $f(n) \in O(g(n))$.

Bestimmen von Vergleichsrelationen

$f(n) \; = \; 800 + 23n + 3n^2$

$f(n) \in \Omega\,(n^2)$

$800 + 23n + 3n^2 \geq \; n^2$

Mit $n_0 = 0$, $c = 1$, $g(n) = n^2$ gilt also $f(n) \in \Omega(g(n))$.

Somit gilt $f(n) \in \theta(g(n))$.