



Computergrundlagen Programmiergrundlagen: Algorithmen und Datenstrukturen

Frank Uhlig, Jens Smiatek, Axel Arnold, Maria Fyta

Institut für Computerphysik
Universität Stuttgart

Wintersemester 2018/19

Literatur:

- Ottmann, Widmayer, Spektrum Akademischer Verlag, Heidelberg 2012
- Weicker, Weicker, Springer Fachmedien, Wiesbaden 2013
- Dietzfelbinger, Mehlhorn, Sanders, Springer, Berlin Heidelberg 2014

Was ist ein Computer?

*Ein Computer (Rechner oder elektronische Datenverarbeitungsanlage) ist ein Gerät, welches mittels **programmierbarer Rechenvorschriften (Algorithmen)** Daten verarbeitet.*

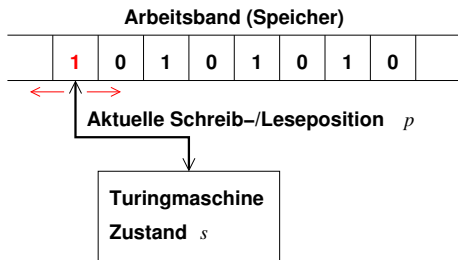
Fragen:

- Kann Rechenmaschine mathematisch, logisch Wahrheiten bestimmen? (Bedarf formaler Sprache!)
- Was heisst programmierbar? . . . berechenbar?
- Wie sehen Rechenvorschriften (Algorithmen) aus?

Theoretische Informatik (Berechenbarkeit)



A. Turing, 1912 - 1954



- ca 1920, D. Hilbert: Was ist berechenbar, was beweisbar?
- 1931, K. Gödel: Entweder widerspruchsfrei oder vollständig!
- Nicht alles ist berechenbar!
- Wir können nicht mal bestimmen, was nicht beweisbar ist

Implikationen

Entscheidungsproblem (David Hilbert):

Ist eine Formel allgemeingültig?

Ist jede Interpretation einer formal logischen Aussage wahr?

**Turingmaschine (Alan Turing):
Definition des Algorithmus und der
Berechenbarkeit als formale, mathematische
Begriffe**

**Turing-berechenbare Funktionen sind genau die von
Menschen intuitiv berechenbaren Funktionen**

A. Turing: *On computable numbers, with an application to the Entscheidungsproblem.* (1936)

Was sind Programme?

- In dieser Vorlesung: Python, C, \LaTeX , bash
- Sind das alle Typen von Programmiersprachen?

Wie kann man Programmiersprachen
klassifizieren?

- Kein Prozessor versteht Python-, C- oder Shell-Befehle
- Was muss alles passieren, um ein Programm laufen zu lassen?

Wie wird aus unserem Programm etwas, dass der
Prozessor ausführen kann?

Programmiersprachen

Imperative Sprachen

Beispiele: Python, C, Shell, BASIC...

- Die meisten Sprachen sind imperativ
- Programme erklären, *wie* ein Problem gelöst werden soll
- Umsetzung des von Neumann-Rechners: Befehle und Schleifen
- **prozedural** heißen Sprachen, die Prozeduren kennen

Beispiele: alle außer einfachem BASIC

Deklarative Sprachen

- Keine von Neumann-artigen Befehle, kein innerer Zustand
- Rekursion anstatt Schleifen
- **funktional**, basierend auf Funktion im mathematischen Sinn

Beispiele: Haskell, Erlang, Scheme...

- **logisch**, basierend auf Fakten, Axiomen und logischer Ableitung

Beispiele: Prolog

Strukturierte Programmierung

Beschränkung auf wenige Kontrollstrukturen:

- Sequenz von Befehlen oder Prozeduren
- Verzweigung
- Wiederholungen
- Rekursion

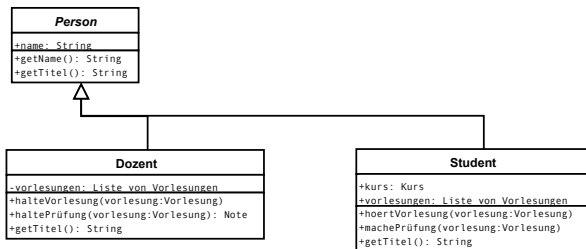
→ eingeschränkter Einsatz von *GO TO*

Vermeidung von:

- “Spagethcode”
- Codewiederholungen (*Don't repeat yourself (DRY)*)

Separation in Einzelprobleme

Objektorientierte Sprachen

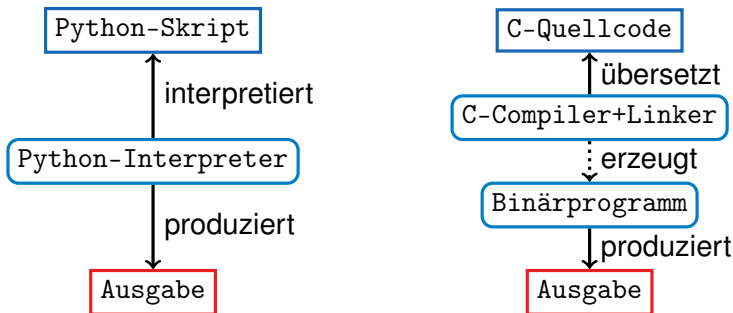


- Objektorientierung (OO) ist ein sprachunabhängiges Programmiermodell
- Ziel ist die bessere Wartbarkeit und Austauschbarkeit von Software durch starke Isolation von Teilproblemen
- Speziell darauf ausgelegt sind z.B. C++, Java, Python,...
- UML (Unified Modelling Language) beschreibt OO-Modelle

Terminologie

- **Klasse:** beschreibt Eigenschaften und Methoden von Objekten
Beispiel: Klassen sind z.B. Dozent, Student, Person
- **Objekt:** eine Instanz einer Klasse. Ein Objekt hat stets eine Klasse, aber es kann viele Instanzen geben
Beispiel: Maria Fyta ist ein Dozent (neutral...), ich auch
- **Eigenschaften:** Datenelemente von Objekten
Beispiel: Dozent hat Name, Titel und zu haltende Vorlesungen
- **Methoden:** Funktionen einer Klasse
Beispiel: Personen können ihren Namen sagen
- **Datenkapselung:** Eigenschaften werden nur durch Funktionen der Klasse verändert
Beispiel: der Name einer Person kann nicht geändert werden
- **Vererbung:** Klassen können von anderen abgeleitet werden, erben dadurch deren Eigenschaften und Funktionen
Beispiel: Jede Person hat einen Namen

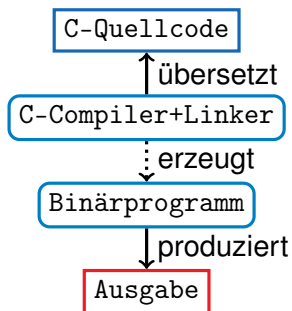
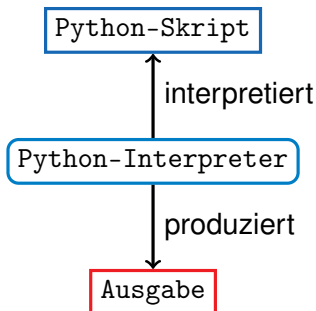
Interpretierte- und kompilierte Sprachen



Interpretierte Sprachen

- Z.B. Python, Java, C#, Basic, PHP, Shellsprachen
- Das Programm wird vom Interpreter gelesen und ausgeführt
- Es wird nie in Maschinencode für den Prozessor übersetzt
- Ausnahme: Just-in-Time (JIT) Compiler

Interpretierte- und kompilierte Sprachen



Kompilierte Sprachen

- Z.B. C/C++, Fortran, Pascal/Delphi
- Compiler übersetzt in maschinenlesbaren Code
- Nicht portabel, erschwerte Fehlersuche, deutlich schneller
- Interpreter selbst müssen kompiliert werden

Maschinensprache

Aber was versteht nun der Prozessor?

Zahlenkolonnen, z.B.

```
AD 34 12 18 69 2A 8D 34
12 AD 35 12 69 00 8d 35
12
```

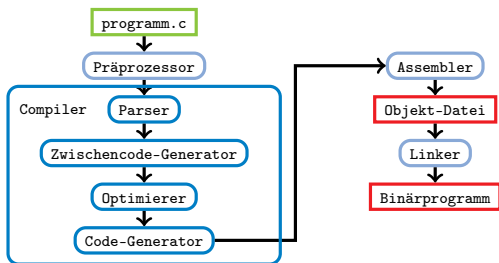
- Maschinencode für den 6502-Mikroprozessor
- Addiert 42 zur 16-bit-Zahl an Speicherstellen 1234h und 1235h

Wie wird aus einem Programm Maschinencode?

Assembler

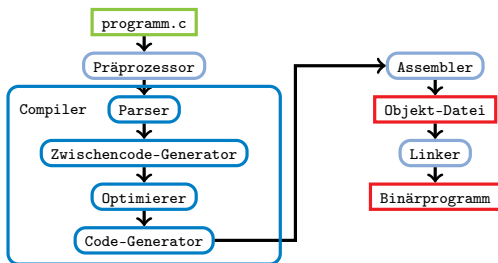
- Assembler ist die menschenlesbare Form der Maschinensprache
 - Assembler bezeichnet auch das Programm, das die Befehle in Zahlen übersetzt und Labels an Speicherzellen knüpft
 - Zeitkritische Anwendungen werden auch heute noch manchmal in Assembler geschrieben
 - sehr nah an Maschinenanweisungen
 - spezifisch für Computerarchitektur
- nicht portabel

Vom Sourcecode zum Programm



- Ein Programm durchläuft viele Schritte bis zur fertigen ausführbaren Datei
- **Präprozessor**, **Compiler**, **Assembler** und **Linker** sind meist separate Programme
- meist mehrere Objektdateien aus verschiedenen Quelltextdateien

Vom Sourcecode zum Programm

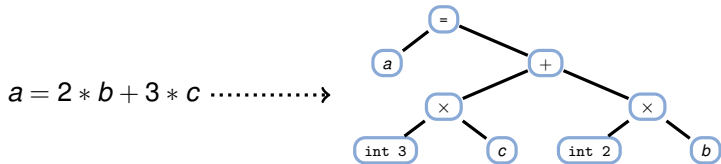


Präprozessor

- Im Prinzip sprachunabhängig, rein textbasiert
- Bindet weitere Quelltextdateien ein
- Erlaubt den Einsatz von *Makros* (Ersetzungen)

Der Compiler

- **Parser** übersetzt den Quellcode in einen Syntaxbaum

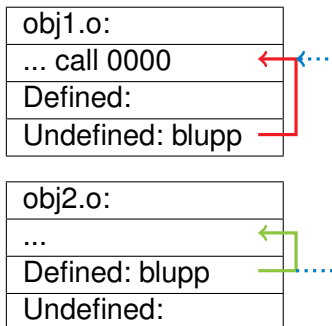
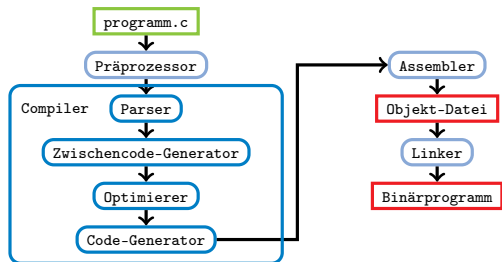


- **Zwischencode-Generator** erzeugt daraus Pseudocode, etwa:

$r1 = b$	$r2 = r1 \times 2$	$r3 = c$	$r4 = r3 \times 3$	$r5 = r2 + r4$	$a = r5$
----------	--------------------	----------	--------------------	----------------	----------

- **Optimierer** versucht, diesen zu verbessern, z.B. durch
 - Prozessorregisterzuweisung
 - Einfügen kurzer Funktionen, Entrollen von Schleifen
 - Suche nach gemeinsamen Termen, ...
- **Code-Generator** erzeugt Assembler aus dem Zwischencode

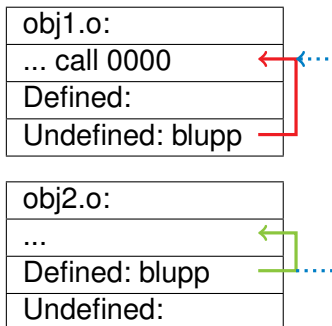
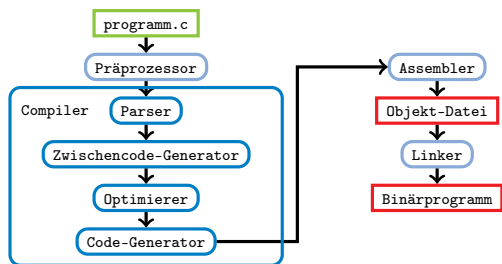
Der Assembler



Der **Assembler** erzeugt eine Objektdatei mit

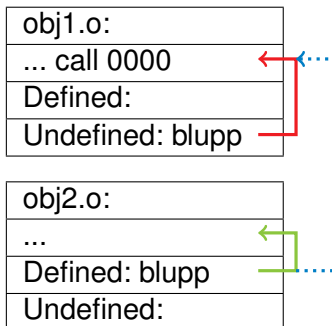
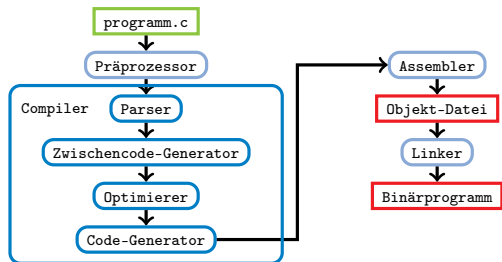
- Maschinencode
- den Speicherpositionen von globalen Variablen und Funktionen
- Stellen, an denen Stellen Information über solche Positionen aus andere Objektdateien benötigt wird

Der Linker



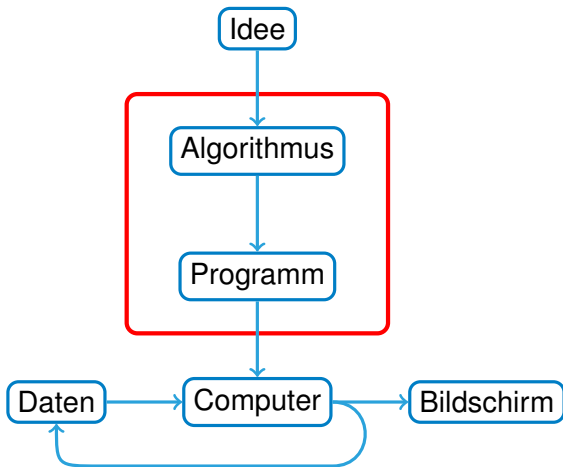
- Verbindet mehrere solcher Objektdateien zu einer ausführbaren Datei (Binary) oder einer **Bibliothek**
- Bibliotheken sind Sammlungen von Objektdateien
- Der Linker verbindet Zugriffe zwischen Objektdateien
- Nur wenn alle Zugriffe aufgelöst wurden, entsteht ein Binary

Dynamisches Linken



- Beim **dynamischen Linken** wird das Linken gegen die Bibliotheken erst beim Starten des Programms erledigt.
- Dadurch können Teile des Programms geupdated werden
- Und mehrfach benutzte Bibliotheken werden nur einmal geladen
- Erfordert spezielle dynamische Bibliotheken (.so, .dylib oder .dll)

Was ist Programmieren?



Algorithmus

Wikipedia:

Ein Algorithmus ist eine aus endlich vielen Schritten bestehende eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen.

Ein Beispiel-**Problem**:

Gegeben

- Liste aller Teilnehmer der Vorlesung

Fragestellung

- ~~Wer wird die Klausur bestehen?~~ ⚡
- Wieviele Studenten haben nur einen Vornamen?
- Wessen Matrikelnummer ist eine Primzahl?

Programm

Ein Programm ist eine Realisation eines Algorithmus in einer bestimmten Programmiersprache.

- Es gibt derzeit mehrere 100 verschiedene Programmiersprachen
- Die meisten sind *Turing-vollständig*, können also alle bekannten Algorithmen umsetzen

Softwareentwicklung und Programmieren

- Entwickeln der Algorithmen
 - Aufteilen in einfachere Probleme
 - Wiederverwendbarkeit
- Umsetzen in einer passenden Programmiersprache

Von der Idee zum Programm

Schritte bei der Entwicklung eines Programms

1. Problemanalyse

- Was soll das Programm leisten?
Z.B. eine Nullstelle finden, Molekulardynamik simulieren
- Was sind Nebenbedingungen?
Z.B. ist die Funktion reellwertig? Wieviele Atome?

2. Methodenwahl

- Schrittweises Zerlegen in Teilprobleme (Top-Down-Analyse)
Z.B. Propagation, Kraftberechnung, Ausgabe
- Wahl von Datentypen und -strukturen
Z.B. Listen oder Tupel? Wörterbuch?
- Wahl der Rechenstrukturen (Algorithmen)
Z.B. Newton-Verfahren, Regula falsi
- Wahl der Programmiersprache



Von der Idee zum Programm

Schritte bei der Entwicklung eines Programms

3. Implementierung und Dokumentation

- Programmieren und *gleichzeitig* dokumentieren
- Kommentare und externe Dokumentation (z.B. Formeln)

4. Testen auf Korrektheit

- Funktioniert das Programm bei erwünschter Eingabe?
Z.B. findet es eine bekannte Lösung?
- Gibt es aussagekräftige Fehler bei falscher Eingabe?
Z.B. vergessene Parameter, zu große Werte

5. Testen auf Effizienz

- Wie lange braucht das Programm bei beliebigen Eingaben?
- Wieviel Speicher braucht es?

6. Meist wieder zurück zur Problemanalyse, weil man etwas vergessen hat ...



Motivation

Viele Problemstellungen wiederholen sich:

- Suchen
- Sortieren
- geometrische Zerteilung (Voronoi)
- Durchlaufen von Datenstrukturen

→ Verwendung optimierter Algorithmen und Datenstrukturen trägt bei zu Übersichtlichkeit und Effizienz

Kontrollstrukturen und Pseudocode

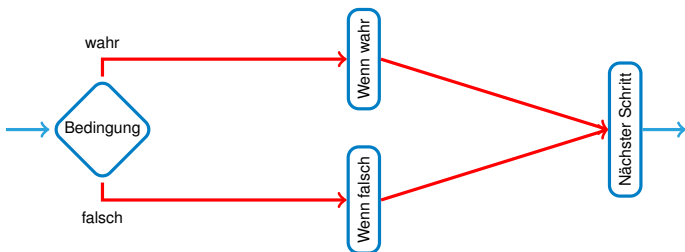
Beeinflussung der Ausführung eines Programms in Abhängigkeit von Bedingungen

- IF, ELSE IF, ELSE
- DO WHILE
- FOR

Pseudocode

- Übliche Kontrollstrukturen ähnlich in verschiedenen Programmiersprachen
- Verwendung (abstrakter) Mathematik möglich
- auslassen von Details wie:
 - Variablendeklaration
 - Subroutinen
 - system-spezifischer Code
- bessere Verständlichkeit

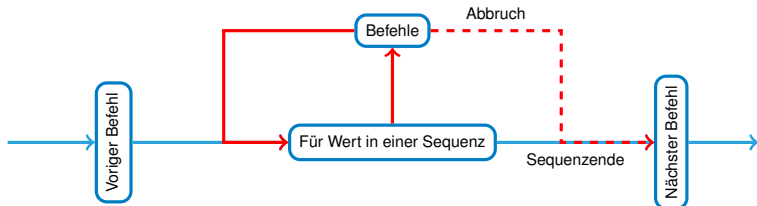
Bedingte Ausführung: if



- Das Programm kann auf Werte von Variablen verschieden reagieren
- Wird als *Verzweigung* bezeichnet
- Auch mehr Äste möglich (z.B. < 0 , $= 0$, > 0)

Student hat mehr als 50% Punkte? \implies zur Klausur zulassen

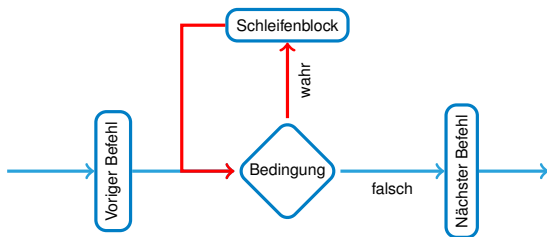
for-Schleifen



- Wiederholen eines Blocks von Befehlen
- *Schleifenvariable* nimmt dabei verschiedene Werte aus einer *Sequenz* (Liste) an
- Die abzuarbeitende Sequenz bleibt fest
- Kann bei Bedarf abgebrochen werden (Ziel erreicht, Fehler, ...)

Für jeden Studenten in den Computergrundlagen finde einen Übungsplatz

Vorprüfende Schleifen: while



- Wiederholte Ausführung ähnlich wie for-Schleifen
- Keine *Schleifenvariable*, sondern Schleifenbedingung
- Ist die Bedingung immer erfüllt, kommt es zur **Endlosschleife**

Solange $a > 0$, ziehe eins von a ab

Solange noch Zeilen in der Datei sind, lese eine Zeile

Beispiel Pseudocode

GD = Ganzzahldivision

Algorithm: GD

```

1 input: int N, int D
2 output: int
3 begin
4   res ← 0
5   while N ≥ D
6     N ← N - D
7     res ← res + 1
8   end
9   return res
10 end
  
```

Algorithm: GD in Worten

```

1 input: int N, int D
2 output: int
3 begin
4   Resultat wird auf Null gesetzt
5   solange N groesser als D ist:
6     N wird auf N minus D gesetzt
7     Resultat wird auf Resultat
8     plus Eins gesetzt
9   Ende der Schleife
10  Resultat wird zurueck gegeben
11 end
  
```

Effizienz und Wachstumsraten

Anzahl der grundlegenden Operationen für Input bestimmter Größe \rightarrow Zeit T abhängig vom Input der Größe n : $T(n)$

Wachstumsrate: Die Rate mit der der (Rechen)Aufwand eines Algorithmus' wächst in Abhängigkeit der Input Größe.

Mittels asymptotischer Analyse bestimmbar:

obere Grenze: Eine nicht-negative Funktion $T(n)$ ist in der Menge $\mathcal{O}(f(n))$, wenn es zwei positive Konstanten c und n_0 gibt, so dass $T(n) \leq cf(n)$ für alle $n > n_0$.
 $\rightarrow T(n)$ wächst nicht schneller als $f(n)$

untere Grenze: Eine nicht-negative Funktion $T(n)$ ist in der Menge $\Omega(g(n))$, wenn es zwei positive Konstanten c und n_0 gibt, so dass $T(n) \geq cg(n)$ für alle $n > n_0$.
 $\rightarrow T(n)$ wächst mindestens so schnell wie $f(n)$

keine absolute Aussage über Geschwindigkeit der Ausführung

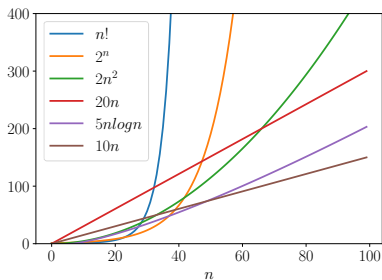
Effizienz und Wachstumsraten

in den meisten Fällen wollen wir wissen wieviel länger es dauert, wenn wir unseren Input vergrößern

- Big-O (obere Grenze - maximaler Aufwand)
- Big- Ω (untere Grenze - minimaler Aufwand)
- Big- Θ (Big-O und Big- Ω gleich (mit Ausnahme einer Konstanten))

Verschiedene Szenarien

- best-case → nützlich in bestimmten Fällen
- average-case → typisches Verhalten über verschiedene Inputs
- worst-case → nützlich zum Vergleich von Algorithmen (umfassendste Garantie)



Beispiel: Verlet-Integrator

Taylorentwicklung für die Position eines Teilchens gemäß der Newtonschen Gleichung:

$$x(t + \Delta t) = x(t) + \dot{x}(t)\Delta t + \frac{1}{2}\ddot{x}(t)\Delta t^2 + \frac{1}{6}\ddot{\ddot{x}}(t)\Delta t^3 + \mathcal{O}(\Delta t^4)$$

und analog

$$x(t - \Delta t) = x(t) - \dot{x}(t)\Delta t + \frac{1}{2}\ddot{x}(t)\Delta t^2 - \frac{1}{6}\ddot{\ddot{x}}(t)\Delta t^3 + \mathcal{O}(\Delta t^4)$$

Addition ergibt das *Verlet*-Verfahren zur numerischen Integration der Bewegungsgleichung:

$$x(t + \Delta t) = 2x(t) - x(t - \Delta t) + a(t)\Delta t^2 + \mathcal{O}_{\Delta t \rightarrow 0}(\Delta t^4)$$

- Fehler ist von der Ordnung $\mathcal{O}_{\Delta t \rightarrow 0}(\Delta t^4)$
- Mit beliebig kleinem Zeitschritt Δt wird das Verfahren exakt
- Man sagt, das Verfahren **konvergiert**

Eigenschaften von Algorithmen

Die wesentlichen Eigenschaften eines Algorithmus können mit Hilfe der Landau-Symbole charakterisiert werden (besser als explizites testen!):

- **Stabilität:** Wir betrachten den Fehler $\Delta f(x)$ in Abhängigkeit von der Genauigkeit Δx der Eingabedaten. Dann soll

$$\Delta f(x) = \mathcal{O}_{\Delta x \rightarrow 0}(\Delta x^n)$$

mit möglichst großem n sein.

- **Konvergenz:** Z.B. der Fehler als Funktion des Zeitschritts Δt . Dann soll

$$\Delta f(x) = \mathcal{O}_{\Delta t \rightarrow 0}(\Delta t^n)$$

sein mit möglichst großem n .

- **Effizienz:** Die Rechenzeit $T(N)$ z.B. bei N Teilchen. Dann soll

$$T(N) = \mathcal{O}_{N \rightarrow \infty}(N^n)$$

mit möglichst *kleinem* n sein.

Beispiele

Algorithm: sum of squares

```
1 input: Liste L mit N Elementen
2 output: quadsum
3 begin:
4     quadsum ← 0
5     for i = 0, N-1, ++i:
6         for j = 0, N-1, ++j:
7             quadsum += L[i] * L[j]
8
9     return quadsum
10 end
```

Kosten:

$$\Theta(n^2)$$

Beispiele

Algorithm: Bubble sort

```
1 input: Liste L, Anzahl Elemente N
2 output: Liste L
3 begin
4     for i = 0 upto N-1:
5         for j = N-1 downto i+1:
6
7             if L[j] > L[j-1]:
8                 swap(L[j], L[j-1])
9
10        return L
11 end
```

Kosten:

$$\Theta(n^2)$$

Datenstrukturen

Datentyp

Datentyp einer Variable entspricht der Menge an Werten, die die Variable annehmen kann.

- int, char, float, double, bool

Abstrakter Datentyp

Menge von Elementen mit einer Sammlung von wohldefinierten Operationen.

- Listen, Mengen, Bäume, Graphen, ...

Datenstruktur

Implementierung die einem abstrakten Datentyp entspricht.

Listen

allg.: Reihe von Einträgen in linearer Reihenfolge angeordnet

Zugriff über:

- Index (wahlfrei) → Arrays (Python Liste: Array von Zeigern)
- vorangehendes/nachfolgendes Element → *verkettete Listen*

Array:

- Bereich kontinuierlich im Speicher, Anfangsposition bekannt
- Elemente, durch mathematische Formel indiziert
- 1-dimensionaler Array: Elemente hintereinander im Speicher

Element verketteter Listen enthält:

- Wert des Elements
 - Adresse des nächsten (und evtl. folgenden) Elemente → *Pointer*
- Knotenstruktur vereinfacht einfügen, anhängen, voransetzen, zusammenführen mehrerer Listen

Doppelt-verkettete Listen

- Folge von n Einträgen von Ring mit $n + 1$ Knoten dargestellt
- Dummyknoten h enthält keinen Eintrag, verknüpft Ende und Anfang
- Linearität erfüllt durch:
 - der Nachfolger des Vorgängers von ik is wieder ik
 - der Vorgängers Nachfolgers von ik is wieder ik
 - Einstiegsknoten bekannt

Knoten einer doppelt-verketteten Liste:



Operationen

- insert, remove \rightarrow beliebiges Element einfügen oder löschen
- pushBack, pushFront \rightarrow Elem. am Ende/Anfang hinzufügen
- popBack, popFront \rightarrow Elem. am Ende/Anfang entfernen
- size, first, last, concat, splice, ...

Hashtabellen und assoziative arrays

Aufgabenstellung: Speichere die Note jedes Studentens, der die CGL-Klausur mitschreibt, in einem array, der über einen Schlüssel indiziert wird.

- *Schlüssel* zB Personalausweisnummer (PAN)
Problem: sehr viele PANs, wenige CGL-Studenten
 - brauchen *kurzen* array, der nur die relevanten Studenten enthält
- assoziativer array

Hashtabelle oft zur Implementierung assoziativer Arrays verwendet

- Hashfunktion bildet die Menge aller möglichen Schlüssel einen kleinen Bereich der ganzen Zahlen ab ($0 \dots n - 1$)
- array mit der Indexmenge $0 \dots n - 1$ → Hashtabelle
- $n \approx$ Anzahl der CGL Studenten
 - idealerweise bildet Hashfunktion so ab, dass Wert direkt in Hashtabelle gespeichert werden kann
 - Kollisionen möglich → möglichst viel Unordnung erzeugen