



# Union-Find-Strukturen

Prof. Dr. S. Albers  
Prof. Dr. Th. Ottmann

# Union-Find Strukturen

---

## Problem:

Verwaltung einer Familie von paarweise disjunkten Mengen unter den Operationen:

### *e.make-set():*

Erzeugt eine neue Menge mit Element  $e$ .

### *e.find-set():*

Liefert diejenige Menge  $M_i$ , die das Element  $e$  enthält.

# Union-Find Strukturen

---

*union*( $M_i, M_j$ ):

Vereinigt die Mengen  $M_i$  und  $M_j$  zu einer neuen Menge.

**Repräsentation** der Mengen  $M_i$  :

$M_i$  wird durch ein repräsentatives (kanonisches) Element aus  $M_i$  identifiziert.

# Union-Find Strukturen

---

## Operationen mit repräsentativen Elementen:

### *e.make-set():*

Erzeugt eine neue Menge mit einzigem Element  $e$ . Die Menge wird durch  $e$  repräsentiert.

### *e.find-set():*

Liefert den Namen des Repräsentanten derjenigen Menge, die das Element  $e$  enthält.

### *e.union(f):*

Vereinigt die Mengen  $M_e$  und  $M_f$ , die die Elemente  $e$  und  $f$  enthalten zu einer neuen Menge  $M$  und liefert ein Element aus  $M_e \cup M_f$  als Repräsentanten von  $M$ .  $M_e$  und  $M_f$  werden zerstört.

# Folgerungen

---

- Ist  $n$  die Anzahl der *e.make-set()* Operationen und werden insgesamt  $m$  Operationen *e.make-set()*, *e.find-set()*, *e.union(f)* ausgeführt, so ist
  - $m \geq n$
  - nach höchstens  $(n - 1)$  union – Operationen besteht die Kollektion von Mengen nur noch aus einer einzigen Menge

# Anwendungs-Beispiel: Zusammenhangstest



**Input:** Graph  $G = (V, E)$

**Output:** die Familie der Zusammenhangskomponenten von  $G$

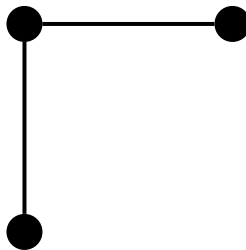
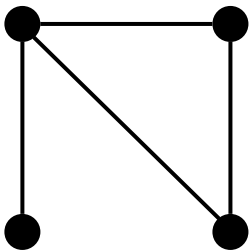
**Algorithmus: Connected-Components**

**for all**  $v$  **in**  $V$  **do**  $v.\text{makeset}()$

**for all**  $(u, v)$  **in**  $E$  **do**

**if**  $u.\text{findset}() \neq v.\text{findset}()$

**then**  $u.\text{union}(v)$

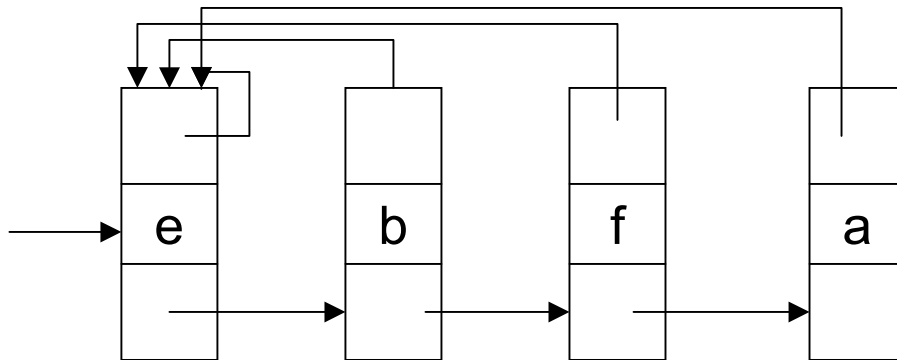


**Same-Component**  $(u, v)$ :

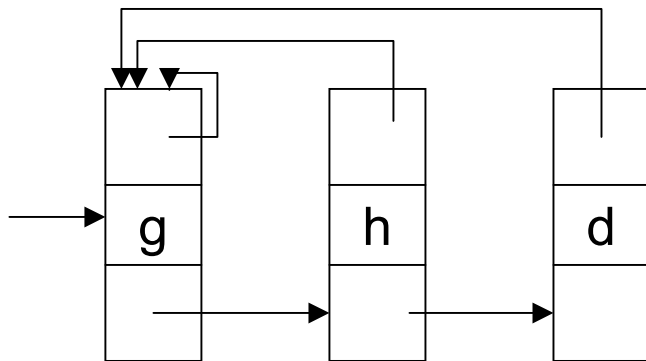
**if**  $u.\text{findset}() = v.\text{findset}()$

**then return true**

# Repräsentation durch verkettete Listen

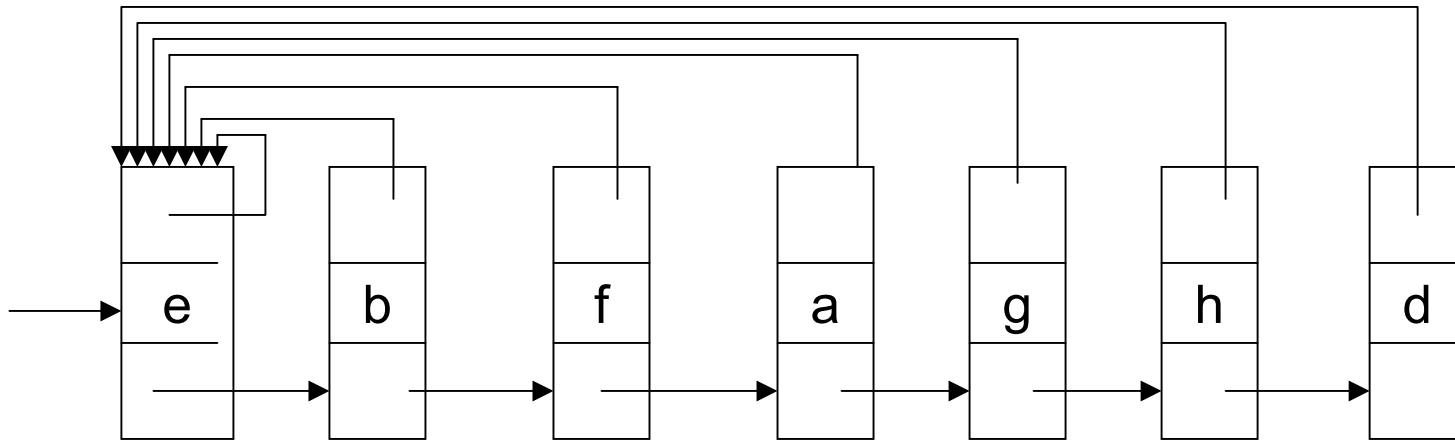


- *x.make-set()*
- *x.find-set()*
- *x.union(y)*



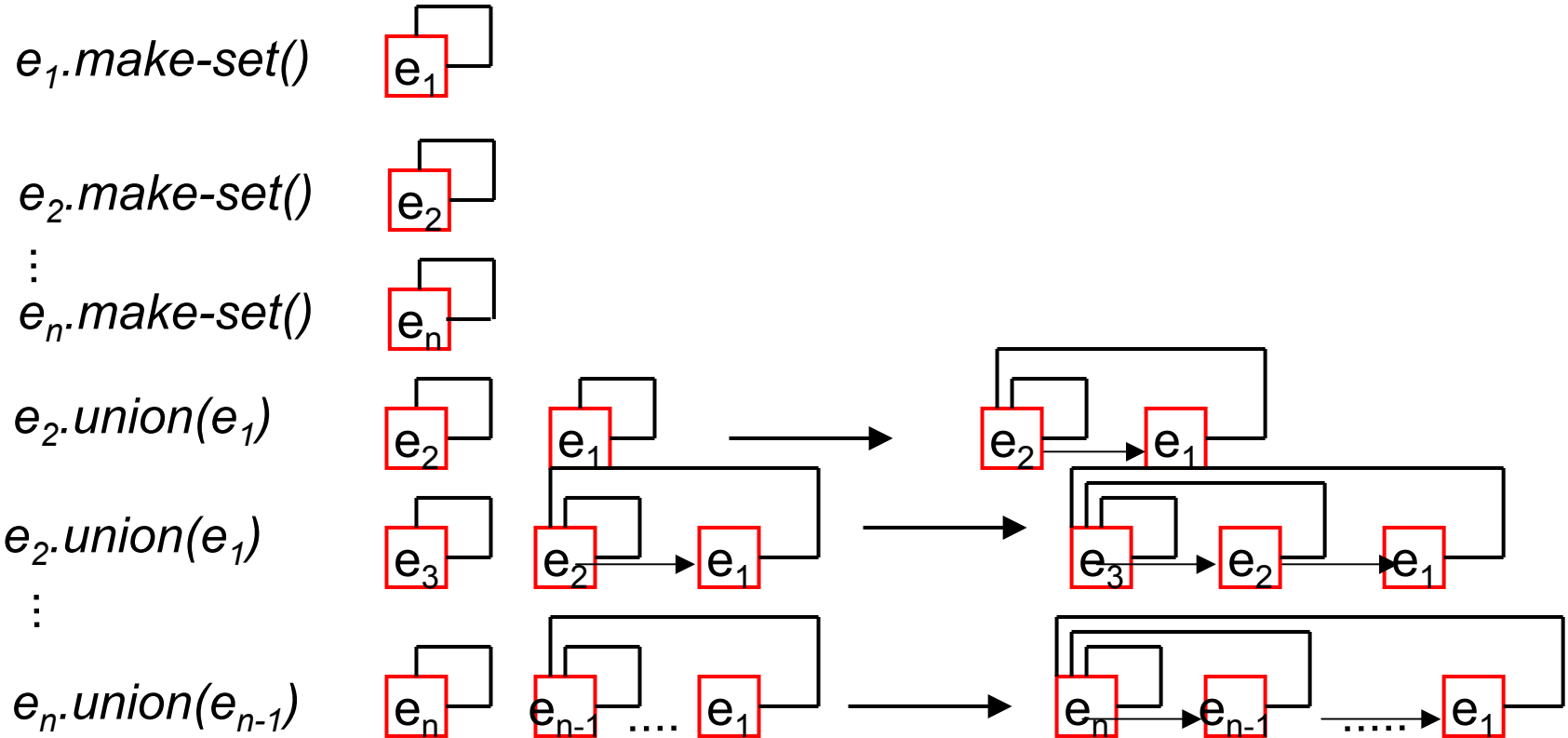
# Repräsentation durch verkettete Listen

*b.union(d)*





# Teurere Operations-Folge



**Längere Liste wird stets an kurze angehängt!**

Zeigerzuweisungen im  $i$ -ten Schritt  $e_i.\text{union}(e_{i-1})$ :

$2n$  Operationen kosten Zeit:

# Verbesserung

## Gewichtete Union-Heuristik

**Hänge stets die kürzere an die längere Liste.  
(Führe Listenlänge als Parameter mit).**

### Satz

Bei Verwendung der gewichteten Union-Heuristik kann jede Folge von  $m$  *make-set()*, *find-set()*, und *union()*-Operationen, die höchstens  $n$  *make-set()* Operationen enthält, in Zeit  $O(m + n \log n)$  ausgeführt werden.

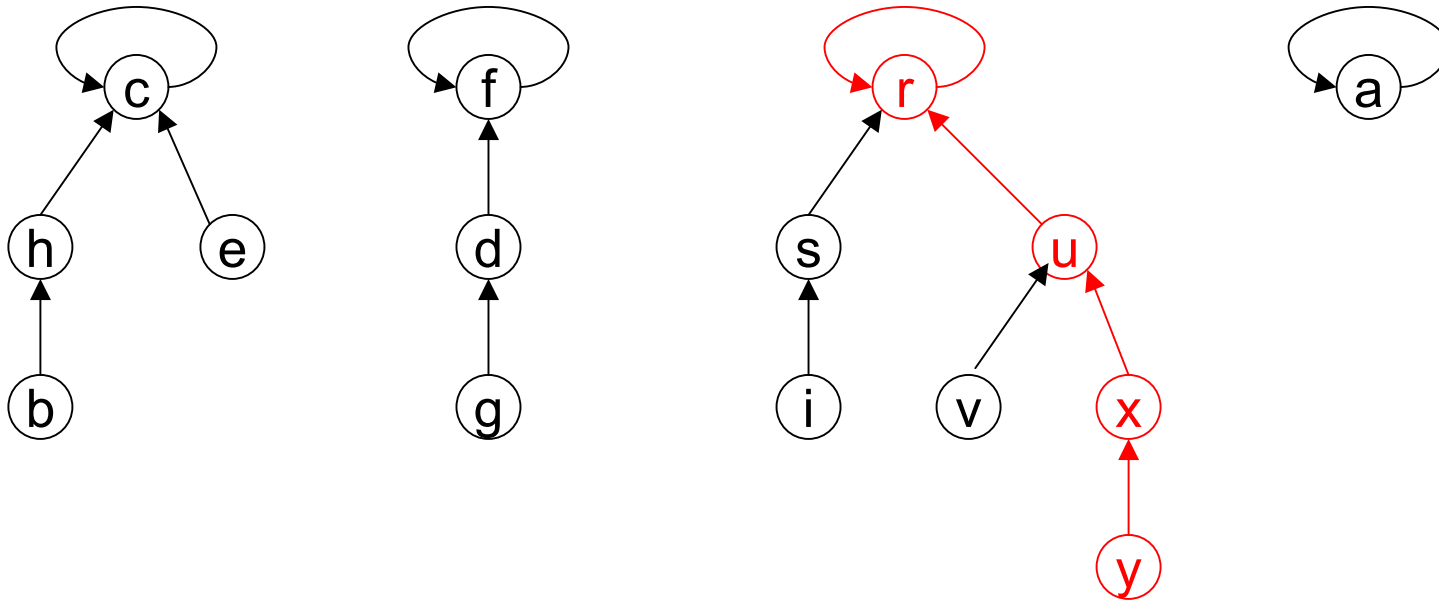
### Beweis

Betrachte Element  $e$

Anzahl der Änderungen des Zeigers von  $e$  auf das repräsentative Element:

$\log n$

# Repräsentation durch Wald von Bäumen



- `a.make-set()`
- `y.find-set()`
- `d.union(e)`: mache den Repräsentanten der einen Menge (z.B. `f`) zum direkten Vater des Repräsentanten.

# Beispiel

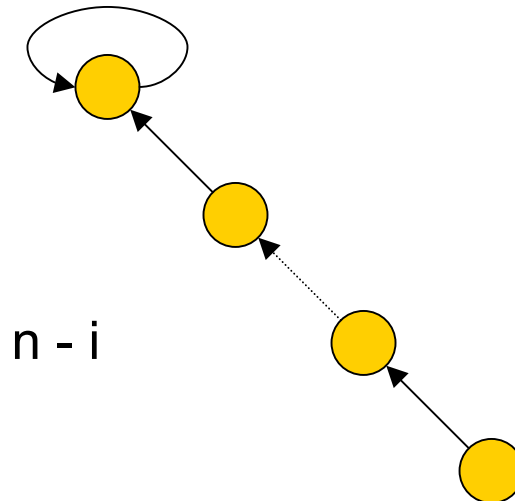
$m$  = Gesamtanzahl der Operationen ( $\geq 2n$ )

**for**  $i = 1$  **to**  $n$  **do**  $e_i$ .make-set( )

**for**  $i = n$  **to**  $2$  **do**  $e_{i-1}$ .union( $e_i$ )

**for**  $i = 1$  **to**  $f$  **do**  $e_n$ .find-set( )

*$i$  – ter Schritt*



Kosten für  $f$  find-set Operationen:  $O(f * n)$

# Vereinigung nach Größe

---

## zusätzliches Feld:

*e.size* = (#Knoten im Teilbaum von *e*)

## *e.make-set()*

1 *e.parent* = *e*

2 *e.size* = 1

## *e.union(f)*

1 *Link*(*e.find-set*( ), *f.find-set*( ))

# Vereinigung nach Größe

---

*Link(e,f)*

```
1 if e.size ≥ f.size
2   then f.parent = e
3         e.size = e.size + f.size
4   else /* e.size < f.size */
5         e.parent = f
6         f.size = e.size + f.size
```

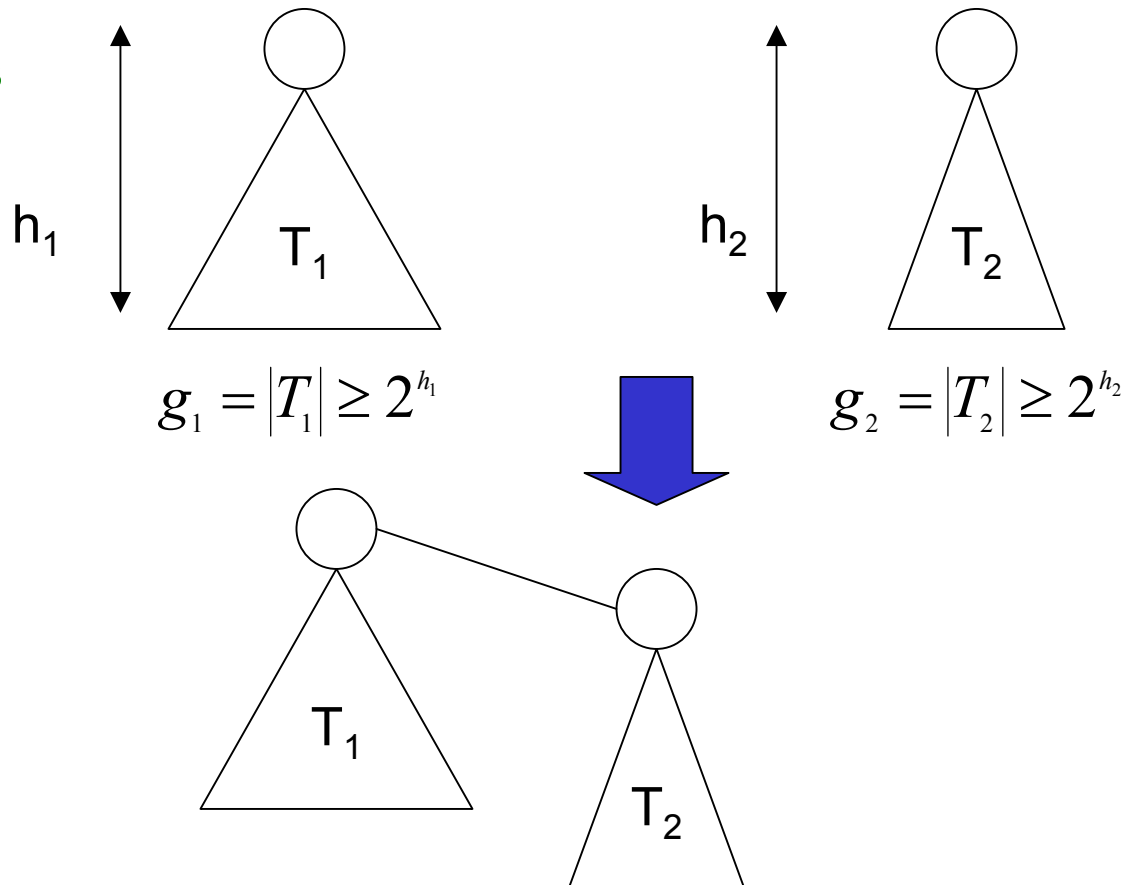
# Vereinigung nach Größe

## Satz

Das Verfahren Vereinigung nach Größe hat folgende Invariante:

Ein Baum mit Höhe  $h$  hat wenigstens  $2^h$  Knoten

## Beweis



# Vereinigung nach Größe

**Fall 1:** Der neue Baum ist genauso hoch wie  $T_1$

$$g_1 + g_2 \geq g_1 \geq 2^{h_1}$$

**Fall 2 :** Der neue Baum ist gewachsen

Höhe von  $T$ :  $h_2 + 1$

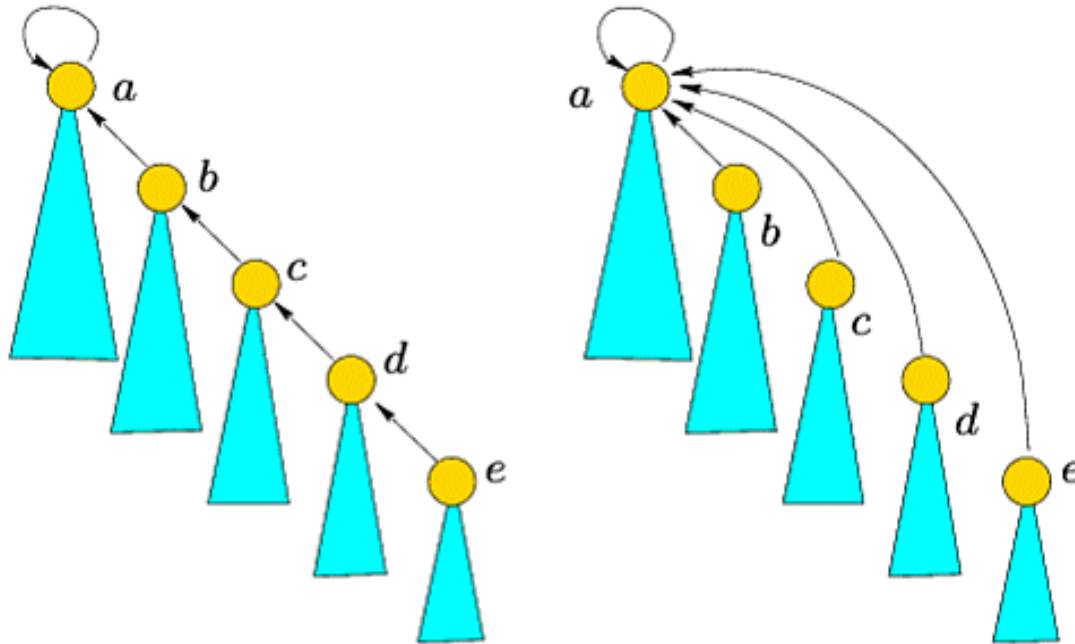
$$g = g_1 + g_2 \geq 2^{h_2} + 2^{h_2} = 2^{h_2+1}$$

## Konsequenz

Eine *find-set*-Operation kostet höchstens  $O(\log n)$  viele Schritte, falls die Anzahl der *make-set*-Operationen gleich  $n$  ist.



# Pfadverkürzung



*e.find-set()*

- 1 **if**  $e \neq e.parent$
- 2     **then**  $e.parent = e.parent.find-set()$
- 3 **return**

# Analyse der Laufzeit

---

$m$  Gesamtanzahl der Operationen, davon

$f$  *find-Set-Operation* und

$n$  *make-Set-Operation*.

→ höchstens  $n - 1$  *union-Operationen*

## Vereinigung nach Größe:

$O(n + f \log n)$

## Vereinigung nach Pfadverkürzung:

Falls  $f < n$ ,  $\Theta(n + f \log n)$

Falls  $f \geq n$ ,  $\Theta(f \log_{1+f/n} n)$

# Analyse der Laufzeit

---

**Satz** ( Vereinigung nach Größe und Pfadverkürzung)

Bei der Verwendung der Strategie *Vereinigung nach Größe* und *Pfadverkürzung* benötigen  $m$  Union-Find-Operation über  $n$  Elementen  $\Theta(m * \alpha(m,n))$  Schritte.

$\alpha(m,n)$  = Inverse der Ackermann-Funktion

# Ackermann-Funktion und Inverse

## Ackermann-Funktion

$$\begin{aligned} A(1, j) &= 2^j && \text{für } j \geq 1 \\ A(i, 1) &= A(i - 1, 2) && \text{für } i \geq 2 \\ A(i, j) &= A(i - 1, A(i, j - 1)) && \text{für } i, j \geq 2 \end{aligned}$$

## Inverse Ackermann-Funktion

$$\alpha(m, n) = \min \{ i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log n \}$$

# Ackermann-Funktion und Inverse

$$A(i, \lfloor m/n \rfloor) \geq A(i, 1)$$

$$A(2, 1) = A(1, 2) = 2^2 = 4$$

$$A(3, 1) = A(2, 2) = A(1, A(2, 1)) = 2^4 = 16$$

$$A(4, 1) = A(3, 2) = A(2, A(3, 1)) = A(2, 16) \\ \geq 2^{2^{2^2}} = 2^{65536}$$

$$\alpha(m, n) \leq 4, \text{ für } \log n < 2^{65536}$$