

Einführung in die Programmierung

Wintersemester 2009/10

Prof. Dr. Günter Rudolph

Lehrstuhl für Algorithm Engineering

Fakultät für Informatik

TU Dortmund

Inhalt

- Rekursion: Technik
- Rekursion vs. Iteration

Definition (einfache, erste Version)

Rekursives Programm := Programm, das sich selbst aufruft

Rekursive Funktion := Funktion, die sich selbst aufruft

offensichtlich:

Es muss eine Abbruchbedingung geben ...

gibt an, wann
Programm / Funktion
aufhören soll, sich
selbst aufzurufen

- ⇒ sonst unendliche Rekursion
- ⇒ entspricht einer Endlosschleife



Arbeitsprinzip:

rekursiver Algorithmus löst Problem

durch Lösung mehrerer kleinerer Instanzen des gleichen Problems

⇒ Zerlegung des Problems in kleinere Probleme gleicher Art

Rekursionsprinzip schon lange bekannt (> 2000 Jahre)

- zunächst in der Mathematik (z. B. Euklid)
- führte in der Informatik zu fundamentalen Techniken beim Algorithmen-Design
 - z.B. „teile und herrsche“-Methode (divide-and-conquer)
 - z.B. dynamisches Programmieren

Thematik inhaltsschwer für eigene 2- bis 4-stündige Vorlesung → hier: nur 1. Einstieg

Rekursion in der Mathematik

Beispiel: Fakultät

$$f(0) = 1$$

Rekursionsverankerung

$$\forall n \in \mathbb{N} : f(n) = n * f(n - 1)$$

Rekursionsschritt

Beispiel: Rekursive Definition logischer Ausdrücke

1. Wenn v logische Variable (**true**, **false**), dann v und **NOT** v logischer Ausdruck.
2. Wenn a und b logische Ausdrücke, dann a **AND** b sowie a **OR** b logische Ausdrücke.
3. Alle logischen Ausdrücke werden mit 1. und 2. aufgebaut.

Rekursion in der Informatik

Beispiel: Fakultät

$$f(0) = 1$$

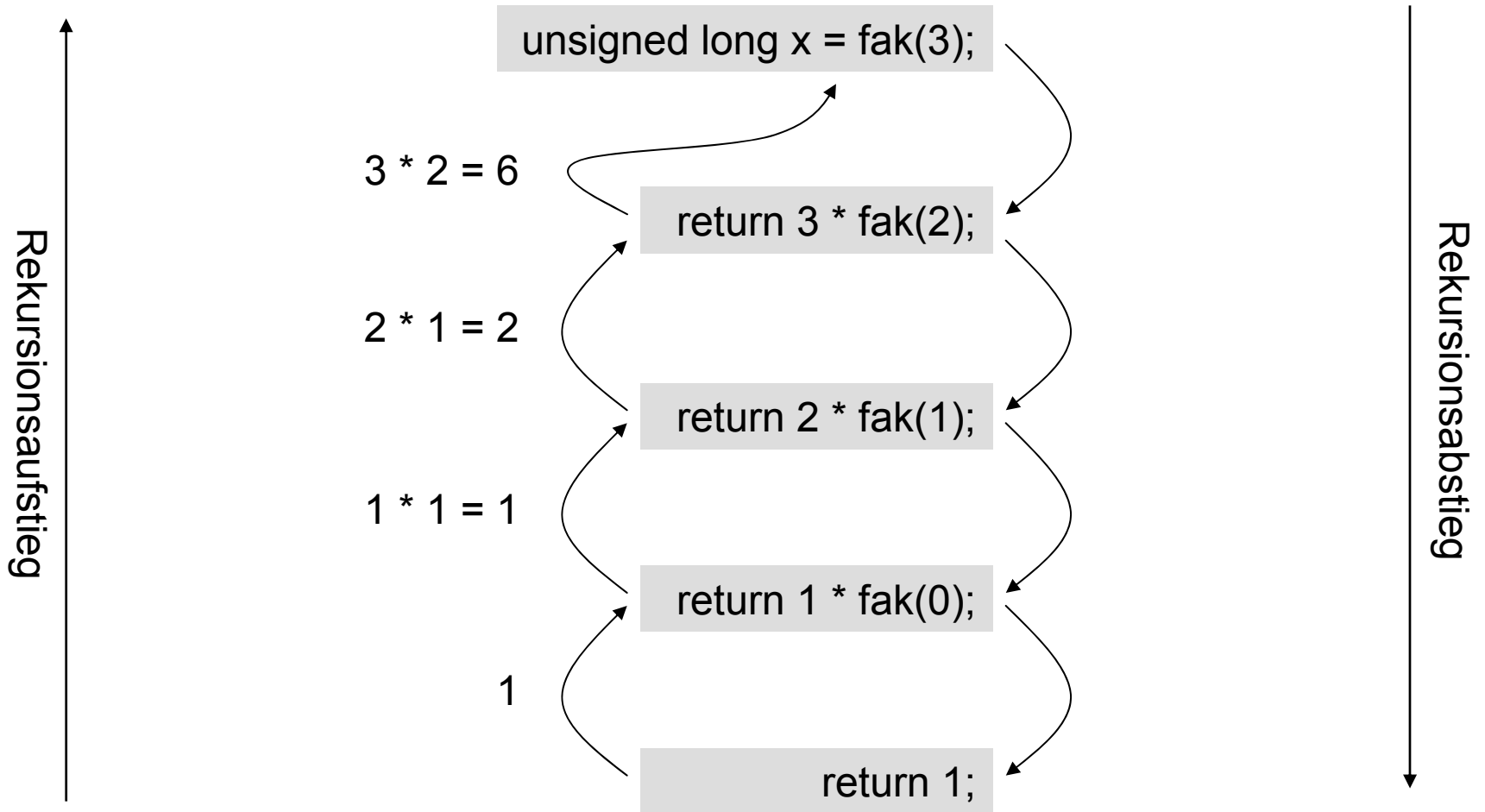
Rekursionsverankerung

$$\forall n \in \mathbb{N} : f(n) = n * f(n - 1)$$

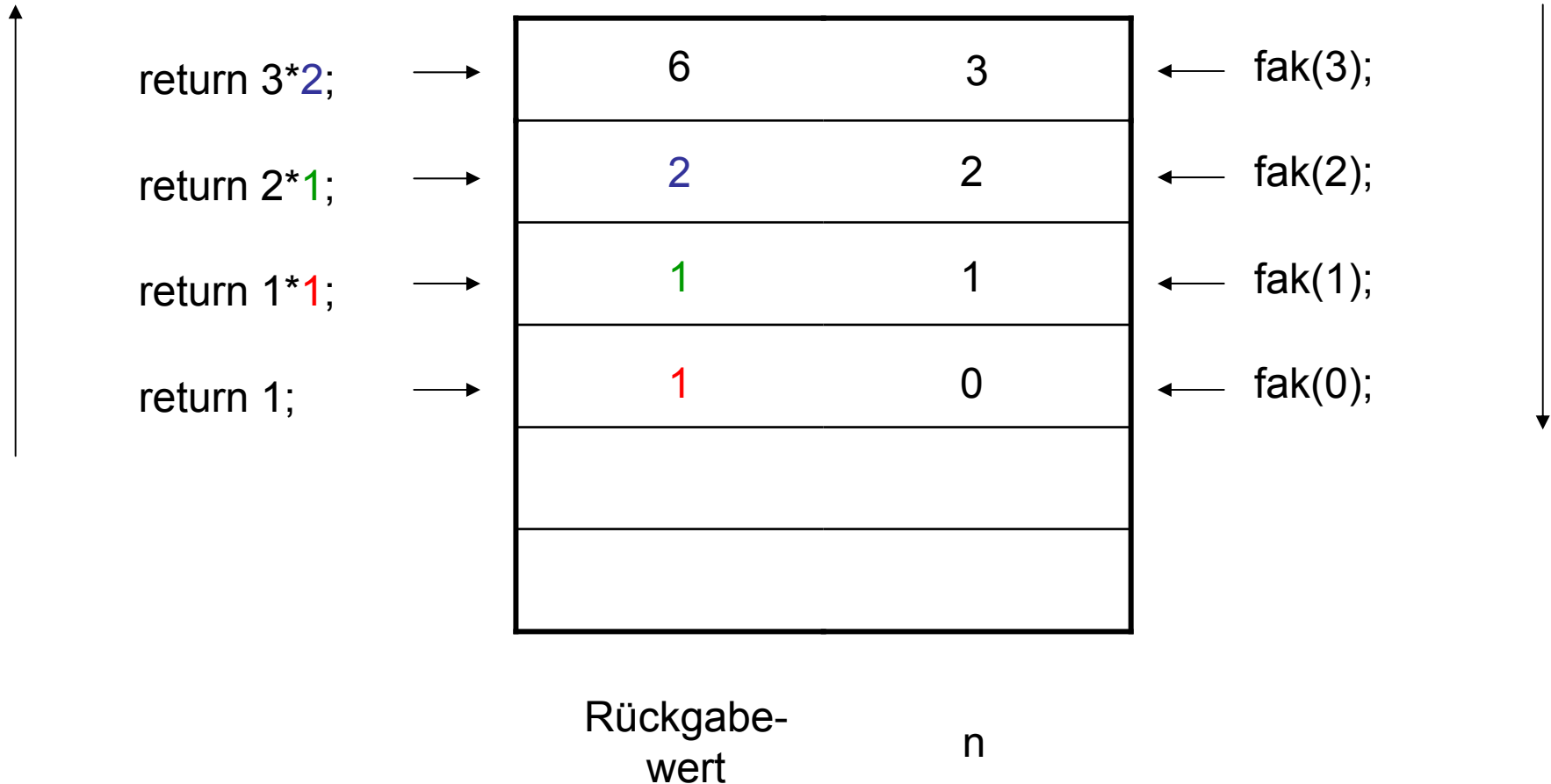
Rekursionsschritt

```
unsigned long fak(unsigned int n) {  
    if (n == 0) return 1;    // Rekursionsverankerung  
    return n * fak(n - 1);  // Rekursionsschritt  
}
```

⇒ Rekursionsverankerung verhindert endlose Rekursion!



Ablagefächer (Stack)




```
unsigned long fak(unsigned int n) {  
    if (n == 0) return 1;    // Rekursionsverankerung  
    return n * fak(n - 1);  // Rekursionsschritt  
}
```

Beobachtung:

1. Der Basisfall des Problems muss gelöst werden können (Rekursionsverankerung).
2. Bei jedem rekursiven Aufruf müssen kleinere Problemgrößen übergeben werden.

Weiteres Beispiel:

Bestimme den größten gemeinsamen Teiler (ggT) zweier Zahlen

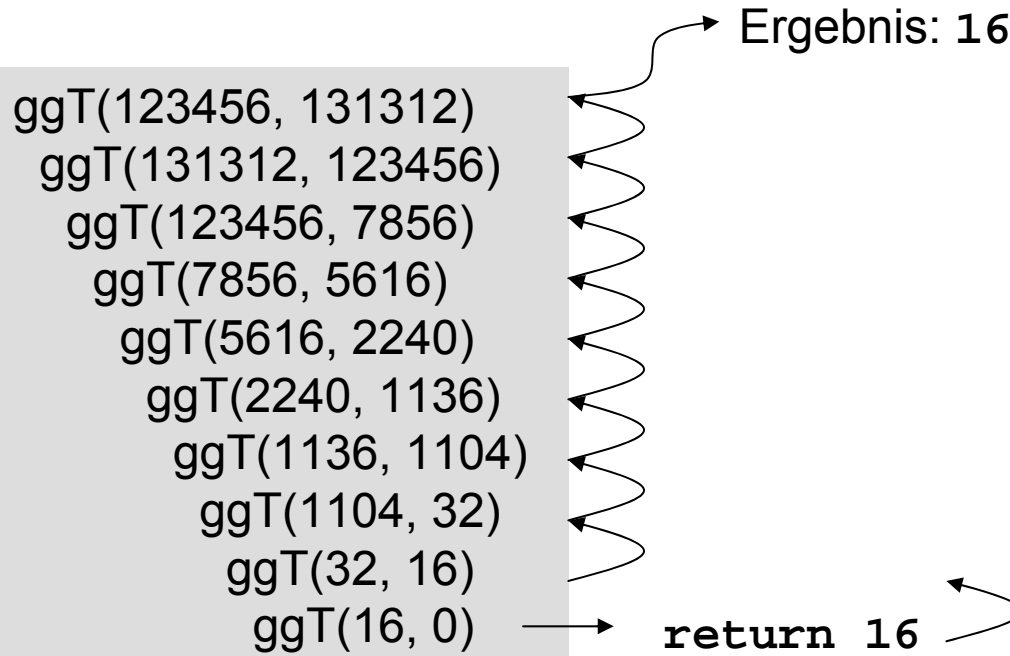
⇒ Euklidischer Algorithmus (> 2000 Jahre)

in C++:

```
unsigned int ggT(unsigned int a, unsigned int b) {  
    if (b == 0) return a;    // Rekursionsverankerung  
    return ggT(b, a % b);    // Rekursionsschritt  
}
```



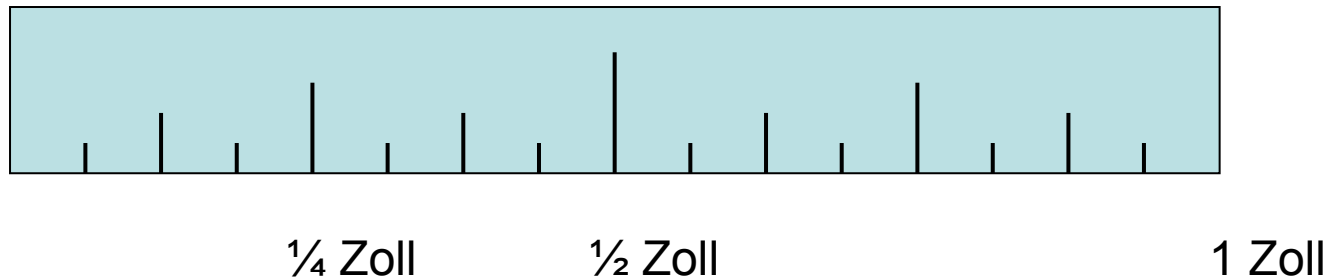
Verkleinerung des Problems



Abbruchbedingung!

Noch ein Beispiel:

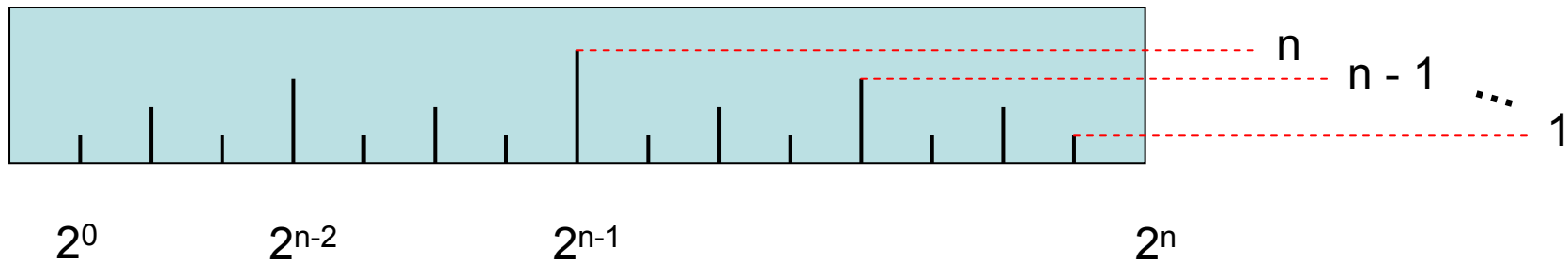
Zeichne Maßstriche auf ein (amerikanisches) Lineal



- Marke bei $\frac{1}{2}$ Zoll
- kleinere Marke bei je $\frac{1}{4}$ Zoll
- noch kleinere Marke bei je $\frac{1}{8}$ Zoll
- u.s.w. immer kleinere Marken bei je $\frac{1}{2^n}$

Annahme: Auflösung soll $1/2^n$ für gegebenes n sein

⇒ Maßstabsänderung:



Idee:

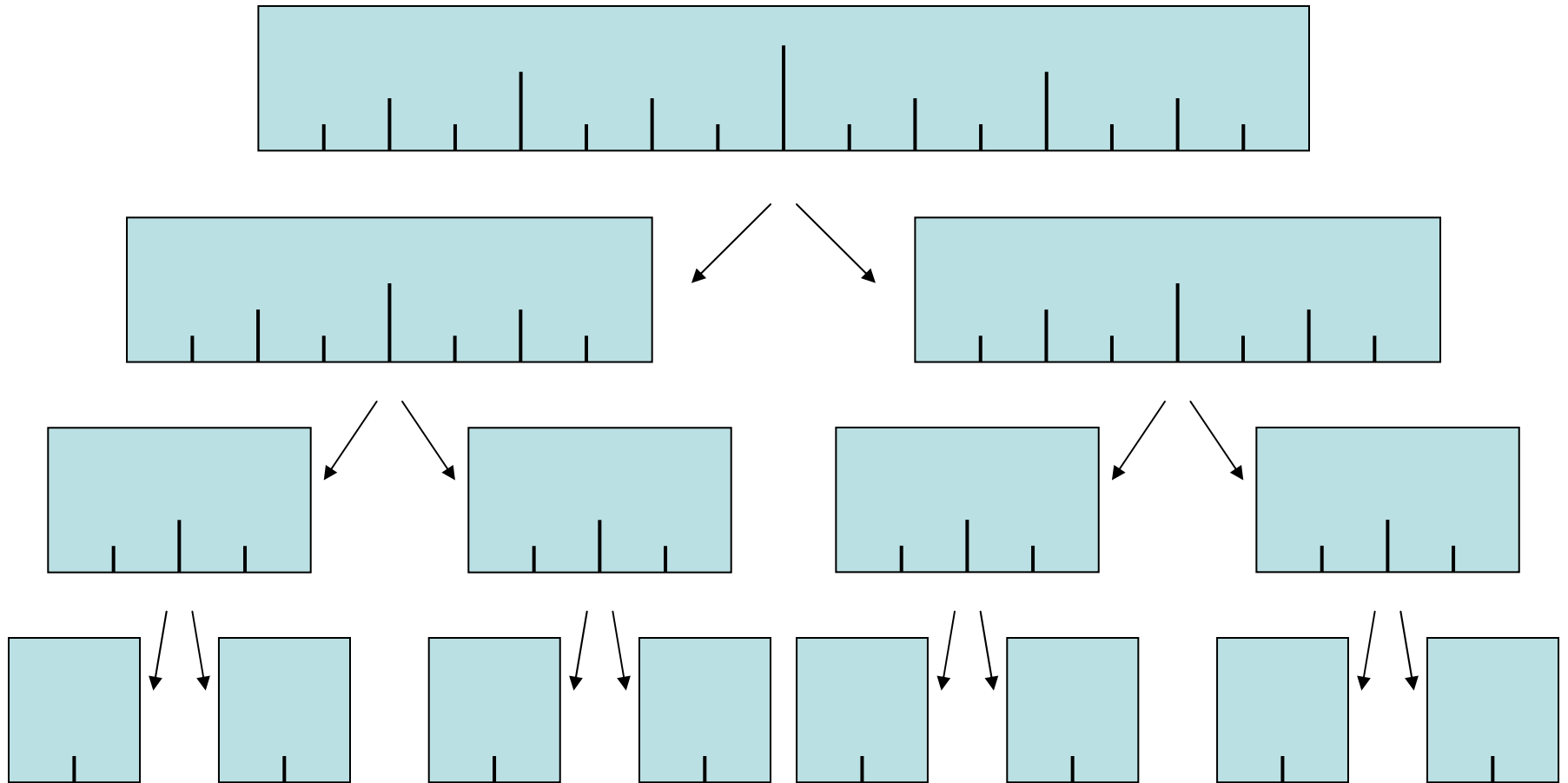
Teile Intervall in 2 gleich große Hälften,

zeichne linkes, halb so großes Lineal mit kürzerer Marke ← rekursiv!

erzeuge längere Marke in der Mitte

zeichne rechtes, halb so großes Lineal mit kürzerer Marke ← rekursiv!

Illustration:



Also:

Zeichnen des Lineals wird so lange auf kleinere Probleme / Lineale vereinfacht,
bis wir das elementare Problem / Lineal lösen können:

„Zeichne eine Marke der Höhe 1“

Jetzt: Rekursionsaufstieg

linkes (elementares) Lineal zeichnen

zeichne Marke der Höhe h ($= 2$)

rechtes (elementares) Lineal zeichnen



⇒ Teilproblem gelöst!

Implementierung

Welche Parameter spielen eine Rolle?

linker Rand des Teil-Lineals → **li**

rechter Rand des Teil-Lineals → **re**

Höhe der Marke → **h**

Mitte des Teil-Lineals (für die Marke) → **mi**

```
void Lineal(unsigned int li, unsigned int re, unsigned int h) {  
    unsigned int mi = (li + re) / 2;  
    if (h > 0) {  
        Lineal(li, mi, h - 1);  
        Marke(mi, h);  
        Lineal(mi, re, h - 1);  
    }  
}
```


Implementierung

Zeichnen der Marken (mehrere Möglichkeiten)

hier: wir wissen, dass Marken von links nach rechts gezeichnet werden

⇒ Testausgabe mit senkrechtem Lineal (Marken von oben nach unten)

```
void Marke(unsigned int position, unsigned int hoehe) {  
    while (hoehe--) cout << '-';  
    cout << endl;  
}
```

Anmerkung:

`position` wird hier nicht gebraucht, aber andere Situationen vorstellbar

Implementierung

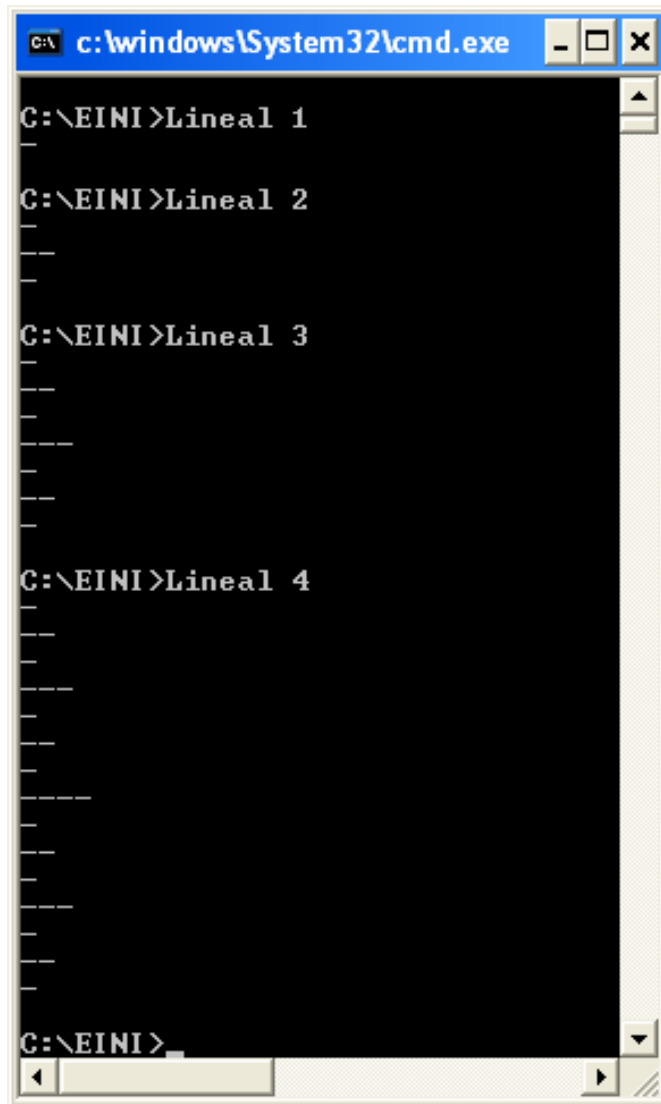
Hauptprogramm zum Testen

```
#include <iostream>
using namespace std;

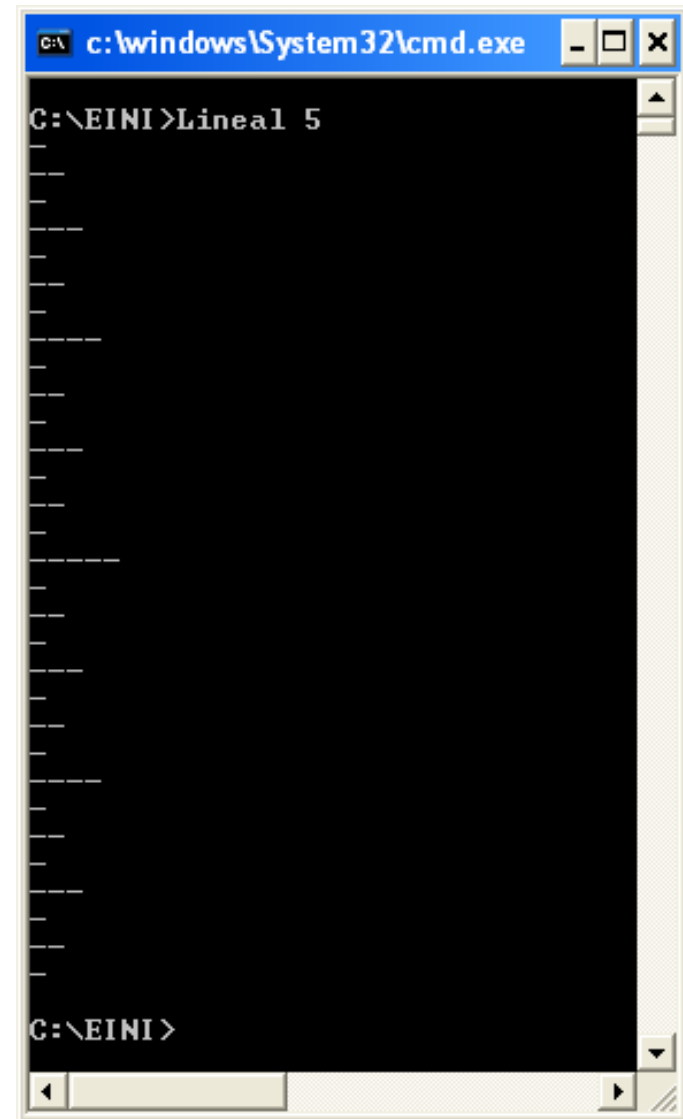
int main(int argc, char *argv[]) {
    if (argc != 2) {
        cerr << "usage: " << argv[0] << ": n" << endl;
        return 1;
    }
    unsigned int n = atoi(argv[1]);
    Lineal(0, 1 << n, n);
    return 0;
}
```

<< im numerischen Ausdruck: $x \ll n$
schiebt Bitmuster von x um n Bits nach links.

Was bedeutet $x \gg n$?

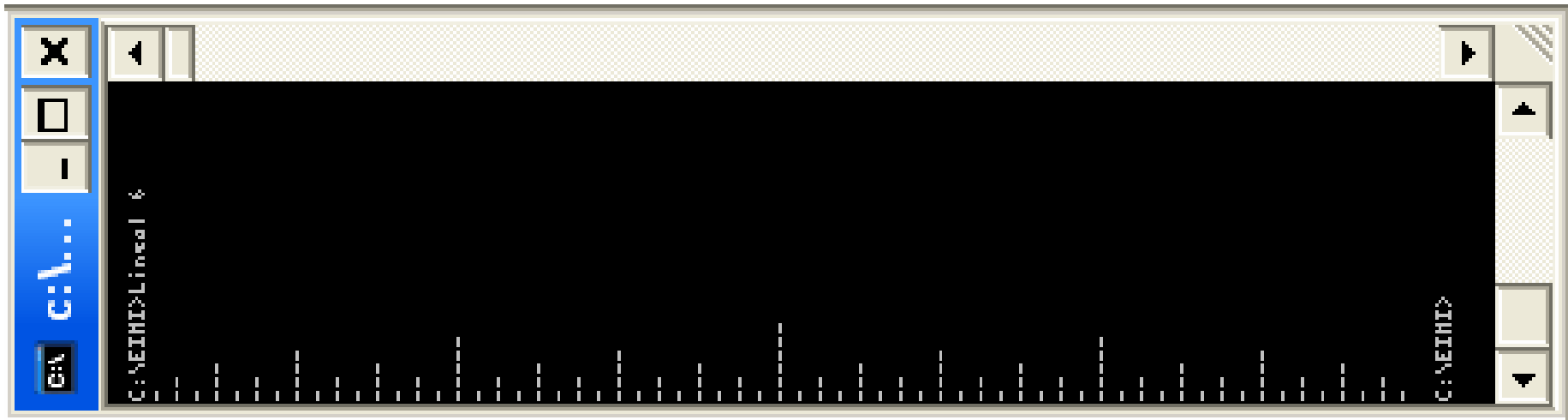


```
C:\windows\System32\cmd.exe
C:\EINI>Lineal 1
_
C:\EINI>Lineal 2
_
_
C:\EINI>Lineal 3
_
_
_
_
_
C:\EINI>Lineal 4
_
_
_
_
_
_
_
_
_
C:\EINI>
```



```
C:\windows\System32\cmd.exe
C:\EINI>Lineal 5
_
_
_
_
_
_
_
_
_
_
_
_
_
_
_
_
_
_
_
_
_
C:\EINI>
```

Lineal mit $2^6 = 64$ Marken:



Rekursion vs. Iteration

Theorem:

Jeder iterative Algorithmus lässt sich rekursiv formulieren und umgekehrt!

Wofür also das alles?

⇒ Manche Probleme lassen sich mit Rekursion sehr elegant + einfach lösen.

⇒ Lösung durch Iteration kann komplizierter sein!

Andererseits:

⇒ Nicht jedes Problem lässt sich durch Rekursion effizient lösen!

⇒ Iterative Lösung kann viel effizienter (auch einfacher) sein.

Rekursion vs. Iteration

beide einfach,
aber nicht gleich effizient

Iterative Lösung zur Fakultät:

```
unsigned long fak(unsigned int n) {
    unsigned int wert = 1;
    while (n > 0) wert *= n--;
    return wert;
}
```

1 Funktionsaufruf
1 x 2 Ablagefächer
1 lokale Variable

Rekursive Lösung zur Fakultät:

```
unsigned long fak(unsigned int n) {
    if (n == 0) return 1;
    return n * fak(n - 1);
}
```

n Funktionsaufrufe
n x 2 Ablagefächer
0 lokale Variable

Rekursion vs. Iteration

```
void Lineal(unsigned int li,unsigned int re,unsigned int h) {  
    for (int t = 1, j = 1; t <= h; j += j, t++)  
        for (int i = 0; li + j + i <= re; i += j + j)  
            Marke(li + j + i, t);  
}
```

Zeichnet erst alle Marken der Höhe 1,
dann 2, usw. mit Auslassungen

```
void Lineal(unsigned int li,unsigned int re,unsigned int h) {  
    unsigned int mi = (li + re) / 2;  
    if (h > 0) {  
        Lineal(li, mi, h - 1);  
        Marke(mi, h);  
        Lineal(mi, re, h - 1);  
    }  
}
```

Rekursion vs. Iteration

Zur einfachen Übertragung rekursiver Algorithmen
in iterative äquivalente Form benötigen wir spezielle Datenstrukturen (**stack**).

Diese und einige andere werden in späteren Kapitel eingeführt.

⇒ **Elementare Datenstrukturen**

Intervallschachtelung

Bestimme Nullstelle einer streng monotonen Funktion $f: [a, b] \rightarrow \mathbb{R}$

Annahme: $f(a) \cdot f(b) < 0$, also haben $f(a)$ und $f(b)$ verschiedene Vorzeichen.

```
double nullstelle(double a, double b) {  
    double const eps = 1.0e-10;  
    double fa = f(a), fb = f(b);  
    double c = (a + b) / 2., fc = f(c);  
    if (fabs(fa - fb) < eps) return c;  
    return (fa < 0 && fc < 0 || fa > 0 && fc > 0) ?  
        nullstelle(c, b) : nullstelle(a, c);  
}
```

