

DATENSTRUKTUREN UND ALGORITHMEN

Vorlesung 1

Diana Cristea

Babeş-Bolyai Universität

Fakultät Mathematik und Informatik

Überblick

- Organisatorisches
- Abstrakte Datentypen und Datenstrukturen
- Pseudocode
- Algorithmenanalyse

Struktur

- Vorlesung: jede Woche (2 Std)
- Seminar: jede **zweite** Woche (2 Std)
- Labor: jede **zweite** Woche (2 Std)

- Dozent: Diana Cristea
- E-mail: [dianat \[at\] cs.ubbcluj.ro](mailto:dianat@cs.ubbcluj.ro)
- Bitte benutze eure *scs.ubbcluj.ro* oder *ubbcluj.ro* E-mail Adresse
- Website: www.cs.ubbcluj.ro/~dianat/
- Für Fragen über Seminar und Labor:
 - Victor Şolea, Kontakt: [vsolea\[at\]cs.ubbcluj.ro](mailto:vsolea@cs.ubbcluj.ro)
- Fragen und Feedback sind immer erwünscht: per E-Mail oder per persönlichem Gespräch
- Anonymes Feedback möglich: auf meine Website

Kursanforderungen

Kursanforderungen und Regeln:

- <http://www.cs.ubbcluj.ro/~dianat/sda-online.php>
- <http://www.cs.ubbcluj.ro/~dianat/SDA/Laborregeln.pdf>
- **Aktive Anwesenheit erfordert!**
 - 5 Seminar-Anwesenheiten aus 7 sind verpflichtend
 - 6 Labor-Anwesenheiten aus 7 sind verpflichtend
- **Ohne diese Anwesenheiten dürft ihr die Klausur nicht mitschreiben (sowohl der erste Versuch, als auch der zweite Versuch)**
- **Es gilt auch für Studenten aus dem zweiten oder dritten Jahr**

Noten

- Labor-Mittelnote **LN**
 - **40%** aus der Endnote
 - LN muss ≥ 5 (keine Rundung) sein um die Prüfung mitzuschreiben
- Schriftliche Prüfung:
 - Findet während der Prüfungszeit statt
 - **60%** aus der Endnote
 - Die Note muss ≥ 5 um die Prüfung zu bestehen
- Bem. Noten und Anwesenheit findet ihr online.

Noten

- Zweitversuch (restanta)
 - **Ihr dürft die Prüfung nur mitschreiben, falls ihr die nötigen Anwesenheiten habt**
 - Nur die schriftliche Prüfung kann man wieder schreiben
 - Die Note wird genauso berechnet wie vorher erwähnt
 - Bei dem Zweitversuch darf man die schriftliche Prüfung mitschreiben auch wenn **LN < 5** ist

Folien, Literatur

- Folien und andere Informationen zur Vorlesung, Seminar und Übungen werden zur Verfügung gestellt:
 - www.cs.ubbcluj.ro/~dianat/
 - Ms Teams
 - Moodle
- Literatur:
 - M. Dietzfelbinger, K. Mehlhorn, P. Sanders. Algorithmen und Datenstrukturen – Die Grundwerkzeuge. Springer Verlag, 2014
 - K. Weicker, N. Weicker. Algorithmen und Datenstrukturen. Springer Verlag, 2013
 - ... Bei Bedarf eine E-mail schreiben um mehr Literatur zu verlangen

Website

- Auf der Website www.cs.ubbcluj.ro/~dianat/ findet ihr:
 - Kursanforderungen und Laborregeln
 - Link zu dem Dokument mit Anwesenheit und Noten

Ms Teams

- für die Vorlesung und Seminar Folien (in Class Materials)
- für Labor Aufgaben (in Class Materials)
- Alle online Meetings finden auf Ms Teams statt

Moodle

- Wird für Quiz benutzt:
 - Anwesenheit Quiz
 - Optionale Quiz
- Wird für Labor benutzt:
 - Es kann verlangt werden, dass die Projekte auf Moodle hochgeladen werden
- Wird für die Prüfung benutzt, falls diese online stattfindet
- **Ihr müsst bis am Ende der ersten Woche in dem Moodle DSA IG Kurs enrolled sein!**

Fragen ?

Dann geht's los!

Ziel der Vorlesung

- Verstehen des Begriffs **abstrakte Datentyp** und Erlernen der grundlegenden abstrakten Datentypen
- Erlernen der unterschiedlichen **Datenstrukturen**, die benutzt werden können um diese abstrakten Datentypen zu implementieren, sowie die Komplexitäten deren Operationen

Algorithmen und Datenstrukturen

- Was Sie aus dieser Vorlesung lernen sollten?
 - Grob abschätzen, ob etwas überhaupt möglich ist.
 - Grob abschätzen, wie teuer etwas wird/ werden kann.
 - Lösungsvorschläge und deren Kosten verstehen.
 - Lösungen selbst erarbeiten können, anfangend von abstrakten Datentypen!
 - Daten verarbeiten, die in verschiedenen Datenstrukturen gespeichert sind.
 - Den abstrakten Datentyp und die Datenstruktur auswählen am besten geeignet für eine gegebene Anwendung
 - *Abstrahieren!*

Abstrakte Datentypen

- Was ist ein Datentyp? Kennt ihr Beispiele?
- **Definition.** Ein *Datentyp* ist eine Menge von Werten (z.B. \mathbb{N}) und Operationen darauf (z.B. $+$).
 - Int
 - Menge von Datenobjekten sind die ganze Zahlen aus einem bestimmten Intervall
 - Operationen: Addition, Subtraktion, Multiplikation, usw.
 - Boolean
 - Menge von Datenobjekten: true, false
 - Operationen: and, or, xor, usw.
- Es gibt vordefinierte Datentypen und benutzerdefinierte Datentypen

Abstrakte Datentypen (ADT)

- ADT ist ein Datentyp mit folgenden zwei Eigenschaften:
 - Die Objekte aus der Domäne des ADT werden unabhängig von der Darstellung spezifiziert
 - Die Operationen des ADT werden unabhängig von ihrer Implementierung spezifiziert

Abstrakte Datentypen (ADT) - Domäne

- Die **Domäne** des ADT beschreibt welche Elemente zu diesem ADT gehören
- Falls die Domäne eine endliche Menge ist, dann können alle Elemente aufgezählt werden.
- Falls die Domäne unendlich ist, dann benutzt man eine Regel um die dazugehörige Elemente zu beschreiben.

Abstrakte Datentypen (ADT) – Interface/Schnittstelle

- Nachdem man die Domäne bestimmt hat, muss man auch die Operationen bestimmen
- Die Menge aller Operationen für einen ADT nennt man ***Interface***
- Das Interface eines ADT besteht aus den ***Methodensignaturen*** zusammen mit den Eingabe- und Ausgabeparametern, Vor- und Nachbedingungen
- D.h. die **Semantik** der Operationen wird beschrieben, aber ohne Details über die Implementierung (also **was** die Operation tut und **nicht wie**)

Abstrakte Datentypen (ADT) - Beispiel

- Sei ein ADT Date, der drei Zahlen enthält: Jahr, Monat und Tag
- Die Elemente, die zu ADT Date gehören, sind gültige Daten
- Eine mögliche Operationen für ADT Date wäre:
 difference(Date d1, Date d2)
 - **Beschreibung:** gibt die Differenz in Tagen zwischen den zwei Daten zurück
 - **Pre:** d1 und d2 sind gültige Daten, $d1 \leq d2$
 - **Post:** difference \leftarrow eine natürliche Zahl welche die Anzahl der Tagen zwischen d1 und d2 darstellt
 - **Throws:** ein Exception falls $d1 > d2$
- Die Spezifikation der Operatin sagt uns schon wie diese funktioniert, wir müssen nichts über die Implementierung kennen um ADT Date zu benutzen.

Behälter/Container ADT

- Ein Container ist eine Sammlung von Daten, in der man neue Daten einfügen kann und von der man Daten löschen kann
- Ein Container ist die Implementierung eines ADTs
- Unterschiedliche Containers basieren sich auf unterschiedliche Eigenschaften:
 - Sind die Elemente eindeutig?
 - Haben die Elemente einen entsprechenden Index?
 - Kann jeder Element zugegriffen werden?
 - Sind die Elemente einfach oder zusammengesetzt (z.B. Schlüssel-Wert Paare)?

Behälter/Container ADT

- Ein Container muss wenigstens folgende Operationen implementieren:
 - Einen leeren Container erstellen
 - Ein neues Element zu dem Container einfügen
 - Ein Element aus dem Container löschen
 - Die Anzahl der Elemente aus dem Container zurückgeben
 - Zugriff zu den Elementen zu geben (meistens mithilfe eines Iterators)

Container vs. Collection

- Python – Collections
- C++ - Containers aus STL
- Java – Collections framework
- .Net – System.Collections framework

- Wir werden den Begriff **Container** benutzen

Behälter/Container ADT

- Habt ihr schon irgendwelche Containers benutzt (z.B. in Python)?
 - *list* und *dict*
- Wissen Sie, wie diese Containers repräsentiert werden?
- Wissen Sie, wie die Operationen implementiert sind?
- Welche sind die Eigenschaften für *list* und *dict*?
 - Sind die Elemente eindeutig?
 - Haben die Elemente einen entsprechenden Index?
 - Kann jeder Element zugegriffen werden?
 - Sind die Elemente einfach oder zusammengesetzt (z.B. Schlüssel-Wert Paare)?

Wofür braucht man Abstrahierung?

- Man braucht die Spezifikation der Operationen ohne erstmal an Implementierungsdetails zu denken
- Es gibt mehrere abstrakte Datentypen, also man muss den besten für die Implementierung auswählen (ein wichtiger Schritt in dem Design der Anwendung)
- Die meisten High-Level Programmiersprachen haben Implementierungen für unterschiedliche abstrakte Datentypen (STL Library in C++, Collections in Java, containers und collections in Python, System.Collection in .Net, usw.)
- Um diese abstrakte Datentypen zu benutzen braucht man die Domäne und das Interface zu kennen.

Beispiel

- Sie müssen eine Anwendung implementieren, die mit römischen Zahlen arbeitet (z.B. um CXXI in 121 umzuwandeln)
- Vielleicht müsst ihr auch neue römische Zahlen definieren (z.B. W für die Zahl 200)
- In der Anwendung muss man oft die Zahl finden, die einer Buchstabe entspricht, und auch umgekehrt.
- Was für einen Container würden Sie verwenden?

Vorteile ADTs zu benutzen

- **Abstrahierung** – die Spezifikation eines Objektes (Domäne und Interface) und die Implementierung sind **getrennt**
- **Datenkapselung** – die Implementierungsdetails sind „verborgen“, d.h. die Anwendung, welche den Datentyp benutzt, hat Zugriff nur zu dem Interface und nicht zu der Implementierung des ADTs
- Die Abstrahierung sichert zu, dass jede Implementierung des ADT zu seiner Domäne gehört und dem Interface entspricht

Vorteile der Abstrahierung und der Datenkapselung

- Ein Teil des Codes, der den ADT benutzt, ist immer noch gültig, wenn sich irgendwas in der Implementierung des ADTs ändert
- **Flexibilität** – ein ADT kann unterschiedliche Implementierungen haben. Man kann mit wenigen Code-Änderungen die Implementierung wechseln.
- Man kann Teile des Codes wieder verwenden
- Die Testphase wird vereinfacht: man muss Tests implementieren für die Operationen aus dem Interface; diese müssen für jedwelle Implementierung funktionieren

Vorteile der Abstrahierung und der Datenkapselung

- Betrachten Sie wieder den Beispiel mit ADT Date
- Nehmen wir an, dass man für ADT Date schon viele Operationen implementiert hat und dass man auch ein ADT TimeInterval implementiert hat, welcher mithilfe von Date repräsentiert wird.
- Man entscheidet jetzt die Repräsentierung für Date zu ändern und anstatt drei Zahlen (für Jahr, Monat und Tag) speichert man eine einzige Zahl, wobei die ersten 4 Ziffern den Jahr darstellen, die nächsten 2 Ziffern den Monat und die letzten 2 Ziffern den Tag.
- Welche Teile aus der Implementierung muss man jetzt ändern?

Begriff: Datentyp & Datenstruktur

- **Definition.** Ein *Datentyp* ist eine Menge von Werten (z.B. \mathbb{N}) und Operationen darauf (z.B. +).
- **Definition.** Bei einer *Datenstruktur* sind die Daten zusätzlich in bestimmter Weise angeordnet und in bestimmter Weise wird Zugriff und Verwaltung ermöglicht. (Beispiele: Array, Liste, Stack, Graph)

Datenstrukturen

- Eine Datenstruktur kann:
 - Statisch sein: hat eine feste Größe zugewiesen
Kann benutzt werden, wenn man von Anfang an weiß, wie viele Elemente es gibt.
 - Dynamisch sein: die Größe kann sich während der Laufzeit verändern
- Eine logische Datenstruktur kann man sowohl statisch als auch dynamisch implementieren

Datenstrukturen

- Für jeden ADT werden wir mehrere mögliche Datenstrukturen für die Implementierung besprechen (zusammen mit den Vorteilen und Nachteilen).
- Meistens gibt es keine *beste* Datenstruktur.

Wofür müssen wir selber ADTs implementieren?

- Die Implementierung von ADTs hilft uns besser zu verstehen, wie diese funktionieren
 - Welche ist die Komplexität in dem folgenden Beispiel?

```
if elem in cont:  
    print("Found")
```

- Es hängt davon ab, was für ein ADT *cont* ist!

Wofür müssen wir selber ADTs implementieren?

- Wenn man einen ADT benutzt, dann muss man verstehen welche Operationen dieser hat und warum wir den Container benutzen wollen
- Manchmal ist das sowieso nötig, in Fälle wie:
 - Die Programmiersprache hat keine Implementierung für den ADT
 - Wir brauchen eine ADT, der einem normalen ADT ähnelt, aber kleine Unterschiede hat

Begriff: Algorithmus

- Ein *Algorithmus* ist ein endlich und präzise beschriebenes Verfahren, das Eingabewerte in Ausgabewerte umwandelt.
- Ein Algorithmus löst ein Problem.
- Es ist **deterministisch** und der Ausgang ist **determiniert**:
 - Ein deterministischer Ablauf bedeutet, dass der Algorithmus eine eindeutige Vorgabe der Schrittfolge der auszuführenden Schritte festlegt
 - Ein determiniertes Ergebnis – ein eindeutiges Ergebnis wird geliefert auch bei mehrfacher Durchführung des Algorithmus (mit denselben Eingabeparametern)
- Die einzelnen Schritte sind zudem elementar/atomar und effektiv ausführbar.
- Meist wird noch die *Termination* sowie die *Korrektheit* des Verfahrens verlangt.

Kriterien für Algorithmen

- Korrektheit (benötigt Beweise)
- Effizienz: Zeit- und Speicherbedarf
 - Analyse basiert nicht auf empirischen Untersuchungen, sondern auf mathematischen Analysen
 - Nutzen hierfür *Pseudocode* und *Basisoperationen*
 - Für die **Komplexität** gibt es:
 - **Statische Analyse:** bezieht sich auf den Quellcode
 - Betrachtung des Codes ohne ihn laufen zu lassen
 - **Dynamische Analyse:** Analyse der Eigenschaften eines laufenden Programms
 - man beobachtet die Ausführung des Programms durch Messen der Laufzeit und Speicherverbrauch

Begriffe

- **Sequentielle Ausführung:** das Hintereinanderausführen von Schritten
- **Parallele Ausführung:** die parallele/gleichzeitige Ausführung von Schritten (auf mehrere Prozessoren)
- **Bedingte Ausführung:** ein Schritt, der nur ausgeführt wird, wenn eine bestimmte Bedingung erfüllt wird
- **Schleife:** die Wiederholung einer Tätigkeit, bis eine vorgegebene Bedingung erfüllt wird
- **Unterprogramm:** eine Bearbeitungsvorschrift, die aufgerufen wird, um dann ausgeführt zu werden
- **Rekursion:** die Anwendung desselben Prinzips auf in gewisser Weise *kleinere* oder *einfachere* Teilprobleme, bis diese Teilprobleme so klein sind, dass sie direkt gelöst werden können

Pseudocode-Notationen für Algorithmen

- Wir benutzen Pseudocode, da wir in dieser Vorlesung die Datenstrukturen allgemein beschreiben wollen (unabhängig von einer Programmiersprache)
- Es gibt zwei Arten von Anweisungen:
 - Standard Anweisungen (Schleifen, Zuweisung, usw.)
 - Non-standard Anweisungen: man beschreibt in Wörtern ein Teil des Algorithmus, das noch nicht implementiert wurde
 - Non-standard Anweisungen fangen mit @ an

Pseudocode-Notationen für Algorithmen

- Schleifen: **for, while, repeat**
- Bedingtes Verzweigen: **if – then – else**
- Zuweisung: \leftarrow
- Kommentar: **//**
- Blockstruktur durch Einrückung
- Daten lesen: **read**
- Daten auszugeben: **print**
- Zwei Variablen zu vergleichen: **=**

Pseudocode – Bedingtes Verzweigen

```
if condition then  
    @Anweisungen  
else  
    @Anweisungen  
end-if
```

- Der **else**- Teil kann fehlen

Pseudocode – **for** Schleife

```
for i ← init, final, step execute  
    @Anweisungen  
end-for
```

- Bei der **for** Schleife ist die Anzahl von Schritte bekannt
- *init* – Anfangswert für Variable i
- *final* – Endwert für Variable i
- *step* – der Wert, der zu *i* bei jedem Durchlauf addiert wird (Aktualisierung)
- Falls *step* fehlt, dann ist es 1 (der Wert wird mit 1 inkrementiert)

Pseudocode – **while** Schleife

```
while condition execute  
    @Anweisungen  
end-while
```

- Bei der **while** Schleife ist die Anzahl von Schritte nicht bekannt

Pseudocode – Subalgorithm

- Subalgorithme sind Unterprogramme, die kein Rückgabewert haben

```
subalgorithm name(formal_parameter_list) is:  
    @Anweisungen  
end-subalgorithm
```

- Um den Unterprogramm aufzurufen:
 name(actual_parameter_list)

Pseudocode – Funktionen

- Funktionen sind Unterprogramme, die ein Wert zurückgeben

function name (formal_parameter_list) **is**:
 @Anweisungen
 name \leftarrow v //Syntax, um den Wert v zurückzugeben
end-function

- Um die Funktion aufzurufen:

Result \leftarrow name (actual_parameter_list)

Pseudocode

- Um eine Variable i vom Typ Integer zu definieren schreiben wir:
 $i : Integer$
- Um ein Array a zu definieren, deren Elemente vom Typ T sind, schreiben wir: $a : T[]$
- Wenn die Größe des Arrays bekannt ist, dann benutzen wir: $a : T [Nr]$ (die Indexierung beginnt bei 1 und geht bis Nr)
- Wenn wir die Indexierung genauer angeben wollen, dann schreiben wir: $a : T [Min...Max]$ (die Indexierung beginnt bei Min und geht bis Max)

Pseudocode – Verbund/Struktur

Array:

n: Integer
elems: T[]

- Diese Struktur besteht aus zwei Feldern: *n* vom Typ Integer und ein Array von Elementen, *elems*, vom Typ T
- Mithilfe einer Variable *var* vom Typ Array, kann man auf die Felder mit . (Punkt) zugreifen:
 - var.n
 - var.elems
 - var.elems[i] – der i-te Element des Arrays

Pseudocode – Zeiger/Pointers

- Unter einem Zeiger/Pointer versteht man eine Variable, deren Wert als Speicheradresse einer anderen Variable verwendet wird.
- Man sagt, diese Variable werde durch den Zeiger referenziert, bzw. **der Zeiger sei eine Referenz auf diese Variable**.
- Für Pointers benutzen wir die Notation \uparrow :
 - $p: \uparrow \text{Integer}$ - p ist eine Variable deren Wert die Adresse eines Integers ist
 - Der Wert von der Adresse aus p kann mit $[p]$ zugegriffen werden
- Speicherallokation (Speicherplatz reservieren): ***allocate(p)***
- Speicherplatz wieder freigeben: ***free(p)***
(in vielen Sprachen muss man den belegten Speicher explizit freigeben)
- Wir benutzen den speziellen Wert **NIL** um eine ungültige Adresse zu bezeichnen

Spezifikationen

- Eine Operation wird folgendermaßen spezifiziert:
 - **pre**: die Vorbedingungen (preconditions) der Operation
 - **post**: die Nachbedingungen (postconditions) der Operation
 - **throws**: ausgelöste Ausnahmen (Exceptions thrown) (optional – nicht jede Operation löst Ausnahmen aus)
- Wenn man den Namen eines Parameters in einer Spezifikation benutzt, dann wird der Wert damit gemeint
- Wenn der Parameter i vom Typ T ist, dann bezeichnet man mit $i \in T$ die Bedingung, dass der Wert von i zu der Domäne des Typs T gehört

Spezifikationen

- Der Wert eines Parameters kann im Laufe der Funktion/Subalgorithmus geändert werden. Wir benutzen ' (Apostroph) um den neuen Wert zu bezeichnen.
- Zum Beispiel, für die Dekrementierungsoperation mit dem Parameter $x : \text{Integer}$ haben wir folgende Spezifikation:
 - **pre:** $x \in \text{Integer}$
 - **post:** $x' = x - 1$

Generische Datentypen

- Man nimmt an, dass die Elemente eines ADT vom generischen Datentypen sind: ***TElem***
- Das Interface des TElem besteht aus folgenden Operationen:
 - Zuweisung ($e_1 \leftarrow e_2$)
 - **pre:** $e_1, e_2 \in \text{TElem}$
 - **post:** $e_1' = e_2$
 - Gleichheitstest ($e_1 = e_2$)
 - **pre:** $e_1, e_2 \in \text{TElem}$
 - **post:** $equal = \begin{cases} \text{True}, & \text{if } e_1 = e_2 \\ \text{False}, & \text{ansonsten} \end{cases}$

Generische Datentypen

- Wenn die Werte eines Datentyps verglichen und sortiert werden können, dann benutzen wir den generischen Datentyp ***TComp***
- Zusätzlich zu den Operationen von TElem, hat der generische Datentyp TComp eine Vergleichoperation:
 - `compare(e1, e2)`
 - **pre:** $e_1, e_2 \in TComp$
 - **post:** $compare = \begin{cases} -1, & \text{if } e_1 < e_2 \\ 0, & \text{if } e_1 = e_2 \\ 1, & \text{if } e_1 > e_2 \end{cases}$
- Als eine Vereinfachung rufen wir manchmal nicht die *compare* Funktion auf, sondern wir benutzen direkt die Notationen: $e_1 < e_2, e_1 \leq e_2, e_1 = e_2, e_1 > e_2, e_1 \geq e_2$

Algorithmen Analyse - RAM Modell

- Die Algorithm Analyse heißt meistens die Laufzeit und den Platzbedarf zu bestimmen.
- Ziel: die Beurteilung der Effizienz von Algorithmen soll unabhängig sein von:
 - dem verwendeten Computer
 - der verwendeten Programmiersprache
 - den Fähigkeiten des Programmierers usw.
- Dazu eignet sich ein axiomatisch definiertes Rechnermodell, sowie die verallgemeinerte Registermaschine (random access machine, RAM)

Algorithmen Analyse - RAM Modell

- **Jede elementare Operation** (+, -, *, /, =, if, call) **braucht eine Zeiteinheit** (ein Schritt)
- Die Integer und Floating Point Datentypen haben feste Größe
- Schleifen und Unterprogramme sind nicht elementare Operationen (es gibt keine spezielle Operation z.B. *sorting*)
- Jeder Zugriff zu dem Speicherplatz zählt als eine Zeiteinheit und es gibt unendlicher Speicherplatz

Algorithmen Analyse - RAM Modell

- Der RAM Modell ist viel vereinfacht, aber es ist in der Praxis ein gutes Modell um die Komplexität eines Algorithmus zu verstehen.
- Die abstrakte Laufzeit ergibt sich aus der Anzahl der elementaren Operationen.
- Anzahl der elementaren Operationen bildet die Basis zur Bestimmung der Wachstumsrate der Zeitkomplexität bei immer längeren Eingaben.
- Die Anzahl der Schritte hängt meistens von der Größe der Eingabeparameter ab

Algorithmen Analyse - Beispiel

subalgorithm something(n) is:

//n ist eine Integer Zahl

rez ← 0

for i ← 1, n **execute**

sum ← 0

for j ← 1, n **execute**

sum ← sum + j

end-for

rez ← rez + sum

end-for

print rez

end-subalgorithm

- Die Anzahl der Schritte in dem Subalgorithmus ist:

$$T(n) = 1 + n * (1 + n + 1) + 1 = n^2 + 2n + 2$$

Wachstumsrate

- Für ein Algorithmus ist nicht die genaue Anzahl von Schritte wichtig, sondern die *Wachstumsrate*
- Dafür ist die dominante Größe der Formel wichtig (n^2 in dem Beispiel), da die anderen relativ unbedeutend sind für große n -Werte.
- Beschreibt eine Funktion die Laufzeit eines Programms, so ist es oft ausreichend, nur ihr asymptotisches Verhalten zu untersuchen.
- Konstante Faktoren werden vernachlässigt
- Um asymptotische Aussagen mathematisch konkret zu fassen, definiert man die O -Notation, Ω -Notation und Θ -Notation

O-Notation

Für eine gegebene Funktion $g(n)$, definiert man $O(g(n))$ als die Menge der Funktionen:

$O(g(n)) = \{ f(n) : \text{es gibt eine positive Konstante } c \text{ und } n_0 \text{ s.d.}$

$$0 \leq f(n) \leq c * g(n) \text{ für alle } n \geq n_0 \}$$

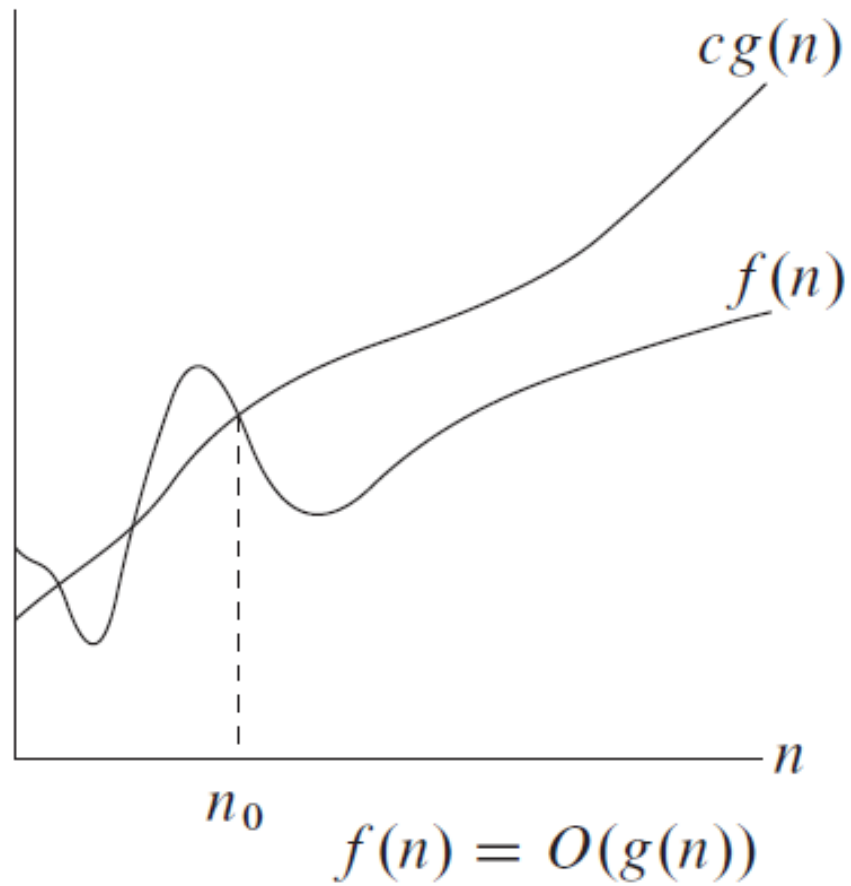
- Die O-Notation ist eine asymptotisch **obere** Schranke (*asymptotic upper bound*) für eine Funktion: für alle n -Werte (größer als n_0) ist der Wert von $f(n)$ kleiner gleich $c * g(n)$
- Anders gesagt, g wächst höchstens so schnell wie f
- Schreibweise: $f(n) \in O(g(n))$ oder $f(n) = O(g(n))$

O-Notation

- Alternative Definition:

$f(n) \in O(g(n))$ wenn $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ entweder 0 oder konstant ist (aber nicht ∞)

- Sei z.B. $T(n) = n^2 + 2n + 2$
 - $T(n) = O(n^2)$ weil $T(n) \leq c * n^2$ für $c = 2$ und $n \geq 3$
 - $T(n) = O(n^3)$ weil $\lim_{n \rightarrow \infty} \frac{T(n)}{n^3} = 0$



Figur aus Corman et. al: Introduction to algorithms, MIT Press, 2009

Ω -Notation

Für eine gegebene Funktion $g(n)$, definiert man $\Omega(g(n))$ als die Menge der Funktionen:

$\Omega(g(n)) = \{ f(n) : \text{es gibt eine positive Konstante } c \text{ und } n_0 \text{ s.d.}$

$$0 \leq c * g(n) \leq f(n) \text{ für alle } n \geq n_0 \}$$

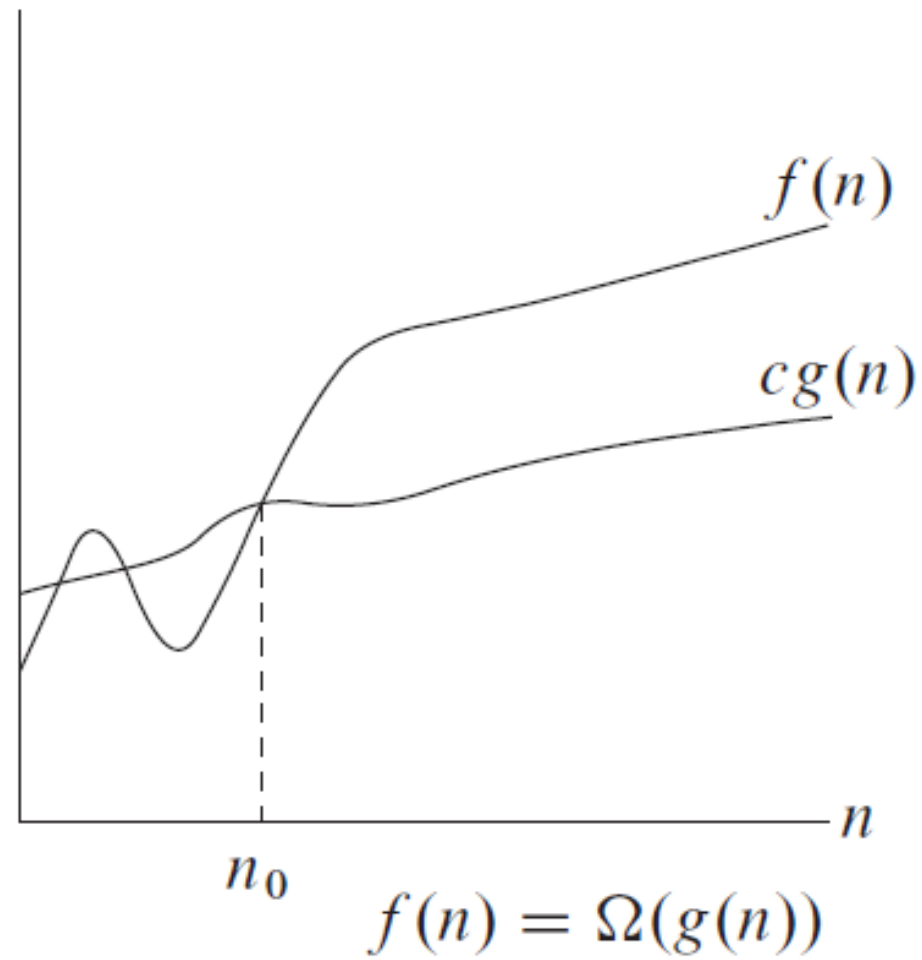
- Die Ω -Notation ist eine asymptotisch **untere Schranke** (*asymptotic lower bound*) für eine Funktion: für alle n -Werte (größer als n_0) ist der Wert von $f(n)$ größer gleich $c * g(n)$
- Anders gesagt, g wächst mindestens so schnell wie f
- Schreibweise: $f(n) \in \Omega(g(n))$ oder $f(n) = \Omega(g(n))$

Ω -Notation

- Alternative Definition:

$f(n) \in \Omega(g(n))$ wenn $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ entweder ∞ oder konstant, aber nicht Null, ist

- Sei z.B. $T(n) = n^2 + 2n + 2$
 - $T(n) = \Omega(n^2)$ weil $T(n) \geq c * n^2$ für $c = 0.5$ und $n \geq 1$
 - $T(n) = \Omega(n)$ weil $\lim_{n \rightarrow \infty} \frac{T(n)}{n} = \infty$



Figur aus Corman et. al: Introduction to algorithms, MIT Press, 2009

Θ -Notation

Für eine gegebene Funktion $g(n)$, dann definiert man $\Theta(g(n))$ als die Menge der Funktionen:

$\Theta(g(n)) = \{ f(n) : \text{es gibt positive Konstanten } c_1, c_2 \text{ und } n_0 \text{ s.d.} \}$

$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ für alle } n \geq n_0 \}$$

- Die Θ -Notation ist die **Wachstumsrate** (*asymptotically tight bound*) einer Funktion: für alle n -Werte (größer als n_0) ist der Wert von $f(n)$ zwischen $c_1 * g(n)$ und $c_2 * g(n)$
- Anders gesagt, g wächst wie f
- Schreibweise: $f(n) \in \Theta(g(n))$ oder $f(n) = \Theta(g(n))$

Θ -Notation

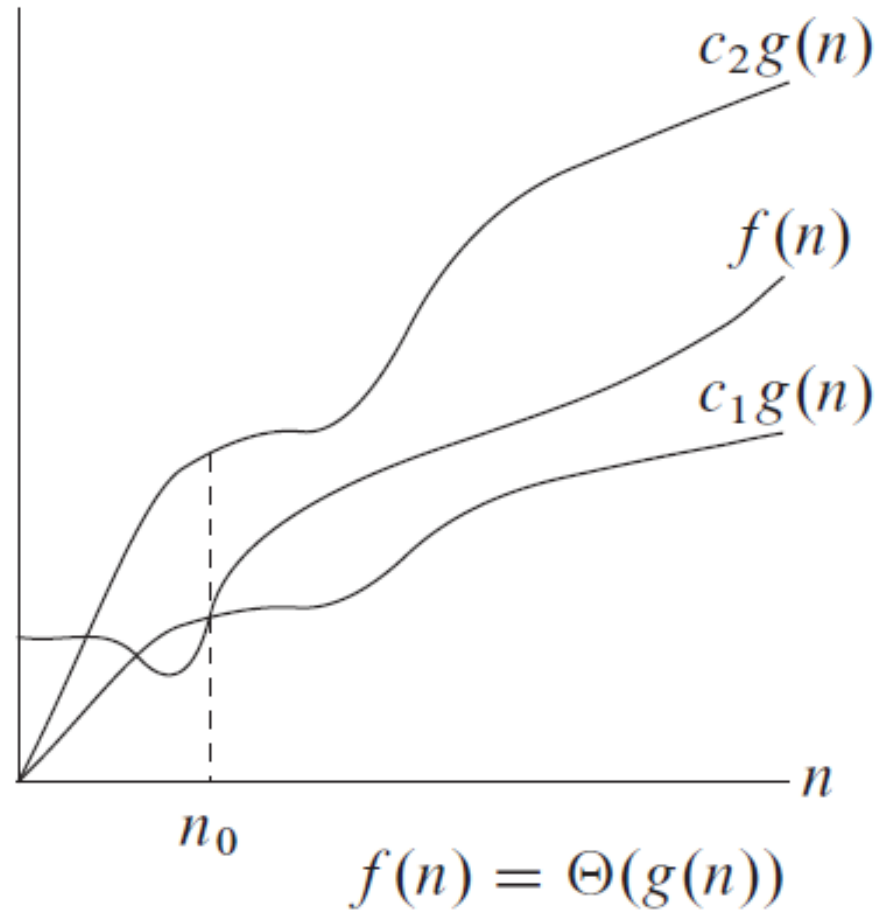
- Alternative Definition:

$f(n) \in \Theta(g(n))$ wenn $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ eine Konstante unterschiedlich von 0 ist (und nicht ∞)

- Sei z.B. $T(n) = n^2 + 2n + 2$

- $T(n) = \Theta(n^2)$ weil $c_1 * n^2 \leq T(n) \leq c_2 * n^2$ für $c_1 = 0.5$, $c_2 = 2$ und $n \geq 3$

- $T(n) = \Theta(n^2)$ weil $\lim_{n \rightarrow \infty} \frac{T(n)}{n^2} = 1$



Figur aus Corman et. al: Introduction to algorithms, MIT Press, 2009

Bester Fall, Schlimmster Fall, Durchschnittlicher Fall

- Aufgabe: schreibe einen Algorithmus, der die Summe aller geraden Zahlen aus einem Array berechnet.
- Wie viele Schritte braucht der Algorithmus für ein Array von Länge n ?

Bester Fall, Schlimmster Fall, Durchschnittlicher Fall

- Aufgabe: schreibe einen Algorithmus, der die erste gerade Zahl in einem Array findet.
- Wie viele Schritte braucht der Algorithmus für ein Array von Länge n ?
- In diesem Fall hängt die Anzahl der Schritte nicht nur vom Array-Länge ab, sondern auch von den genauen Werten aus dem Array
 - Der Algorithmus hört auf, wenn er eine gerade Zahl findet, diese kann die erste Zahl sein, oder die zweite, usw.

Bester Fall, Schlimmster Fall, Durchschnittlicher Fall

- Für solche Algorithmen betrachten wir drei Fälle:
 - Der beste Fall – die minimale Anzahl von Schritten
 - Der schlimmste Fall – die maximale Anzahl von Schritten
 - Der durchschnittliche Fall – der Durchschnitt aller möglichen Fälle
- Die Komplexitäten für den besten und schlimmsten Fall werden durch die Code-Analyse berechnet
- Für unser Beispiel:
 - Bester Fall: $\Theta(1)$ – nur die erste Zahl wird überprüft (egal welche Länge der Array hat)
 - Schlimmster Fall - $\Theta(n)$ – man muss alle Zahlen überprüfen

Durchschnittliche Fall

- Um die Komplexität für den durchschnittlichen Fall zu berechnen benutzen wir die Formel:

$$\sum_{I \in D} P(I) * E(I)$$

wobei:

- D – Domäne des Problems, die Menge aller möglichen Eingabeparametern für das Problem
- I – ein bestimmter Eingabeparameter (oder Eingabeparameter-Menge)
- P(I) – die Wahrscheinlichkeit, dass I Eingabeparameter ist
- E(I) – die Anzahl der Operationen für den Eingabeparameter I

Durchschnittliche Fall

- Für unser Beispiel:
 - D – die Menge aller Arrays von Länge n
 - Für I gibt es mehrere Möglichkeiten:
 - Die erste Zahl ist gerade
 - Die zweite Zahl ist gerade, und die erste ungerade
 - ...
 - Alle Zahlen sind ungerade
 - P(I) ist meistens gleich für alle I, in diesem Fall $\frac{1}{n+1}$

$$T(n) = \frac{1}{n+1} (1 + 2 + \dots + n + n) = \frac{n(n+1)}{2(n+1)} + \frac{n}{n+1} \in \Theta(n)$$

Bester Fall, Schlimmsten Fall, Durchschnittliche Fall

- Nachdem wir die Komplexität für alle drei Fälle berechnet haben, wählt man als Komplexität des Algorithmus die höchste Komplexität (aus dem schlimmsten Fall) und wenn die drei Komplexitäten unterschiedlich sind, dann gibt man die allgemeine Komplexität mit der O-Notation
- Für unser Beispiel:
 - Bester Fall: $\Theta(1)$
 - Schlimmster Fall: $\Theta(n)$
 - Durchschnittlicher Fall: $\Theta(n)$
 - Allgemeine Komplexität: $O(n)$

Komplexitätsklassen

- Um Algorithmen bezüglich ihrer Komplexität vergleichen zu können, teilt man diese in Komplexitätsklassen ein:

Klasse	Bezeichnung	Beispiel
1	konstant	elementarer Befehl
$\log n$	logarithmisch	binäre Suche
n	linear	lineare Suche
n^2	quadratisch	einfache Sortiervverfahren
2^n	exponentiell	Backtracking
$n!$	Fakultät	Zahl der Permutationen (Traveling-Salesman-Problem)

- Bei den logarithmischen Komplexitätsklassen spielt die Basis in der Regel keine Rolle, da ein Basiswechsel einer Multiplikation mit einem konstanten Faktor entspricht.

Komplexitäten - Wachstumsverhalten

- Die Komplexität ist **wichtig!**

Alter des Universums

1000 Jahre

1 Jahr

1 Tag

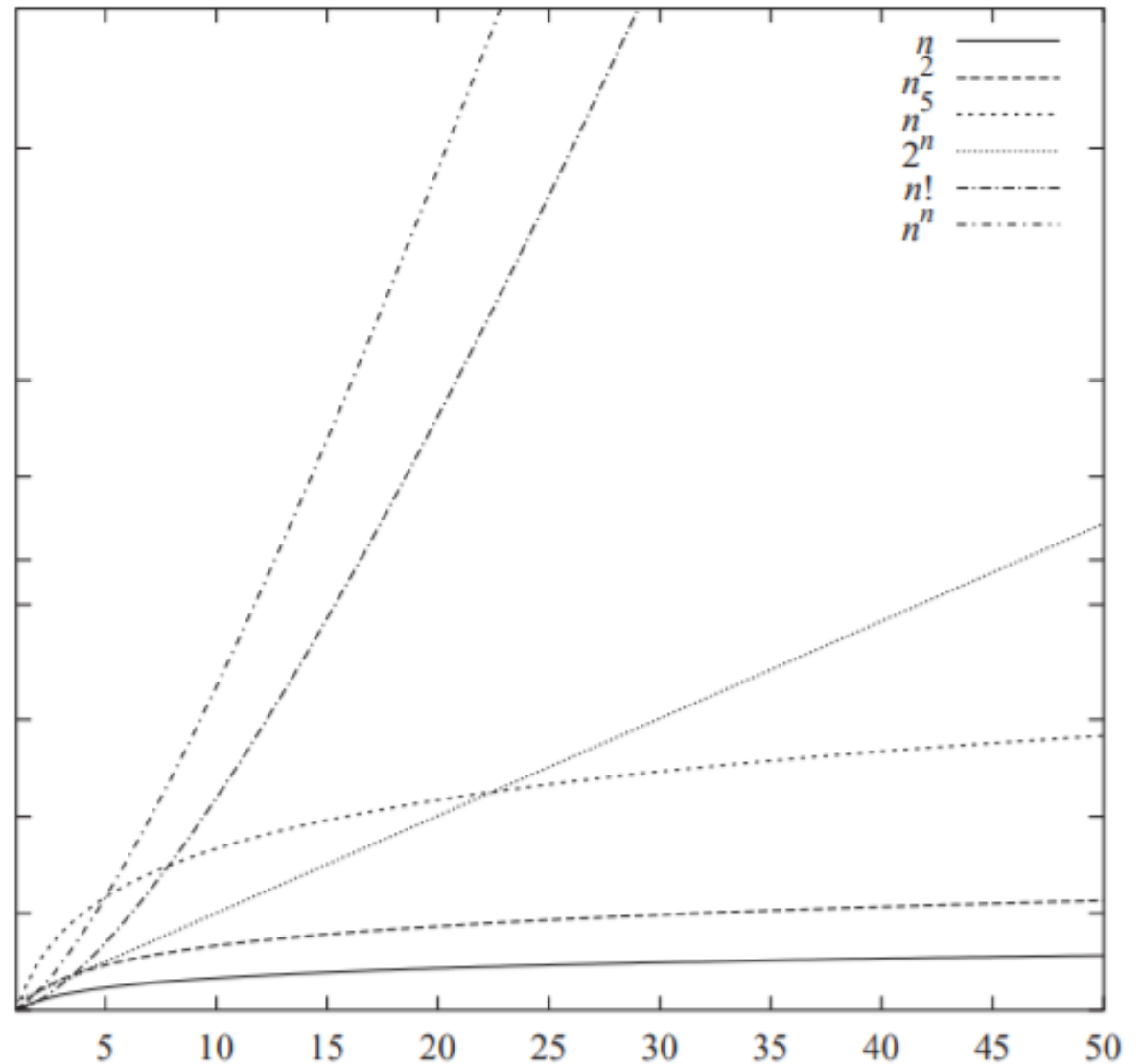
1h

1 s

1 ms

1 μ s

1 ns



Figur aus Prof. Dr.-Ing. H. Ney:
Algorithmen und Datenstrukturen,
Skript, 2004

Algorithm Analyse für rekursive Funktionen - Beispiel

```
function BinsarySearchR (array, elem, start, end) is:  
  //array – eine sortierte Array, die ganze Zahlen enthält  
  //elem – der gesuchte Element  
  //start – Anfang des Intervalls wo man suchen muss (inklusive)  
  //end – Ende des Intervalls wo man suchen muss (inklusive)  
  if start > end then  
    BinarySearchR ← False  
  end-if  
  middle ← (start + end) / 2  
  if array[middle] = elem then  
    BinarySearchR ← True  
  else if elem < array[middle] then  
    BinarySearchR ← BinarySearchR( array, elem, start, middle-1)  
  else  
    BinarySearchR ← BinarySearchR(array, elem, middle+1, end)  
  end-if  
end-function
```

Algorithm Analyse für rekursive Funktionen

- Aufruf der Funktion für ein Array mit nr Elemente:

BinarySearchR(array, elem, 1, nr)

- Wir notieren mit n die Länge der Sequenz, in dem man bei einem Durchlauf sucht (d.h. $n = \text{end} - \text{start}$)
- Man muss die rekursive Formel schreiben um die Komplexität zu berechnen:
- $$T(n) = \begin{cases} 1, & \text{wenn } n \leq 1 \\ T\left(\frac{n}{2}\right) + 1, & \text{ansonsten} \end{cases}$$

Algorithm Analyse für rekursive Funktionen

- Nehmen wir an, dass $n = 2^k$
- $T(2^k) = T(2^{k-1}) + 1 =$
 $= T(2^{k-2}) + 1 + 1 = T(2^{k-2}) + 2 =$
 $\dots = T(2^0) + k =$
 $= k + 1$

Aber wenn $n = 2^k$, dann ist $k = \log_2 n$.

D.h. $T(n) = 1 + \log_2 n \in \Theta(\log_2 n)$ (logarithmisch)

Algorithm Analyse für rekursive Funktionen - Beispiel 2

Subalgorithm operation(n, i) **is:**

//n und i sind ganze Zahlen, $n > 0$

if $n > 1$ **then**

$i \leftarrow 2 * i$

$m \leftarrow n/2$

operation(m, i-2)

operation(m, i-1)

operation(m, i+2)

operation(m, i+1)

else

write i

end-if

end-subalgorithm

Algorithm Analyse für rekursive Funktionen - Beispiel 2

- Rekursive Formel für Algorithm Analyse:

$$T(n) = \begin{cases} 1, & \text{wenn } n \leq 1 \\ 4 * T\left(\frac{n}{2}\right) + 1, & \text{ansonsten} \end{cases}$$

- Nehmen wir an, dass $n = 2^k$
- $$\begin{aligned} T(2^k) &= 4 * T(2^{k-1}) + 1 = 4 * (4 * T(2^{k-2}) + 1) + 1 \\ &= 4^2 * T(2^{k-2}) + 4 + 1 = \\ &= 4^3 * T(2^{k-3}) + 4^2 + 4 + 1 = \\ &\dots = 4^k * T(2^0) + 4^{k-1} + 4^{k-2} + \dots + 4 + 1 = \\ &= 4^k + 4^{k-1} + \dots + 4 + 1 \end{aligned}$$

Algorithm Analyse für rekursive Funktionen - Beispiel 2

$$\sum_{i=0}^n p^i = \frac{p^{n+1} - 1}{p - 1}$$

$$T(2^k) = \frac{4^{k+1} - 1}{4 - 1} = \frac{4^k * 4 - 1}{3} = \frac{(2^k)^2 * 4 - 1}{3}$$

Da $n = 2^k$, gilt:

$$T(n) = \frac{4 * n^2 - 1}{3} \in \Theta(n^2)$$

Algorithm Analyse für rekursive Funktionen – Master-Theorem

- Bei der Analyse von rekursiven Algorithmen, treten meist Rekursionsgleichungen der folgenden Form auf:

$$T(n) = \begin{cases} 1, & \text{wenn } n = 1 \text{ (oder } n \leq 1) \\ a * T\left(\frac{n}{b}\right) + f(n), & \text{ansonsten} \end{cases}$$

Mit Konstanten $a \geq 1, b > 1$ und eine positive Funktion $f(n)$