2 Algorithmen

Begriffe

- Effizienz
  = gute Ausnutzung von Resourcen
- Resourcen
  = Rechenzeit, Speicherplatz
- Aufwand / Komplexität
  = tatsächlicher Verbrauch von Resourcen

Performanzfaktoren

- Prozessorleistung
- Hauptspeicher
- Caches
- Compiler-Version
- Betriebssystem
- ...

Beispiel

- Matrix mit 2048×2048 Einträgen
- Entry(i,j) = A[i×2048+j]
- Löschen 1:

```
for i←0 to 2047 do
    for j←0 to 2047 do
        Entry(i,j) ← 0
```

- Löschen 2:

```
for j←0 to 2047 do
    for i←0 to 2047 do
        Entry(i,j) ← 0
```

Effizienz eines Algorithmus

- Implement / Run / Test
  - Abhängig vom speziellen System
  - Abhängig von der aktuellen Auslastung
  - Abhängig von den Testdaten
  - Abhängig von der Begabung des Programmierers
  - Gibt es noch bessere Algorithmen?

Effizienz eines Algorithmus

- Tatsächliche Laufzeit variiert auf unterschiedlichen Computer-Systemen
- Erwartete Laufzeit verschiedener Algorithmen auf dem selben Computer kann nur für lange Berechnungen verglichen werden (variierende Systemauslastung).

Effizienz eines Algorithmus

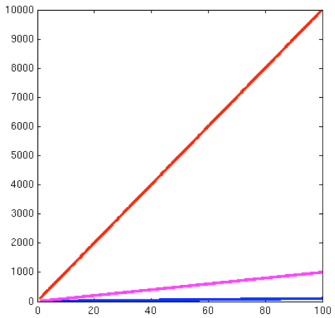
- In der Regel interessiert man sich nicht für die exakte Anzahl von Operationen, sondern nur für die Komplexitätsklasse.
- Beispiel: lineare Liste vs. Suchbaum:
  - 1000 vs. 10
  - 2000 vs. 11
  - ...
  - 1000000 vs. 20
  - 2000000 vs. 21

Komplexitätsklassen

- Performance Charts
  - X-Achse : Größe der Eingabedaten
  - Y-Achse : Rechenzeit
- Lineare vs. logarithmische Achsen
  - Exakte Werte
  - Größenordnungen

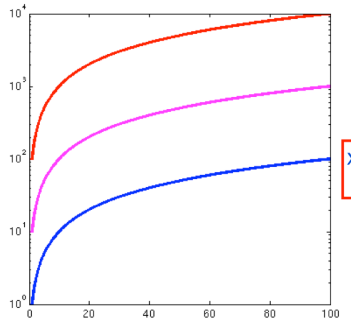
Je größer die Datenmengen, desto größer die Auswirkungen der Komplexitätsklassen

Komplexitätsklassen



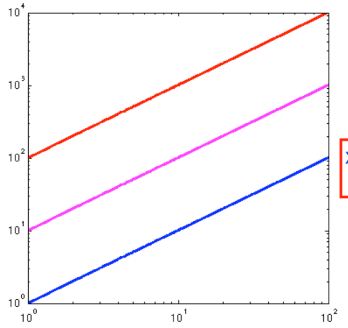
x, 10x, 100x
[lin / lin]

Komplexitätsklassen



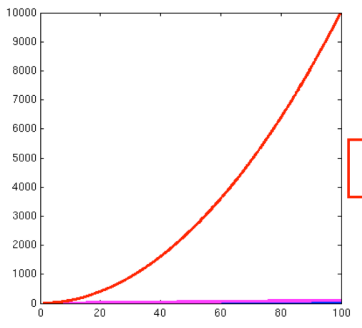
x, 10x, 100x
[lin / log]

Komplexitätsklassen



x, 10x, 100x
[log / log]

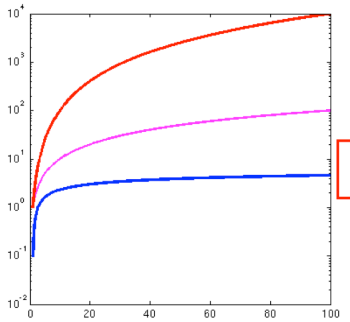
Komplexitätsklassen



Log(x), x, x^2
[lin / lin]

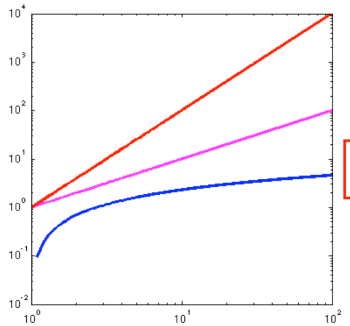
Logarithmische Skalierung
Gleiche Steigung => Gleiche Komplexitätsklasse

Komplexitätsklassen



Log(x), x, x^2
[lin / log]

Komplexitätsklassen



Log(x), x, x^2
[log / log]

Multiplikation mit kleinem Faktor bringt für kleine Probleme was, für große nicht

Abstrakte Analyse

- Abstrakte Analyse
  Zähle die wesentlichen Operationen auf einem idealen Computer.
- Die Komplexität eines Algorithmus ist unabhängig von der Programmiersprache.

Stripped Java

- Teilmenge von Java
  - Prozedur-Aufrufe
    - Integer-Arithmetik
    - Zuweisungen (Lesen, Schreiben)
    - if-then-else
    - while
- Vollständig aber einfacher zu analysieren als full Java

Berechnungsaufwand

- Prozedur-Aufruf
  - Speichere Kontext auf dem Stack
  - Speichere Rücksprungadresse
  - Kopiere aktuelle Parameter
  - Generiere neuen Prozedurkontext
  - ...
  - Kopiere Rückgabewert
  - Stelle alten Kontext wieder her
- Kosten: $C_{PC} = C_{PC}(S_{old}, Args, S_{new})$

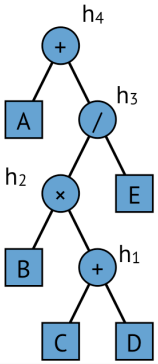
Berechnungsaufwand

- Integer-Arithmetik (Einheitskosten der Elementaroperationen)
  $C_+ \leqslant C_- \leqslant C_\times \leqslant C_/$
- Konvertiere komplizierte Terme in die 3-Adress-Form
- Achtung: bei genauer Analyse muß die Anzahl der Stellen berücksichtigt werden

3-Adress-Form

A + B × (C + D) / E

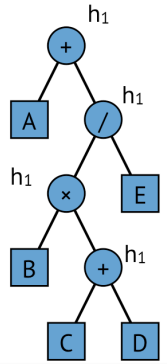
$h_1 = C + D$
$h_2 = B \times h_1$
$h_3 = h_2 / E$
$h_4 = A + h_3$



3-Adress-Form

A + B × (C + D) / E

$h_1 = C + D$
$h_1 = B \times h_1$
$h_1 = h_1 / E$
$h_1 = A + h_1$



Kostenmaße

- $C_+ \leqslant C_- \leqslant C_\times \leqslant C_/$
- Addition
  - Addition einzelner Ziffern:
    $x + y = z + c$
    - Berücksichtigung des Überlaufs
    - # Sub-Operationen = # Stellen
      $\approx \log z$
    - Java-Integer : # Stellen = 32

Kostenmaße

- $C_+ \leqslant C_- \leqslant C_\times \leqslant C_/$
- Multiplikation
  - Multiplikation einzelner Ziffern: $x \cdot y = z$
    - $(10 \cdot x_1 + x_0) \cdot (10 \cdot y_1 + y_0) =$
      $100 \cdot x_1 \cdot y_1 + 10 \cdot (x_1 \cdot y_0 + x_0 \cdot y_1) + x_0 \cdot y_0$
    - # Sub-Operationen = # Stellen^2
      $\approx (\log z)^2$
    - Java-Integer : # Stellen = 32

Kostenmaße

- $C_+ \leqslant C_- \leqslant C_\times \leqslant C_/$
- Multiplikation
  - Multiplikation einzelner Ziffern: $x \cdot y = z$
    - $(b \cdot x_1 + x_0) \cdot (b \cdot y_1 + y_0) =$
      $b^2 \cdot x_1 \cdot y_1 + b \cdot (x_1 \cdot y_0 + x_0 \cdot y_1) + x_0 \cdot y_0$
    - # Sub-Operationen = # Stellen^2
      $\approx (\log z)^2$
    - Java-Integer : # Stellen = 32

Kostenmaße

- $C_+ \leqslant C_- \leqslant C_\times \leqslant C_/$
- Multiplikation
  - Multiplikation einzelner Ziffern: $x \cdot y = z$
    - $(b \cdot x_1 + x_0) \cdot (b \cdot y_1 + y_0) =$
      $x_0 \cdot y_0 + b \cdot ((x_0 + x_1) \cdot (y_0 + y_1) - x_0 \cdot y_0 - x_1 \cdot y_1) +$
      $b^2 \cdot x_1 \cdot y_1$
    - # Sub-Operationen = # Stellen$^{1.58}$
      $\approx (\log z)^{\log 3}$
    - Java-Integer : # Stellen = 32

Kostenmaße

- $C_+ \leqslant C_- \leqslant C_\times \leqslant C_/$
- Multiplikation
  - Karatsuba-Multiplikation
    - Spart eine Sub-Multiplikation aber benötigt drei zusätzliche Additionen
    - Lohnt sich nur für sehr große Zahlen !!!
    - Es gibt theoretisch noch bessere Algorithmen

Abstrakte Analyse

- Theorie: Effiziente Algorithmen lohnen sich vor allem für große Probleme
- Praxis: Für kleinere Probleme können einfache Algorithmen unter Umständen sinnvoller sein.
- Die Klassifizierung groß/klein hängt vom Problem und vom Algorithmus ab.

Berechnungsaufwand

- Zuweisung (Lesen, Schreiben)
  - Einheitskosten C_{ass}
  - Caching-Effekte werden vernachlässigt
- Array-Zugriff A[i] enthält eine versteckte Addition.

Berechnungsaufwand

- if X then Y else Z
  - Worst Case:
    #OP = #OP(X) + max{ #OP(Y), #OP(Z) }
  - Average Case:
    P = Probability(X = true)
    #OP = #OP(X) + P×#OP(Y) +
    (1−P)×#OP(Z)

Berechnungsaufwand

- while X do Y
  - Konstanter Aufwand:

    #OP = #Loops × [#OP(X) + #OP(Y)]
  - Variabler Aufwand:

    #OP = #Loops × #OP(X) + SUM(#OP(Y_i))

Beispiel

```
• Sortiere(A[1..n])
    i ← 1
    while i < n do
        min ← i
        j ← i+1
        while j ⩽ n do
            if A[j] < A[min] then
                min ← j
            j ← j+1
        swap(A[i],A[min])
        i ← i+1
```

Beispiel

- Aufwand der inneren Schleife C_{inner}
- Aufwand der äußeren Schleife C_{outer}
- Anzahl der Durchläufe

$$C(n) = \sum_i C_{outer} + (n-i) \times C_{inner}$$
$$= n \times C_{outer} + n \times (n-1)/2 \times C_{inner}$$
$$\approx n^2 \times C_{inner}/2 \text{... für „große“ n}$$

Best / Worst / Average Case

- Verschieden optimistische Abschätzungen
  - Untere Schranke
  - Erwartungswert
  - Obere Schranke
- Erwartungswert: Mittelwert des Aufwandes über alle möglichen Eingaben gewichtet mit der Auftrittswahrscheinlichkeit.

Beispiel

- Multipliziere zwei n-bit Zahlen in Binärdarstellung
- Multiply(x,y)

```
i ← 0
result ← 0
while i < n do
    if bit(x,i) = 1 then
        result ← result + shift(y,i)
    i ← i+1
```


Beispiel

- Aufwand
  ≈ Anzahl der Additionen
  = Anzahl der Einsen in der Binärdarstellung von x
- Best Case : $x = 0$ ... 0 Additionen
- Worst Case: $x = 2^n - 1$ ... n Additionen
- Average Case ???

Beispiel

- Anzahl der möglichen Eingaben $N = 2^n$
- Anzahl der Eingaben mit k Einsen in x

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- Mittlerer Aufwand ...

Beispiel

- Mittlerer Aufwand

$$\begin{aligned} k\binom{n}{k} &= k\frac{n!}{k!(n-k)!} \\ &= \frac{n(n-1)!}{(k-1)!(n-1-k+1)!} \\ &= n\binom{n-1}{k-1} \end{aligned}$$

$$\sum_k k\binom{n}{k} = \sum_k n\binom{n-1}{k-1} = n2^{n-1}$$

Beispiel

- Mittlerer Aufwand

  $S / N = n \times 2^{n-1} / 2^n = n / 2$ Additionen

- Bemerkung zur Kombinatorik
  - Permutationen $n!$
  - Kombinationen $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Asymptotische Komplexität

- In der Regel ist das quantitative Zählen der Elementaroperationen mühsam und wenig ergiebig.
- Man ist daher eher an qualitativen Aussagen interessiert, z.B. wie verändert sich der Rechenaufwand, wenn man die Eingabedaten verdoppelt?

Beispiele

- Addition
  - Verdopplung der Stellen
  - Verdopplung der Sub-Operationen
- Multiplikation / Sortierbeispiel
  - Verdopplung der Stellen / Elemente
  - Vervierfachung der Sub-Operationen
- Suchen im Suchbaum:
  - Verdopplung der Knoten im Baum
  - Erhöhung des Suchaufwandes um einen Test (vollständiger Baum erhöht sich um ein Level)

Asymptotische Komplexität

- Definition von Schranken für Funktionen $f,g : N \rightarrow R_+$
  - $O(f) = \{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : g(n) \leq c \cdot f(n) \}$
  - $\Omega(f) = \{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : g(n) \geq c \cdot f(n) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$ $\{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : f(n)/c \leq g(n) \leq c \cdot f(n) \}$

Funktionen: Rechenzeit/Eingaberöße
O(f) besteht aus allen Funktionen, für welche f eine Oberschranke ist

Asymptotische Komplexität

- Definition von Schranken für Funktionen $f,g : N \to R_+$
  - $O(f) = \{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : g(n) \leq c \cdot f(n) \}$
  - $\Omega(f) = \{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : g(n) \geq c \cdot f(n) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$
    $\{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : f(n)/c \leq g(n) \leq c \cdot f(n) \}$

O(f) besteht aus allen Funktionen, für welche f eine Oberschranke ist

Asymptotische Komplexität

- Definition von Schranken für Funktionen $f,g : N \to R_+$
  - $O(f) = \{ g \mid$ $: g(n) \leq$ $f(n) \}$
  - $\Omega(f) = \{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : g(n) \geq c \cdot f(n) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$
    $\{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : f(n)/c \leq g(n) \leq c \cdot f(n) \}$

O(f) besteht aus allen Funktionen, für welche f eine Oberschranke ist

Asymptotische Komplexität

- Definition von Schranken für Funktionen $f,g : \mathbb{N} \to \mathbb{R}_+$
  - $O(f) = \{ g \mid \exists c>0 \quad : g(n) \leq c \cdot f(n) \}$
  - $\Omega(f) = \{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : g(n) \geq c \cdot f(n) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$
    $\{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : f(n)/c \leq g(n) \leq c \cdot f(n) \}$

Asymptotische Komplexität

- Definition von Schranken für Funktionen $f,g : \mathbb{N} \to \mathbb{R}_+$
  - $O(f) = \{ g \mid \exists c>0\ \exists n_0>0 \quad : g(n) \leq c \cdot f(n) \}$
  - $\Omega(f) = \{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : g(n) \geq c \cdot f(n) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$
    $\{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : f(n)/c \leq g(n) \leq c \cdot f(n) \}$

Konstanter Faktor ändert nichts an Komplexitätsklasse!

Asymptotische Komplexität

- Definition von Schranken für Funktionen $f,g : N \rightarrow R_+$
  - $O(f) = \{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : g(n) \leq c \cdot f(n) \}$
  - $\Omega(f) = \{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : g(n) \geq c \cdot f(n) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$ $\{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : f(n)/c \leq g(n) \leq c \cdot f(n) \}$

Asymptotische Komplexität

- Definition von Schranken für Funktionen $f,g : N \rightarrow R_+$
  - $O(f) = \{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : g(n) \leq c \cdot f(n) \}$
  - $\Omega(f) = \{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : g(n) \geq c \cdot f(n) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$ $\{ g \mid \exists c>0\ \exists n_0>0\ \forall n \geq n_0 : f(n)/c \leq g(n) \leq c \cdot f(n) \}$

Für alle n>n_0 ist der Berechnungsaufwand höchstens c*F(n)
c kann beliebigen Wert haben
Untere Schranke omega
Schnitt: theta

Rechenregeln

- $f + g \in O(\max\{f,g\}) = \begin{cases} O(f) & \text{falls } g \in O(f) \\ O(g) & \text{falls } f \in O(g) \end{cases}$
- $f \times g \in O(f \times g)$
- $g(n) := a \cdot f(n) + b \wedge f \in \Omega(1) \Rightarrow g \in O(f)$
- Schreibweise
  $n \times C_{outer} + n \times (n-1)/2 \times C_{inner} = O(n^2)$

Fibonacci-Zahlen

- $F(n) = \begin{cases} 0 & \dots n = 0 \\ 1 & \dots n = 1 \\ F(n-1)+F(n-2) & \dots n > 1 \end{cases}$
- F(n) positiv für alle n
- F(n) wächst für n > 2 streng monoton

Summe zweier Komplexitätsklassen liegt in maximaler KK
Affine bbildung: g(n) in O(f), da a konstant ist

Fibonacci-Zahlen

- $F(n) = F(n-1) + F(n-2)$
  $< 2 \times F(n-1)$
  $< 4 \times F(n-2)$
  $< \ldots$
  $< 2^{n-1} \times F(1) = 2^n/2$

- $F \in O(2^n)$, $c = 1/2$

Fibonacci-Folge wächst höchstens exponentiell schnell!

Fibonacci-Zahlen

- $F(n) = F(n-1) + F(n-2)$
  $> 2 \times F(n-2)$
  $> 4 \times F(n-4)$
  $> \ldots$

  $> \begin{cases} 2^{(n-1)/2} \times F(1) & \ldots \text{ n ungerade} \\ 2^{n/2-1} \times F(2) & \ldots \text{ n gerade} \end{cases}$

- $F \in \Omega(2^{n/2})$, $c = 1/\sqrt{2}$

Folge wächst mindestens exponentiell schnell (2^(n/2) = 1/sqrt(2)*2^n), 1/sqrt(2) konstanter Faktor!

Alternative Definition

- „Wachstumsrate“ $(f,g : N \to R_+)$
- $\lim_{n\to\infty} g(n) / f(n) = \begin{cases} 0 & \dots \text{ g wächst langsamer} \\ c & \dots \text{ g und f wachsen gleich} \\ \infty & \dots \text{ g wächst schneller} \end{cases}$

- g wächst langsamer : $g \in O(f) \setminus \Theta(f)$
- g und f wachsen gleich : $g \in \Theta(f)$
- g wächst schneller : $g \notin O(f)$

Typische Komplexitätsklassen

- O(1) Elementaroperation
- O(logn) Binäre Suche
- O(n) Lineare Suche
- O(n×logn) Sortieren (später)
- $O(n^2)$ Sortieren (vorhin)
- $O(n^3)$ Invertieren von Matrizen
- $O(2^n)$ Labyrinth-Suche (vollständig)
- O(n!) Zahl von Permutationen

O(1): Dauer der Berechnung ist unabhängig von Eingabegröße!!
Labyrinth Suche: Jede neue Zelle verdoppelt die Anzahl der Pfade, welche durchsucht werden müssen

Fibonacci-Zahlen

- $F(n) = \begin{cases} 0 & \dots n = 0 \\ 1 & \dots n = 1 \\ F(n-1)+F(n-2) & \dots n > 1 \end{cases}$

- F(n) positiv für allen
- F(n) wächst für n > 2 streng monoton

Fibonacci rekursiv

- F(n)

```
if n> 1 then
    return F(n−1) + F(n−2)
else
    return n
```

- T(0) = T(1) = t
- T(n+2) = t + T(n+1) + T(n)

Fibonacci rekursiv

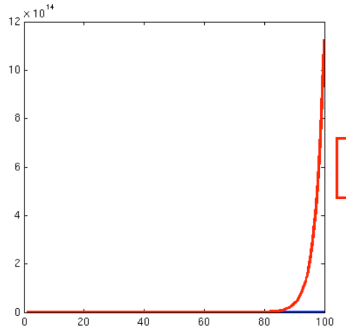
- Behauptung: $T(n) \geqslant t \times F(n+1)$
- Beweis: Vollständige Induktion …
  - Anfang: $T(0) = t \geqslant t \times F(1) = t$
    $T(1) = t \geqslant t \times F(2) = t$
  - Schritt: $T(n+2) = t + T(n+1) + T(n)$
    $\geqslant t + t \times F(n+2) + t \times F(n+1)$
    $\geqslant t + t \times F(n+3)$
    $\geqslant t \times F(n+3)$
- $T(n) \in \Omega(2^{n/2})$

Fibonacci iterativ

- F(n)
  $f_{old} = 1$
  $f_{new} = 0$
  while n > 0 do
    $f_{new} \leftarrow f_{new} + f_{old}$
    $f_{old} \leftarrow f_{new} - f_{old}$
    $n \leftarrow n-1$
  return f_{new}
- $T(n) = t \times n = \Theta(n)$

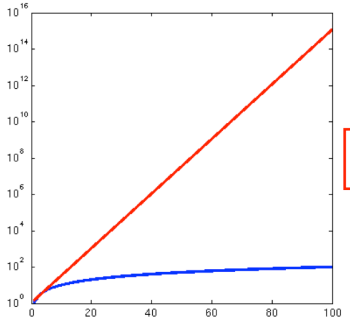
Vollständige Induktion: Stelle Behauptung auf, gilt für eine Zahl n! Mache Schritt von n -> n+1

Fibonacci Vergleich



$x, 2^{x/2}$
[lin / lin]

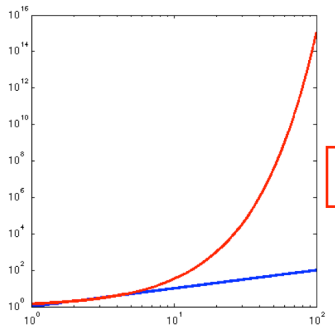
Fibonacci Vergleich



$x, 2^{x/2}$
[lin / log]

Iterative Algorithmus deutlich effizienter als rekursiver!

Fibonacci Vergleich



x, $2^{x/2}$
[log / log]

Standardabschätzungen

- Reihensummen
  - Geschachtelte Schleifen
    - Abschätzung durch Integrale
    - Vollständige Induktion
- Rekursionsgleichungen
  - Eliminierung
    - Master-Theorem
    - Direkter Ansatz

Reihensummen

- GeschachtelteSchleifen
  - for i ←1 to n do
    O(1) = O(n)
  - for i ←1 to n do
    O(i) = $O(n^2)$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2 + n) = O(n^2)$$

Reihensummen

- GeschachtelteSchleifen
  - for i ←1 to n do O(1) = O(n)
  - for i ←1 to n do O(i) = $O(n^2)$
  - for i ←1 to n do
    for j ←1 to n do O(1) = $O(n^2)$
  - for i ←1 to n do
    for j ←1 to i do O(1) = $O(n^2)$
  - …

Schleifendurchlauf bis i ist trotzdem O(n), da die Anzahl der Iterationen effektiv halviert wird

Reihensummen

- Geschachtelte Schleifen
  - Generell: der Aufwand $T_k(n)$ einer k-fach geschachtelten Schleife (mit linear beschränkten Laufintervallen) besitzt als obere Schranke ein Polynom vom Grad k

$$T_k(n) = O\left(\sum_{i=0}^{k} \alpha_i n^i\right) = O(n^k)$$

Höchter Exponent des Polynoms!

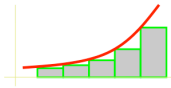
Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals

$$\int_0^n f(x)dx \approx \sum_{i=0}^{n-1} f(i+h), \quad h \in [0,1)$$

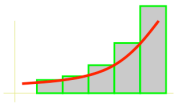
Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals



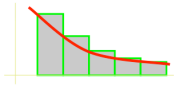
Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals



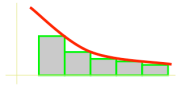
Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals



Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals



Abschätzung durch Integral

- Beispiel

$$\sum_{i=0}^{n-1} i^2 \leq \int_0^n x^2 dx = \frac{x^3}{3}|_0^n = \frac{1}{3}n^3 = O(n^3)$$

$$\sum_{i=0}^{n-1} i^2 = \sum_{i=1}^{n-1} i^2 \geq \int_0^{n-1} x^2 dx = \frac{x^3}{3}|_0^{n-1}$$

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

$$\sum_{i=0}^{n-1} i^2 = \Theta(n^3)$$

Vollständige Induktion

- Behauptung
- Zeige: Behauptung gilt für $n = n_0$
- Zeige: Wenn die Behauptung für ein n gilt, dann gilt sie auch für $n+1$

Vollständige Induktion

- Behauptung: $\sum_{i=0}^{n-1} i^2 = \frac{1}{6}\left(2n^3 - 3n^2 + n\right)$
- Beweis: n = 1 $\sum_{i=0}^{0} i^2 = \frac{1}{6}\left(2 - 3 + 1\right) = 0$

Vollständige Induktion

- Behauptung: $\sum_{i=0}^{n-1} i^2 = \frac{1}{6}\left(2n^3 - 3n^2 + n\right)$
- Beweis: n → n+1

$$\begin{aligned}
\sum_{i=0}^{n} i^2 &= n^2 + \sum_{i=0}^{n-1} i^2 \\
&= n^2 + \frac{1}{6}\left(2n^3 - 3n^2 + n\right) \\
&= \cdots \\
&= \frac{1}{6}\left(2(n+1)^3 - 3(n+1)^2 + (n+1)\right)
\end{aligned}$$

nach Induktions-voraussetzung

Rekursionsgleichungen

- Elimination
- Master-Theorem
- Direkter Ansatz

Rekursionsgleichungen

- $F(n) = a + b \times F(n-1)$
- $F(n) = a + b \times F(n-1) + c \times F(n-2)$
- ...
- $F(n) = a + b \times F(n/c)$
- $G(m) := F(c^m)$
  → $G(m) = a + b \times G(m-1)$

Rekursionsgleichungen

- $F(n) = a(n) + b(n) \times F(n-1)$
- $F(n) = a(n) + b(n) \times F(n-1) + c(n) \times F(n-2)$
- ...
- $F(n) = a(n) + b(n) \times F(n/c)$
- $G(m) := F(c^m)$
  → $G(m) = a(n) + b(n) \times G(m-1)$

Elimination

- $F(n) = n + F(n-1)$
  $= n + (n-1) + F(n-2)$
  $= \ldots$
  $= n + (n-1) + \ldots + 1 + F(0)$
  $= n \times (n-1) / 2 + F(0)$
  $= O(n^2)$

einfaches Sortieren

1. Rekursionsgleichungaus Algorithmus ableiten
2. Rekursionsgleichung lösen!

Elimination

- $F(n) = F(n-1) + F(n-2)$ Fibonacci
  $= 2 \times F(n-2) + F(n-3)$
  $= 3 \times F(n-3) + 2 \times F(n-4)$
  $= 5 \times F(n-4) + 3 \times F(n-5)$
  $= \ldots$
  $= ???$

Formel nicht einfach ableitbar

Abschätzung nach oben

- Typisch: $T(n) = a \times T(n/2) + b$
- Einfach, wenn $n = 2^k$
- Sonst: finde $m = 2^k$ mit $m/2 < n \leqslant m$
- Dann gilt: $T(n) \leqslant T(m)$

Elimination

- T(n) = T(n/2) + 1
  = T(n/4) + 1 + 1
  = T(n/8) + 1 + 1 + 1
  = …
  = T(1) + log n
  = O(logn)

binäre Suche

Elimination

- T(n) = 4×T(n/2)
  = 16×T(n/4)
  = 64×T(n/8)
  = …
  = $4^{\text{ld } n}$×T(1)
  = n^2×T(1)
  = O(n^2)

Multiplikation großer Zahlen

ld: logarithmus dualis (Zweierlogarithmus)
Wahl des Logarithmus ist egal, da sie sich nur durch konstante Multiplation unterscheiden. Dies gilt nur solange die logarithmen nicht im Exponenten auftauchen!

Elimination

Multiplikation großer Zahlen

- $T(n) = 4 \times T(n/2) + 3 \times n$
  $= 16 \times T(n/4) + 12 \times n/2 + 3 \times n$
  $= 64 \times T(n/8) + 48 \times n/4 + 12 \times n/2 + 3 \times n$
  $= \ldots$
  $= 3 \times n \times (n + \ldots + 4 + 2 + 1)$
  $= 3 \times n \times (2^{(\log n)+1} - 1)$
  $= 3 \times n \times (2 \times n - 1) = O(n^2)$

Ausführlich mit Additionen!

Elimination

effizientes Sortieren

- $T(n) = 2 \times T(n/2) + n$
  $= 4 \times T(n/4) + 2 \times n/2 + n$
  $= 8 \times T(n/8) + 4 \times n/4 + 2 \times n/2 + n$
  $= \ldots$
  $= 2^{(\log n)-1} \times n / 2^{(\log n)-1} + \ldots + 2 \times n/2 + n$
  $= n \times \log n$

Schwieriger zu sehen. Grenzen der Elimination!

Master-Theorem

- $T(n) = \begin{cases} c & \dots\ n = 1 \\ a \times T(n/b) + c \times n & \dots\ n > 1 \end{cases}$
- $T(n) = \begin{cases} O(n) & \dots\ a < b \\ O(n \times \log n) & \dots\ a = b \\ O(n^{\log_b a}) & \dots\ a > b \end{cases}$

Master-Theorem

- ... sei $n = b^k$:
- $T(n) = a \times T(n/b) + c \times n$
  $= a^2 \times T(n/b^2) + a \times c \times n/b + c \times n$
  $= a^3 \times T(n/b^3) + a^2 \times c \times n/b^2 + a \times c \times n/b + c \times n$
  $= \dots$
  $= cn \sum_{i=0}^{k} \left(\frac{a}{b}\right)^i$

Master-Theorem

- T(n) = $cn \sum_{i=0}^{k} \left(\frac{a}{b}\right)^i$
- a < b: T(n) $\leqslant$ $cn \frac{1}{1-a/b} = O(n)$
- a = b: T(n) = $cn(k+1) = O(n \log n)$
- a > b: T(n) = $cn \frac{(a/b)^{k+1}-1}{a/b-1}$

Master-Theorem

$$\begin{aligned} \frac{(a/b)^{k+1}-1}{a/b-1} &= O\left(\left(\frac{a}{b}\right)^k\right) \\ &= O\left(\left(\frac{a}{b}\right)^{\log_b n}\right) \\ &= O(a^{\log_b n}/n) \end{aligned}$$

Master-Theorem

$$\begin{aligned}
\frac{(a/b)^{k+1}-1}{a/b-1} &= O\left(\left(\frac{a}{b}\right)^{k}\right) \\
&= O\left(\left(\frac{a}{b}\right)^{\log_b n}\right) \\
&= O(a^{\log_b n}/n) \\
cn\frac{(a/b)^{k+1}-1}{a/b-1} &= O(a^{\log_b n}) \\
&= O(n^{\log_b a})
\end{aligned}$$

Master-Theorem

- T(n) = $\begin{cases} c & \dots\ n = 1 \\ a \times T(n/b) + c \times n & \dots\ n > 1 \end{cases}$

- T(n) = $\begin{cases} O(n) & \dots\ a < b \\ O(n \times \log n) & \dots\ a = b \\ O(n^{\log_b a}) & \dots\ a > b \end{cases}$

Master-Theorem

- $T(n) = \begin{cases} c & \dots\ n = 1 \\ a \times T(n/b) + O(n^p) & \dots\ n > 1 \end{cases}$

- $T(n) = \begin{cases} O(n^p) & \dots\ a < b^p \\ O(n^p \times \log n) & \dots\ a = b^p \\ O(n^{\log_b a}) & \dots\ a > b^p \end{cases}$

Multiplikation großer Zahlen

- Standard: $T(n) = 4 \times T(n/2) + 3 \times n$
  $= O(n^{\text{ld } 4})$
  $= O(n^2)$
- Karatsuba: $T(n) = 3 \times T(n/2) + 6 \times n$
  $= O(n^{\text{ld } 3})$
  $= O(n^{1.58})$

Direkter Ansatz

- $F(n) = F(n-1) + F(n-2)$
- Ansatz : $F(n) = \varphi^n$
- $\varphi^n = \varphi^{n-1} + \varphi^{n-2}$
  $\Leftrightarrow \quad \varphi^2 = \varphi + 1$
  $\Leftrightarrow \quad \varphi^2 - \varphi - 1 = 0$
  $\Leftrightarrow \quad \varphi_{\pm} = (1 \pm \sqrt{5}) / 2$

Direkter Ansatz

- $\varphi_{\pm} = (1 \pm \sqrt{5}) / 2$
- $F(n) := \alpha \times \varphi_+^n + \beta \times \varphi_-^n$
- $F(0) = \alpha + \beta = 0 \Rightarrow \alpha = -\beta$
- $F(1) = \alpha \times \varphi_+ + \beta \times \varphi_- = \alpha \times (\varphi_+ - \varphi_-) = 1$
  $\Rightarrow \alpha = 1/\sqrt{5}$
- $F(n) = (\varphi_+^n - \varphi_-^n) / \sqrt{5}$