

Vorlesung Wirtschaftsinformatik

Prof. Dr. Clemens Espe, MBA
Fakultät für Informatik
Wirtschaftsinformatik
Hochschule Augsburg



Inhaltsverzeichnis

4. Algorithmen

1. Definition und Beschreibung von Algorithmen
2. Komplexität und O-Notation
3. Rekursive Algorithmen
4. Suchalgorithmen

5. Datenstrukturen

1. Elementare Datenstrukturen und Listen
2. Binäre Suchbäume
3. Balancierte Suchbäume
4. Hashtabellen

6. Datenkommunikation

1. Kommunikationssysteme
2. Internet und Web 2.0

Kapitel 4: Algorithmen

4.1. Definitionen und Beschreibung von Algorithmen

Kapitel 4: Algorithmen

Eigenschaften von Algorithmen

1. **Finitheit:** Der Algorithmus kann in einer endlichen Zeit beschrieben und beendet werden.
2. **Dynamische Finitheit:** Das Verfahren darf zu jedem Zeitpunkt nur endlich viel Speicherplatz benötigen.
3. **Ausführbarkeit:** Jeder Schritt des Verfahrens muss tatsächlich ausführbar sein.
4. **Terminierung:** Das Verfahren muss in endlich vielen Schritten abschließbar sein.
5. **Determiniertheit:** Jeder Einzelschritt muss eindeutig, verständlich, vollständig und exakt beschreibbar sein und unter identischen Voraussetzungen immer zum gleichen Ergebnis führen.

Kapitel 4: Algorithmen

Vorgehen beim Entwurf von Algorithmen

Ausgangspunkt zum Entwurf von Algorithmen ist in der Informatik generell ein Problem, das mit Hilfe einer Maschine/eines Computers gelöst werden muss.

Auch wenn dies ein kreativer Prozess ist, sollten folgende Schritte berücksichtigt werden:

- 1. Spezifikation des Problems:** Es geht darum **vollständig, detailliert und unzweideutig** das zu beschreiben und **was** zu machen ist (aber nicht, **wie** es zu machen ist).
- 2. Abstraktion des Problems:** Die Spezifikation wird derart in eine **formale Form** gebracht, dass eine Bearbeitung mit einer DV-Anlage möglich ist.
- 3. Eigentlicher Algorithmenentwurf:** Dabei wird häufig auf Beschreibungswerkzeuge wie Pseudocode, Ablaufdiagramm und Struktogramm zurückgegriffen
- 4. Nacharbeiten:** Dies beinhaltet idealerweise einen Korrektheitsnachweis, die Komplexitätsanalyse und die Dokumentation des Algorithmus.

Kapitel 4: Algorithmen

Gruppenarbeit zu einem Algorithmus

Aufgabe:

Erarbeiten Sie mit Ihrem Nachbarn einen Algorithmus, mit dessen Hilfe Sie die durchschnittliche Programmiererfahrung aller Personen in einem Raum bestimmen können.

Beschreiben Sie dabei die einzelnen Schritte des Algorithmus in einfachen Sätzen.

Zeit: 10 Minuten

Ergebnis:

- 1) Hebe 0 Jahre für Programmiererfahrung und 0 für Anzahl Befragter
- 2) Gehe zu einer Person, die du bisher noch nicht gefragt hast, und frage sie nach ihrer Programmiererfahrung in Jahren
- 3) Addiere individuelle PE zu Gesamtprogrammiererfahrung und erhöhe Anzahl Befragter
- 4) Wiederhole Schritte 2 und 3 solange, bis alle Personen befragt sind
- 5) Teile die Gesamtprogrammiererfahrung durch die Anzahl der Befragten
- 6) Das Ergebnis ist die \varnothing Programmierserfahrung

Kapitel 4: Algorithmen

Beschreibung von Algorithmen

Zur Beschreibung von Algorithmen wird häufig auf so genannten Pseudocode und auf Ablaufdiagramme oder Struktogramme zurückgegriffen.

Pseudocode beinhaltet dabei:

- **Syntaktische Elemente:** Sind angelehnt an gängige Programmiersprachen, müssen aber nicht den genauen Anforderungen konkreter Sprachen entsprechen. Ziel sind hierbei die Verständlichkeit, Eindeutigkeit und Übersichtlichkeit und weniger die Lauffähigkeit in einer konkreten Sprache.
- Ggf. umgangssprachliche Einschübe, die im weiteren Entwicklungsprozess formalisiert werden können
- Eine sehr hilfreiche Veranschaulichung von Algorithmen und Programmabläufen stellen Ablaufdiagramme und Struktogramme dar

Beschreibung von Algorithmen in Pseudocode

Datentypen, Anweisungen

Datentypen: Auch im Pseudocode werden die gängigen Datentypen verwendet:

char: Alle darstellbaren Zeichen,

int: Alle ganzen darstellbaren Zahlen,

float: Alle darstellbaren Gleitkommazahlen mit einfacher Genauigkeit,

double: Alle darstellbaren Gleitkommazahlen mit doppelter Genauigkeit,

array: Zusammenfassung zusammengehöriger Daten gleichen Typs,

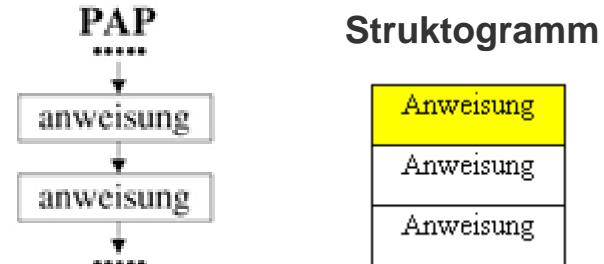
Anweisungen: Typische Anweisungen:

Zuweisung: Variable := Ausdruck;

 q:= vaterzeiger(p);

Abfragen: p = nil, x < n,

Aufrufe: Procedure: suchen (Wert), Funktion: Alter := ermittle.Alter(Name)



Beschreibung von Algorithmen in Pseudocode

Operationen: Verzweigung mit if

If bedingung

then begin

anweisung;

anweisung;

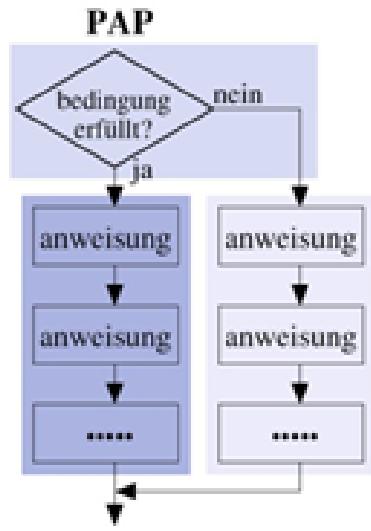
end

else begin

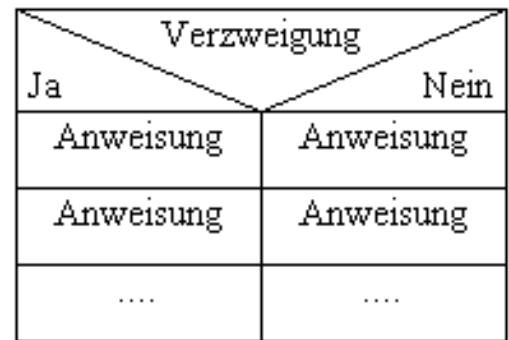
anweisung;

anweisung;

end



Struktogramm



Beschreibung von Algorithmen in Pseudocode

Operationen: Verzweigung mit switch

Switch bedingung

```
case auswahl 1 begin
```

```
    anweisung;
```

```
    anweisung;
```

```
end
```

```
case auswahl 2 begin
```

```
    anweisung;
```

```
    anweisung;
```

```
end
```

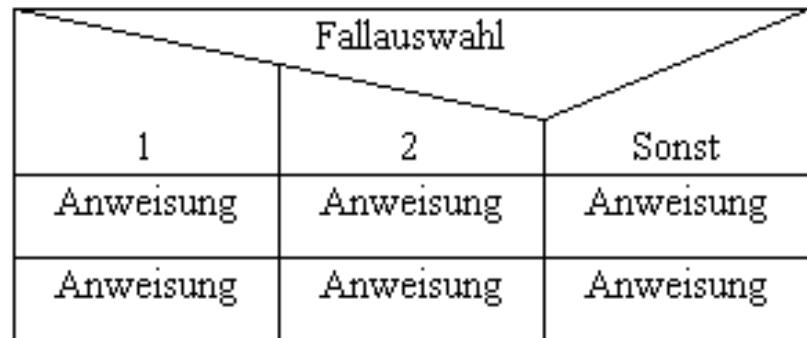
```
default sonst begin
```

```
    anweisung;
```

```
    anweisung;
```

```
end
```

Struktogramm



Beschreibung von Algorithmen in Pseudocode

Operationen: Kopfgesteuerte Schleifen (z. B. if, while)

For Zählvariable := von **to** bis

begin

 anweisung;

 anweisung;

end

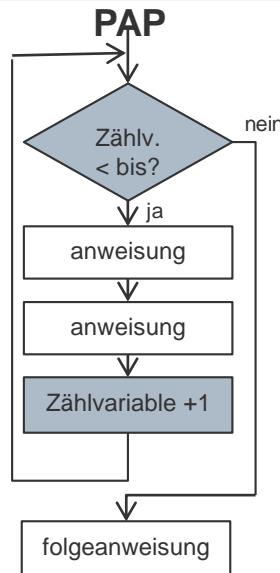
While bedingung **do**

begin

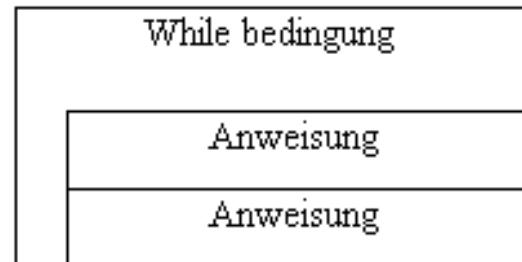
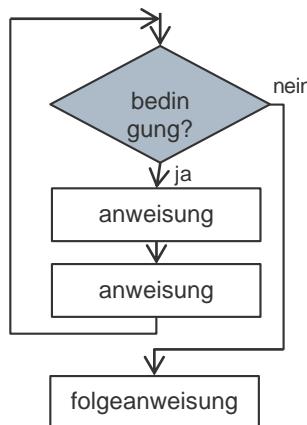
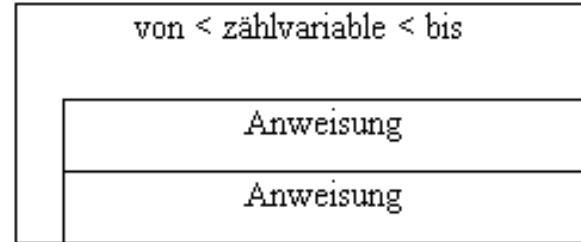
 anweisung;

 anweisung;

end



Struktogramm



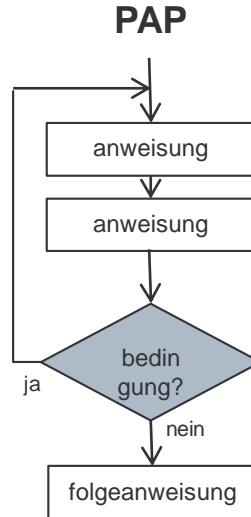
Beschreibung von Algorithmen in Pseudocode

Operationen: Fußgesteuerte Schleife Repeat Until

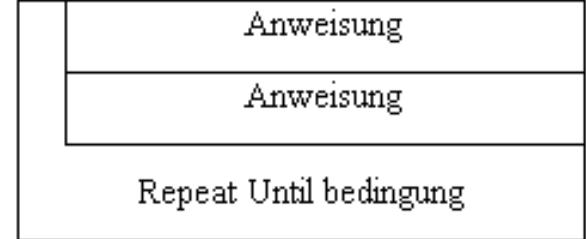
Repeat

anweisung;
anweisung;

Until bedingung



Struktogramm

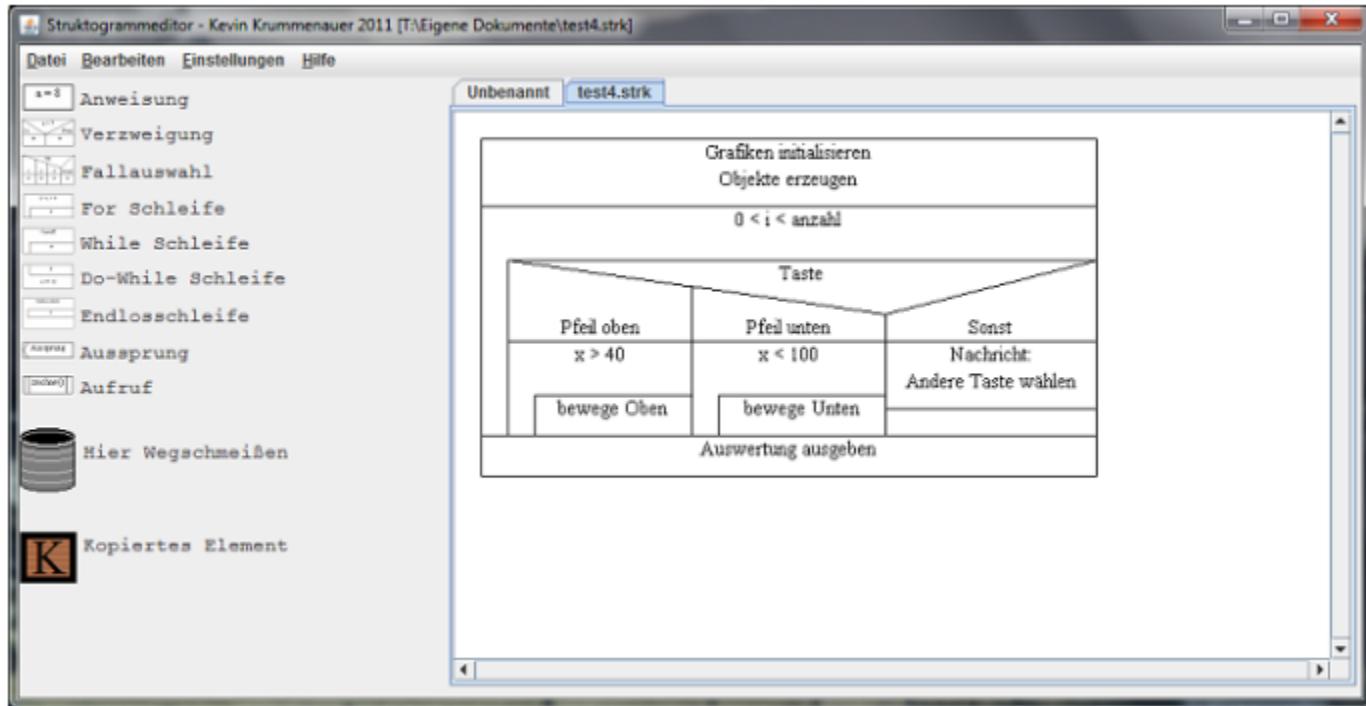


Beschreibung von Algorithmen

Hilfsmittel zur einfachen Erstellung von Struktogrammen

Unter der folgenden Internetadresse finden Sie eine Freeware, mit der sehr einfach Struktogramme erstellt werden können:

<http://whiledo.de/index.php?p=struktogrammeditor>



Kapitel 4: Algorithmen

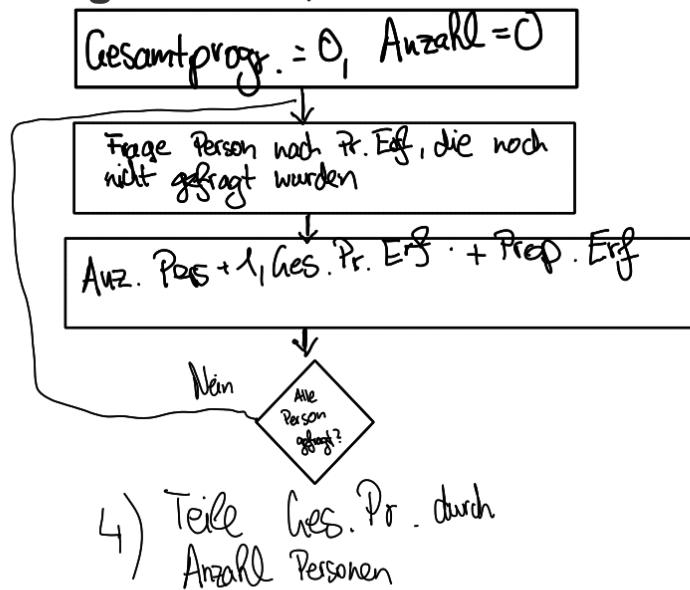
Gruppenarbeit zur Beschreibung von Algorithmen

Aufgabe:

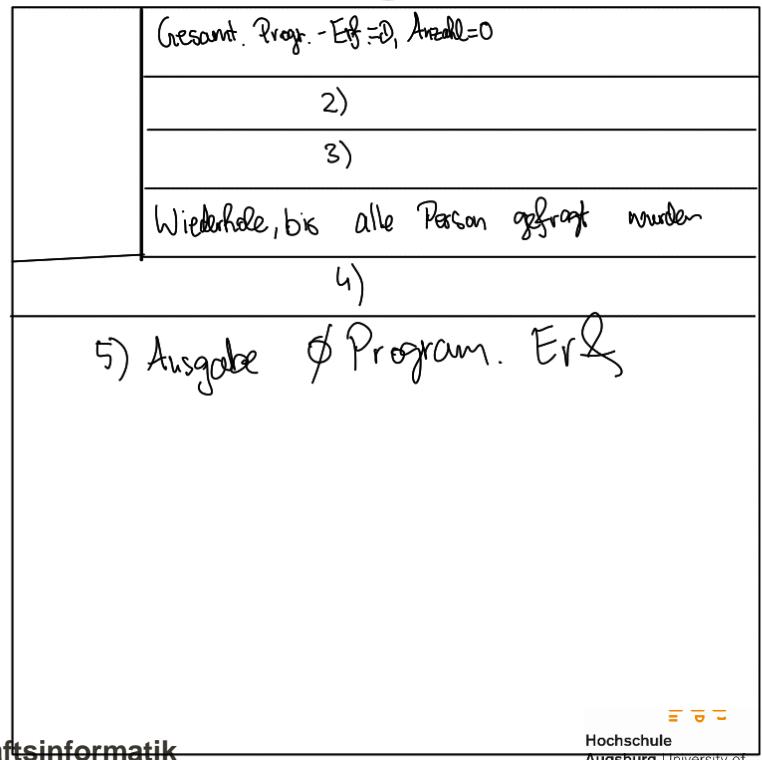
Zeichnen Sie zusammen mit Ihrem Nachbarn das zugehörige Ablaufdiagramm und Struktogramm zum in der vorigen Aufgabe ermittelten Algorithmus.

Zeit: 10 Minuten

Ergebnis: PAP



Struktogramm



Kapitel 4: Algorithmen

4.2. Komplexität und O-Notation

Um die **Brauchbarkeit von Algorithmen**, insbesondere für Speziallösungen mit beschränkten Rechner- und Speicherkapazitäten, abschätzen zu können, wird häufig eine Betrachtung von deren **Zeit- und Speicherplatzverhalten** durchgeführt.

Folgende **Rahmenbedingungen** haben dabei einen starken Einfluss:

1. Art und Geschwindigkeit des **Rechners**
2. **Qualität des Codes**, der von Programmierer und Compiler erzeugt wird
3. Konkrete Situation der **Eingabedaten** (worst, average, best case)

Deshalb empfiehlt sich eine nähere Analyse von:

- **Laufzeitkomplexität** und
- **Speicherplatzbedarf**

des zu untersuchenden Algorithmus um dessen **Effizienz** beurteilen zu können.

Komplexität und O-Notation

Zeitverhalten von Algorithmen

Beispiel:

Die maximale Abschnittssumme einer Zahlenreihe ergibt sich aus der Teilfolge von aufeinander folgenden Zahlen, die unter allen möglichen Teilsummen die größte Summe bildet:

z. B. gegeben: 45, -33, 15, -26, -13, 55, 33, -13, 35, -67, 25, 13

dann bilden die folgenden Zahlen die maximale Abschnittssumme:

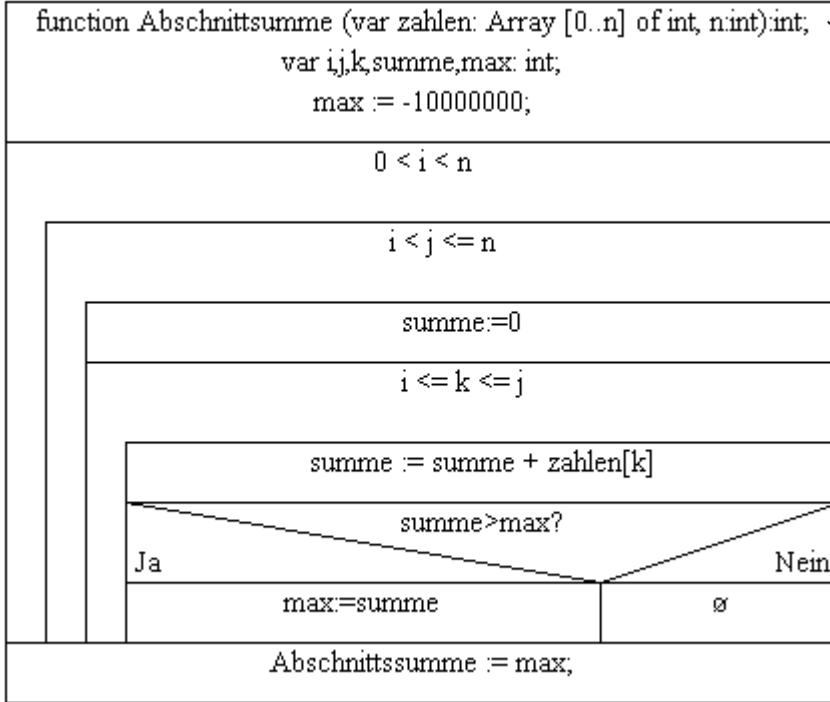
$$55, 33, -13, 35 = 110$$

Zur Veranschaulichung des Laufzeitverhaltens sollen im folgenden hierzu 3 unterschiedliche Algorithmen zur Lösung angewandt werden

Komplexität und O-Notation

Zeitverhalten von Algorithmen: Kubische Algorithmen

Ein erster Ansatz soll in einer dreifachen Schleifenprogrammierung liegen:



1. Schleife wird n -mal durchlaufen,
2+3. Schleife $\frac{n}{2}$ mal
→ Zeitbedarf ist proportional zu n^3
→ Kubischer Algorithmus
→ Algorithmus nicht effizient, da bereits errechnete Summen neu berechnet werden
→ Kub. Algorithmen haben sehr lange Laufzeiten und eignen sich nur für sehr kleine Probleme

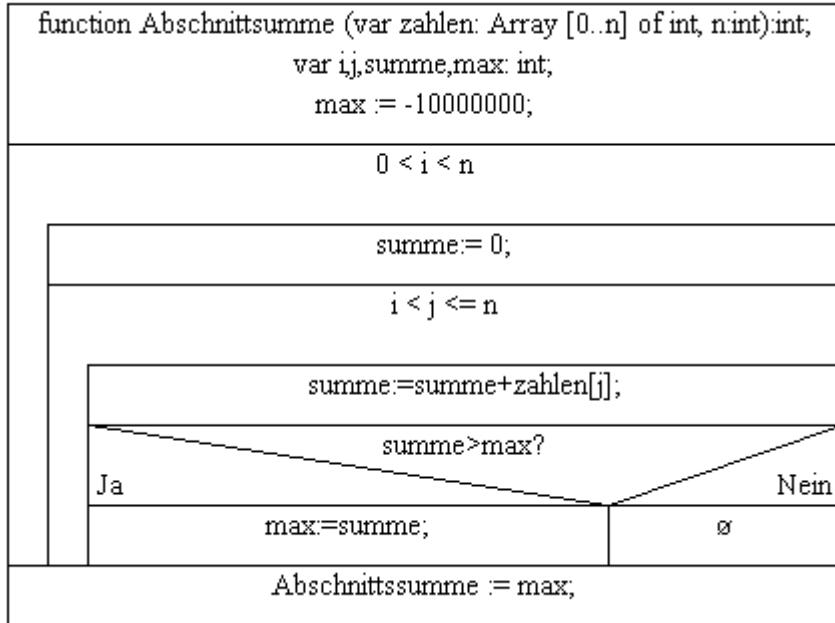
Aufgabe: Diskutieren Sie mit Ihrem Nachbarn, was der dargestellte Algorithmus macht. Wie oft werden die Anweisungen durchlaufen?



Komplexität und O-Notation

Zeitverhalten von Algorithmen: Quadratische Algorithmen

Im zweiten Ansatz gibt es nur noch eine zweifache Schleifenprogrammierung:



- Zurückgreifen auf berechnete Summen
- Erste Schleife wird n -durchlaufen,
2. $\frac{n^2}{2}$ mal
 - Zeitbedarf ist proportional zu n^2
 - quadr. Algorithmus
 - bereits dramatische Verbesserung

Aufgabe: Diskutieren Sie mit Ihrem Nachbarn, was der dargestellte Algorithmus macht. Wie oft werden die Anweisungen durchlaufen?



Komplexität und O-Notation

Zeitverhalten von Algorithmen: Lineare Algorithmen

Dritter Ansatz: Optimierung auf eine Schleife:

→ Die größte Abschnittssumme wird immer in

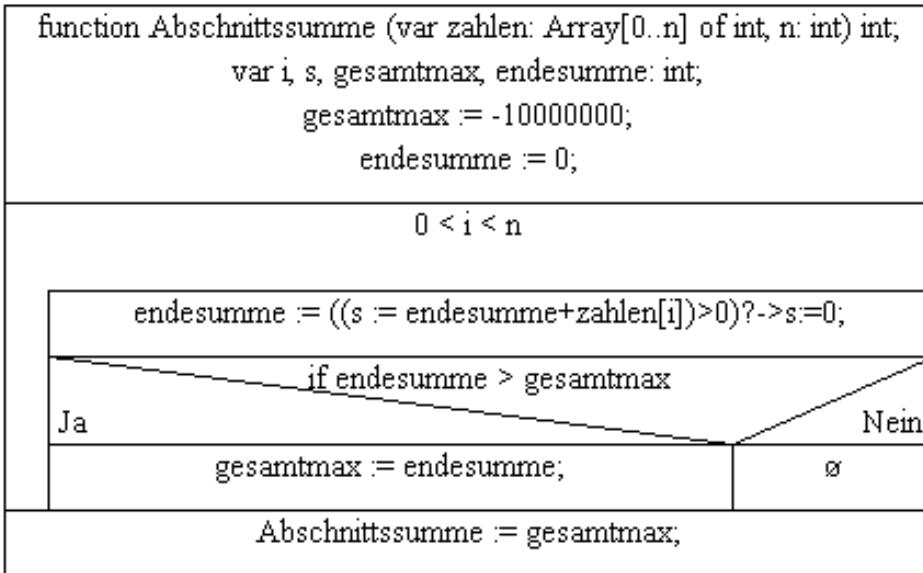
→ In Schleife wird die Abschnittssumme des aktuellen Teilstücks in Ende Summe berechnet.

Wenn sie größer ist, Überschreitet sie Gesamt

→ Wenn Endsumme < 0, dann wird sie aufgesetzt

→ nur in Durchläufe

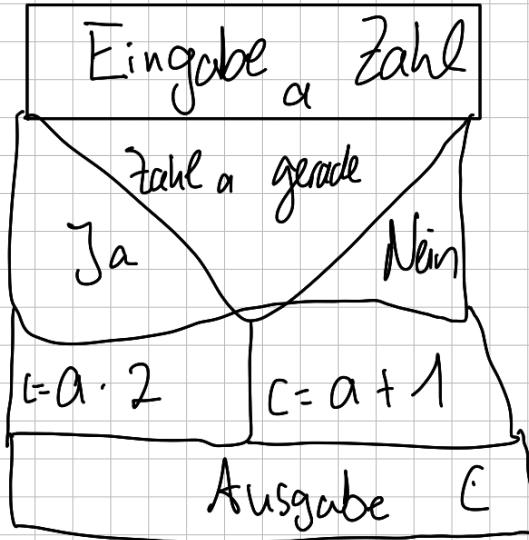
→ lineare Algorithmen



Aufgabe: Diskutieren Sie mit Ihrem Nachbarn, was der dargestellte Algorithmus macht. Wie oft werden die Anweisungen durchlaufen?



Strukturgramm



Komplexität und O-Notation

Zeitverhalten von Algorithmen: Zusammenfassung

Um das Laufzeitverhalten eines Algorithmus allgemeingültig (unabhängig von Rechnerleistung und zu verarbeitenden Daten) zu beschreiben, ist eine Angabe in Zeit (sec.) ungünstig.

Sinnvoller ist es, das Zeitverhalten als Funktion der Größe der Eingabedaten n anzugeben.

Beispiel polynomiale Laufzeitkomplexität:

$$T(n) = a_k * n^k + a_{k-1} * n^{k-1} + \dots + a_1 * n^1 + a_0$$

Für kleine n spielen hier die Komponenten a_0 und $a_1 * n$, für sehr große n aber die Komponente $a_k * n^k$.

O-Notation: Weist ein Modul ein Verhalten auf, das sich für große Zahlen proportional zu n verhält, spricht man von der Komplexität $O(n)$

Komplexität und O-Notation

Typische Komplexitätsfunktionen zu Algorithmen

Typische Komplexitätsfunktionen zu Algorithmen

1	<i>konstant</i>	Jede Anweisung eines Programms wird höchstens einmal ausgeführt. Dies ist der Idealzustand für einen Algorithmus.
$\log n$	<i>logarithmisch</i>	Speicher- oder Zeitverbrauch wachsen nur mit der Problemgröße n . Die Basis des Logarithmus wird häufig 2 sein, d. h. vierfache Datenmenge verursacht doppelten Ressourcenverbrauch, 8-fache Datenmenge verursacht 3-fachen Verbrauch und 1024-fache Datenmenge 10-fachen Verbrauch.
n	<i>linear</i>	Speicher- oder Zeitverbrauch wachsen direkt proportional mit der Problemgröße n .
$n \log n$	$n \log n$	Der Ressourcenverbrauch liegt zwischen n (<i>linear</i>) und n^2 (<i>quadratisch</i>).
n^2	<i>quadratisch</i>	Speicher- oder Zeitverbrauch wachsen quadratisch mit der Problemgröße. Solche Algorithmen lassen sich praktisch nur für kleine Probleme anwenden.
n^3	<i>kubisch</i>	Speicher- oder Zeitverbrauch wachsen kubisch mit der Problemgröße. Solche Algorithmen lassen sich in der Praxis nur für sehr kleine Problemgrößen anwenden.
2^n	<i>exponentiell</i>	Bei doppelter, dreifacher und 10-facher Datenmenge steigt der Ressourcenverbrauch auf das 4-, 8- bzw. 1024-fache. Solche Algorithmen sind praktisch kaum verwendbar.

Quelle: Herold/Lurz/Wohlrab 2012



\mathcal{O} -Notation

- Maß für Performance eines Algorithmus
 $\mathcal{O}(n^2)$ quadratisches Wachstum



Datensetze

- $\mathcal{O}(n \cdot \log n)$ schneller als $\mathcal{O}(n^2)$ bei größer werdenden Zahlen

$$\mathcal{O}(1) = \text{konsstant}$$

$$\mathcal{O}(\log N) =$$

$$\mathcal{O}(n) =$$

$$\mathcal{O}(n \cdot \log n)$$

$$\mathcal{O}(n^2)$$

$$\mathcal{O}(n^3)$$

$$\mathcal{O}(n^c)$$

$$\mathcal{O}(c^n)$$

$$\mathcal{O}(n!)$$

$$\mathcal{O}(\infty)$$

$$\mathcal{O}(\log n) = \mathcal{O}\left(\frac{\log N}{\log c}\right)$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in \{0, \infty\}$$

$$\text{Bsp.: } \mathcal{O}(n); \mathcal{O}\left(\frac{n}{2}\right) \lim_{n \rightarrow \infty} \frac{n}{\frac{n}{2}} = \lim \frac{1}{\frac{1}{2}} = 2$$

$$\lim_{n \rightarrow \infty} \frac{n^2}{n^3} = \lim_{n \rightarrow \infty} \frac{n \cdot n}{n \cdot n \cdot n} = \frac{1}{n} = 0$$

Kapitel 4: Algorithmen

Gruppenarbeit zu logarithmischen Algorithmen

Aufgabe:

Überlegen Sie sich mit Ihrem Nachbarn ein Anwendungsbeispiel für einen logarithmischen Algorithmus und geben Sie Beispiele für die Anzahl der Berechnungen für verschiedene Eingaben an.

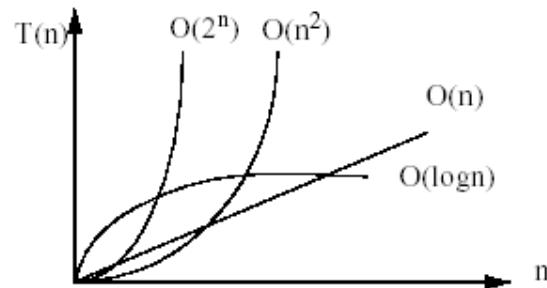
Zeit: 10 Minuten

Ergebnis: Bsp.: Logarithmischer Algorithmus
Namenssuche Telefonbuch

Suche in alphabetisch sortieren TB einen Namen ergibt

Komplexität und O-Notation

Typische Komplexitätsfunktionen in Abhängigkeit von n



n	$\lg(n)$	\sqrt{n}	$n * \lg(n)$	$n * \lg^2(n)$	n^2	n^3	e^n
10	3	3	33	110	100	1000	22026
100	7	10	664	4414	10000	1000000	3E+43
1000	10	32	9966	99317	1000000	1000000000	oo oo
10000	13	100	132877	1765633	100000000	1E+12	oo
100000	17	316	1660964	27588016	1E+10	1E+15	oo
1000000	20	1000	19931569	397267426	1E+12	1E+18	oo

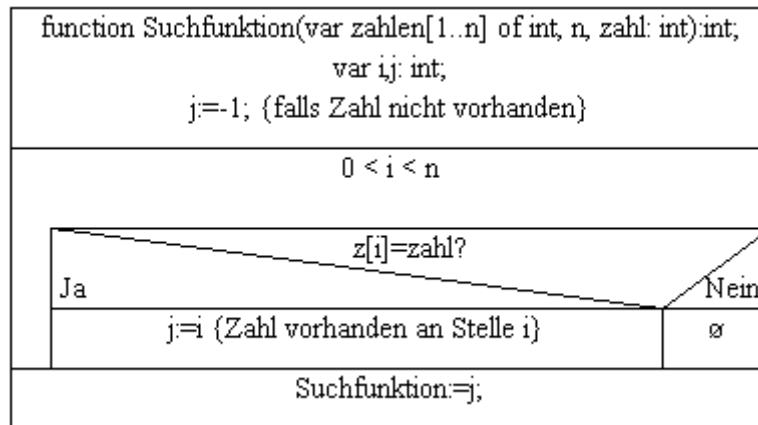
-> In der Praxis sind Algorithmen mit Komplexitäten schlechter $O(n^2)$ (z. B. $O(n^3)$, $O(x^n)$) nur für sehr kleine Problemstellungen geeignet!!!



Komplexität und O-Notation

Günstigster, durchschnittlicher und ungünstigster Fall

Gegeben ist ein Algorithmus zur Suche einer definierten Zahl in einem Array:



Wir können 3 Fälle unterscheiden:

Best Case: Die gesuchte Zahl ist in erster Stelle im
im Array $\rightarrow O^1$

Average Case: Die gesuchte Zahl befindet sich in der Mitte
 $\rightarrow \frac{n}{2}$ Vergleiche $\rightarrow O(\frac{n}{2})$

Worst Case: Die gesuchte Zahl befindet sich an letzter Position oder gar
nicht im Array $\rightarrow n$ -Vergleiche $\rightarrow O(n)$

Häufig ist der Nachweis der Terminierung eines Algorithmus wichtig.

Vorgehensweise:

- Analysiere die Menge der zu bearbeitenden Elemente
- weise nach, dass bei jedem Aufruf der Rekursion die Menge kleiner wird

Oder

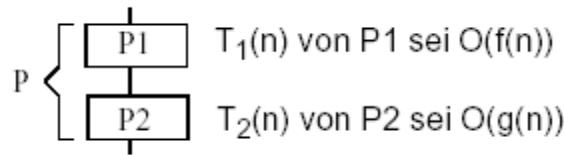
- Weise nach, dass Ergebniswert streng monoton fallend (steigend) ist.
- Zeige, dass der Algorithmus bei Erreichen einer unteren (oberen) Grenze terminiert.

① Gilt nur für rekursive Algorithmen

② Gilt nur für alle anderen Algorithmen

Regeln zur Berechnung der Laufzeitkomplexität

1. Als ersten Schritt werden kleine Programmteile identifiziert, deren Laufzeit unabhängig von der Problemgröße (worst/best case) sind. Ihre Laufzeit ist $O(1)$.
2. Für Programmteile, die hintereinander ablaufen, gilt die **Summenregel**:

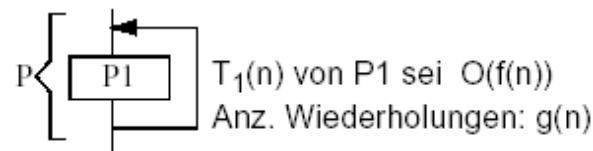


Gesamlaufzeit P:

$$T_{(n)} = T_{1(n)} + T_{2(n)}$$

$$O_{(p)} = \text{Max} (O_{f(n)}, O_{g(n)})$$

3. Schleifen im Programm: **Produktregel**:



Gesamlaufzeit P:

$$T_{(n)} = g(n) * T_{1(n)}$$

$$O_{(p)} = O(g(n) * O(f_{(n)}))$$

Komplexitätsbestimmung von Algorithmen (I)

Beispiel: Algorithmus zur Berechnung der Anzahl von Tripel in einem Array, die sich auf 0 aufsummieren:

```
int N = a.length;    E
int count = 0;
for (int i = 0; i < N; i++) D
    for (int j = i+1; j < N; j++) C
        for (int k = j+1; k < N; k++) B
            if (a[i] + a[j] + a[k] == 0) A
                count++;
            E
```



Komplexitätsbestimmung

Block Häufigkeit Gesamtzeit

E x (hängt von
Eingabe ab) $\asymp N^3$

① $\frac{N^3}{6} + \frac{N^2}{2} + \frac{N}{3}$

$x \in \mathbb{O}$

$t_1 \left(\frac{N^3}{1} - \frac{N^2}{2} + \frac{N}{3} \right)$

C $\frac{N^2}{2} + \frac{N}{2} \asymp N^2$

$t_2 \left(\frac{N^2}{2} + \frac{N}{2} \right)$

B N

$t_3 + N$

A 1

t_4

Kapitel 4: Algorithmen

Gruppenarbeit zu logarithmischen Algorithmen

Aufgabe:

Bestimmen Sie die O-Notation zu folgenden Beispielen:

1) $2n^2 + 5n + 12$

$$O(n^2)$$

2) $5n^5 - 27n^3 + 2n - 5112$

$$O(n^5)$$

3) $2^n + n^2$

$$O(2^n)$$

4) $n^2 \log^2 n + 10 \frac{n^3}{\log n} + 5n$

$$O\left(\frac{n^3}{\log n}\right)$$

Zeit: 5 Minuten

Ergebnis:

Charakteristika von Algorithmen

Iteration: Ein Algorithmus heißt iterativ, wenn bestimmte Abschnitte des Algorithmus innerhalb einer einzigen Ausführung des Algorithmus mehrfach durchlaufen werden.

Realisiert werden Iterationen meist durch Schleifen (for, while, ...)

Rekursion: Ein Algorithmus heißt rekursiv, wenn er Abschnitte enthält, die sich selbst direkt oder indirekt aufrufen. Dabei wird aus einer zu bearbeitenden Menge von Elementen jeweils ein Element entnommen, dieses bearbeitet und das verfahren wiederum auf die Restmenge angewendet.

Direkt rekursiv: Der Anweisungsteil von P enthält explizit einen Aufruf von P

Indirekt rekursiv: Der Anweisungsteil von P enthält den Aufruf eines Unterprogramms Q, das seinerseits P direkt oder indirekt aufruft.

Linear rekursiv: Der Anweisungsteil von P führt nur zu maximal einem Aufruf von P (dynamisch betrachtet)



Beispiele

Achtung: Fallstricke beim Bestimmen des Laufzeitverhaltens

- **Große Konstanten:**

Befinden sich große konstante Koeffizienten in Termen niedriger Ordnung, können diese ggf. nicht mehr ignoriert werden.

Beispiel: $aN^2 + bN \sim aN^2$ -> gilt nicht für sehr große b und kleine a

- **Befehlszeit:**

Diese sind nicht immer konstant. Verfahren, wie Caching, die den Datenzugriff verkürzen können z. B. bei mehrfachem Zugriff auf die gleichen Daten zu erheblichen Zeitverkürzungen führen.

- **Abhängigkeit von Eingaben:**

Wie in der best, average und worst case Betrachtung gezeigt, können diese erheblichen Einfluss auf die Laufzeit haben.

- **Systembetrachtung:**

Die Laufzeit hängt häufig stark von der aktuellen Systemsituation ab (z. B. parallel starke Rechnerbelastung, Downloads etc.)



Kapitel 4: Algorithmen

4.3. Rekursive Algorithmen

Baumstrukturen

Exkurs: Rekursion (1)

Definition: Eine Funktion heißt rekursiv, wenn ihr Anweisungsteil mindestens einen Aufruf von sich selbst enthält

Warum Rekursion?

- Wirkungsvolle, ungewohnte Herangehensweise
- Kann Code einfacher, verständlicher und effizienter machen

Typische Anwendungen:

- Kombinatorische Suche über Baum-Datenstrukturen
- Fast Fourier-Transformation in der Signalverarbeitung

Hinweis:

Die Rekursion steht in direktem Bezug mit der mathematischen Induktion.

Die mathematische Induktion kann verwendet werden, um die Richtigkeit einer rekursiven programmtechnischen Umsetzung nachzuweisen.



Exkurs: Rekursion (2)

Beispiel Berechnung der Fakultät von n (n!)

$$n! = 1 * 2 * 3 * \dots * n$$

Rekursiv:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n - 1)!, & n > 0 \end{cases}$$

Implementierung:

Procedure faku(N: integer);

begin

if (N = 0) then return (1)
 else return(faku(N-1)*N)

end



Geben Sie eine natuerliche Zahl ein: 3

N = 3

faku(3)

begin

if (N = 0) then return (1)
else return(faku(N-1)*N)

end



N = 3

faku(3)

begin

if (N = 0) then return (1)
else return(faku(N-1)*N)

end

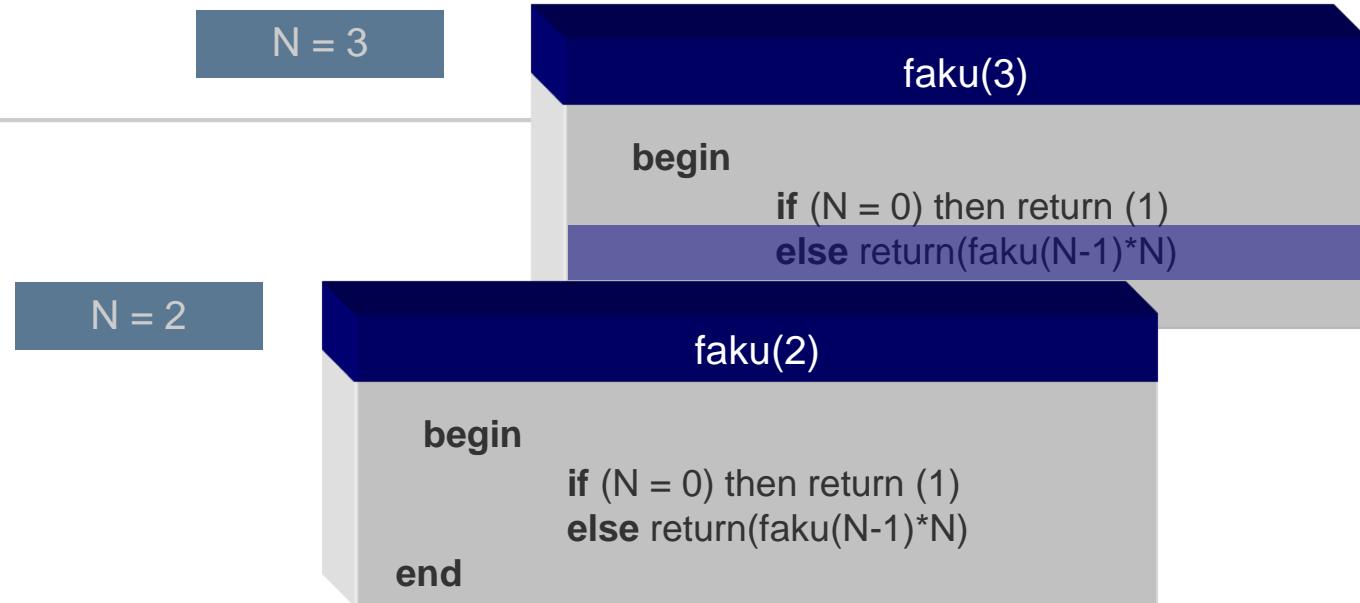
N = 3

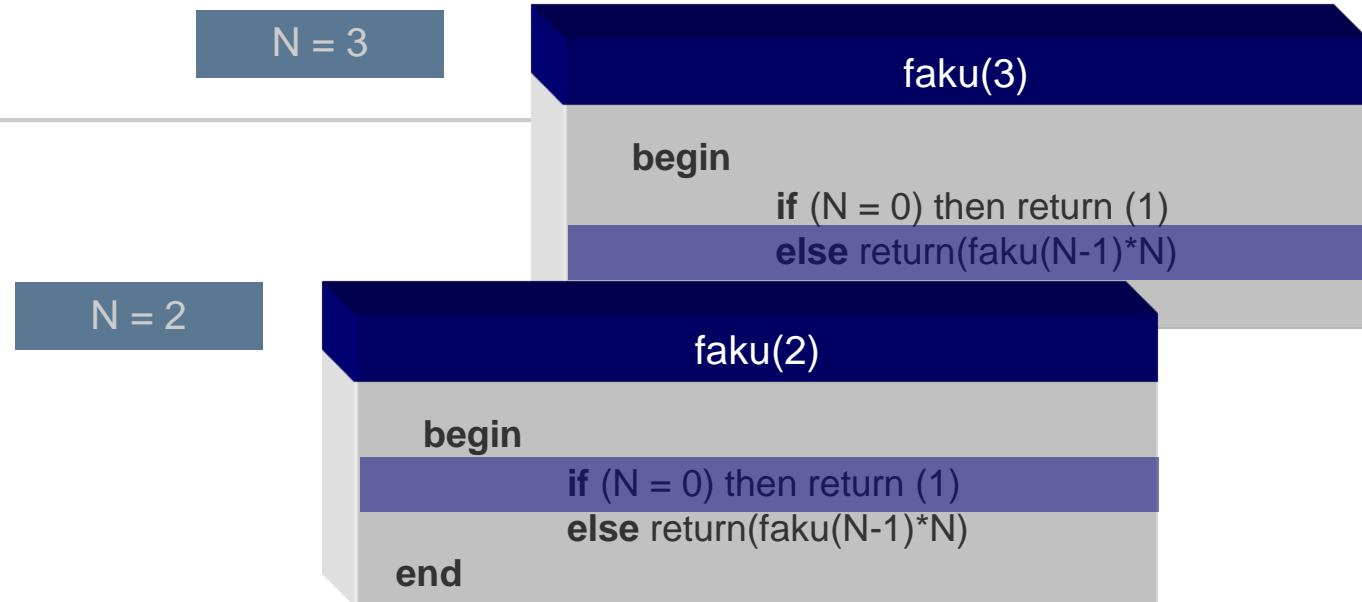
faku(3)

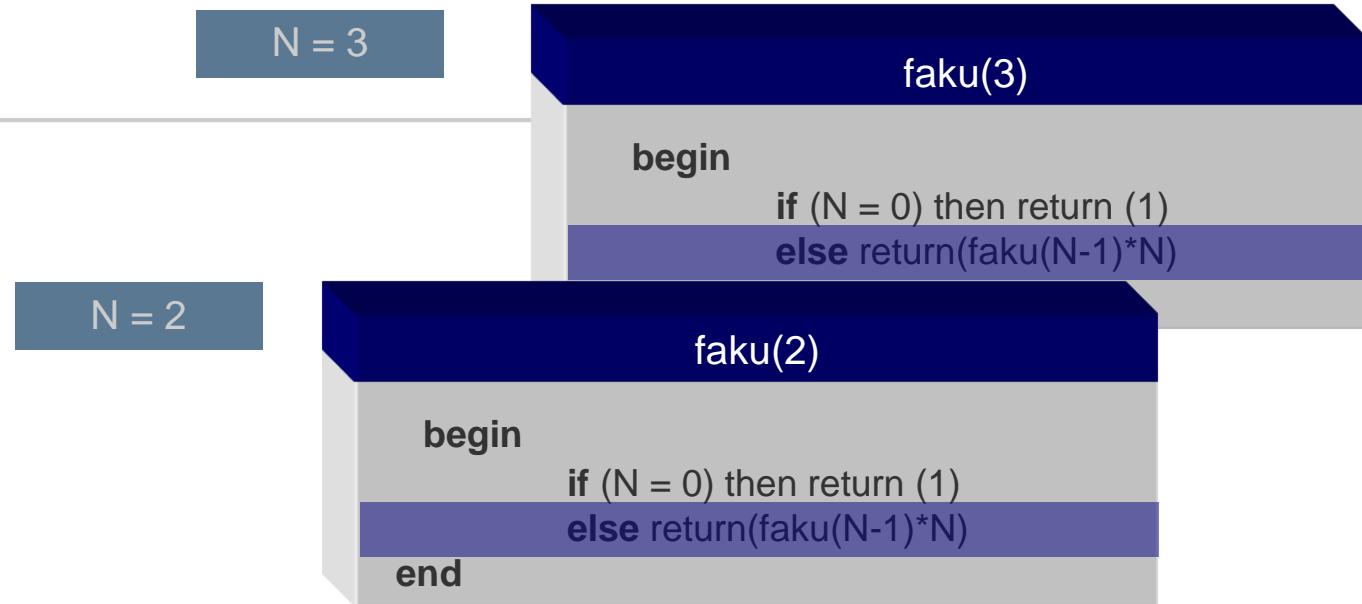
begin

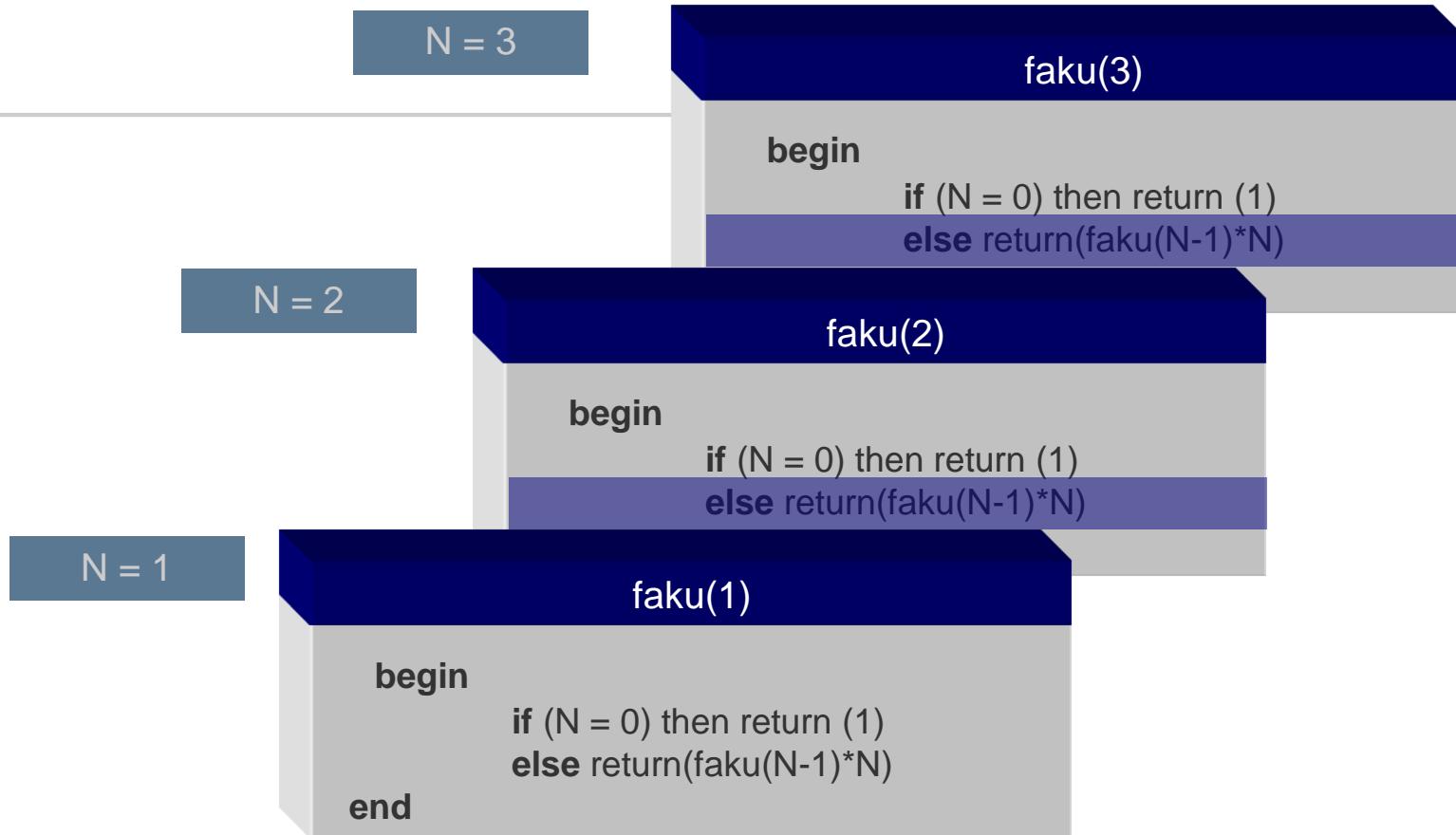
if (N = 0) **then return** (1)
else return(faku(N-1)*N)

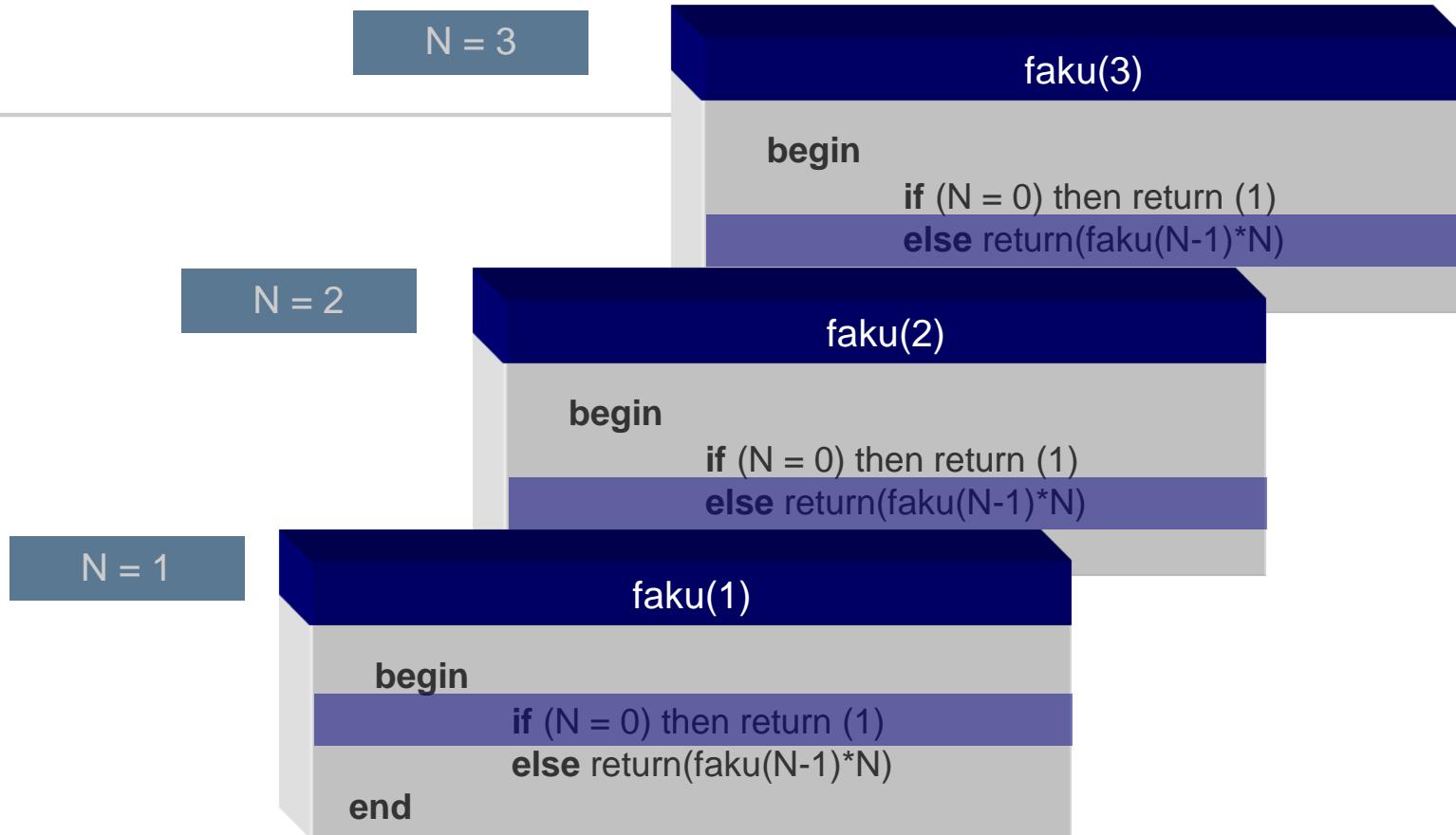
end

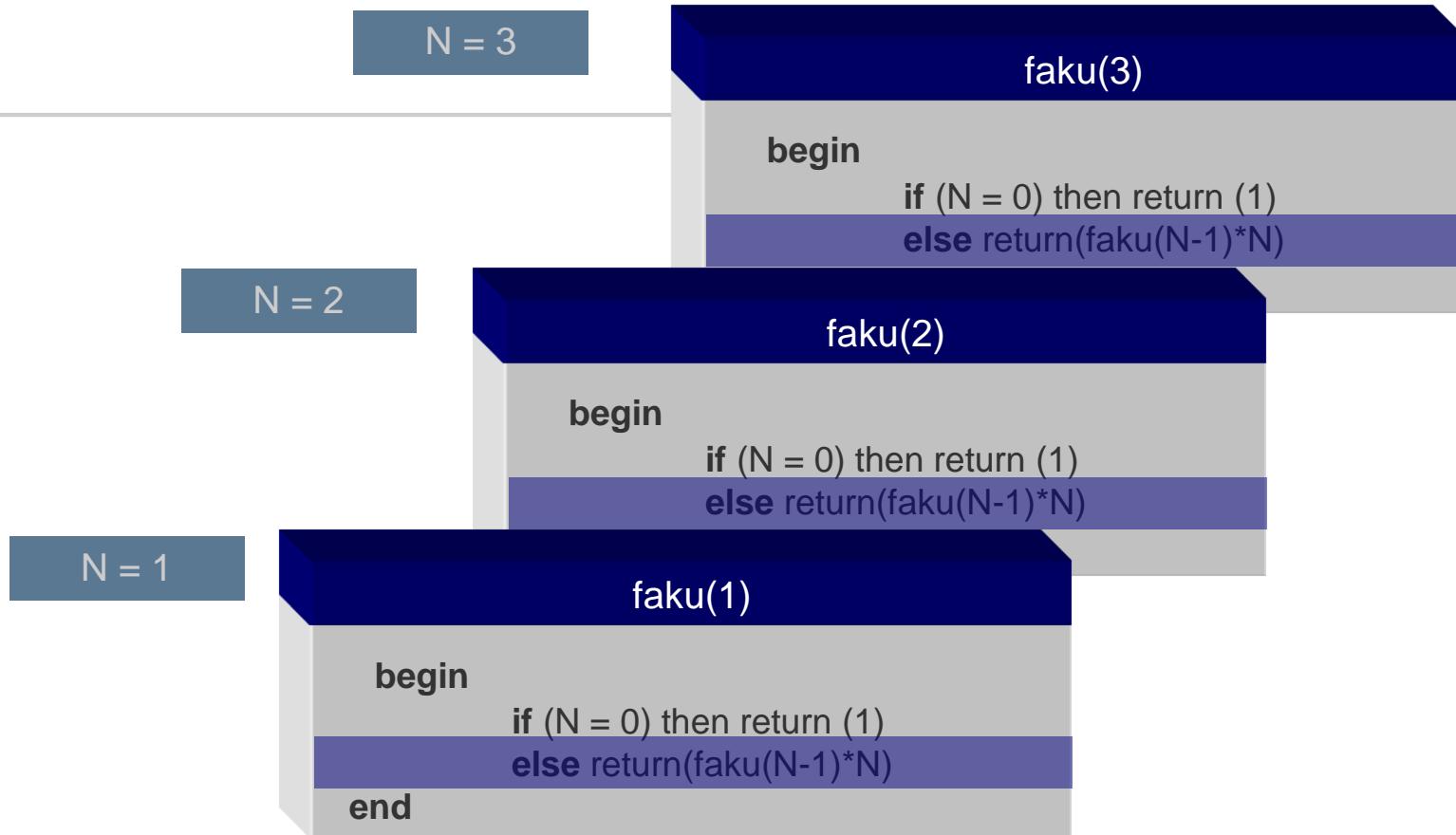


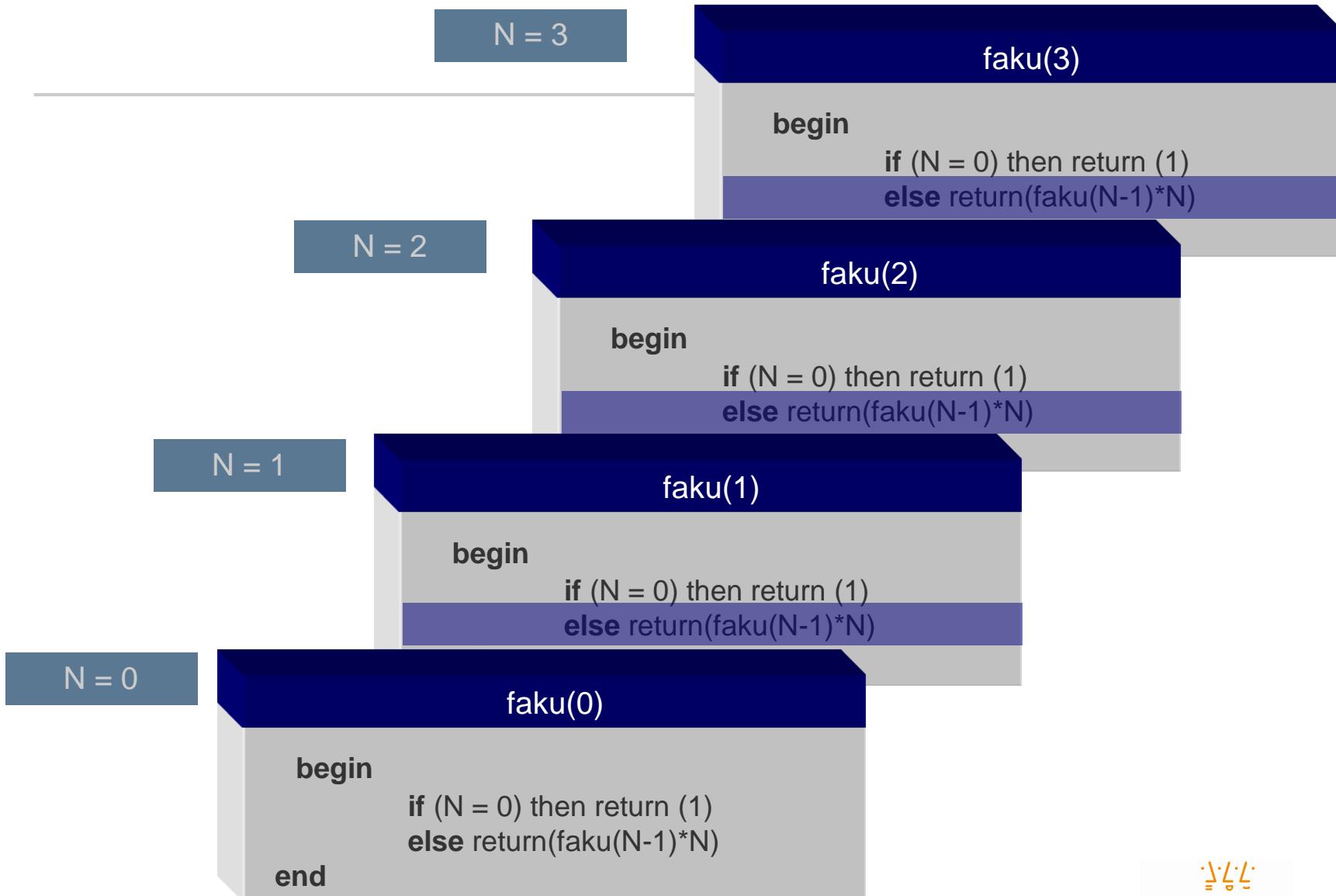


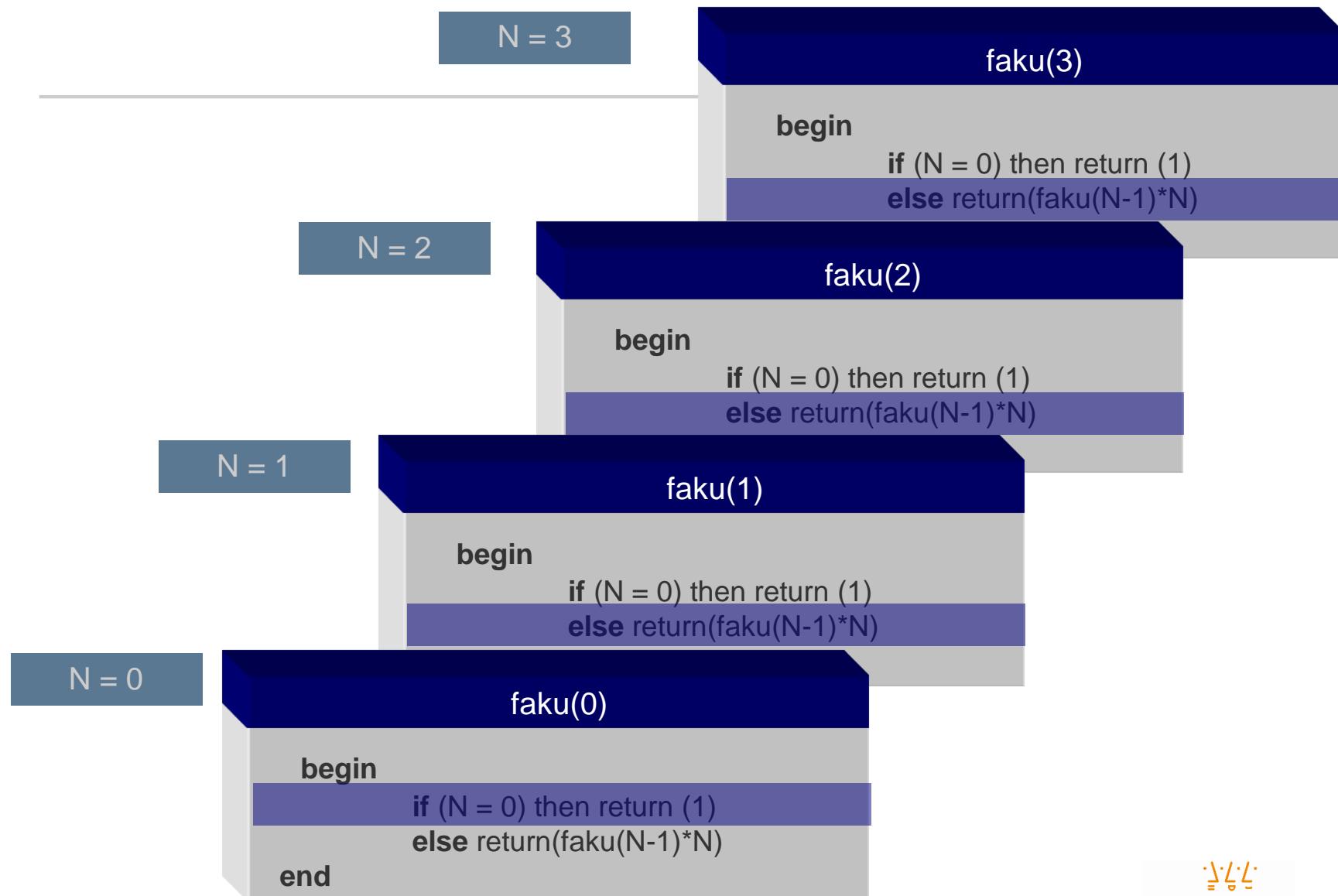


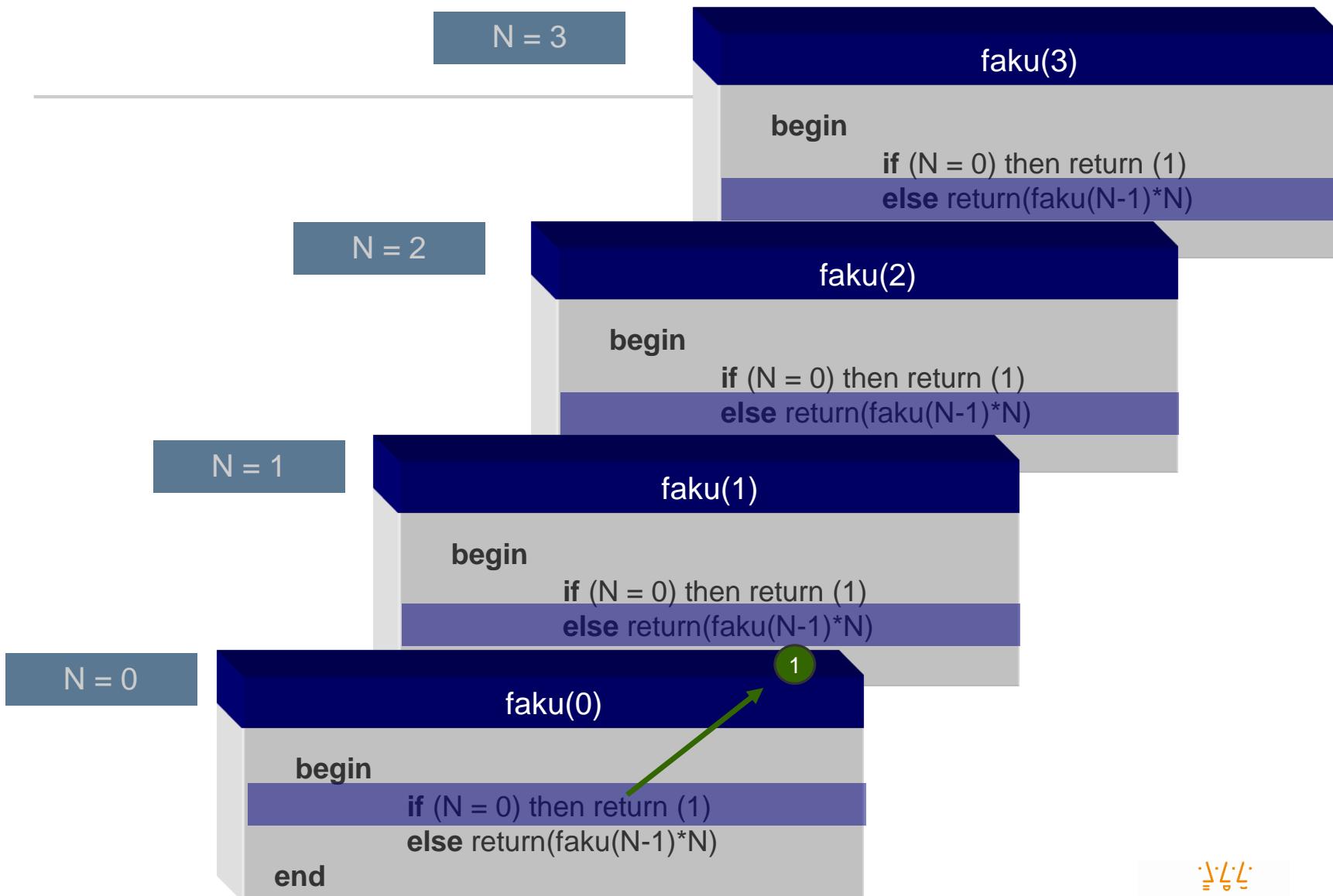


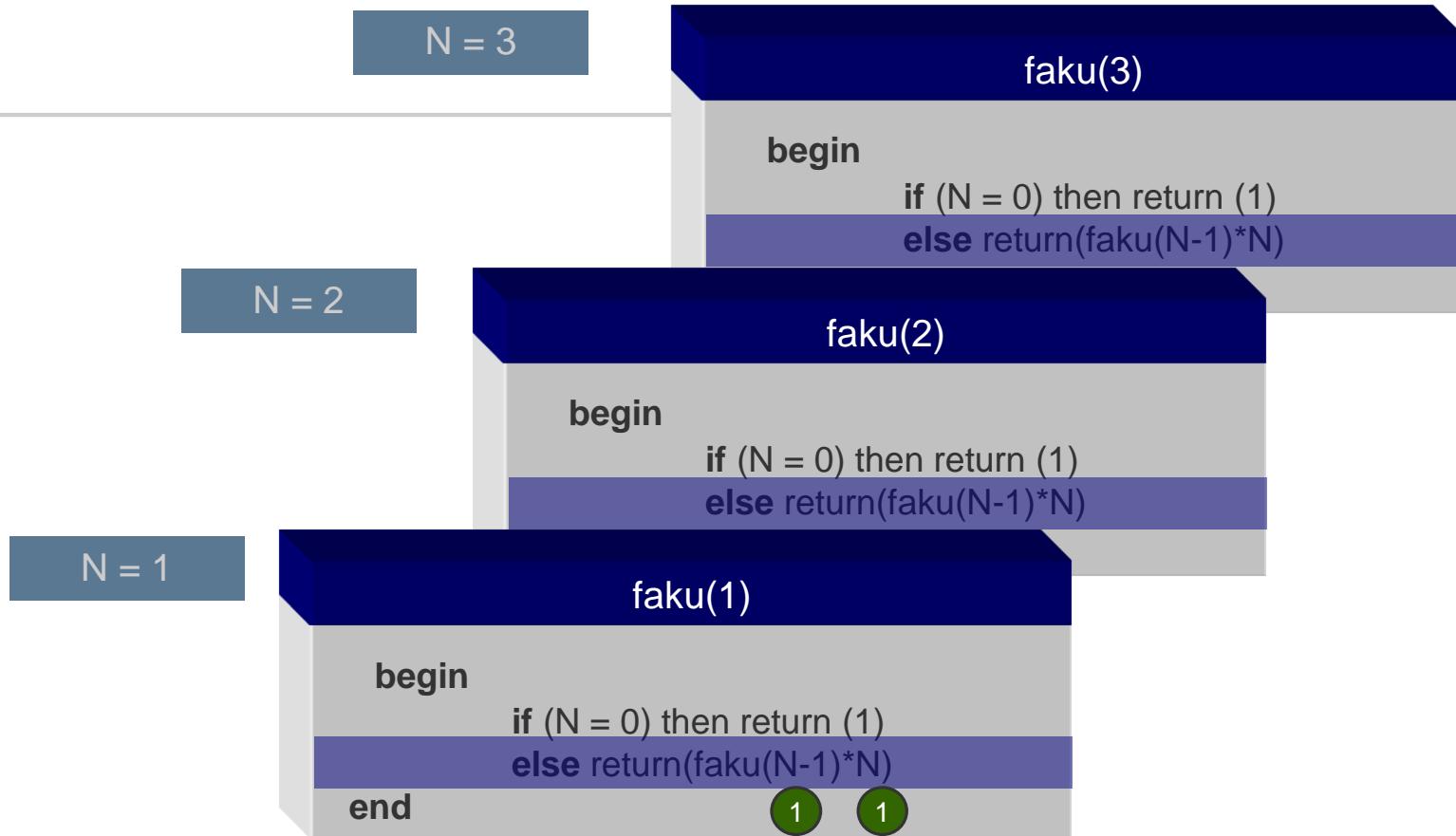


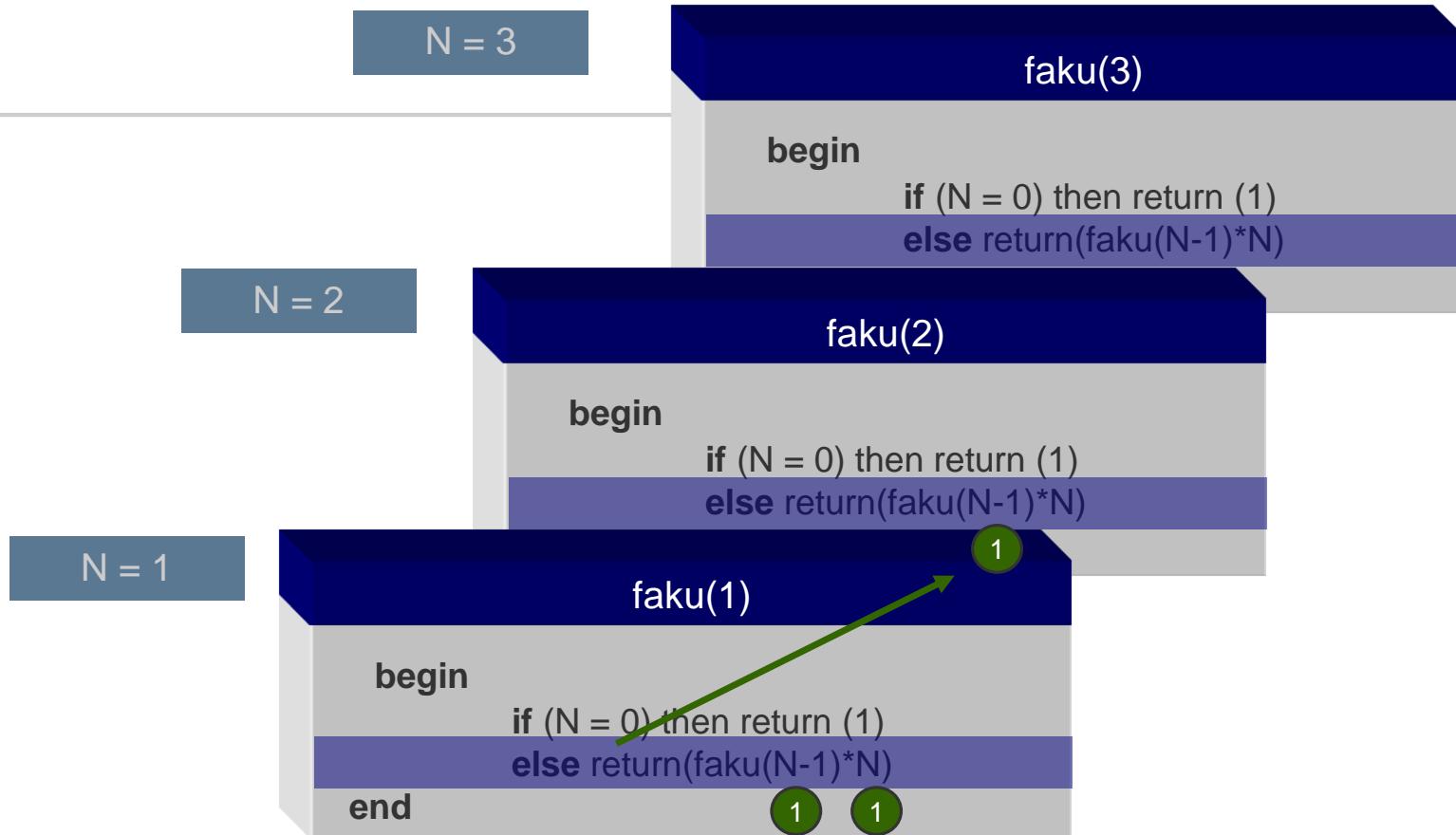


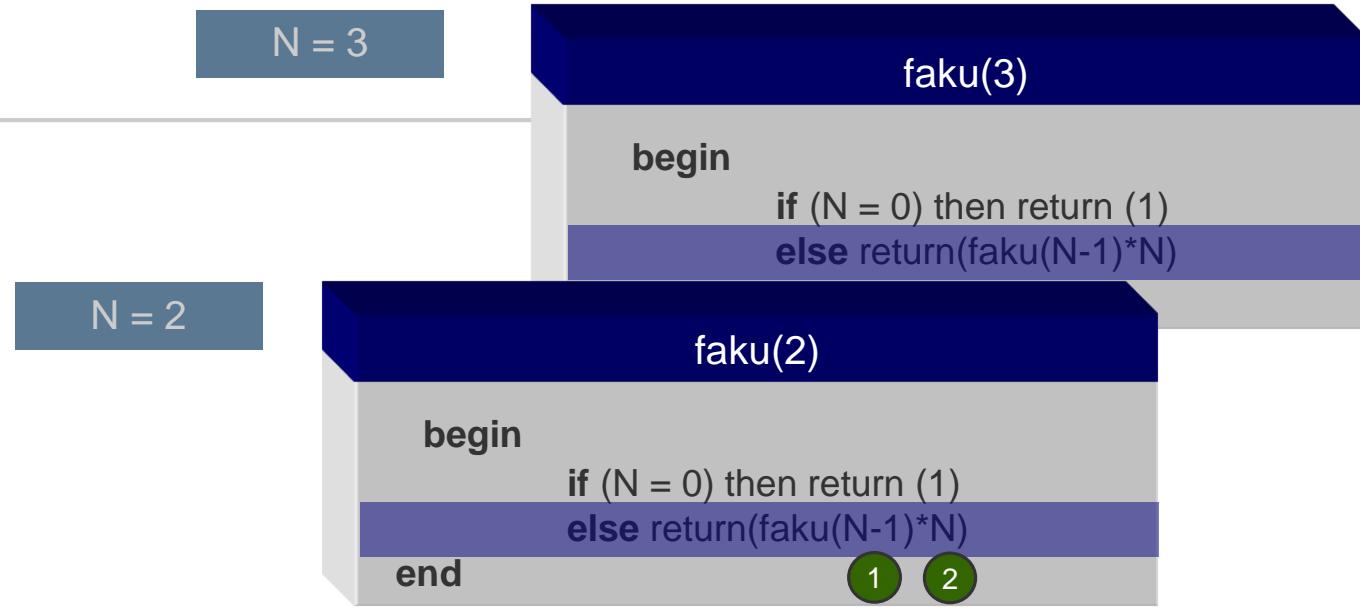












$N = 3$

faku(3)

begin

if ($N = 0$) **then return** (1)
else return(faku($N-1$)* N)

$N = 2$

faku(2)

begin

if ($N = 0$) **then return** (1)
else return(faku($N-1$)* N)

end

1

2



N = 3

faku(3)

begin

if (N = 0) then return (1)
else return(faku(N-1)*N)

end

2

3

N = 3

faku(3)

begin

if (N = 0) then return (1)
else return(faku(N-1)*N)

end

2

3

6

Die 3-te Fakultaet lautet 6.



Klassisches Beispiel für einen rekursiven Algorithmus

Türme von Hanoi (I)

Legendenumwobenes Spiel, bei dem es darum geht, turmhaft gestapelte Scheiben von einem linken Stab unter Zuhilfenahme eines mittleren Stabes auf den rechten Stab zu bringen. Dabei sind folgende Spielregeln einzuhalten:

- Es darf bei jedem Schritt immer nur eine Scheibe bewegt werden
- Es darf nie eine größere Scheibe auf eine kleinere Scheibe gelegt werden



Teil der Sage ist, dass die Welt untergeht, wenn eine bestimmte Gruppe von Mönchen diese Aufgabe in einem Tempel mit 64 goldenen Scheiben auf Diamantnadeln erfüllt.

Wie lange meinen Sie würden die Mönche für diese Aufgabe benötigen???

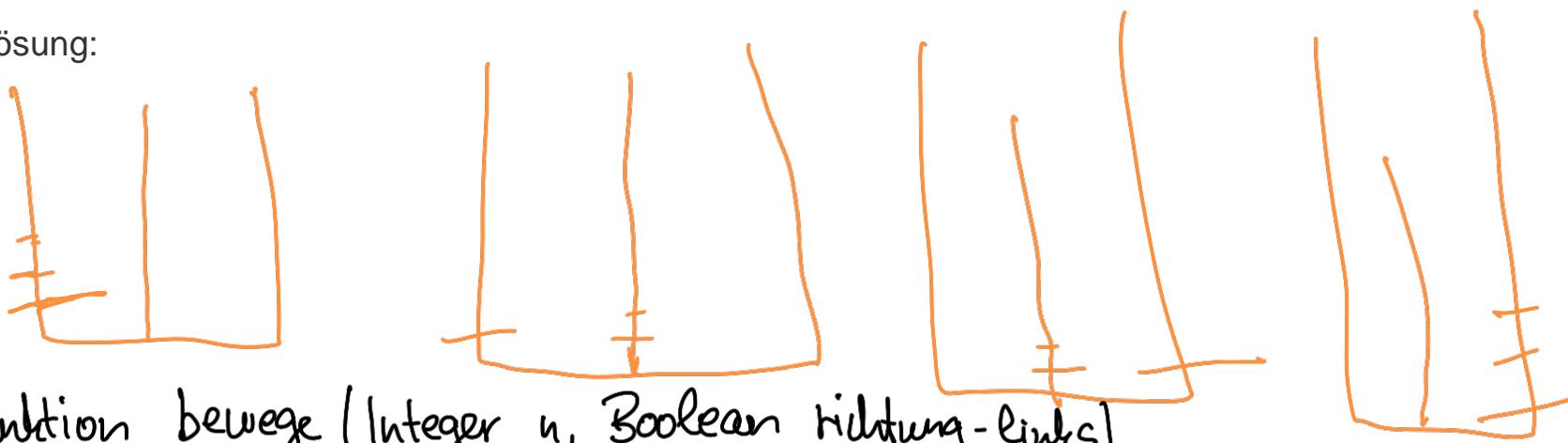
Klassisches Beispiel für einen rekursiven Algorithmus Türme von Hanoi (II)

Aufgabe:

Diskutieren Sie mit Ihrem Sitznachbarn, wie eine rekursive Lösung des Problems aussehen könnte.

Zeit: 10 Minuten

Lösung:



funktion bewege (Integer n, Boolean richtung-links)

{

if ($n=0$) return;

bewege ($n-1$, not richtung-links);



Unter https://www.mathematik.ch/spiele/hanoi_mit_grafik/ können Sie das Spiel probieren

if (links) then "schiebe nach links";

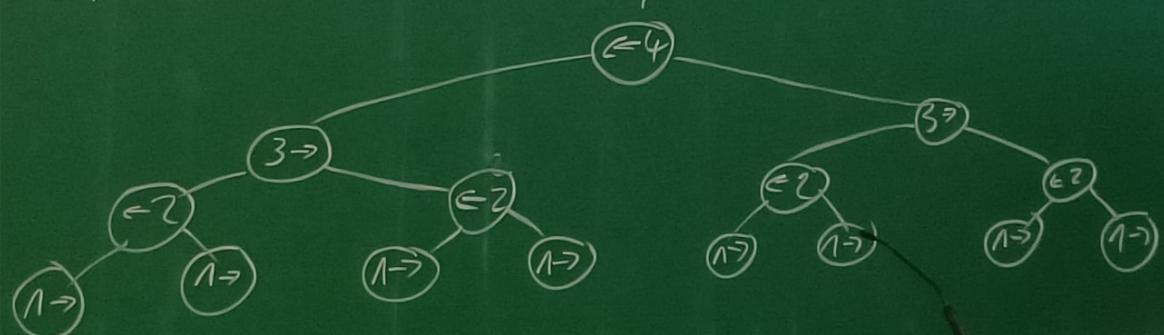
else then "schiebe nach rechts";

bewege ($n-1$, not links);

}

$$3! = 3 \times 2! = 3 \times \underbrace{2 \times 1!}_2 = 3 \times 2 \times 1 \times 1$$

$$n! = n \times (n-1)!$$



Klassisches Beispiel für einen rekursiven Algorithmus

Türme von Hanoi (III)

Umsetzung in Java:

```
public class TurmVonHanoi{  
    public static void bewege(int n, boolean links) {  
        if(n==0) return;  
        bewege(n-1, !links);  
        if (links) System.out.println("'" + n + " links");  
        else System.out.println("'" + n + " rechts");  
        bewege(n-1, !links);}  
  
    public static void main(String[] args){  
        int n = Integer.parseInt(javax.swing.JOptionPane.showInputDialog(null,"Anzahl Scheiben"));  
        bewege(n, true);}  
}
```

Im Funktionsaufruf-Baum lässt sich der Ablauf des Programms nachvollziehen:

Klassisches Beispiel für einen rekursiven Algorithmus

Türme von Hanoi (IV)

Aufgabe:

Diskutieren Sie mit Ihrem Sitznachbarn, ob bzw. wie sich der Algorithmus auch iterativ darstellen lässt.

Welcher der beiden Algorithmen ist effizienter, warum?

Zeit: 10 Minuten

Lösung: Iterative Lösung

```
Funktion TürmeVonHanoi_iterativ(Integer n) {  
    while (Scheibe in t und B vorhanden)  
        if (n%2 = 0) then Bewege Scheibe 1 Rechts;  
        else Bewege Scheibe 1 Links;  
        Bewege Andere Scheibe;  
    }
```

Laufzeitanalyse Türme von Hanoi

Aufgabe: Diskutieren Sie mit Ihrem Nachbarn, wie viele Züge abhängig von der Anzahl n der Scheiben benötigt werden und geben Sie eine Formel für die Komplexität in Abhängigkeit von n an

Aus rekursiven Algorithmus lässt sich folgende Formel für Anzahl der Zuweisungen ableiten:

$$T(n) = 2 T(n-1) + 1 \quad \text{für } n > 1, \text{ mit } T(1) = 1$$

→ durch Probieren ergibt: $T(n)$

Frage: Wie lange würden die Mönche für die Aufgabe mit 64 Scheiben brauchen?

Laufzeitanalyse Türme von Hanoi

Aufgabe: Diskutieren Sie mit Ihrem Nachbarn, wie viele Züge abhängig von der Anzahl n der Scheiben benötigt werden und geben Sie eine Formel für die Komplexität in Abhängigkeit von n an

Aus rekursivem Algorithmus lässt sich folgende Formel für Anzahl der Zuweisungen ableiten:

$$T(n) = 2 T(n-1) + 1 \text{ für } n > 1, \text{ mit } T(1) = 1$$

→ durch Probieren ergibt: $T(1) = 1, T(2) = 3, T(3) = 7, T(4) = 15$

$$\rightarrow T_n = 2^n - 1$$

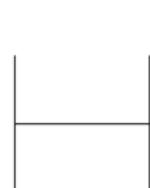
Frage: Wie lange würden die Mönche für die Aufgabe mit 64 Scheiben brauchen?

$$2^{64} - 1 \times 15\text{s} (\text{Annahme: 15s pro Zug})$$
$$1,84 \times 10^{19} \times \frac{15\text{s}}{60(\text{Minuten}) + 60(\text{Stunden}) \times 24(\text{Tage}) \approx 365(\text{Jahre})}$$
$$= 585 \text{ Milliarden Jahre}$$

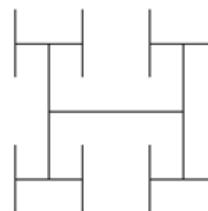
Rekursive Grafik, Beispiel H-Baum (1)

Die Funktion draw führt folgende Aktionen aus:

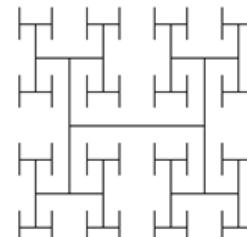
- Es werden drei Linien in Form eines Buchstabens H gezeichnet.
- Darauf ruft sich die Funktion draw vier mal rekursiv auf und zeichnet vier H's der Ordnung n-1, die halb so groß sind und an den vier Spitzen des ersten H's zentriert werden
- Dies wird solange wiederholt, bis die Ordnung N erreicht wurde.



N=1



N=2



N=3

Rekursive Grafik, Beispiel H-Baum (2)

Umsetzung in Java:

```
1 public class Htree {
2
3     // plot an H, centered on (x, y) of the given side length
4     public static void drawH(double x, double y, double size) {
5
6         // compute the coordinates of the 4 tips of the H
7         double x0 = x - size/2;
8         double x1 = x + size/2;
9         double y0 = y - size/2;
10        double y1 = y + size/2;
11
12        // draw the 3 line segments of the H
13        StdDraw.line(x0, y0, x0, y1);    // left vertical segment of the H
14        StdDraw.line(x1, y0, x1, y1);    // right vertical segment of the H
15        StdDraw.line(x0, y, x1, y);    // connect the two vertical segments
16    }
17    // plot an order n H-tree, centered on (x, y) of the given side length
18    public static void draw(int n, double x, double y, double size) {
19        if (n == 0) return;
20        drawH(x, y, size);
21
22        // compute x- and y-coordinates of the 4 half-size H-trees
23        double x0 = x - size/2;
24        double x1 = x + size/2;
25        double y0 = y - size/2;
26        double y1 = y + size/2;
27
28        // recursively draw 4 half-size H-trees of order n-1
29        draw(n-1, x0, y0, size/2);    // lower left H-tree
30        draw(n-1, x0, y1, size/2);    // upper left H-tree
31        draw(n-1, x1, y0, size/2);    // lower right H-tree
32        draw(n-1, x1, y1, size/2);    // upper right H-tree
33    }
34    // read in a command-line argument N and plot an order N H-tree
35    public static void main(String[] args) {
36        int N = Integer.parseInt(args[0]);
37
38        double x = 0.5, y = 0.5;    // center of H-tree
39        double size = 0.5;        // side length of H-tree
40        draw(N, x, y, size);
41    }
42}
```

Rekursive Grafik, Beispiel H-Baum (3)

Aufgabe: Ermitteln Sie die Komplexität des Algorithmus

Wachstumsfunktion der Aufrufe von draw mit wachsender Ordnung n

$$T(n) 4^{n-1} \text{ für } n > 1$$

Es handelt sich um einen exponentiellen Anstieg der rekursiven Aufrufe

Von draw mit Ordnung n

Dies führt zu n sehr hohen Laufzeiten mit Ordnung n.

Eine programmtechnische Begrenzung der Ordnungen verhindert den Programmabsturz



Rekursive Algorithmen – Fallstricke (1)

Fehlender Basisfall

Was passiert, wenn Sie bei der besprochenen Fakultätsberechnung vergessen, den Basisfall zu programmieren?:

Korrekt:

```
Procedure faku(N: integer);
begin
    if (N = 0) then return (1)
    else return(faku(N-1)*N)
end
```

Falsch:

```
Procedure faku(N: integer);
begin
    return(faku(N-1)*N)
end
```

Antwort: Die Prozedur ruft sich endlos selbst auf.
was zu einem Stackoverflow mit zugehöriger Fehlermeldung führt

!

Rekursive Algorithmen – Fallstricke (2)

Keine Konvergenz

Was passiert, wenn Sie bei der besprochenen Fakultätsberechnung einen rekursiven Aufruf einbauen, der nicht kleiner ist als das ursprüngliche Problem?:

Korrekt:

```
Procedure faku(N: integer);
begin
    if (N = 0) then return (1)
    else return(faku(N-1)*N)
end
```

Antwort:

Falsch:

```
Procedure faku(N: integer);
begin
    if (N = 0) then return (1)
    else return(faku(N)*N)
end
```

Rekursive Algorithmen – Fallstricke (3)

Extremer Speicherbedarf

Wie wir gesehen haben, steigt beim Beispiel der Türme von Hanoi der rekursive Selbstaufruf exponentiell an. Dies führt dazu, dass extrem viel Speicherplatz mit steigender Zahl n der Scheiben benötigt wird, was relativ bald zu einem stack overflow führen kann.

Deshalb sollten Sie im Vorfeld (insbesondere für rekursive Algorithmen) immer die Komplexität prüfen, um ggf. die Eingabe zu hoher n programmtechnisch abzufangen.

Rekursive Algorithmen – Fallstricke (4)

Extrem häufige Neuberechnungen

Die im Jahr 1202 beschriebene Fibonacci Folge:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 ...

kann durch einen einfachen rekursiven Algorithmus beschrieben werden:

```
Function Fibonacci(N: integer):integer;
begin
  if n=0 then Fibonacci := 0;
  if n=1 then Fibonacci := 1;
  Fibonacci := (Fibonacci(n-1) + Fibonacci(n-2));
end
```

Frage1: Was fällt Ihnen bei dem Algorithmus bezüglich der rekursiven Aufrufe auf?

Frage 2: Wie könnte dies verhindert werden?

Kapitel 4: Algorithmen

4.4. Sortieralgorithmen



Gruppenarbeit Jigsaw

Aufteilung der Sortieralgorithmen:

- | | |
|-------------------|---------------|
| 1) Selection Sort | 4) Quick Sort |
| 2) Insertion Sort | 5) Merge Sort |
| 3) Bubble Sort | 6) Shell Sort |

Schritt 1: Treffen Sie sich in Ihrer Gruppe. Jeder erarbeitet in Einzelarbeit den zugeteilten Sortieralgorithmus. Bereiten Sie dabei eine Kurzpräsentation (Papier) für die Expertengruppe vor. **30 Minuten**

Schritt 2: Die Experten treffen sich pro Sortieralgorithmus. Jedes Mitglied stellt seine Präsentation kurz vor (Anfangs ca. 5 – 10 Minuten, später nur noch auf Ergänzungen/Unterschiede eingehen, 1-2 Minuten), Zuhörer optimieren ihre Präsentation um das von den anderen gehörte. **45 Minuten**

Schritt 3: Treffen Sie sich wieder in ihrer ursprünglichen Gruppe. Jedes Mitglied präsentiert seinen Sortieralgorithmus und beantwortet zugehörige Fragen. Vorschlag: Gegebene Reihenfolge einhalten. **60 Minuten**



Einfache Sortieralgorithmen

Selection-Sort (1)

Beschreibung:

Es wird schrittweise erst das kleinste Element gesucht und an die erste Stelle des Arrays gehängt, dann folgt das zweite Element an der zweiten Stelle usw.

Algorithmus:

```
procedure Selection_Sort (var A : Array [0] in, int)
```

```
    var links, min, tausch int;
```

```
    links = 0;
```

```
    while links < n
```

```
        min := links
```

```
        for i = links+1 to n
```

```
            Ja
```

```
A[i] < A[min] ?
```

```
Nein
```



$\left\{ \begin{array}{l} \min = c \\ \text{tausch} = A[\min]; A[\min]; A[\text{links}]; \\ A[\text{links} = \text{tausch}]; \end{array} \right.$

$\text{links} = \text{links} + 1;$

Einfache Sortieralgorithmen

Selection-Sort (2)

Beispiel:

Startfolge: 31 57 70 22 97 62 61 84 78 5

1. DL 5 | 57, 70, 22, 97, 62, 61, 84, 78, 31

2. DL 5, 22, 70, 57, 97, 62, 61, 84, 78, 31

3. DL 5, 22, 31, 57, 97, 62, 61, 84, 78, 70

4. DL 5, 22, 31, 57, 61, 62, 97, 84, 78, 70

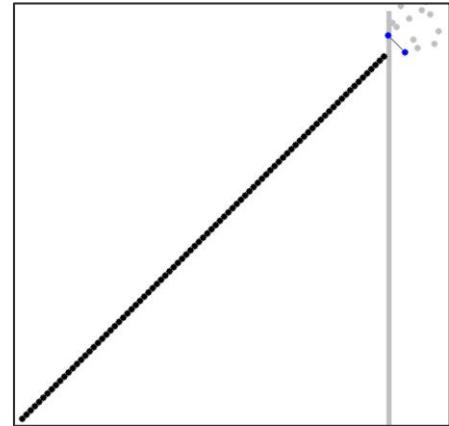
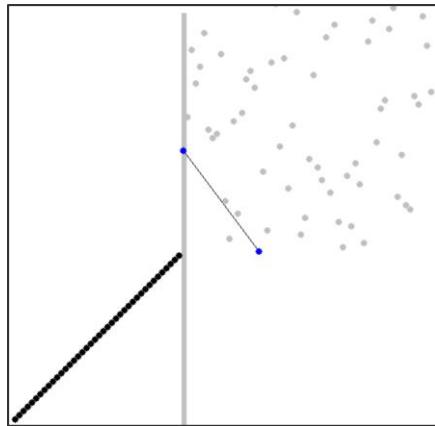
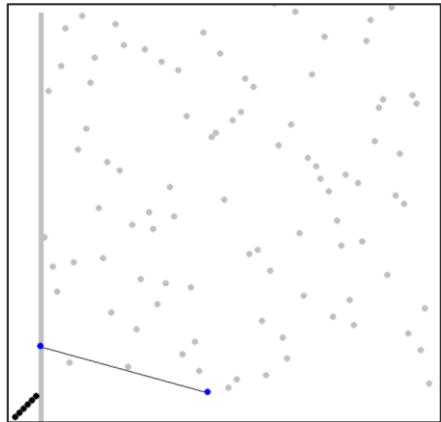
5. DL 5, 22, 31, 57, 61, 62, 70



Einfache Sortieralgorithmen

Selection-Sort (3)

Veranschaulichung:



Laufzeitkomplexität:

Best Case: Da unabhängig von den vorsortierten Daten der bestimmte Array durchlaufen werden muss: $O(n^2)$

Average Case:

Worst Case:

Einfache Sortieralgorithmen

Bubble-Sort (1)

Beschreibung:

Es werden schrittweise alle benachbarten Paare miteinander verglichen und ggf. miteinander vertauscht. Dabei steigen/sinken große/kleine Werte wie „Gasbladen“ nach oben/unten

Algorithmus:

```
for (int i=0; i < arr.length; i++) {  
    if (arr[i] > arr[i+1]) {  
        int a = arr[i+1];  
        arr[i+1] = arr[i];  
        arr[i] = a  
    }  
}
```

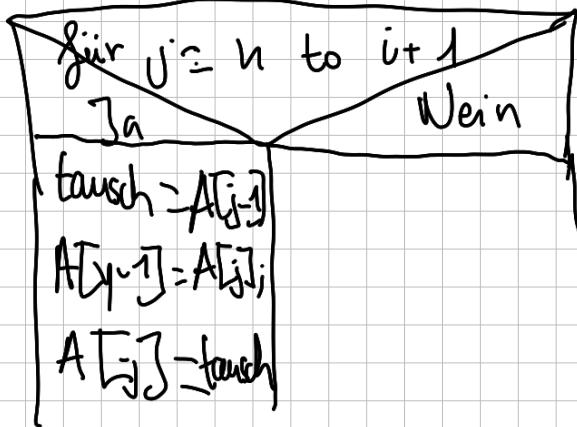


Lösung 1

```
procedure bubble_sort (var A: Array [0..n] ; n : int);
```

```
    var i,j,tausch: int;
```

```
    for i = 0 to n-1
```



Lösung 2:

```
procedure bubble_sort2(var A: Array [0..n] ; int);
```

```
    var i,j,tausch,int;
```

```
    var sortiert: boolean;
```

```
    sortiert := falsche
```

```
    for j := 0 to n
```



Einfache Sortieralgorithmen

Bubble-Sort (2)

Beispiel:

Vor Bubble-Sort: 5 7 3 6 4

1. DL:

5 7 3 4 6

5 3 7 4 6

2. DL:

3 5 7 4 6

↑↑

3 5 4 7 6

↑↑

3. DL

3 4 5 7 6

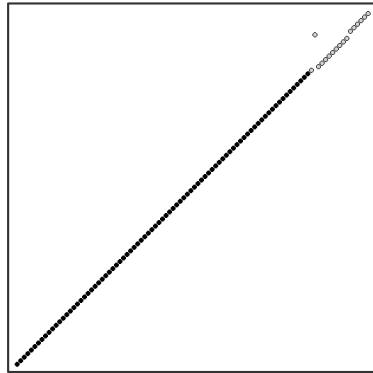
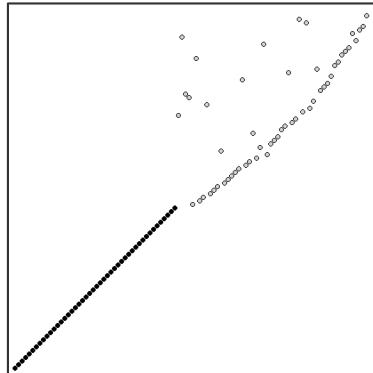
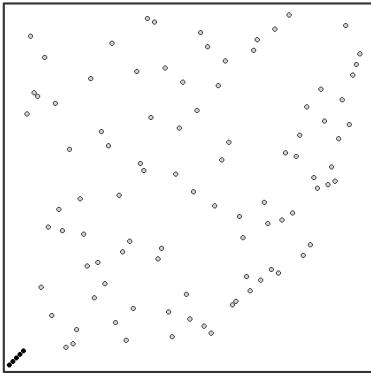
4. DL

3 4 5 6]

Einfache Sortieralgorithmen

Bubble-Sort (3)

Veranschaulichung:



Laufzeitkomplexität:

Best Case: Array bereits voll vorsortiert \rightarrow nur ein Durchlauf
 \rightarrow Linear $O(n)$ \rightarrow gut für vorsortige Daten geeignet

Average Case: Abschätzung schwierig, zwischen $O(n)$ und $O(n^2)$,
abhängig von Daten

Worst Case: Absteigende sortierendes Array, erste, zweite Schleife
werden n -mal durchlaufen \rightarrow quadratisch $O(n^2)$

Einfache Sortieralgorithmen

Insertion-Sort (1)

Beschreibung:

Wie beim Aufdecken eines Skat-Blattes wird jede Zahl einzeln aufgerufen und an der richtigen Stelle in das Array eingefügt.

Algorithmus:

Procedure insertion-sort(var A: Array[0..n] of int,
 n int);

var i,j,einsortierwert int;

1 < c <= n

einsortiert = A[i]; j = i;

while j>1 & A[j-1] > einsortierwert



$$A[j] = A[j-1];$$

$$\ddot{j} = j - 1$$

$A[j]$ = eingesortierter Wert;

Einfache Sortieralgorithmen

Insertion-Sort (2)

Beispiel:

Vor Insertion-Sort: 26 14 83 11 7 51 32 4 21 40

1. DL 14, 26, 83, 11, 7, 51, 32, 4, 21, 40

2. DL 14, 26, 83, 11, 7, 51, 32, 4, 21, 40

3. DL 11, 14, 26, 83, 7, 51, 32, 4, 21, 40

4. DL 7, 11, 14, 26, 51, 83, 32, 4, 21, 40

5. DL 7, 11, 14, 26, 51, 83, 32, 4, 21, 40

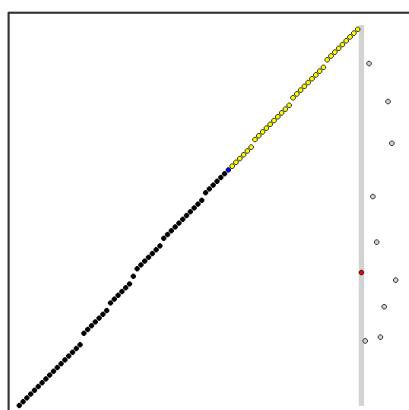
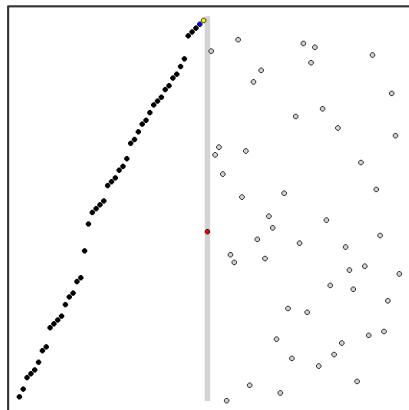
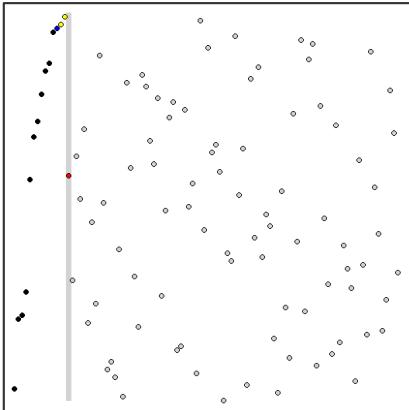
6. - 11, 12, 14, 26, 32, 51, 83, 21, 40



Einfache Sortieralgorithmen

Insertion-Sort (3)

Veranschaulichung:



Laufzeitkomplexität:

Best Case: Array → Linear $O(n)$ → bereit vorsortiert → nur ein Durchlauf
for vorsortierte Daten geeignet

Average Case Abhängigkeiten von Daten ⇒ zw. 0

Worst Case:

Effiziente Sortieralgorithmen

Shell-Sort (1)

Beschreibung:

Der Shell-Sort arbeitet wie der Insertion-Sort, nur dass dieser auf mehrere Sequenzen aufgeteilt angewandt wird, um das Verschieben über „weite Strecken“ zu vermeiden. Dabei haben sich die Schrittweiten 1-3-7-21-48... als besonders effizient herausgestellt.

Algorithmus:

```
procedure shell_sort (var A: Array[0:n]; n int);
    var h, h, i, j, v, s[n]: int
    s[i]:= [198768; 86961; 33936; 13776; 4592; 1968;
            861; 366; 112; 48; 21; 7; 3; 1]
    for 0 < h < length of s, h++
```



$n = \sup \{k\}$

Effiziente Sorteralgorithmen

Shell-Sort (2)

Beispiel:

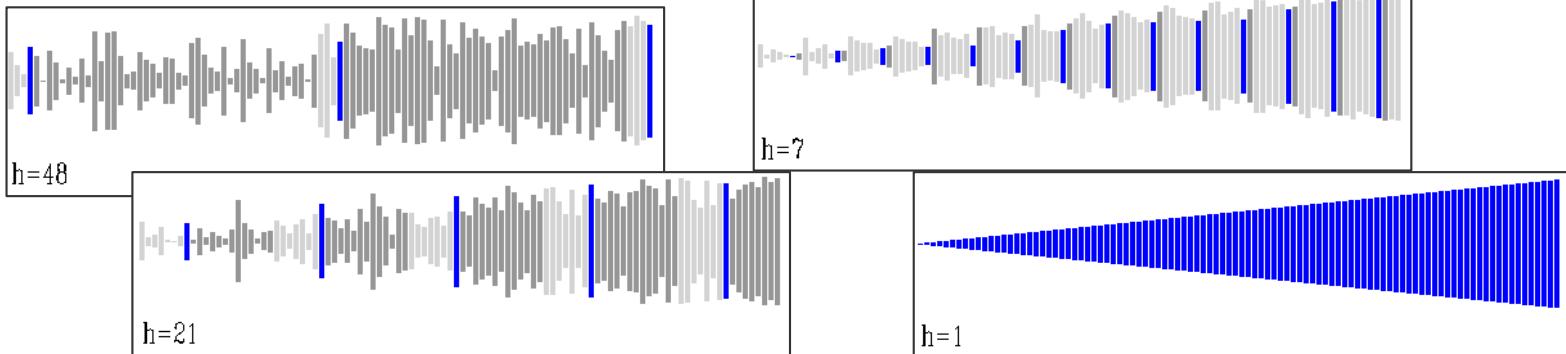
Vor Shell-Sort: 9 6 3 5 2 1 8 4 7



Effiziente Sortieralgorithmen

Shell-Sort

Veranschaulichung:



Laufzeitkomplexität:

Effiziente Sortieralgorithmen

Quick-Sort (1)

Beschreibung:

Mit Hilfe eines Pivot-Elements wird die Liste rekursiv immer wieder in zwei Teillisten geteilt, die für sich sortiert werden, wobei immer die Elemente der linken Liste \leq der Elemente der rechten Liste sind.

Algorithmus:



Effiziente Sortieralgorithmen

Quick-Sort (2)

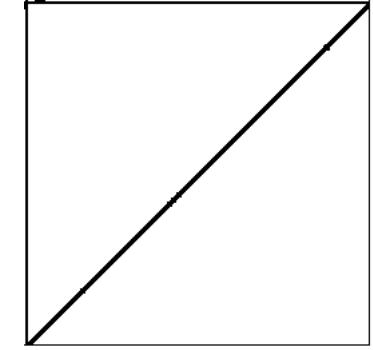
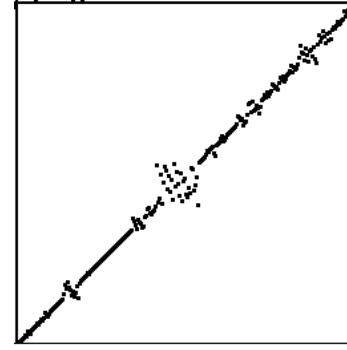
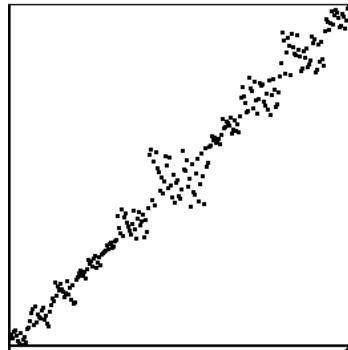
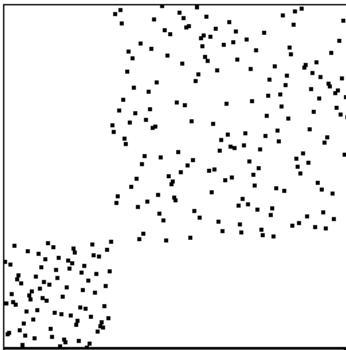
Beispiel: Vor Quicksort: 1 4 6 9 7 1 1 4 6 7



Effiziente Sortieralgorithmen

Quick-Sort (3)

Veranschaulichung:



Laufzeitkomplexität:

Best Case:

Average Case:

Worst Case:

Speicherplatz:

Effiziente Sortieralgorithmen

Merge-Sort (1)

Beschreibung:

Die Daten werden in kleinere Arrays zerlegt, die jeweils für sich sortiert werden. Dann werden diese Arrays im Reißverschlussverfahren wieder zu einem großen Array zusammengefasst.

Algorithmus:

Effiziente Sorteralgorithmen

Merge-Sort (2)

Beispiel:

Vor Mergesort: P O N M L K J I H G F E D C B A

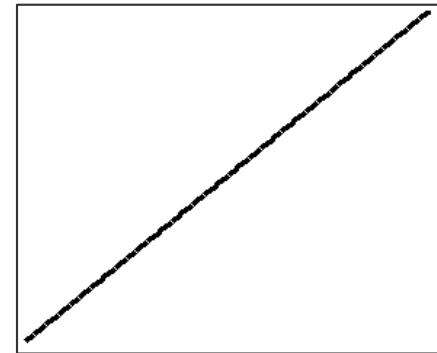
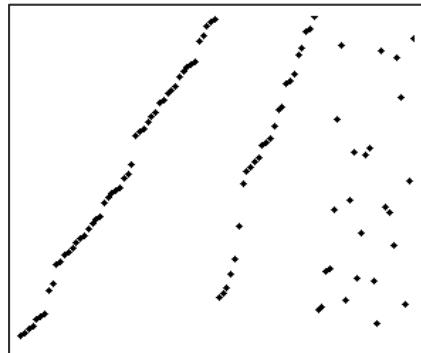
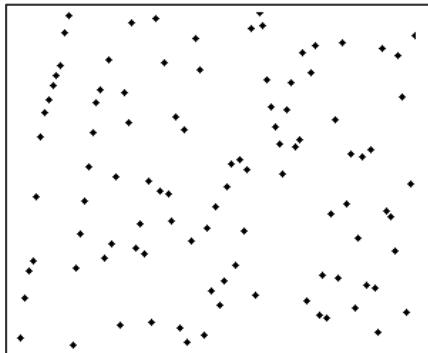
Effiziente Sortieralgorithmen

Merge-Sort (3)

Effiziente Sortieralgorithmen

Merge-Sort (4)

Veranschaulichung:



Laufzeitkomplexität:

Sortieralgorithmen

Leistungsvergleich der Algorithmen

<https://i.imgur.com/fq0A8hx.gif>