Vorlesung Algorithmen und Datenstrukturen (AD)

Prof. Dr. Matthias Rarey

Zentrum für Bioinformatik, Universität Hamburg
Bundesstraße 43, 20146 Hamburg
rarey@zbh.uni-hamburg.de

www.zbh.uni-hamburg.de

Organisatorisches: Vorlesung

- **Module:**
  - BSc Informatik: Modul InfB-AD / IP04
  - BSc Computing in Science
  - MSc Bioinformatik: Modul MBI-04 (AD)
  - Wahlpflichtveranstaltung in weiteren Informatik-Studiengänge
  - Nebenfach Informatik
- **Vorlesung**
  - Mi 10-12h c.t. Phil C 7tg,
  - Fr 14-16h c.t. Phil B 14tg, erstmalig 19.10
- **Sprechstunden**
  - nach Vereinbarung (oder nach der Vorlesung)
  - Zentrum für Bioinformatik, Bundesstraße 43, Raum 205
  - Terminvereinbarung: geringhoff@zbh.uni-hamburg.de

Organisatorisches: Übungen

- **Übungen (14tg, erstmalig am 24. Okt.)**

1.	Mi	12-14	Frank Heitmann	F-009
2.	Mi	14-16	Florian Lauck	ZBH-R16
3.	Mi	14-16	Mathias von Behren	F-534
4.	Mi	14-16	Matthias Hilbig	F-635
5.	Mi	16-18	Mathias von Behren	D-129
6.	Mi	16-18	Matthias Hilbig	F-534
7.	Do	10-12	Frank Heitmann	F-334
8.	Do	10-12	Jan Henrik Röwekamp	F-534
9.	Fr	10-12	Thomas Otto	ZBH-R16
10.	Fr	12-14	Thomas Otto	ZBH-R16

- **In den Übungen besteht Anwesenheitspflicht [6 von 7].**

Organisatorisches: Übungszettel, zeitlicher Ablauf

- **Übungsblätter**
  - Ausgabe der Zettel: Mo ca. 12h, 14tg, erstmalig am 17.10 (29.10)
    Online unter: www.stine.uni-hamburg.de
    Oder: ZBH-Foyer
    Oder: Stellingen: Treppenhaus Haus C/D, 1. Stock
  - Abgabe der Lösungen: Mo 12h [Woche +1]
    < Absprache mit den Ü-Leitern>
    Online: **nur 1 pdf-Dokument**
    Oder: ZBH-Foyer, bzw.
    Stellingen: Treppenhaus Haus C/D, 1. Stock
  - Besprechung/Vorrechnen: in der Übung [Woche +1]

Organisatorisches: Studienleistung

- **Bearbeitung der Ü-Blätter in Gruppen mit 2-3 Studierenden**
- **Fristgerechte Abgabe der Lösungen**
- **50% der erreichbaren Punkte auf den Ü-Blättern**
- **Mindestens 1 Punkt auf 6 von 7 Ü-Blättern**
- **Anwesenheitspflicht bei 6 von 7 Übungen (abzgl. Krankmeld.)**
- **Keine Punkte für offensichtlich abgeschriebene Lösungen (für alle beteiligten Gruppen)**

- **Mind. eine erfolgreiche Präsentation einer Übungsaufgabe durch die Gruppe:**
  - Ü-Gruppenleiter wählt unter korrekten Lösungen und anwesenden Gruppen eine Gruppe aus
  - Ü-Gruppenleiter bestimmt für Teillösungen, wer aus der Gruppe diese präsentiert.

Organisatorisches: Literatur

- **T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein**
  engl. Originalausgabe:
  Introduction to Algorithms, MIT Press, 2009, (3. Aufl.)
  deutsche Übersetzung:
  Algorithmen – Eine Einführung, Oldenbourg Wissenschaftsverlag, überarbeitete Auflage 2010

- **Foliensammlung:**
  - Ausgedrucktes Skript, Updates über STINE
    [nur für den eigenen Gebrauch!]

Themenschwerpunkte

- **Themen:**
  - Entwurf von Algorithmen und Datenstrukturen
  - Beschreibung von Algorithmen
  - Analyse der Platz- und Zeitkomplexität
  - Algorithmen für häufig auftretende Probleme

- **Kapitel**
  1. Algorithmen und deren Komplexität
  2. Grundlegende Datenstrukturen
  3. Sortieren
  4. Suchen
  5. Graphen
  6. Dynamische Programmierung
  7. Lösen schwerer Probleme

Kapitel 1: Algorithmen und deren Komplexität

Beschreibung von Algorithmen
Analyse von Algorithmen
O-Notation
Das Maxsum-Subarray-Problem

1.1 Beschreibung von Algorithmen

- **Informatik = Information + Mathematik**
  - entstand mit der Entwicklung der ersten Rechenanlagen (40‘er J.)
  - anglo-amerikanisch: Computer Science
  - Wissenschaft vom ‚mechanischen Rechnen‘
- **Algorithmus (von Al-Chowarizmi, persischer Math., ca. 780)**
  = mechanisch ausführbares Rechenverfahren

BSP [Algorithmus]: GGT (Euklid, 300 v.Chr.)

ggt(a,b): *1. a = b*q + r mit r < b (ganzzahlige Division)*
2. falls r=0: output b
3. a ← b; b ← r;
4. gehe zu Schritt 1.

Der Algorithmenbegriff

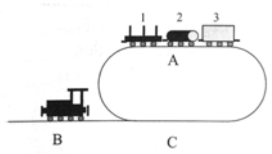
Begriffe

- **„Problem“:** definiert eine Eingabe-Ausgabe-Beziehung
- **„Instanz“:** eine mögliche Eingabe für das Problem
- **„Algorithmus“:** definiert eine Folge elementarer Anweisungen zur Lösung eines Problems
- **„Korrektheit“:** Ein Algorithmus **stoppt** für **jede** mögliche Eingabe mit der korrekten Ausgabe

- **Eigenschaften von Algorithmen:**
  - mechanische Verfahren
  - bestehen aus mehreren elementaren Schritten
  - Schritte werden ggf. wiederholt durchlaufen [Iteration]
  - Schritte werden ggf. bedingt durchlaufen [Selektion]
  - Das Verfahren führt sich ggf. selbst mit veränderten Parametern aus [Rekursion]

Beschreibung von Algorithmen

- Spezifikation: **Beschreibung des Problems**
  - vollständig: alle Anforderungen und Rahmenbedingungen
  - detailliert: welche Hilfsmittel / Basisoperation sind erlaubt
  - unzweideutig: klare Kriterien für akzeptable Lösungen
- **Bsp: Eine Lokomotive soll die auf Gleis A stehenden Wagen 1,2,3 in Reihenfolge 3,1,2 auf Gleis C abstellen.**
  - Vollständigkeit:
    - Wie viele Wagen kann die Lok auf einmal ziehen?
  - Detailliertheit:
    - Welche Aktionen kann die Lok ausführen?
  - Unzweideutigkeit:
    - Darf die Lok am Ende zwischen den Wagen stehen?



- **Beschreibung durch Vorbedingung / Nachbedingung**

Beschreibung von Algorithmen

- **Beschreibungsformen:**
  1. Natürliche Sprache
  2. Computerprogramme
  3. Hardwareentwurf
  4. Mischung aus 1. und 2. → Pseudo-Code

  Regel: Wie die Spezifikation muss der Algorithmus vollständig, detailliert und unzweideutig beschrieben sein.
- **Pseudo-Code:**
  - Angelehnt an imperative Programmiersprachen (Pascal, C, …)
  - Kontroll- und Datenstrukturen werden aus P-Sprachen übernommen
  - Bedingungen, Funktionen werden ggf. natürlich-sprachlich formuliert

Beschreibung von Algorithmen

- **Pseudo-Code Konventionen (aus Cormen et al, ab 3. Aufl.)**
  1. Einrücken kennzeichnet die Blockstruktur
  2. **if – [elseif] – else** Anweisungen für die bedingte Ausführung
  3. **while, for – to/downto, repeat – until** Anweisungen für die iterative Ausführung
     - **while** <Bedingung ist wahr>
     - **for** <variable> ← <Wert/Ausdruck> **to** <Wert/Ausdruck>
       Anw1
       Anw2 …
     - **repeat** Anw1
       Anw2 …
       until <Bedingung ist wahr>
     - *Achtung: Variable der for-Schleife ist auch noch nach Schleifenende definiert (C-Konvention)*
  4. // leiten Kommentare ein
  5. == steht für den Vergleich von Ausdrücken, = für die Zuweisung

Beschreibung von Algorithmen

- **Pseudo-Code Konventionen**
  6. Feldelemente werden durch eckige Klammern indiziert, Teilfelder durch Indexbereiche, *A[i], A[i..j]*
  7. Zusammenhängende Daten werden als Objekte mit Attributen dargestellt, das Objektattribut wird durch den .-Operator spezifiziert, A.*länge* bezeichnet die Anzahl der Elemente von Array A
  8. Funktionsparameter werden als Wert übergeben (call-by-value), Objekt- und Array-Bezeichner repräsentieren die Adresse des Objektes
  9. Boole'sche Operatoren werden träge ausgewertet (lazy evaluation), d.h. von links nach rechts bis der Ausdruck garantiert falsch oder wahr ist.

     Bsp: *x* und *y* : y wird nur ausgewertet, wenn x wahr ist.
     x oder *y*: y wird nur ausgewertet, wenn x falsch ist.

Beispiel: Euklids GGT-Algorithmus

- **Algorithmus zur Berechnung des GGT:**

BSP [Algorithmus]: GGT (Euklid, 300 v.Chr.)
ggt(a,b):
*1. a = b*q + r mit r < b (ganzzahlige Division)*
2. falls r=0: output b
3. a ← b; b ← r;
4. gehe zu Schritt 1.

Iterative Variante:

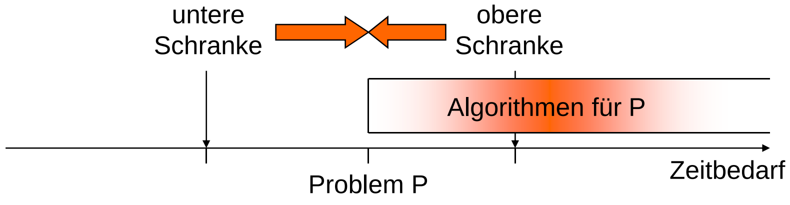
```
GGT(a, b)            // Annahme: a > b
  while  a mod b > 0
      r = a mod b
      a = b; b = r
  return b
```

Rekursive Variante:

```
GGT(a, b)            // Annahme: a > b
  r = a mod b
  if r > 0    return GGT(b, r)
  else        return b
```


1.2 Analyse von Algorithmen

- **Ziel:** theoretische (d.h. ohne ein Computerprogramm zu schreiben) Analyse von Problemen und Algorithmen
  - Welche Probleme sind lösbar?
  - Welche Unterschiede gibt es in der Mächtigkeit von Computermodellen?
  - Welche Ressourcen (Zeit, Speicherplatz) werden mindestens benötigt?
  - **Ist der Algorithmus für das Problem korrekt?**
  - **Welche Ressourcen benötigt ein gegebener Algorithmus?**



Korrektheit von Algorithmen

- **Formale Korrektheit**
  - Angabe von Vor- und Nachbedingung für jede Anweisung
  - Schleifeninvariante: Bedingung, die vor, während (d.h. nach jeder Iteration) und nach Ausführung einer Schleife gültig ist
- **Bsp: Berechnung von a^k für k > 0**

```
POTENZ(a, k)      // Annahme: k > 0 und ganzzahlig
  b = 1; i = 0;
  for i = 0 to k
    b = b * a
  return b
  Invariante:  { b = a^i }
```

- **Beweistechniken sind analog zur Mathematik**
  - Insbesondere: vollständige Induktion, Beweis durch Widerspruch

Asymptotische Laufzeit

- **physikalische Laufzeit**
  - hängt stark vom Computer ab
  - hängt von vielen Details der Eingabedaten ab
- **Modellannahmen Laufzeit**
  - **Uniformes Kostenmaß**: Math. Operationen kosten unabhängig von der Größe der Operanden eine Zeiteinheit
  - **RAM-Modell** (Random-Access-Maschine):
    - Zugriff auf Daten kostet eine konstante Zeiteinheit
    - Algorithmen werden sequentiell ausgeführt (nur ein Prozessor)
  - Statt der genauen Laufzeit wird eine Schranke angegeben
  - Konstante Faktoren werden vernachlässigt.
- **Modellannahmen Speicherplatz**
  - **RAM-Modell:** Speicherung eines elementaren Datenobjekts kostet unabhängig vom Wert konstanten Speicherplatz

Asymptotische Laufzeit

- **Wie können wir die Effizienz eines Algorithmus unabhängig von Details der Eingabe bewerten?**
  - Instanzen (Eingaben) verursachen aufgrund
    1. der Größe
    2. der individuellen Werte

    unterschiedliche physikalische Laufzeiten.

  Bsp: Sortieren einer Zahlenfolge
    1. Wie viele Zahlen sollen sortiert werden?
    2. In welcher initialen Reihenfolge liegen die Zahlen vor?
- **Asymptotische Laufzeit:**
  - Wie verhält sich der Algorithmus bei immer größeren Instanzen?
    - im **worst case**: im ungünstigsten Fall
    - im average case: im statistischen Mittel
    - im best case: im besten Fall

    (bzgl. der Menge aller Instanzen gleicher Länge)
  - Falls nichts weiter angegeben ist, bezieht sich eine Laufzeitanalyse immer auf den Worst Case.

1.3 O-Notation

- **Größenordnungen von Funktionen**

f(N)	Bezeichnung	10	1.000	1.000.000
1	konstant	1	1	1
log(N)	logarithmisch	3	10	20
$\log^2(N)$	log-quadrat	10	100	400
$\sqrt{N}$		3	30	1000
N	linear	10	1.000	1.000.000
N log(N)		30	10.000	20.000.000
N^2	quadratisch	100	1.000.000	10^{12}
N^3	kubisch	1000	10^9	10^{18}
2^N	exponentiell	1000	10^{300}	10^{300000}

Hinweis: zur Berechnung der Beispielzahlen wurde $\log_2()$ verwendet.

O-Kalkül / O-Notation

- O-Kalkül: **Eine Funktion *f* ist *höchstens von der Ordnung g*, falls Konstanten c und n_0 existieren mit $0 \leq f(n) \leq c\, g(n)$ für alle $n > n_0$, d.h.**
  $\exists c > 0\ \exists n_0 > 0\ \forall n > n_0 : 0 \leq f(n) \leq c\, g(n)$
- **Man schreibt f(n) = O(g(n)).**
  - O(g(n)) ist die **Menge** der Funktionen, die nicht stärker wachsen als g (= hat die Bedeutung von $\in$, mit O ist manchmal eine Menge, manchmal ein Repräsentant gemeint)

O	**$\exists c$ ... mit $f(n) \leq c\, g(n)$**	**... f ist höchstens von Ordnung g**
o	**$\forall c$ ... mit $f(n) < c\, g(n)$ ...**	**... f ist von echt kleinerer Ordnung als g**
Ω	**$\exists c$ ... mit $f(n) \geq c\, g(n)$ ...**	**... f ist mindestens von Ordnung g ...**
ω	**$\forall c$ ... mit $f(n) > c\, g(n)$ ...**	**... f ist von echt größerer Ordnung als g**
Θ	**$\exists c_1, c_2$ mit $c_1 g(n) \leq f(n) \leq c_2\, g(n)$**	**... f ist von Ordnung g ...**

O-Notation (O-Kalkül)

- **Rechenregeln für O**
  1. f = O(f)
  2. f,g = O(F) => f + g = O(F)
  3. f= O(F) und c konstant => c * f = O(F)
  4. f= O(F) und g= O(f) => g = O(F)
  5. f= O(F) und g= O(G) => f * g = O(F * G)
  6. f= O(F * G) => f = |F| * O(G)
  7. f= O(F) und $|F| \leq |G|$ => f = O(G)

- Beweis zu 5. (andere Beweise erfolgen analog):
  - $f(n)= O(F(n))$, d.h. $\exists n_0, c_0\ \forall n \geq n_0 : f(n) \leq c_0 F(n)$
  - $g(n)= O(G(n))$, d.h. $\exists n_1, c_1\ \forall n \geq n_1 : g(n) \leq c_1 G(n)$
  - Sei $h(n) = f(n) * g(n)$. Dann gilt $\forall n \geq n_2 = \max(n_0,n_1)$:
    $f(n) * g(n) \leq c_0 F(n)\, c_1 G(n) = c_2 F(n)G(n)$, also $f * g = O(F * G)$

O-Notation

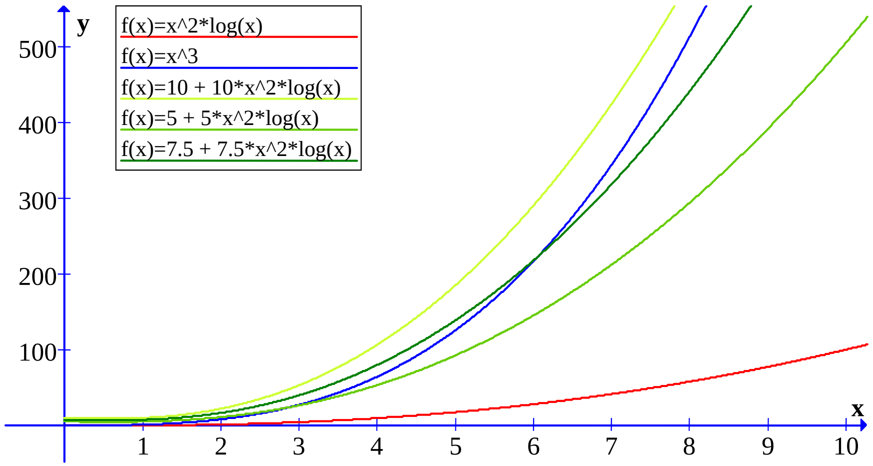
- **Polynome:**
  - Ist p ein Polynom von Grad m gilt: $p = O(N^m)$
    - $n^k = O(N^l)$ für alle $l \geq k$, Anwendung von Regel 2

- **Weitere Beispiele:**
  - $f(n)=3n^2 + 17\sqrt{n} = O(N^2)$
    - $\sqrt{n} = O(N)$, Anwendung von Regel 4 und 2
  - $f(n)= 10^{300}n + 2n \log(n) = O(N \log N)$
  - $f(n)= 2^{2*n} = (2^2)^n = 4^n = O(4^N)$
  - $f(n) = \log_{10} n = O(\log N)$
    - $\log_{10} n = \log_2 n / \log_2(10) = 1/\log_2(10) * \log_2 n = O(\log_2 N) = O(\log N)$
    - Eine Änderung der Basis führt zu einem konstanten Faktor, die Basis muss im O-Kalkül nicht berücksichtigt werden.

O-Notation

- **O-Notationen in Gleichungen und Ungleichungen**
  1. Was bedeutet $2n^2 + 3n +1 = 2n^2 + \Theta(n)$?
     - $\Theta(n)$ bezeichnet eine Menge, gemeint ist:
       $2n^2 + 3n +1 = 2n^2 + f(n)$ mit $f(n) = \Theta(n)$
     - $\Theta(n)$ repräsentiert eine *anonyme Funktion,* d.h. der genaue Funktionsverlauf ist nicht bekannt, lediglich das Wachstumsverhalten.
  2. Achtung: O-Notationen in parametrisierten Summen vermeiden!
     $$\sum_{i=1}^{n} O(i) \neq O(1) + O(2) + \ldots + O(n)$$
  3. Was bedeutet $2n^2 + \Theta(n) = \Theta(n^2)$?
     - $\Theta(n)$ repräsentiert eine anonyme Funktion mit linearem Wachstum f(n)
     - Für jede Funktion $f(n) = \Theta(n)$ gibt es eine Funktion $g(n) = \Theta(n^2)$ und Konstanten $c_1>0$, $c_2>0$, $n_0>0$ mit
       $c_1\, g(n) \leq 2n^2 + f(n) \leq c_2\, g(n)$ für alle $n > n_0$

Einfluss konstanter Faktoren auf das Funktionswachstum



1.4 Laufzeitanalysen

- **Gegeben ist ein Algorithmus A mit Eingabe der Länge N**
  Gesucht wird die Laufzeit des Algorithmus: T_A(N) oder T(N)

- **Welche Operation dauert wie lang?**
  - math. Operationen; Zuweisungen — O(1)
    uniformes Kostenmaß: — O(1)
    [logarithmisches Kostenmaß: — O(log x)
    x ist der Wert des größten Operanden]
  - Klammerung von Anweisungen — O(1)
  - Bedingte Ausführungen — O(1)
  - Schleifen — O(#Schleifendurchläufe)
  - Funktionsaufrufe — O(1)
  - Rekursion: Aufruf der eigenen Funktion mit
    Eingabe der Länge N‘ — T(N‘)

Laufzeitanalysen

- **Bsp: Berechnung von a^k für k > 0**

```
EXPONENT (a, k)
  b = 1                        c0
  for i = 1 to k               c,  k+1 Vergl., k Durchläufe
    b = b * a                  c1
  return b                     c2
```

T(a,k)= c_0 + (k+1) * c + k*c_1 + c_2 = O(k)

- **Bsp: Berechnung von a^k für k > 0**

```
EXPONENT2(a, k)              // Invariante: a^k = p * q^l
  p = 1; q = a; l = k          c0
  while l ≥ 1                  c,  (log2 k+1) Durchläufe
    if( l MOD 2 = 1) p ← p * q c1
    l = l DIV 2                c2
    q = q * q                  c3
  return p                     c4
```

T(a,k) = c_0 + ($\log_2$ k+1) (c + c_1 + c_2 + c_3) + c_4 = O(log k)

Korrektheit von exponent2()

- Schleifeninvariante: **$a^k = p * q^l$**
- **Beweis der Korrektheit:**
  - Vor Schleifenbeginn (Zeile 2) gilt:
    p=1, q=a, l=k   => $a^k = p * q^l$
  - Nach jedem Schleifendurchlauf (nach Zeile 6) gilt:
    vor Zeile 3 gilt $a^k = p * q^l$ (Schleifeninvariante)
    Fall 1: l ist gerade, d.h. l mod 2 = 0
    Seien l‘ und q‘ die Werte von l und q vor Zeile 4. Dann gilt nach
    Zeile 6: $a^k = pq'^{l'} = p(q'^2)^{l'/2} = pq^l$

    Fall 2: l ist ungerade, d.h. l mod 2 = 1
    Seien p‘, l‘, q‘ die Werte von p, l, und q vor Zeile 4, Dann gilt nach
    Zeile 6: $a^k = p'q'^{l'} = p'q'(q'^2)^{(l'-1)/2} = p'q'q^l = pq^l$
  - Nach Schleifenende (Zeile 7) gilt:
    $a^k = p * q^l$ und l=0 (Schleifenterminierung), also $a^k = p$

1.5 Asymptotische Laufzeit rekursiver Funktionen

- **Rekurrenz: Gleichung/Ungleichung, die durch sich selbst mit kleinerem Eingabewert beschrieben wird:**
- **Bsp:**

$$T(n) = \begin{cases} c_0 & : n = 1 \\ T(n-1) + c_1 & : n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & : n = 1 \\ 2T(n/2) + c & : n > 1 \end{cases}$$

- **Eigenschaften:**
  - T(n) ist typischerweise nur für n $\in$IN definiert
  - T(n) = c für kleine n
  - Auf- und Abrundungen für n können i.d.R. vernachlässigt werden.
- **Ziel:**
  1. bestimme die asymptotische Laufzeit
  2. finde eine geschlossene Formel für T(n)

Substitutionsmethode

- **Schritt 1: Setze T(n) wiederholt ein:**

$$\begin{aligned} T(n) &= T(n-1) + c_1 \\ &= T(n-2) + c_1 + c_1 \\ &= T(n-(n-1)) + \underbrace{c_1 + c_1 \cdots + c_1}_{n-1\,\text{mal}} \\ &= T(1) + (n-1)c_1 = (n-1)c_1 + c_0 = O(n) \end{aligned}$$

$$\begin{aligned} T(n) &= 2T(n/2) + c \\ &= 2(2T(n/4) + c) + c \\ &= 4T(n/4) + 2c + c \\ &= 2^i T(n/2^i) + 2^{i-1}c + 2^{i-2}c + \cdots + c \end{aligned}$$

Substitutionsmethode

- **Schritt 2: Intuition**

$$\begin{aligned} T(n) &= 2^i T(n/2^i) + 2^{i-1}c + 2^{i-2}c + \cdots + c \\ &= 2^i T(n/2^i) + \sum_{k=0}^{i-1} 2^k c \end{aligned}$$

  - Wie viele Substitutionsschritte sind notwendig?

$$n/2^i \leq 1 \Leftrightarrow n \leq 2^i \Leftrightarrow \log_2 n \leq i$$

$$T(n) = 2^i T(n/2^i) + c\sum_{k=0}^{i-1} 2^k \qquad \text{mit } i = \log_2 n$$

$$= 2^{\log_2 n} T(1) + c\sum_{k=0}^{\log_2 n - 1} 2^k$$

$$\boxed{\text{Geometrische / Exponentielle Reihe:} \quad \sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1} \quad \text{für } x \neq 0, x \in \mathrm{IR}}$$

$$= nc + c(2^{\log_2 n} - 1) = nc + c(n-1) = 2cn - c = O(n)$$

  - Beweis erfolgt durch vollständige Induktion

Substitutionsmethode

- **Schritt 3: Beweis der Hypothese** (durch vollständige Induktion)
  - Annahme: T(n) $\leq$ kn
  - Induktionsanfang: T(1) = c $\leq$ k*1 für k $\geq$ c
  - Induktionsschritt:

$$T(n) = 2T(n/2) + c \leq 2kn/2 + c \leq kn + c$$

  - Neue Annahme: T(n) $\leq$ kn – c
  - Induktionsanfang: T(1) = c $\leq$ k*1 – c für k $\geq$ 2c
  - Induktionsschritt:

$$T(n) = 2T(n/2) + c \leq 2(kn/2 - c) + c \leq kn - c$$

  - In der Regel werden wir unserer Intuition vertrauen, formal ist der Beweis aber notwendig.

Hilfsmittel: Variablentransformation

- Transformation der Variablen hilft häufig, einfachere, intuitiv leichter lösbare Gleichungen zu erhalten:
- Bsp:

$$T(n) = 2T\left(\left\lfloor \sqrt{n} \right\rfloor\right) + \log n$$

- Ersetze m = log n

$$T(n) = 2T\left(\sqrt{2^{\log n}}\right) + \log n$$

$$T(2^m) = 2T\left(2^{m/2}\right) + m$$

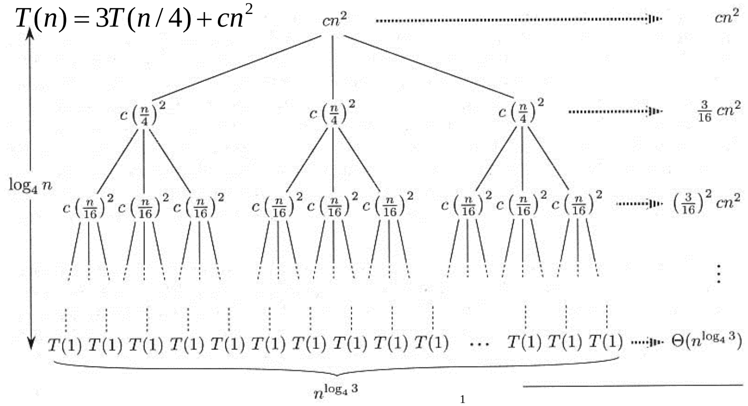
- Setze S(m) = T(2^m)

$$S(m) = 2S(m/2) + m = O(m \log m)$$

$$T(n) = O(\log n \log\log n)$$

Hilfsmittel: Rekursionsbaum

- **Was tun, wenn die Intuition fehlt?**
  - Aufbau des Rekursionsbaums
  - Abschätzung der Höhe, der Knoten pro Ebene, der Kosten pro Knoten
- **Bsp:** $T(n) = 3T(n/4) + cn^2$



Hinweis: $3^{\log_4 n} = 3^{\frac{\log_3 n}{\log_3 4}} = \left(3^{\log_3 n}\right)^{\frac{1}{\log_3 4}} = n^{\frac{1}{\log_3 4}} = n^{\frac{1}{\left(\frac{\log_4 4}{\log_4 3}\right)}} = n^{\log_4 3}$

Hilfsmittel: Rekursionsbaum

- **Was tun, wenn die Intuition fehlt?**
  - Aufbau des Rekursionsbaums
  - Abschätzung der Höhe, der Knoten pro Ebene, der Kosten pro Knoten
- **Bsp:** $T(n) = 3T(n/4) + cn^2$

$$= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$\le \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$\boxed{\text{Unendl. Geom. / Exp. Reihe:} \quad \sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad \text{für } x \in \mathrm{IR}, |x| < 1}$$

$$= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3})$$

$$= O\left(n^2\right)$$

Das Master-Theorem

- **Auflösung von Rekurrenzen der Form**

$$T(n) = \begin{cases} c & : n = 1 \\ aT(n/b) + f(n) & : n > 1 \end{cases}$$

mit a ≥ 1 und b > 1, a und b konstant

- **Algorithmische Bedeutung:**
  - Ein Problem wird in a Teilprobleme zerlegt
  - Jedes Teilproblem hat die Größe n/b (genauer ⌈n/b⌉)
  - Zum Aufteilen des Problems und zum Zusammenfügen benötigt man f(n) Zeit.

Fall 1.	$f(n) = O(n^{\log_b a - \varepsilon}), \varepsilon > 0$	:	$T(n) = \Theta(n^{\log_b a})$
Fall 2.	$f(n) = \Theta(n^{\log_b a})$	:	$T(n) = \Theta(n^{\log_b a} \log_2 n)$
Fall 3.	$f(n) = \Omega(n^{\log_b a + \varepsilon}), \varepsilon > 0$ und $af(n/b) \le cf(n), c < 1$	:	$T(n) = \Theta(f(n))$

Das Master-Theorem

- **Bsp:**

$$T(n) = \begin{cases} c & : n = 1 \\ 2T(n/2) + c & : n > 1 \end{cases}$$

$$a = 2, b = 2, \; f(n) = c$$

- **Welcher Fall des Master Theorems?**

$$\log_b a = 1 \text{ für } a = b = 2 \text{ und } \; f(n) = c = O(n^{1-\varepsilon})$$

- **Die Bedingungen für Fall 1 sind erfüllt, also folgt**

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n)$$

1.6 Ein Beispiel: Das Maximum-Subarray-Problem

[aus Ottmann, Widmeyer, Algorithmen und Datenstrukturen, Spektrum Verlag, 2002]

- **Problem (maximum-subarray):**
  - geg.: eine Folge X von N ganzen Zahlen
  - ges.: Teilfolge, deren Summe maximal ist (max. Teilsumme)
- **Bsp.:**
  - N=10, X: 3, -4, 5, 2, -5, 6, 9, -9, -2, 8 (d.h. X[1]=3, X[2]=-4, …)
  - Teilsumme von X[1..4] : 3-4+5+2 = 6
  - max. Teilsumme X[3..7] : 5+2-5+6+9 = 17
- **Algorithmus 1: Probiere alle Möglichkeiten**
  - u: untere Grenze der Teilsumme
  - o: obere Grenze der Teilsumme
  - tsum: aktuelle Teilsumme
  - maxtsum: maximale Teilsumme

Das Maximum-Subarray-Problem

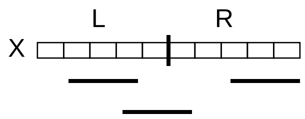
- **Algorithmus 1: Probiere**

```
TEILSUMME(X)                             Zeitbedarf
  maxtsum = 0                            c
  for u = 1 to X.length                  N
    for o = u to X.length                N-u+1  ≤ N
      tsum = 0                     c
      for i = u to o                     o-u+1  ≤ N
        tsum = tsum+X[i]                 c
      maxtsum = max( maxtsum, tsum) c
  return maxtsum                         c
```

- Laufzeit: Sei N = X.length die Anzahl der Array-Elemente.
  Offensichtlich gilt $T_p(N) = O(N^3)$.
  Es gilt sogar $T_p(N) = \Theta(N^3)$ (ohne Beweis).

Das Maximum-Subarray-Problem

- **Verbesserung 1:** *Divide & Conquer Verfahren* (Teile und Herrsche)
- **Idee:**
  - Teile die Folge in linke und rechte Hälfte L, R an der Stelle $\lfloor N/2 \rfloor$
  - Fall 1: max. Teilsumme ist in L
  - Fall 2: max. Teilsumme ist in R
  - Fall 3: max. Teilsumme enthält $X[\lfloor N/2 \rfloor]$ und $X[\lfloor N/2 \rfloor+1]$



- **Teilproblem: maximale Randteilsumme**

```
RTS_links(X,l,r)
  // finde max. Randteilsumme am linken Rand in
  X[l,...,r]
  lmax = 0; sum = 0;
  for i = l to r
    sum = sum + X[i]
    lmax = max( lmax, sum )
  return lmax
```

Laufzeit: $T_{RTS}(N) = O(N)$

Das Maximum-Subarray-Problem

- **Algorithmus 2:** Divide&Conquer (Aufruf Teilsumme(X,1,X.length))

```
TEILSUMME(X,l,r)                                      Laufzeit T_DC(N)
  if l == r
    return max(X[l],0)                                      c
  else// DIVIDE: Teile die Daten
    m = ⌊(l+r)/2⌋                                           c
      // CONQUER: Löse Teilprobleme
Fall 1: maxtsum_Links =Teilsumme(X,l,m)                 T_DC(N/2)
Fall 2: maxtsum_Rechts = Teilsumme(X,m+1,r)             T_DC(N/2)
Fall 3: maxtsum_LRand = RTS_rechts(X,l,m)                   c N
    maxtsum_RRand = RTS_links(X,m+1,r)                      c N
    maxtsum_Mitte = maxtsum_LRand + maxtsum_RRand           c
      // MERGE: Füge Ergebnis zusammen
    maxtsum = max( maxtsum_Links, maxtsum_Rechts,           c
             maxtsum_Mitte)
  return maxtsum
```


Das Maximum-Subarray-Problem

- **Rekurrenz für Divide&Conquer:**

$$T_{DC}(N) = \begin{cases} c & : N = 1 \\ 2T_{DC}(N/2) + cN & : N > 1 \end{cases}$$

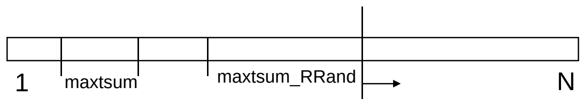
- **Master-Theorem:** a = b = 2; f(N)= c N = O(N)

  $T_{DC}(N) = \Theta(N \log N)$

- **Wie viel Zeit benötigt man mindestens zur Lösung des Problems?**
  - Man muss jedes X[i] mindestens ein mal betrachten
  - $T_{opt}(N) = \Omega(N)$

Das Maximum-Subarray-Problem

- **Verbesserung 2: *Scan-Line Verfahren***
- **Idee:**
  - durchlaufe X von links nach rechts
  - merke dir das bisherige Maximum   maxtsum
  - merke dir die rechte Randteilsumme   maxtsum_RRand



Scan-Line

- Scan-Line am linken Rand: maxtsum = maxtsum_RRand= 0
- Scan-Line geht von i nach i+1:
  - maxtsum_RRand: addiere X[i+1]; setze auf 0 falls negativ
  - maxtsum: Maximum von maxtsum oder maxtsum_RRand
- Scan-Line ist rechts angekommen: maxtsum ist das Ergebnis

Das Maximum-Subarray-Problem

- **Algorithmus 3:** Scan-Line-Verfahren

```
TEILSUMME(X)                                       Laufzeit T_SL(N)
  maxtsum = 0                                              c
  maxtsum_RRand = 0                                        c
  for i=1 to X.length                                      N
    maxtsum_RRand = max(maxtsum_RRand + X[i],0)            c
    maxtsum = max(maxtsum, maxtsum_RRand)                  c
  return maxtsum
```

$T_{SL}(N) = c\,N = \Theta(N)$

- Das Scan-Line Verfahren löst das Maximum-Subarray-Problem in optimaler asymptotischer Laufzeit.