DATENSTRUKTUREN UND ALGORITHMEN
Vorlesung 1

Babeș-Bolyai Universität

Fakultät für Mathematik und Informatik

Überblick

- Organisatorisches
- Abstrakte Datentypen und Datenstrukturen
- Pseudocode
- Algorithmenanalyse

Victor Șolea

Kontakt: victor.solea[at]ubbcluj.ro oder MS Teams

Auf Microsoft Teams **code 6z9lo2r:**

- Kursanforderungen und Regeln
- Unterlagen, Folien
- Laboraufgaben
- Anwesenheiten und Noten
- Mitteilungen

Bitte regelmäßig überprüfen!

Fragen und Feedback sind immer erwünscht: per E-Mail, Teams oder persönlichem Gespräch

Struktur

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

BEIM SEMINAR UND LABOR IMMER MIT DER EIGENEN GRUPPE KOMMEN!

Kursanforderungen

- Aktive Anwesenheit erwünscht!
  - **5** Seminar-Anwesenheiten aus 7 sind verpflichtend laut Studienordnung
  - **6** Labor-Anwesenheiten aus 7 sind verpflichtend laut Studienordnung
- **Ohne diese Anwesenheiten dürft ihr die Klausur nicht mitschreiben. Dies gilt auch für Studierende aus vorherigen Jahrgängen**
- Motivierte Abwesenheiten beim Seminar / Labor (Krankheit, dringender Familiennotfall o.ä.): Zeugnis spätestens in der nächsten Stunde bringen. Später wird es nicht mehr angenommen!

Noten

- Labormittelnote – **30%** aus der Endnote
  - Note muss ≥ 5 (ohne Rundung) sein um in die Erstprüfung eingelassen zu werden
  - Studierende aus vorigen Jahrgängen mit Labormittelnote ≥5 können bei Verlangen ihre Note anerkennen und müssen dann bei den Laborstunden nicht mehr teilnehmen
- Seminarnote (parțial) – **10%** aus der Endnote
  - Besteht aus einer schriftlichen Arbeit im Seminar 5
  - Note muss nicht unbedingt ≥ 5 sein für das Bestehen der Endprüfung
  - Teilnahme ist nicht verpflichtend
  - Seminarnoten aus den vorigen Jahren werden **NICHT** anerkannt
- Schriftliche Klausur – **60%** aus der Endnote
  - Findet während der Prüfungszeit statt
  - Die Note muss ≥ 5 (ohne Rundung) um die Klausur zu bestehen
- Bonus am Ende für Aktivität, z.B. eine 8.30 kann auf 9 aufgerundet werden

Noten

- Zweitversuch (restanță)
  - **Ihr dürft die Prüfung nur mitschreiben, falls ihr die nötigen Anwesenheiten habt. Dies gilt auch für Studierende aus vorherigen Jahrgängen**
  - Nur die schriftliche Prüfung kann man wieder schreiben (Labormittelnote und Seminarnote bleiben gleich)
  - Die Note wird genauso berechnet wie vorher erwähnt

Rugăminte

- **De programat examenul în prima săptămână de sesiune!**

- **Restanța poate fi în orice zi**

Reorganizare vineri – pentru tot semestrul

- Sapt 1:
  - ora 14: lab cu 713/2
  - ora 16: seminar cu 713
  - ora 18: lab cu 713/1
- Sapt 2:
  - ora 16: lab cu 712/1
  - ora 18: lab cu 712/2

Salile raman aceleasi. Orarul de pe site urmeaza sa se actualizeze

Folien, Literatur

- 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
  - T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms. MIT Press

  - ... Bei Bedarf eine E-mail schreiben um mehr Literatur zu verlangen

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.
  - Auswahl des abstrakten Datentyps und der Datenstruktur, die für eine bestimmte Anwendung am besten geeignet sind
  - *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 ganzen 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 ein 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/Schnittstelle***

- 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 Operation 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 Container basieren sich auf unterschiedliche Eigenschaften:
  - Sind die Elemente eindeutig?
  - Haben die Elemente einen entsprechenden Index?
  - Kann man auf jedes Element zugreifen?
  - 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 man auf jedes Element zugreifen?
  - 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 muss man auch neue "römische Zahlen" definieren (z.B. W für die Zahl 200)
- In der Anwendung muss man oft die Zahl finden, die einem Buchstaben 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**
- **Dateneinkapselung** – die Implementierungsdetails sind „verborgen“, d.h. die Anwendung, welche den Datentyp benutzt, hat Zugriff nur zu der Schnittstelle und nicht zu der Implementierung des ADTs

- Die Abstrahierung sichert dass jede Implementierung des ADTs 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 nur für die Operationen aus der Schnittstelle Teste implementieren; diese müssen für jedwelce Implementierung funktionieren

Vorteile der Abstrahierung und der Datenkapselung

- Betrachten Sie wieder das 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 das 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, aber es gibt eine *passende* für die zu lösende Aufgabe.

Warum 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!

Warum 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 ein 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 es 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: ←
- Kommentar: **//**
- Blockstruktur durch Einrückung
- Daten lesen: **read**
- Daten ausgeben: **print**
- Zwei Variablen 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 Schritten 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 Schritten 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 ← v //Syntax, um den Wert v zurückzugeben
end-function
```

- Um die Funktion aufzurufen:

```
Result ← 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)

Verbund/Struktur (record/struct)

- Ein Verbund oder Struktur ist eine statische Datenstruktur
- Ein Verbund ist eine Menge von Elementen (meistens von unterschiedlichen Typen), die eine logische Einheit bilden
- Die Elemente eines Verbunds werden *Felder* genannt
- Beispiel: ein Verbund, um eine *Person* zu speichern mit den Feldern *Name, Geburtsdatum, Adresse*:

<u>Person:</u>

name: String

gd: String

adresse: String

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 ↑ :
  - p: ↑ 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/Subalgorithms geändert werden. Wir benutzen ’ (Apostroph) um den neuen Wert zu bezeichnen.

- Zum Beispiel, für die Dekrementierungsoperation mit dem Parameter *x : 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:** $e_1, e_2 \in$ TElem
    - **post:** $e_1' = e_2$
  - Gleichheitstest ($e_1 = e_2$)
    - **pre:** $e_1, e_2 \in$ TElem
    - **post:** $equal = \begin{cases} True, & if \ e_1 = e_2 \\ False, & ansonsten \end{cases}$

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(e_1, e_2)
    - **pre:** $e_1, e_2 \in$ TComp
    - **post:** *compare* = $\begin{cases} -1, \ if \ e_1 < e_2 \\ 0, \ if \ e_1 = e_2 \\ 1, \ if \ e_1 > e_2 \end{cases}$
- Als eine Vereinfachung rufen wir manchmal nicht die *compare* Funktion auf, sondern wir benutzen direkt die Notationen: $e_1 < e_2$, $e_1 \leq e_2$, $e_1 = e_2$, $e_1 > e_2$, $e_1 \geq e_2$

Algorithmenanalyse - RAM-Modell

- Die Algorithmennalyse bedeutet meistens die Laufzeit und den Platzbedarf eines Algorithmus 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)

Algorithmenanalyse - 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

Algorithmenanalyse - RAM-Modell

- Der RAM-Modell ist viel vereinfacht, aber es ist in der Praxis ein guter 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 Eingabeparametern ab

Algorithmenanalyse - 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 Schritten wichtig, sondern die *Wachstumsrate*
- Dafür ist die dominante Größe der Formel wichtig (n^2 in dem Beispiel), da alle andere 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)* : es gibt eine positive Konstante *c* und n_0 s.d.**

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

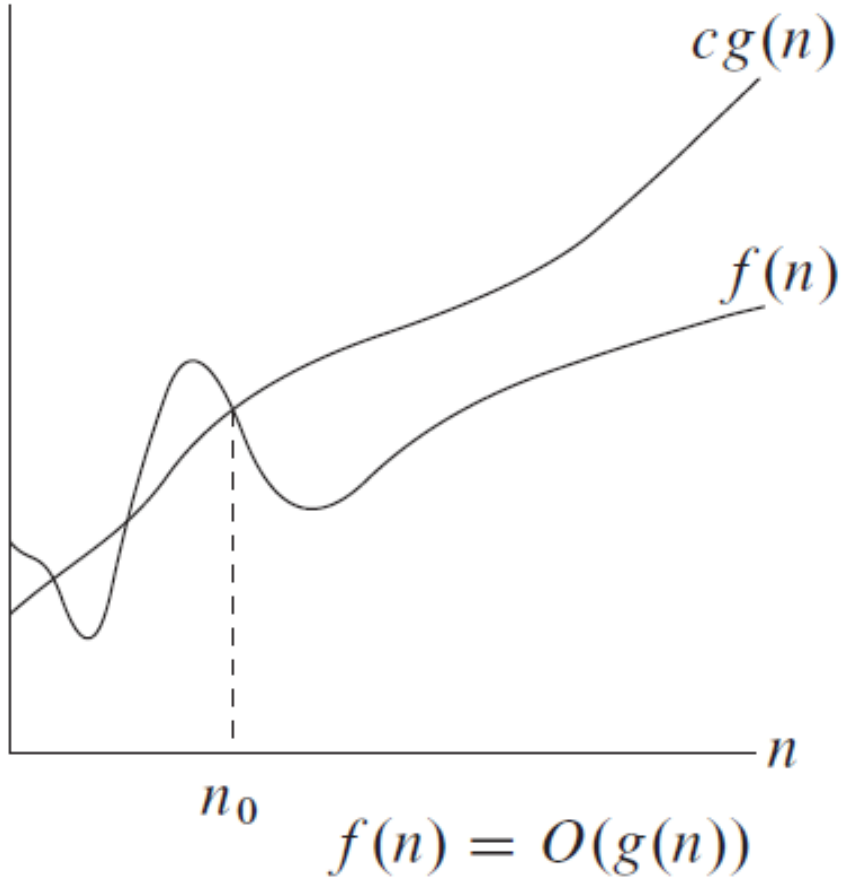
- Die O-Notation ist eine asymptotisch ***obere*** *Schr*anke *(asymptotic upper bound)* für eine Funktion: für alle n-Werte (größer als n_0) ist der Wert von f(n) kleiner gleich c*g(n)
- Anders gesagt, g wächst höchstens so schnell wie f
- Schreibweise: $f(n) \in O(g(n))$ oder $f(n) = O(g(n))$

O-Notation

- Alternative Definition:

  f(n) ∈ O(g(n)) wenn $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ 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 $\lim_{n\to\infty} \frac{T(n)}{n^3} = 0$



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

Ω -Notation

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

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

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

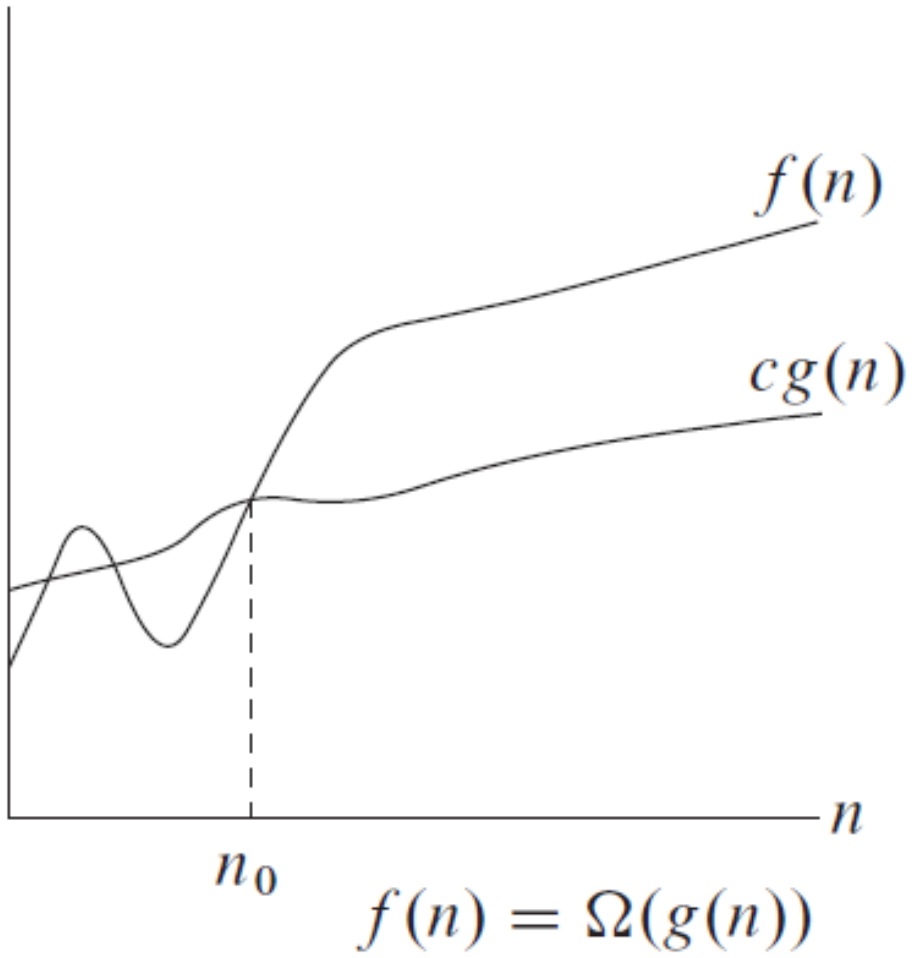
- Die Ω -Notation ist eine asymptotisch ***untere*** *Schranke (asymptotic lower bound)* für eine Funktion: für alle n-Werte (größer als n_0) ist der Wert von f(n) größer gleich c*g(n)
- Anders gesagt, g wächst mindestens so schnell wie f
- Schreibweise: $f(n) \in \Omega(g(n))$ oder $f(n) = \Omega(g(n))$

Ω -Notation

- Alternative Definition:

  f(n) $\in \Omega$(g(n)) wenn $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ 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 ≥ 1
  - T(n) = Ω(n) weil $\lim_{n\to\infty} \frac{T(n)}{n} = \infty$



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

Θ -Notation

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

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

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

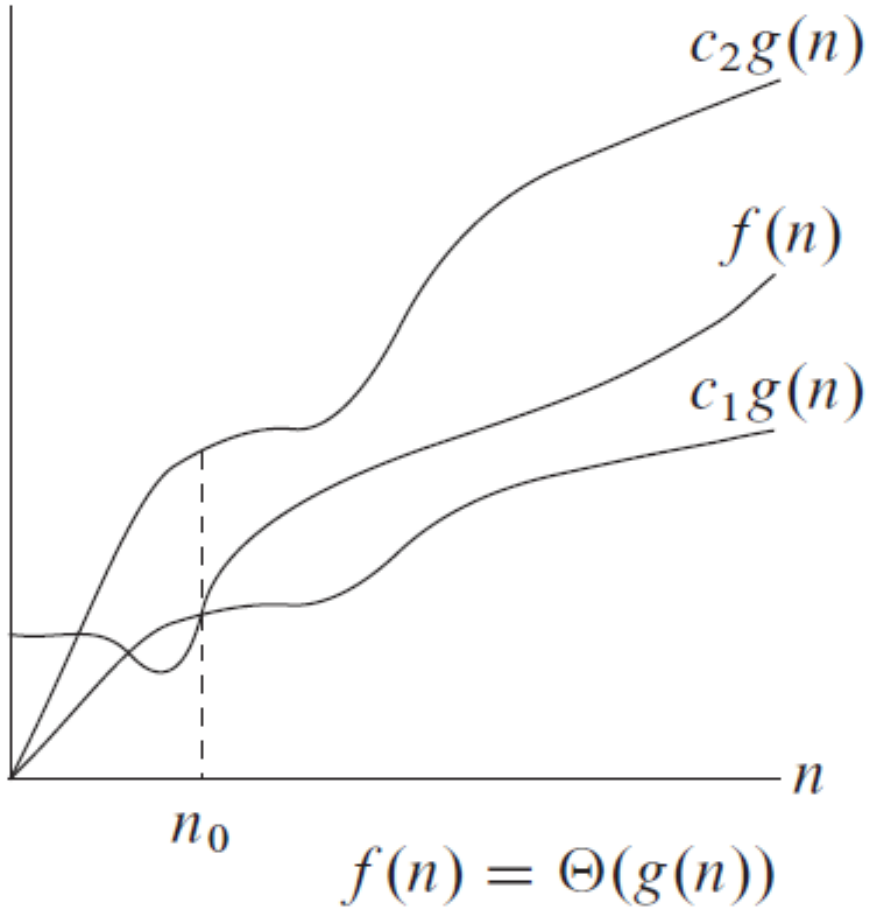
- Die Θ -Notation ist die ***Wachstumsrate*** *(asymptotically tight bound)* einer Funktion: für alle n-Werte (größer als n_0) ist der Wert von f(n) zwischen c_1 *g(n) und c_2 *g(n)
- Anders gesagt, g wächst wie f
- Schreibweise: f(n) ∈ Θ(g(n)) oder f(n) = Θ(g(n))

Θ -Notation

- Alternative Definition:

  f(n) ∈ Θ(g(n)) wenn $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ 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 $\lim_{n\to\infty} \frac{T(n)}{n^2} = 1$



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

Durchschnittlicher 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

Durchschnittlicher 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 $\frac{1}{n+1}$

$$\mathrm{T(n)} = \frac{1}{n+1}\,(1 + 2 + \ \ldots + n + n) = \frac{n\,(n+1)}{2\,(n+1)} + \frac{n}{n+1} \in \Theta(n)$$

Bester Fall, Schlimmsten Fall, Durchschnittlicher 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

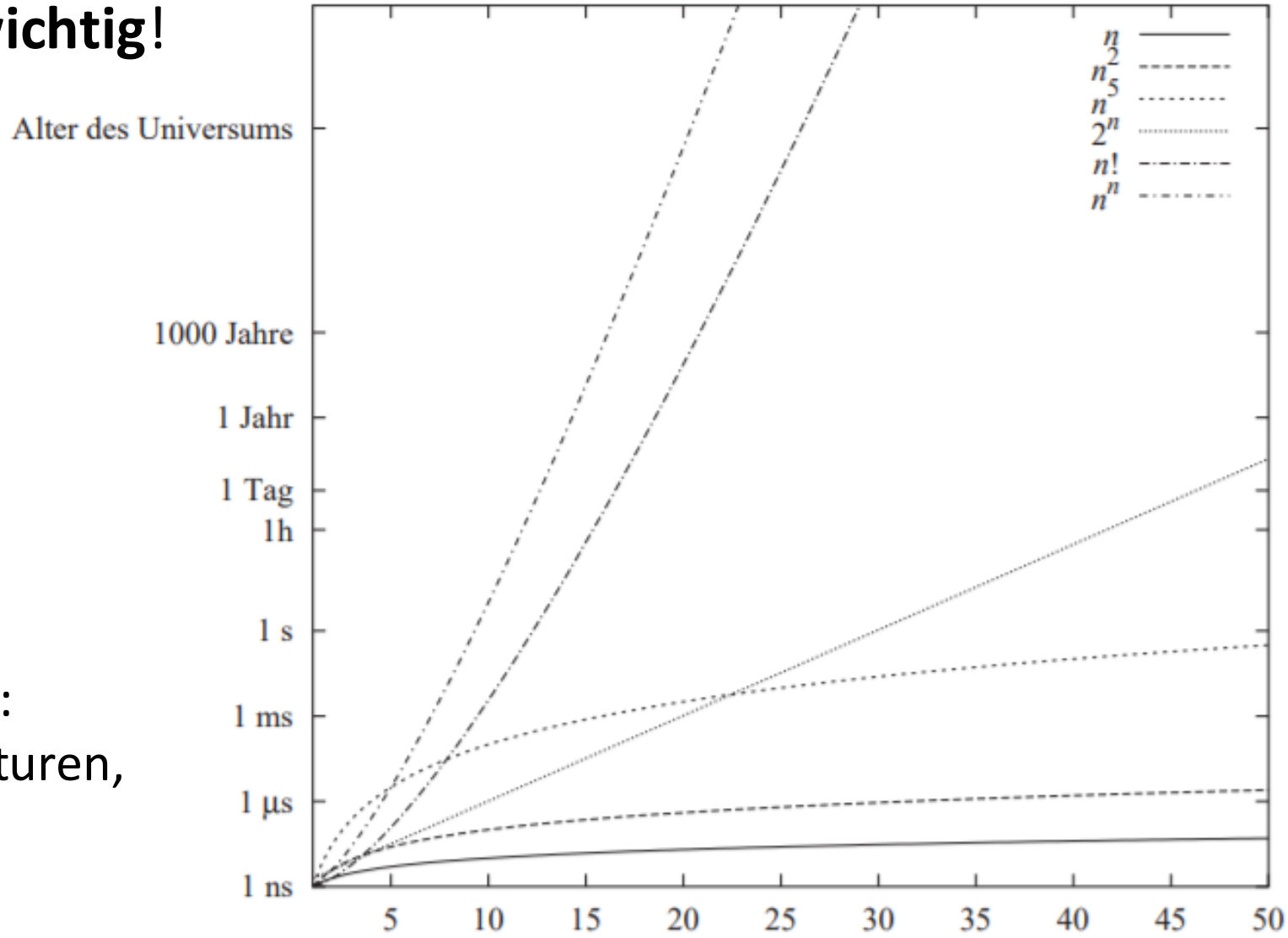
- Um Algorithmen bezüglich ihrer Komplexität vergleichen zu können, teilt man diese in Komplexitätsklassen ein:

Klasse	**Bezeichnung**	**Beispiel**
1	konstant	elementarer Befehl
log n	logarithmisch	binäre Suche
n	linear	lineare Suche
n^2	quadratisch	einfache Sortierverfahren
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**!



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

Algorithmenanalyse für rekursive Funktionen - Beispiel

```
function BinarySearchR (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 (inklusiv)
  //end – Ende des Intervalls wo man suchen muss (inklusiv)
  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
```

Algorithmenanalyse 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 = end – start)
- Man muss die rekursive Formel schreiben um die Komplexität zu berechnen:
- $$T(n) = \begin{cases} 1, & wenn\ n \leq 1 \\ T\left(\frac{n}{2}\right) + 1, & ansonsten \end{cases}$$

Algorithmenanalyse für rekursive Funktionen

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

$= T(2^{k-2}) + 1 + 1 = T(2^{k-2}) + 2 =$

$\ldots = T(2^0) + k =$

$= k + 1$

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

D.h. $T(n) = 1 + log_2 n \in \boldsymbol{\Theta(log_2 n)}$ **(logaritmisch)**

Bem. Wenn wir den Algorithmus betrachten, dann merken wir, dass es auch ein bester Fall gibt mit der Komplexität $\Theta(1)$, also die allgemeine Komplexität für `BinarySearchR` ist $\boldsymbol{O(log_2 n)}$.

Algorithmenanalyse für rekursive Funktionen - Beispiel 2

```
Subalgorithm operation(n, i) is:
//n und i sind ganze Zahlen, n>0
   if n > 1 then
            i ← 2 * i
            m ← 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
```

Algorithmenanalyse für rekursive Funktionen - Beispiel 2

- Rekursive Formel für Algorithm Analyse:

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

- Nehmen wir an, dass $n = 2^k$

- $T(2^k) = 4 * T(2^{k-1}) + 1 = 4 * (4 * T(2^{k-2}) + 1) + 1$
  $= 4^2 * T(2^{k-2}) + 4 + 1 =$
  $= 4^3 * T(2^{k-3}) + 4^2 + 4 + 1 =$
  $\ldots = 4^k * T(2^0) + 4^{k-1} + 4^{k-2} + \ldots + 4 + 1 =$
  $= 4^k + 4^{k-1} + \ldots + 4 + 1$

Algorithmenanalyse 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:

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

Algorithmenanalyse für rekursive Funktionen – Master-Theorem

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

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

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