Hochschule Deggendorf

Algorithmen und Datenstrukturen

Teil 1 – Einleitung

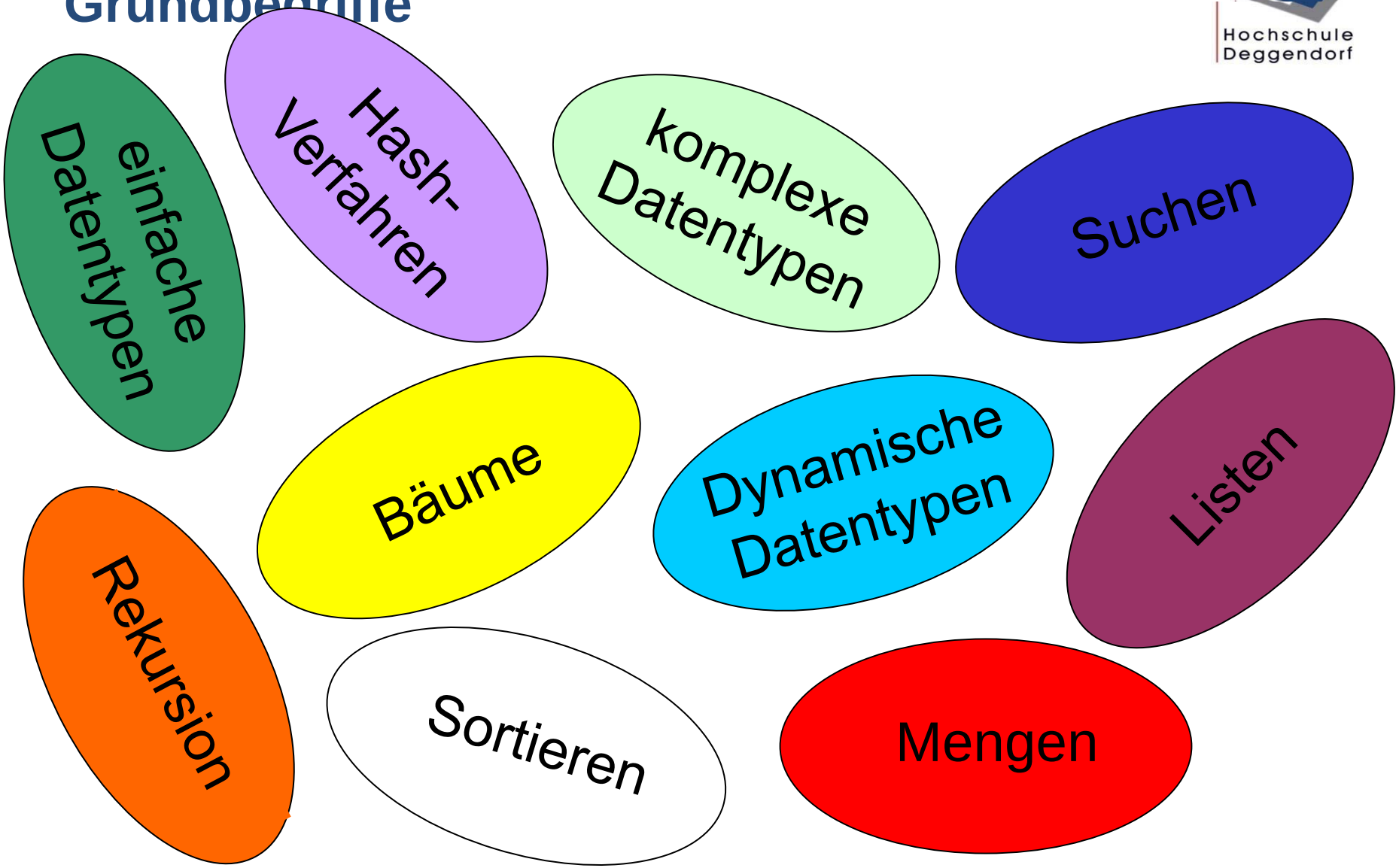
Prof. Dr. Peter Jüttner

Ziele der Vorlesung

Hochschule Deggendorf

1. **Studenten können Komplexität von Algorithmen bewerten**
2. **Studenten kennen Rekursion und deren Vor- und Nachteile**
3. **Studenten kennen wichtige Such- und Sortieralgorithmen und ihre Vor- und Nachteile**
4. **Studenten beherrschen wichtige Datenstrukturen**
5. **Studenten haben 1. 2. und 3. in Übungen und Beispielen aktiv erfahren**

Grundbegriffe



Vorbemerkungen

Hochschule Deggendorf

- **Vorlesung nicht spezifisch für eine Programmiersprache**
- **Beispiel in C/C++**
- **Klausur mit Unterlagen**
-

Zeitplan

Hochschule Deggendorf

- **Vorlesung/Übung Donnerstags**
- **8:00 – 9:30 Vorlesung**
- **9:45 – 11:15 Übung (2 Gruppen)**

Sprechstunde Dozent:

- **Mi. 8.00 – 9.30 Uhr im Büro nach Voranmeldung**
- **auch nach Vereinbarung**
- **vor und nach der Vorlesung**

Fragen

Falls Sie Fragen haben …

fragen Sie bitte

- **sofort**
- **oder vor / nach der Vorlesung**
- **oder in der Sprechstunde**
- **oder nach Terminvereinbarung**

Motivation

Hochschule Deggendorf

- **Effiziente Algorithmen und Datenstrukturen sind ein zentrales Thema der Informatik.**

- **Direkter und enger Zusammenhang zwischen der Organisation von Daten (ihrer Strukturierung) und dem Entwurf von Algorithmen, die diese Daten bearbeiten.**
  z.B. komplexe Datenstrukturen ermöglichen schnelle Algorithmen, kosten aber (tendenziell) viel Speicher (und umgekehrt)

Inhalt

Algorithmen und Datenstrukturen

Hochschule Deggendorf

Literatur

- Algorithmen und Datenstrukturen, Skript zur Vorlesung, Dieter Hofbauer und Friedrich Otto, FB Elektrotechnik / Informatik und FB Mathematik / Informatik, Universität Kassel
- Algorithmen und Datenstrukturen, Vorlesungsskript, Gunter Saake, Kai-Uwe Sattler Universität Magdeburg, Juli 2000
- Uwe Schöning: Algorithmik, oder Algorithmen - kurz gefasst, Spektrum, ST 134 S365
- Uwe Schöning: Theoretische Informatik - kurz gefaßt, Spektrum, ST 130 S365
- R. Sedgewick: Algorithmen in Java, Pearson, oder Algorithms in Java, Addison-Wesley, ST 250 J35
- M. Goodrich, R. Tamassia: Data Structures and Algorithms in Java, Wiley, ST 265 G655
- V. Heun, Grundlegende Algorithmen, Vieweg, ST 134 H593
- H. Gumm, M. Sommer: Einführung in die Informatik, Oldenbourg, ST 110 G974
- W. Küchlin, A. Weber: Einführung in die Informatik, Springer-Verlag, ST 110 K95
- T.H. Cormen, C.E. Leiseron, R.R. Rivest, C. Stein: Introduction to Algorithms, 2nd ed., The MIT Press / McGraw-Hill, ST 134 C811 (2)
- J. Kleinberg, E. Tardos: Algorithm Design, Addison-Wesley, ST 134 K64

Motivation

Hochschule Deggendorf

Zum Schluss dieses Abschnitts …

Noch Fragen ??

Inhalte

Algorithmen

Algorithmus:

Definierte Berechnungsvorschrift, die aus einer Menge von Eingaben eine Menge von Ausgaben erzeugt.

Algorithmen

Beispiele (aus dem täglichen Leben ...)

- **Kochrezepte „... man nehme ...“**

  300 g Butter, weich, 270 g Zucker, 1 Beutel Vanillezucker, 1 Rum-Aroma, 1 Prise Salz, 5 Eier, 375 g Mehl (Weizen), 12 g Backpulver, 3 EL Milch, 20 g Kakaopulver, 20 g Zucker, 3 EL Milch, Puderzucker

  Weiche Butter geschmeidig rühren, nach und nach Zucker, Vanillezucker, Rum-Aroma und Salz zugeben und solange rühren, bis eine gebundene Masse entstanden ist. Die Eier einzeln einrühren.
  Mehl und Backpulver vermischen und abwechseln esslöffelweise mit der Milch einrühren (nur so viel Milch verwenden, dass der Teig schwer reißend von einem Löffel fällt). 2/3 des Teiges in eine Kuchenform füllen. Kakao mit Zucker vermischen und Milch einrühren, das ganze unter den restlichen Teig rühren. Den dunklen Teig auf dem hellen verteilen und mit einer Gabel spiralförmig durch die Teigschichten ziehen. Den Kuchen backen bei 190°C, ca. 60 Min. (wenn er oben zu dunkel wird, nach der Hälfte der Backzeit mit einem Stück Alufolie abdecken). Den erkalteten Kuchen mit Puderzucker bestäuben.

Algorithmen

Beispiele (aus dem täglichen Leben ...)

- **Kochrezepte „... man nehme ...“**

Algorithmen

Beispiele (aus dem täglichen Leben ...)

- **Bedienungsanleitungen (Beispiel Heizlüfter)**

Sehr bedienen für Heiss.
AB 2000 hat Connector Stecks an Cable mit 1,5 m für Stecks Betrieb. Du stecks Connector in Stecks an Wand oder an Leitung / Line. Leitung muss AC sein und egal 180 bis 250 Volts ist gut. ... Power ist bei drehen Schalter auf hat Step 1 bei Kalt 18 Watts zum Fan blasen. Power kommt bei Drehen Heiss Step 2 1.200 Watts mit Heiss und Fan blasen. Ganz fully hat bei Drehen viel Heiss Step 3 2.000 Watts mit vielen Heiss und Fan blasen. Blasen mit Sicherheit immer bei Heiss sonst kewin Gefahr kein hat.

Algorithmen

Beispiele (aus dem täglichen Leben ...)

- **Beipackzettel für Medikamente „Tablette in Wasser auflösen ...“**
- **Anleitung zum Zusammenbau von für IKEA Möbeln**
- **Anleitung zum Zusammenbau von LEGO Modellen (Besonderheit?)**
- **Computerprogramme „if a<b ... else ... „**

Algorithmen

Hochschule Deggendorf

Eigenschaften von Algorithmen

- ***Endlichkeit der Beschreibung***
  (Gegenbeispiel 1 + 1/2 + 1/4 + 1/8 + 1/16 + ...)

- ***Effektivität*, d.h. jeder Schritte des Algorithmus ist ausführbar**

- ***Terminierung*, d.h. der Algorithmus kommt immer(!) in endlicher Zeit zu einem Ende*)**
  (Gegenbeispiel: Berechnung der Zahl Pi)

*) diese Eigenschaft ist nicht immer auf Anhieb zu erkennen und nicht allgemein zu beweisen!

Algorithmen

Hochschule Deggendorf

Eigenschaften von Algorithmen

- ***Effizienz***, **d.h. Verhältnis von Aufwand und Leistung, ein Algorithmus ist z.B. effizienter, wenn er mit weniger Rechenoperation auskommt oder mit weniger Speicherplatz als ein anderer*)**
  (Achtung: Effizienz ≠ Effektivität !)

*) Effizienz ist „relativ“, bei kleinem Speicherplatz ist ein langsamer Algorithmus u.U. effizienter

Algorithmen

Hochschule Deggendorf

Eigenschaften von Algorithmen

- ***Determinismus****)*, d.h. der Ablauf ist eindeutig vorgeschrieben**
  (Gegenbeispiel „... man nehme ein Pfund Hackfleisch von Rind oder vom Schwein ...“)

- ***Determiniertheit*****)*, d.h. das Ergebnis des Algorithmus ist bei gleichen Eingabedaten immer gleich**

*) das zugehörige Adjektiv ist *deterministisch*

**) das zugehörige Adjektiv ist *determiniert*

Algorithmen

Hochschule Deggendorf

Eigenschaften von Algorithmen

Anmerkung:

1.) Determinismus und Determiniertheit sind nicht abhängig voneinander.

D.h. ein nicht-deterministischer Algorithmus muss nicht automatisch nicht-determiniert sein.

Beispiel: Finden des kleinsten Elements in einem Integer Array:

Durchsuche das Array beginnend mit dem kleinsten Index oder durchsuche das Array beginnend mit dem größten Index.

Algorithmen

Hochschule Deggendorf

Eigenschaften von Algorithmen

Anmerkung:

Ein nicht-deterministischer Algorithmus muss nicht automatisch nicht-determiniert sein.

2.) Beispiel: Nehmen Sie eine Zahl x ungleich 0;

Addieren Sie das Dreifache von x zu x und teilen das Ergebnis durch x oder subtrahieren Sie 4 von x und subtrahieren das Ergebnis von x

Algorithmen

Hochschule Deggendorf

Eigenschaften von Algorithmen

Anmerkung:

1.) Die gängigen Programmiersprachen C, C++, Java, C# sind deterministisch

2.) Nicht-deterministische erscheinende Abläufe entstehen bei parallelen Abläufen von Software, z.B. in verschiedenen Prozessen auf einem Mikrocontroller

Algorithmen

Hochschule Deggendorf

Eigenschaften von Algorithmen

- ***Semantik***, d.h. die Bedeutung des Algorithmus

Anmerkungen:

- **Die tatsächliche Semantik eines Algorithmus kann von der beabsichtigten Semantik abweichen. In diesem Fall ist der Algorithmus falsch!**
- **Verschieden Algorithmen können die gleiche Semantik haben.**

Algorithmen

Hochschule Deggendorf

Eigenschaften von Algorithmen

***Korrektheit*, d.h. die beabsichtigte Semantik entspricht der tatsächlichen Semantik des Algorithmus**

Achtung: Die Korrektheit von Algorithmen ist nur sehr eingeschränkt beweisbar!

Algorithmen

Hochschule Deggendorf

Zum Schluss dieses Abschnitts …

Noch Fragen ??

Algorithmen

Hochschule Deggendorf

Bausteine von Algorithmen (Wiederholung)

- **elementare Operationen, z.B: a+b, , c*d, a<5, >**
- **sequentieller Ablauf, z.B. a+b; c-d;**
- **paralleler Ablauf (Mehrprozessorsysteme!)**
- **bedingte Ausführung, z.B. if a<b ...**
- **Schleifen, z.B. for (i=1 ...), while (h==7) ...**
- **Unterprogramme, z.B. a=f(b);**
- **Rekursion, z.B. f(int i) { if (...) else f(i-1) }**
- **(Sprünge, z.B. goto marke;)**

Algorithmen

Hochschule Deggendorf

Bausteine von Algorithmen (Wiederholung)

Anmerkungen:

1. **Die Konstrukte elementare Operationen, Sequenz, Bedingung, Schleifen (oder Sprünge) sind ausreichend, alle auf Rechnern programmierbar Algorithmen zu beschreiben (auch andere Kombinationen möglich)**
2. **Es gibt Probleme, die sich nicht mittels eines Programms lösen lassen, z.B. Terminierungsproblem**

➔ **Theorie der Berechenbarkeit**

Algorithmen

Existenz nicht berechenbarer Funktionen

Vorbedingung:

Jeder Algorithmus läßt sich in einem endlichen, fest definierten Alphabet beschreiben

(andernfalls ließe sich der Algorithmus nicht als Programm einer Programmiersprache darstellen)

Algorithmen

Hochschule Deggendorf

Existenz nicht berechenbarer Funktionen

- **A := {a_1; ... ; a_n} ein Alphabet mit der Ordnung $a_1 < a_2 < ... < a_n$.**

- **A* := Menge der Texte (Zeichenketten, endlichen Folgen, Worte), die aus A gebildet werden können**

- **A* = { „_“, „a_1“, „a_2“, ... „a_1a_1“, „a_1a_2“ ... }**

- **Die Elemente von A* können der Länge nach aufgelistet werden. Zu einer Länge l gibt es n^l verschiedene Texte (endlich viele)**

Algorithmen

Hochschule Deggendorf

Existenz nicht berechenbarer Funktionen

- **Durch „$b_1b_2...b_k$“ < „$c_1c_2...c_k$ $\Leftrightarrow$ $b_1 < c_1$ oder**
  $b_1 = c_1$ und $b_2 < c_2$ oder
  ... oder
  $b_k < c_k$

wird eine Ordnung auf A* definiert.

- **Aus der Tatsache, dass es nur endlich viele Texte einer Länge l gibt und dass über A* eine Ordnung definiert werden kann, folgt A* ist abzählbar (d.h. A* kann durchnummeriert werden)**

Algorithmen

Hochschule Deggendorf

Existenz nicht berechenbarer Funktionen

- **Betrachten wir nun speziell einstellige Funktionen $f : Z \rightarrow Z$ (Z ganze Zahlen)**
- **Wie oben erläutert, gibt es nur abzählbar viele solcher Funktionen, die auch berechenbar sind.**

Algorithmen

Hochschule Deggendorf

Existenz nicht berechenbarer Funktionen

- **Es gibt aber insgesamt mehr Funktionen, wie die folgende Überlegung zeigt:**
  - **Wir betrachten die Menge F := { f : Z → Z } der einstelligen Funktionen auf Z und nehmen an, dass diese Menge ebenfalls abzählbar ist.**
    ➔ jedes f aus F hat eine Nummer i, d.h.
    F= {f_1, f_2, ... }
  - **Sei nun g: Z → Z definiert durch g(x) = $f_{abs(x)}(x) + 1$**

Algorithmen

Hochschule Deggendorf

Existenz nicht berechenbarer Funktionen

- **Es gibt aber insgesamt mehr Funktionen, wie die folgende Überlegung zeigt:**
  - **➔es gilt für i = 1, 2, ... $g(i) \neq f_i(i)$**
  - **➔für i = 1, 2, ... gilt immer $g \neq f_i$**
  - **➔g kommt in $\{f_1, f_2, \ldots\}$ nicht vor, ist aber eine einstellige Funktion auf Z und müsste somit in F vorkommen ➔ Widerspruch**
  - **➔Der Widerspruch lässt sich nur auflösen, wenn die Annahme fallengelassen wird, F sei abzählbar.**
  - **➔Es gibt nicht berechenbare Funktionen*)**

***) diese Tatsache war schon in den 30Jahren des vorigen Jahrhunderts bekannt**

Algorithmen

Beispiele nicht berechenbarer Funktionen*)

- **Halteproblem: Terminiert ein Algorithmus x mit Eingabe y**
- **Erreicht ein Algorithmus eine bestimmte Stelle**
- **Sind zwei Algorithmen gleich (d.h. haben Sie immer das gleiche Ergebnis) ?**
- **Ist ein Algorithmus korrekt?**

*) für einen einzelnen gegebenen Algorithmus läßt sich das u.U. entscheiden, nicht aber allgemein

Übung

Beweis derTerminierung von Algorithmen

Beschreibung:
Die Terminierung eines Algorithmus läßt sich allgemein nicht beweisen. Für bestimmte Algortihmen ist dies aber möglich.

Aufgabe:
Überlegen Sie, wie Sie die Terminierung der folgenden Algorithmen zeigen können:
1.) fakultät (n) := if (n==0) return 1 else return (n * fakultät(n-1));

Zeit:
10min

Algorithmen

Hochschule Deggendorf

Zum Schluss dieses Abschnitts …

Noch Fragen ??

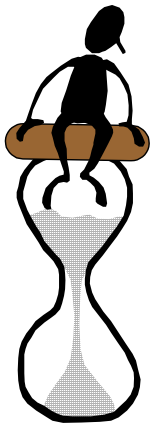
Inhalte

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

- 1. Ziel für die Lösung eines Problems mittels eine Algorithmus: Korrektheit des Algorithmus.
- 2. Ziel: Problemlösung durch Algorithmus mit geringem oder geringstem Aufwand*)
- 2. Ziel u.U. genau so wichtig wie 1., speziell im harten Echtzeitbereich, z.B.
  - Airbag, ABS, ESP (lebenswichtig)
  - Motorsteuerung (gesetzliche Vorgaben)
  - Fernbedienung, Navigation (Komfort)



*) mit Aufwand kann Zeit und/oder auch Platzbedarf gemeint sein

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

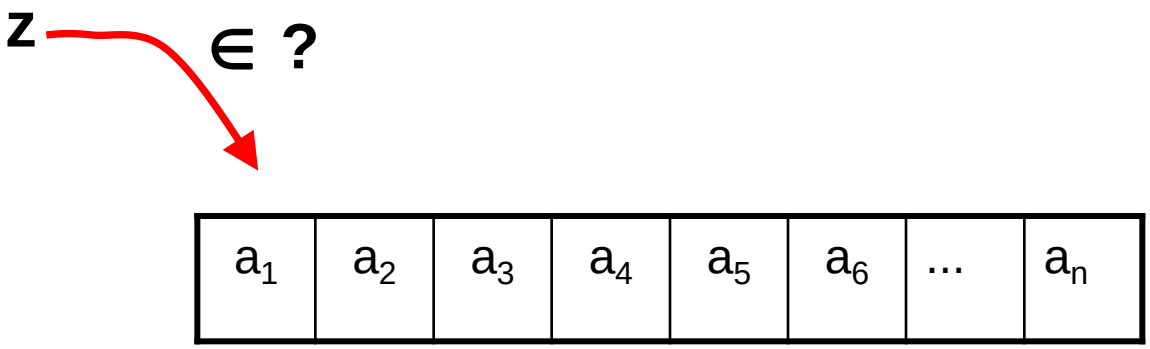
- **Komplexitätstheorie für Algorithmen**
  - ➔ **Schätzen des Aufwands eines konkreten Algorithmus**
  - ➔ **Angabe eines Mindestaufwands für die Lösung eines bestimmten Problems oder einer Klasse von Problemen.**

Algorithmen

Komplexität von Algorithmen

Beispiel: Suchen in einem Feld

- **gegeben ist a, ein unsortiertes Feld n verschiedener Zahlen. a habe die Länge n (n>0). Es soll festgestellt werden, ob eine bestimmte Zahl z in a enthalten ist und wenn ja in welchem Element des Felds.**



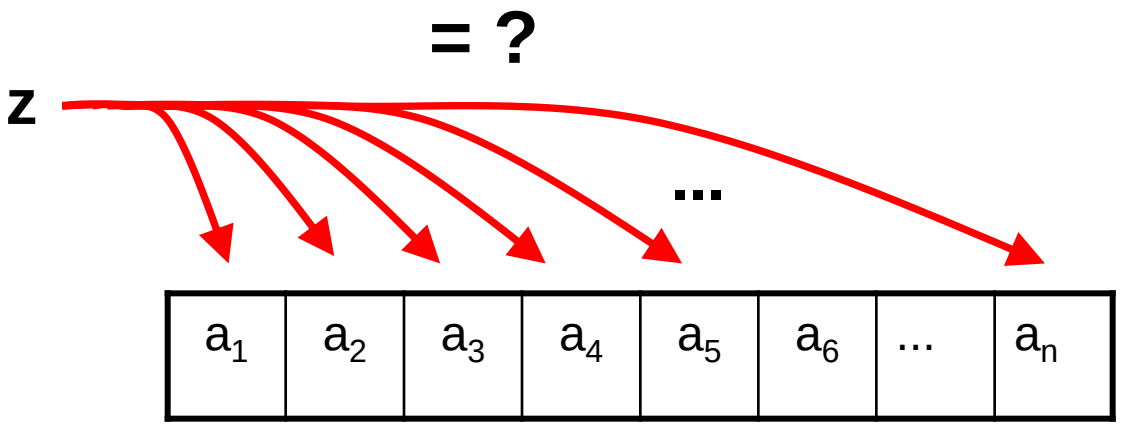
Algorithmen

Komplexität von Algorithmen

Beispiel: Suchen in einem Feld

➔ **Lösung: Durchlaufen des Feldes von ersten Index an bis z gefunden oder das Ende des Feldes erreicht ist.**

```
...
i=0;
while ((i<n) && (a[i] != z)) i++;
...
```



Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Beispiel: Suchen in einem nicht sortierten Feld

➔ **Aufwand für die Suche ist abhängig von**

- **n (Länge des Feld)**
- **a[0], ..., a[n-1] (Inhalt des Feld)**
- **z (gesuchte Zahl)**

➔ **erfolgreiche Suche: z = a[i] ➔ i Schritte**

➔ **erfolglose Suche ➔ n Schritte**

➔ **nicht hilfreich, da abhängig von zu vielen Parametern (Aufwand = f(n, a[1], a[2], ..., a[n], z)**

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Beispiel: Suchen in einem nicht sortierten Feld

➔ Änderung der Fragestellung(en):

- **Anzahl Schritte im schlechtesten Fall?**
- **Anzahl Schritte im Durchschnitt?**

➔ Anzahl Schritte im schlechtesten Fall

- **Element wird nicht gefunden: n Schritte**

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Beispiel: Suchen in einem nicht sortierten Feld

➔ **Anzahl Schritte um Durchschnitt ?**

- **abhängig von der Verteilung der Elemente im Feld**

➔ **Annahme: Gleichverteilung der Zahlen im Feld**

➔ **P(z im 1. Schritt gefunden) = P(z im 2. Schritt gefunden) = ... = P(z im n-ten Schritt gefunden)*)**

➔ **P(z im i-ten Schritt gefunden) = 1/n**

*) P steht hier für Wahrscheinlichkeit (Propability)

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Beispiel: Suchen in einem nicht sortierten Feld

➔ **Anzahl Schritte um Durchschnitt ?**

- ➔ **P(b im i-ten Schritt gefunden) = 1/n**
- ➔ **Anzahl der notwendigen Schritte:**

  1/n * 1 + 1/n * 2 + ... + 1/n * n =

  1/n (1+2+ ...+n) = 1/n * n(n+1)/2 = (n+1)/2

- ➔ **es werden im Schnitt (n+1)/2 Schritte benötigt um b zu finden**

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Komplexitätsklassen

- **für Algorithmen zur Lösung eines Problems**
- **für Probleme, die mittels eines Algorithmus gelöst werden sollen**

Algorithmen

Komplexität von Algorithmen

Komplexitätsklassen (Algorithmus a löst Problem p)

- **Komplexitätsklasse (Algorithmus a) > Komplexitätsklasse (Problems p)**

➔ **besseren Algorithmus suchen**

- **Komplexitätsklasse (Algorithmus a) = Komplexitätsklasse (Problems p)**

➔ **nur noch Feintuning möglich**

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Komplexitätsklassen

- **Problem abhängig von einem (oder mehreren) Größenparametern (z.B. Feld der Länge n, Baum der Tiefe t, Struktur mit x Bytes)**
  - **➔ wie wächst die Komplexität der Lösung der Problems mit dem Größenparameter**

➔ die Komplexität der Lösungsalgorithmen wächst ebenfalls mit dem Größenparameter

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Komplexitätsklassen

- **möglichst unabhängig von einem konkreten Rechner/HW/Controller**

➔ **Berechnung der Komplexität in Rechenschritten (für die Laufzeit)**

- **wie häufig wird eine Operation ausgeführt**
- **wie laufzeitintensiv ist eine Operation im Vergleich zu anderen Operationen (ggf. ausschließliche Betrachtung laufzeitintensiver Operationen)**

➔ **maschinenunabhängiges Maß für die Rechenzeit**

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Allgemeine Form:

f : N -> N (N hier für die natürlichen Zahlen)

wobei f(n) = a steht für: “Ein konkretes Problem der Größe n erfordert Aufwand a.

- **Die “Problemgröße” n bezeichnet dabei meist ein grobes Maß für den Umfang einer Eingabe, z.B.**
  - **die Anzahl der Elemente einer Liste,**
  - **Anzahl Elemente in einem Feld**
  - **Größe eines bestimmten Eingabewertes.**

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Allgemeine Form:

f : N -> N (N hier für die natürlichen Zahlen)

wobei f(n) = a steht für: “Ein konkretes Problem der Größe n erfordert Aufwand a.

- **Der “Aufwand” a ist in der Regel ein grobes Maß für die Rechenzeit (und ggf. den Speicherplatz)**
- **Die Rechenzeit zählt die Anzahl der Rechenoperationen und ggf. Anzahl der Speicherzugriffe,**

➔ Maschinenunabhängiges Maß

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Komplexitätsklassen „O-Notation“

- **Abschätzungen mit vereinfachenden Annahmen**
- **obere Schranke für die Komplexität**
- **abhängig vom Größenparameter n (n nat. Zahl) des Problems**

➔ **Komplexität Problem p(n) <= O (g(n)), d.h. es gibt eine Funktion g(n) in die natürlichen Zahlen für die gilt: die Lösung des Problems p (abhängig vom Größenparameter n) erfordert mindestens g(n) Schritte**

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Komplexitätsklassen „O-Notation“ Mathematische Definition:

$$f(n) \in O(g(n)) \Leftrightarrow \exists c , n_0 : \forall n \geq n_0 : f(n) \leq c * g(n)$$

„f wächst nicht stärker als g“

„f ist beschränkt durch g“

„f(n) hat höchstens die Komplexität g(n)“

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Komplexitätsklassen „O-Notation“ Mathematische Definition:

$$f(n) \in O(g(n)) \Leftrightarrow \exists c , n_0 : \forall n \geq n_0 : f(n) \leq c * g(n)$$

Anwendung bei der Analyse von Algorithmen:

Aufwandsfunktionen $f : N \rightarrow N$ wird durch Angabe einer einfachen Vergleichsfunktion $g : N \rightarrow N$ abgeschätzt.

Algorithmen

Komplexität von Algorithmen

Komplexitätsklassen „O-Notation“

- **O(1) ➔ konstanter Aufwand**
- **O(log n) ➔ logarithmischer Aufwand**
- **O(n) ➔ linearer Aufwand**
- **O(n * log n)**
- **O(n^2) ➔ quadratischer Aufwand**
- **O(n^k) ➔ polyminaler Aufwand (k>2)**
- **O(2^n) ➔ exponentieller Aufwand**

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Komplexitätsklassen „O-Notation“ – Anmerkungen:

- **Summanden werden weggelassen, d.h. O(n+5) = O(n).**
- **Faktoren werden weggelassen, d.h. O(5*n) = O(n)**
- **Basen von Logarithmen werden (meist) weggelassen, d.h. $O(_2\log n) = O(\ln n) = O(_{10}\log n)$**

Algorithmen

Hochschule Deggendorf

Komplexität von Algorithmen

Beispiele:

- **Suchen mittels Hashverfahren*) ➔ O(1)**
- **binäres Suchen in einem sortierten Array ➔ O(log n)**
- **lineares Suchen in einem unsortierten Array ➔ O(n)**
- **Syntaktische Analyse von Programmen (Compiler) ➔ O(n)**
- **Multiplikation Matrix-Vektor ➔ $O(n^2)$**
- **Matrizen-Multiplikation ➔ $O(n^3)$**

*) unter bestimmten Randbedingungen

Algorithmen

Hochschule Deggendorf

Zum Schluss dieses Abschnitts …

Noch Fragen ??

Inhalte

Rekursion

Rekursion

- **Idee: Löse ein Problem dadurch, dass durch eine leicht durchzuführende Aktion das Problem auf einen einfacheren Fall des gleichen Problems zurückgeführt werden kann.**

Rekursion

Hochschule Deggendorf

Rekursion

- **Beispiel: Suchen einer bestimmten Stelle (z.B. Foto) in einem Buch durch Blättern.**

- **Einfache Aktion: Blättern um eine Seite, nachschauen, ob das Foto gefunden ist. Falls nein: Suchen im restlichen Buch.**

Rekursion

Rekursion

- **Weitere Beispiele ?**

Rekursion

Hochschule Deggendorf

Rekursion

- „etwas auf sich selbst zurückführen“
- im Sinn der Programmierung „zurückführen auf einen einfacheren Fall des selben Problems“
- mathematische Definitionen
  - 1 ist eine natürliche Zahl
  - ist n eine natürliche Zahl, dann ist auch n+1 eine natürliche Zahl
- rekursiver Algorithmus
- rekursive Datenstruktur

Rekursion

Hochschule Deggendorf

Rekursion

- **erlaubt eine unendliche Menge von Objekten mit einer endlichen Definition zu beschreiben**
- **erlaubt eine unendliche Anzahl von Berechnungen durch eine endliche Definition zu beschreiben**
- **rekursive Algorithmen beschreiben die Lösung „rekursiver“ Probleme auf einfache Art*)**
- **rekursive Algorithmen arbeiten auf rekursiven Datenstrukturen auf einfache Art*)**

*) der Algorithmus ist dabei nicht immer der effizienteste oder im Extremfall nicht effektiv

Rekursion

Rekursion

- **Im Sinn der Programmiersprache bedeutet Rekursion, dass eine Funktion sich direkt oder indirekt selbst aufruft.**
- **Direkte Rekursion: Funktion ruft sich selbst im eigenen Funktionsrumpf auf.**

Rekursion

Rekursion

- **Beispiel für direkte Rekursion:**

```
long fakultät (unsigned long i /* i > 0 */)
{ if (i==1)
    return 1
  else return i * fakultät(i-1);
}
```

(diese Form entspricht der mathematischen Definition der Fakultät)

Rekursion

Hochschule Deggendorf

Rekursion

- **indirekte Rekusion: Funktion f ruft eine Funktion g auf, die direkt oder indirekt wiederum f aufruft**
  Beispiel:
  - **Eine Zahl ist gerade, wenn sie entweder gleich 0 ist, andernfalls wenn die Zahl verringert um 1 ungerade ist.**
  - **Eine Zahl ist nicht ungerade, wenn sie gleich 0 ist, andernfalls ist sie ungerade, wenn die Zahl verringert um 1 gerade ist.**

Rekursion

Rekursion

- **indirekte Rekusion: Beispiel:**

```
int zahl_ist_gerade (unsigned int i)
  { if (i == 0)
      return 1
    else return zahl_ist_ungerade (i-1)
  };

  int zahl_ist_ungerade (unsigned int i)
  { if (i == 0)
      return 0
    else return zahl_ist_gerade (i-1)
  };
```

Rekursion

Rekursion

- **kaskadenartige Rekusion: In der Funktion f werden mindestens zwei Aufrufe von f „nebeneinander“ aufgerufen.**

  „nebeneinander“ bedeutet hier dass die rekursiven Aufrufe verknüpft sind, z.B. durch einen Operator oder Parameter eines anderen Funktionsaufrufs sind. Der äußere Aufruf kann erst beendet werden, wenn kaskadenartigen rekursiven Aufrufe beendet sind.

Rekursion

Rekursion

- **kaskadenartige Rekusion: Beispiele**

  f(...)
  { ... f(...) + f(...) ... }

  oder

  f(...)
  { ... g(f(...), ... ,f(...)) ... }

Rekursion

Rekursion

- **kaskadenartige Rekusion: Beispiel Fibonacci Zahlen fib (n), n natürliche Zahl**

  Mathematische Definition:
  fib(0) = 0, fib(1) = 1, fib(n, n>1) = fib(n-1)+fib(n-2)

  Lösung in C:

```
long fib(long n)
{ if (n==0) return 0;
  if (n==1) return 1;
  else return(fib(n-1) + fib(n-2));
};
```

Rekursion

Hochschule Deggendorf

Rekursion

- **geschachtelte Rekusion: In der Funktion f wird f in der Form f(..., f(...), ...) aufgerufen.**

  Beispiel: ?

  kommt eher selten vor

Rekursion

- **Rekursion benötigt immer eine Terminierung, d.h. einen Fall, der zu keinen weiteren rekursiven Aufrufen führt (z.B. bei Fakultät i==0)**
- **Mittels Rekursion lassen sich viele Aufgabenstellungen „einfach“ lösen, falls die Aufgabe selbst rekursiv lösbar ist. z.B:**
  - **Fakultät n = n * Fakultät (n-1) für n>1, 1 für n==1**
  - **$a^n = a * a^{(n-1)}$ für n>0, 1 für n==0**
  - **Suchen in einer Liste: Falls Liste leer ➔ nicht gefunden, falls gesuchtes Element gleich dem ersten Listenelement ➔ gefunden, sonst im Rest der Liste weitersuchen.**

Rekursion

Rekursion ➔ Ausflug in den Compilerbau

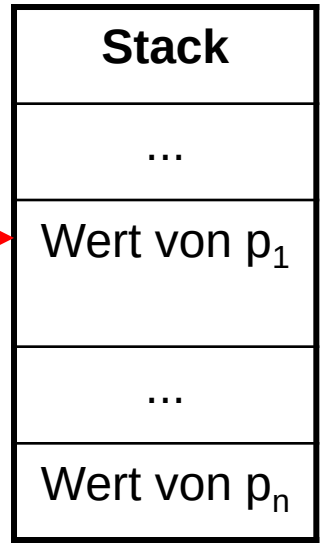
Behandlung von nicht-rekursiven Funktionsaufrufen

C-Code:

```
...
int f(t1 p1, ..., tn pn)
{ ... Rumpf von f ... };
...
int main(void)
{ ...
  x = f(p1, ..., pn);
  ...
}
```

Funktion f wird aufgerufen. Laufzeitsystem merkt sich Rücksprungadress und speichert die aktuellen Parameter p_1, ..., p_n);auf dem Stack.

Die Ausführung „springt“ zum Code von f, der mit dem Parametern auf dem Stack abgearbeitet wird.
Nach der Ausführung von f wird das Ergebnis in x abgelegt und die Parameter auf dem Stack gelöscht (Stack wird freigegeben).
Danach wird an die Rücksprungadresse zurückgekehrt.



Rekursion

Hochschule Deggendorf

Behandlung von rekursiven Funktionsaufrufen

- **„im Prinzip“ genau so, wie nicht-rekursive Funktionsaufrufe**
  aber:
  - **Jeder rekursive Funktionsaufruf hat seinen eigenen Parametersatz auf dem Stack (die Namen der Parameter sind dabei immer gleich)**
  - **Jeder rekursive Funktionsaufruf hat seinen eigenen Satz lokaler Variabler auf dem Stack (die Namen der Variablen sind dabei immer gleich)**

Rekursion

Rekursion ➔ Ausflug in den Compilerbau

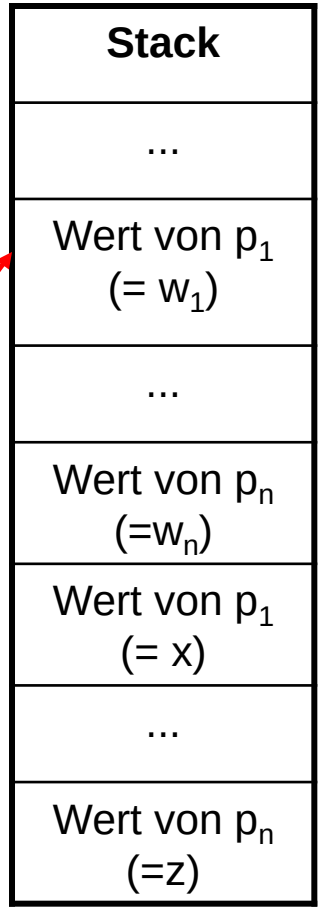
- **Behandlung von rekursiven Funktionsaufrufen**

C-Code:

```
...
 int f(t1 p1, ..., tn pn)
{ ... z = f(x, ..., z)
  ...
};
...
int main(void)
{ ...
  x = f(w1, ..., wn);
  ...
}
```

rekurisver Aufruf von f: Neue Instanzen von p1, ..., pn mit den Werten x, ..., z werden auf dem Stack angelegt. Die Rücksprungadresse wird gemerkt und f wird mit den Parameterwerten x, ..., z ausgeführt. Der „äußere Aufruf von f „hängt“, d.h. wird nicht beendet.
Nach Beendigung des inneren Aufrufs wird der Stack freigegegeben und die Ausführung an der Aufrufstelle fortgesetzt.

Aufruf von f wie beim nicht rekursiven Fall



Rekursion

Hochschule Deggendorf

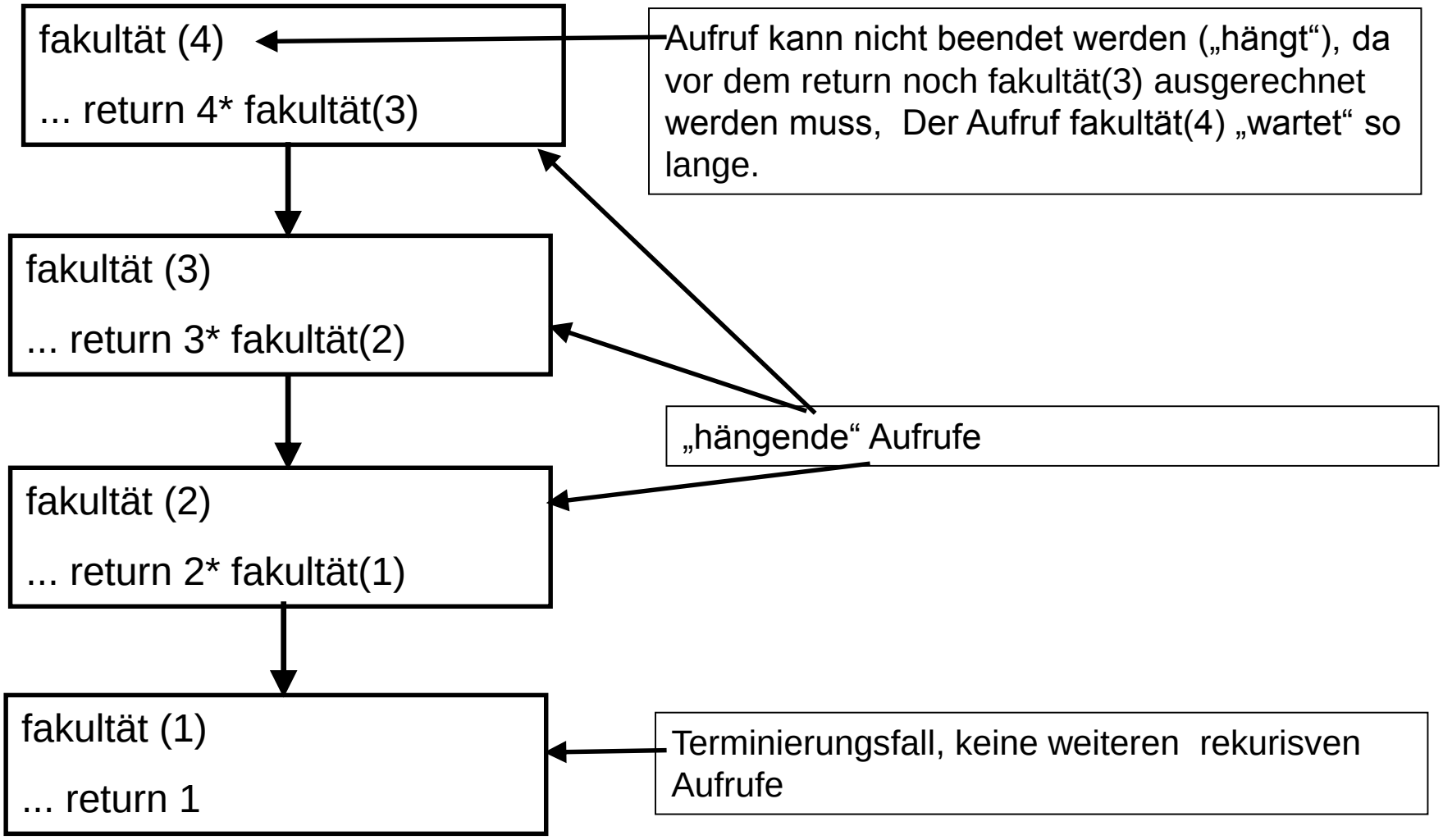
Behandlung von rekursiven Funktionsaufrufen

- **Beispiel fakultät**

```
long fakultät (unsigned long i /* i > 0 */)
{ if (i==1)
    return 1
  else return i * fakultät(i-1);
}
```

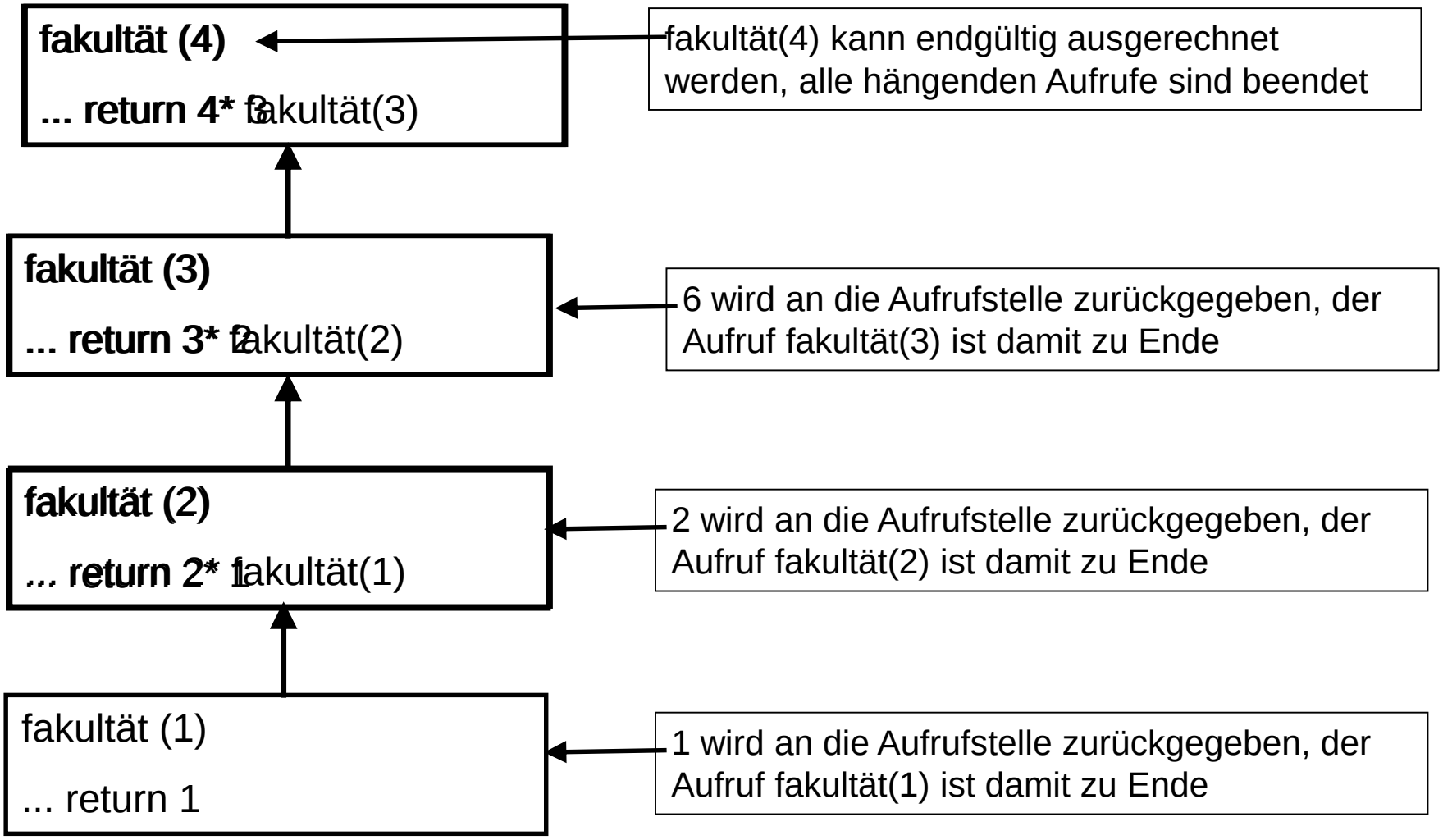
Rekursion

- **Ablauf zur Laufzeit am Beispiel fakultät**



Rekursion

- Ablauf zur Laufzeit am Beispiel fakultät (4)



Rekursion

Hochschule Deggendorf

Behandlung von rekursiven Funktionsaufrufen

- **jeder rekursive Aufruf besitzt seine eigenen Parameter, auch wenn diese den selben Namen wie in der aufrufenden Funktion haben**
- **jeder rekursive Aufruf besitzt seine eigenen lokalen Variablen (Ausnahme static)**
- **Parameter und lokale Variable bleiben so lange gültig, bis der rekursive Aufruf beendet ist**

Rekursion

```
long ggt(long a, long b)
   { if (a==0)
        return b;
     else if (b==0)
           return a;
     else if (a>=b)
          return ggt(a-b,b);
        else return ggt(b-a,a);
   };

int main(int argc, char *argv[])
{
   long x,y = 0;
   long result = 0;

  ...
   result = ggt(x,y);
   ...

}
```

formale Parameter a,b der Funktion

- definieren Parametertyp und Name
- sind nur innerhalb der Funktion bekannt

return gibt das Ergebnis der Funktion (sofern ein Ergebnistyp definiert ist) an den Aufrufer zurück.

Der Typ des Ergebnisses ist in der Funktion festgelegt, z.B. long bei long ggt (...)

aktuelle Parameter x,y der Funktion

- werden an Stelle der formalen Parameter verwendet, d.h. z.B. überall wo der formale Parameter a im Rumpf der Funktion vorkommt, wird beim Aufruf der Funktion die Variable x „eingesetzt“
- könne, müssen aber nicht den gleichen Namen wie die formalen Parameter der Funktion haben
- können auch Konstante sein, z.B. Aufruf ggt(10,6)

Rekursion

Terminierung, d.h.

keinen weiteren rekursiven Aufrufe

```
long ggt(long a, long b)
   { if (a==0)
       return b;
     else if (b==0)
          return a;
     else if (a>=b)
            return ggt(a-b,b);
          else return ggt(b-a,a);
   };

int main(int argc, char *argv[])
{
   long x,y = 0;
   long result = 0;

  ...
   result = ggt(x,y);
   ...

}
```

rekursive Aufrufe der Funktion ggt

- ggt ruft sich selbst auf mit neuen Parametern
- der „äußere“ Aufruf wartet, bis der „innere“ rekursive Aufruf beendet ist

Rekursion

```
long ggt(long a, long b)
   { if (a==0)
        return b;
     else if (b==0)
            return a;
     else if (a>=b)
           return ggt(a-b,b);
         else return ggt(b-a,a);
   };
```

Aufrufreihenfolge am Beispiel ggt(10,6)
ggt(10,6) führt zum rekursiven Aufruf ggt(10-6,6), da der Wert der Parameters a (=10) >= Wert des Parameters b (=6). ggt(10-6,6) = ggt(4,6)
ggt(4,6), führt zum rekursiven Aufruf ggt(6-4,4), da der Wert der Parameters a (=4) < Wert des Parameters b (=6). ggt(6-4,4) = ggt(2,4)
ggt(2,4), führt zum rekursiven Aufruf ggt(4-2,2), da der Wert der Parameters a (=2) < Wert des Parameters b (=4). ggt(4-2,2) = ggt(2,2)
ggt(2,2), führt zum rekursiven Aufruf ggt(2-2,2), da der Wert der Parameters a (=2) >= Wert des Parameters b (=2). ggt(2-2,2) = ggt(0,2)
ggt(0,2) führt zu keinem weiteren Aufruf, da Parameter a=0 (Terminierungsfall!). Als Ergebnis wird 2 zurückgegeben (return ggt...) an den vorherigen Aufruf ggt(2,2). Dieser gibt 2 zurück an seinen Aufrufer ggt(2,4) usw. bis als Gesamtergebnis des Aufrufs ggt(10,6) 2 zurückgegeben wird

Rekursion

Hochschule Deggendorf

Einsatzgebiete der Rekursion

- ☺ **Umgebungen, in denen ausreichend Speicherplatz auf dem Stack zur Verfügung steht ➔ PC Umgebungen, „sehr große“ eingebettete Systeme (Multimediasysteme im Kfz)**
- ☹ **Systeme mit „wenig“ Speicherplatz, z.B. Mikrocontroller im Kfz**

Rekursion

Hochschule Deggendorf

Schlußbemerkung:

- **Rekursive Funktionen können in äquivalente nicht-rekursive Programme übergeführt werden („Entrekursivierung“).**

  ➔einfach bei einfacher Rekursion

  ➔komplex bei kaskadenartiger Rekursion (erfordert in der Regel einen eigenen Stack zur Verwaltung)

Übung

McCarthy‘s 91 Funktion

Beschreibung:

Aufgabe:
Implementieren Sie die Funktionen
f(x) := x-10 falls x>100, sonst f(f(x+11))
und
g(x) := x-10 falls x>100, sonst 91

1.) Testen Sie die Funktionen und vergleichen Sie die Resultate. Welche Vermutung ergibt sich aus den Resultaten? Beweisen Sie Ihre Vermutung durch Vollständige Induktion für alle natürlichen Zahlen.

2.) Erweitern Sie Ihre Implementierung um einen Zähler für die rekursiven Aufrufe.

Zeit:

Rekursion

Hochschule Deggendorf

Zum Schluss dieses Abschnitts …

Noch Fragen ??