

# Die Ackermannfunktion

Prof. Dr. David Sabel

LFE Theoretische Informatik



# Die Ackermannfunktion

---

- Von Wilhelm Ackermann in 1920er Jahre vorgeschlagen
- sehr schnell wachsende Funktion
- Variante von Rózsa Péter:

Ackermann-Funktion  $a : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$

$$a(x, y) = \begin{cases} y + 1, & \text{falls } x = 0 \\ a(x - 1, 1) & \text{falls } x \neq 0 \text{ und } y = 0 \\ a(x - 1, a(x, y - 1)) & \text{falls } x > 0 \text{ und } y > 0 \end{cases}$$

# Die Ackermannfunktion

- Von Wilhelm Ackermann in 1920er Jahre vorgeschlagen
- sehr schnell wachsende Funktion
- Variante von Rózsa Péter:

Ackermann-Funktion  $a : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$

$$a(x, y) = \begin{cases} y + 1, & \text{falls } x = 0 \\ a(x - 1, 1) & \text{falls } x \neq 0 \text{ und } y = 0 \\ a(x - 1, a(x, y - 1)) & \text{falls } x > 0 \text{ und } y > 0 \end{cases}$$

Unser nächstes Ziel:

Die Ackermannfunktion ist WHILE-berechenbar, aber:  
die Ackermannfunktion ist **nicht** LOOP-berechenbar  
(obwohl sie total ist)

# Einige Werte der Ackermannfunktion

Tabelle mit  $a(x, y)$ -Einträgen:

| $y \backslash x$ | 0 | 1 | 2  | 3   | 4                     |
|------------------|---|---|----|-----|-----------------------|
| 0                | 1 | 2 | 3  | 5   | 13                    |
| 1                | 2 | 3 | 5  | 13  | 65533                 |
| 2                | 3 | 4 | 7  | 29  | $2^{65536} - 3$       |
| 3                | 4 | 5 | 9  | 61  | $a(3, 2^{65536} - 3)$ |
| 4                | 5 | 6 | 11 | 125 | $a(3, a(4, 3))$       |

$$a(x, y) = \begin{cases} y + 1, & \text{falls } x = 0 \\ a(x - 1, 1) & \text{falls } x \neq 0 \text{ und } y = 0 \\ a(x - 1, a(x, y - 1)) & \text{falls } x > 0 \text{ und } y > 0 \end{cases}$$

Mehrmaliges Entfalten der dritten Zeile zeigt:

$$a(x, y) = \underbrace{(a(x - 1, a(x - 1, a(x - 1, \dots, a(x - 1, 1) \dots))))}_{y+1\text{-mal}}$$

Daher sind alle rekursiven Aufrufe echt kleiner und daher terminiert die Ackermannfunktion stets.

## Lemma

Die Ackermannfunktion ist total.

# Berechenbarkeit der Ackermannfunktion

---

Intuitiv klar: Mit jeder modernen Programmiersprache ist die Ackermannfunktion implementierbar. Daher ist sie auch (intuitiv) berechenbar.

# Berechenbarkeit der Ackermannfunktion

---

Intuitiv klar: Mit jeder modernen Programmiersprache ist die Ackermannfunktion implementierbar. Daher ist sie auch (intuitiv) berechenbar.

## **Satz**

Die Ackermannfunktion ist WHILE-berechenbar.

# Berechenbarkeit der Ackermannfunktion

---

Intuitiv klar: Mit jeder modernen Programmiersprache ist die Ackermannfunktion implementierbar. Daher ist sie auch (intuitiv) berechenbar.

## Satz

Die Ackermannfunktion ist WHILE-berechenbar.

Beweis: Erstelle ein WHILE-Programm, welches die rekursive Berechnung durchführt mit einem Stack.

Daher ist der erste Schritt im Beweis:

- Darstellung des Stacks
- Implementierung von Operationen auf dem Stack als WHILE-Programm.

Danach wird das Programm angegeben, dass die Stackoperationen durchführt.



Darstellung von Tupelfolgen mit fester Länge  $k$  als eine einzige Zahl:

- $\langle n_1, \dots, n_k, 0 \rangle = c(n_1, c(n_2, (\dots, c(n_k, 0) \dots)))$
- wobei  $c(x, y)$  eine Bijektion  $(\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$  ist
- Wir nehmen an, dass  $left$  und  $right$  existieren mit  $left(c(x, y)) = x$  und  $right(c(x, y)) = y$
- Genauere Details zu  $c, left, right$  werden später nochmal erörtert

# Ackermannfunktion mit WHILE berechnen

---

Stack:

- Stack: Folge von Zahlen  $n_1, \dots, n_k, 0$  sodass  $n_1$  ganz oben liegt. Leerer Stack: 0 markiert Kellerboden.
- Stack im WHILE-Programm: Variable *stack*, die Zahl  $\langle n_1, \dots, n_k, 0 \rangle = c(n_1, c(n_2, (\dots, c(n_k, 0) \dots)))$  speichert, und *stacksize*, die Größe des Stacks speichert

# Ackermannfunktion mit WHILE berechnen

Stack:

- Stack: Folge von Zahlen  $n_1, \dots, n_k, 0$  sodass  $n_1$  ganz oben liegt. Leerer Stack: 0 markiert Kellerboden.
- Stack im WHILE-Programm: Variable  $stack$ , die Zahl  $\langle n_1, \dots, n_k, 0 \rangle = c(n_1, c(n_2, (\dots, c(n_k, 0) \dots)))$  speichert, und  $stacksize$ , die Größe des Stacks speichert

Stack-Operationen:

- $push(x, stack)$  legt Zahl  $x$  oben auf den Stack.  
WHILE-Programm dazu:  $stack := c(x, stack); stacksize := stacksize + 1$ .
- $x := pop(stack)$  entfernt das oberste Element vom Stack und setzt  $x$  auf dessen Wert.  
WHILE-Programm:  $x := left(stack); stack = right(stack); stacksize := stacksize - 1$
- $stacksize \neq 1$ : WHILE-Programm dazu existiert

## WHILE-Programm zur Berechnung von $a(x, y)$

---

```
stack := 0;
stacksize := 0;
push(x, stack);
push(y, stack);
WHILE size(stack)  $\neq$  1 DO
  y := pop(stack);
  x := pop(stack);
  IF x = 0
    THEN push(y + 1, stack);
    ELSE IF y = 0 THEN push(x - 1, stack); push(1, stack)
      ELSE push(x - 1, stack); push(x, stack), push(y - 1, stack);
    END
  END
END
result := pop(stack);
```

## Lemma

Die folgenden Monotonie-Eigenschaften gelten für die Ackermannfunktion  $a$ :

- 1  $y < a(x, y)$
- 2  $a(x, y) < a(x, y + 1)$
- 3  $a(x, y + 1) \leq a(x + 1, y)$
- 4  $a(x, y) < a(x + 1, y)$
- 5 Falls  $x \leq x'$  und  $y \leq y'$ , dann gilt auch  $a(x, y) \leq a(x', y')$

Beweise im Skript (Induktion über  $x$  bzw.  $y$ , bzw  $(x, y)$ )

# Maximale LOOP-berechenbare Zahlen

- Sei  $P$  ein LOOP-Programm
- Seien  $x_0, \dots, x_k$  die in  $P$  vorkommenden Variablen.
- Sei

$$f_P(n) = \max \left\{ \sum_{i=0}^k \rho'(x_i) \mid \sum_{i=0}^k \rho(x_i) \leq n, (\rho, P) \xrightarrow[\text{LOOP}]^* (\rho', \varepsilon) \right\}$$

- $f_P(n)$  ist die maximale Zahl, die als Summe aller Endbelegungen  $\rho'$  der Variablen  $x_0, \dots, x_k$  zustande kommt, über alle initialen Variablenbelegung  $\rho$ , die in der Summe der Werte  $\rho(x_0), \dots, \rho(x_k)$  den Wert  $n$  nicht überschreiten.

## Maximale LOOP-berechenbare Zahlen (2)

### Satz

Für jedes LOOP-Programm  $P$  gibt es eine Konstante  $k$ , so dass  $f_P(n) < a(k, n)$  für alle  $n \in \mathbb{N}$ .

Beweis: Normalisiere  $P$  zunächst:

- Ersetze Zuweisungen  $x_i := x_j + c$  durch  $x_i := x_j + 1; \underbrace{x_i := x_i + 1; \dots x_i := x_i + 1}_{c-1 \text{ mal}}$ .
- Für **LOOP**  $x_i$  **DO**  $Q$  **END** und  $x_i$  kommt in  $Q$  vor:  
Ersetze **LOOP**  $x_i$  **DO**  $Q$  **END** durch  $x'_i := x_i; \text{LOOP } x'_i \text{ DO } Q \text{ END}; x'_i := 0$ , wobei  $x'_i$  eine neue Variable ist.
- Beides verändert  $f_P$  nicht!

Zeige Behauptung für normalisierte Programme.

## Maximale LOOP-berechenbare Zahlen (3)

---

Zu zeigen: Für jedes normalisierte LOOP-Programm  $P$  gibt es eine Konstante  $k$ , so dass  $f_P(n) < a(k, n)$  für alle  $n \in \mathbb{N}$ .



## Maximale LOOP-berechenbare Zahlen (3)

---

Zu zeigen: Für jedes normalisierte LOOP-Programm  $P$  gibt es eine Konstante  $k$ , so dass  $f_P(n) < a(k, n)$  für alle  $n \in \mathbb{N}$ .

Beweis durch strukturelle Induktion über normalisiertes Programm.

## Maximale LOOP-berechenbare Zahlen (3)

---

Zu zeigen: Für jedes normalisierte LOOP-Programm  $P$  gibt es eine Konstante  $k$ , so dass  $f_P(n) < a(k, n)$  für alle  $n \in \mathbb{N}$ .

Beweis durch strukturelle Induktion über normalisiertes Programm.

Fälle:

- 1 Zuweisung  $x_i := x_j + c$  mit  $c \in \mathbb{Z}$  und  $c \leq 1$
- 2 Sequenz  $P_1; P_2$
- 3 LOOP-Schleife **LOOP**  $x_i$  **DO**  $Q$  **END**,  
wobei  $x_i$  nicht in  $Q$  vorkommt.

## Maximale LOOP-berechenbare Zahlen (4)

Fall: Zuweisung  $x_i := x_j + c$  mit  $c \in \mathbb{Z}$  und  $c \leq 1$

Dann gilt  $f_P(n) \leq 2n + 1$ , da im maximalen Fall:

- $\rho(x_k) = 0$  für  $k \neq j$
- $\rho(x_j) = n$
- $c = 1$  und
- $\rho'(x_i) = n + 1$ ,
- $\rho'(x_j) = n$  und
- $\rho'(x_k) = 0$  für  $k \neq j$  und  $k \neq i$

Mit  $a(2, y) = 2y + 3$  (s. Skript) folgt

$$f_P(n) \leq 2n + 1 < 2n + 3 = a(2, n)$$

D.h. die Aussage  $f_P(n) < a(k, n)$  gilt mit  $k = 2$ .

## Maximale LOOP-berechenbare Zahlen (5)

Fall: Sequenz  $P_1; P_2$

Induktionsannahme:  $f_{P_i}(n) < a(k_i, n)$  für  $i = 1, 2$ .

Es gilt:

$$\begin{aligned} f_P(n) &= f_{P_2}(f_{P_1}(n)) \\ &< a(k_2, a(k_1, n)) \\ &\leq a(\max\{k_1, k_2\}, a(\max\{k_1, k_2\}, n)) && (1) \\ &\leq a(\max\{k_1, k_2\}, a(\max\{k_1, k_2\} + 1, n + 1)) && (1) \\ &= a(\max\{k_1, k_2\} + 1, n + 1) && (\text{Definition von } a) \\ &\leq a(\max\{k_1, k_2\} + 2, n) && (2) \end{aligned}$$

(1) Monotonie:  $x \leq x', y \leq y' \implies a(x, y) \leq a(x', y')$

(2) Monotonie:  $a(x, y + 1) \leq a(x + 1, y)$

Daher gilt  $f_P(n) < a(k, n)$  für  $k = \max\{k_1, k_2\} + 2$ .

## Maximale LOOP-berechenbare Zahlen (6)

---

Fall: **LOOP**  $x_i$  **DO**  $Q$  **END** und  $x_i$  kommt nicht in  $Q$  vor

Induktionsannahme:  $f_Q(n) < a(k_Q, n)$  für ein  $k_Q$ .

$f_P(n)$  berechnet Maximum. Sei  $\rho$  so, dass  $f_P(n)$  maximal ist.

- Wenn  $\rho(x_i) = 0$ , dann ist  $f_P(n) = n < a(0, n)$ .
- Wenn  $\rho(x_i) = 1$ , dann ist  $f_P(n) = f_Q(n) < a(k_Q, n)$
- Sei  $\rho(x_i) \geq 2$ . ...

## Maximale LOOP-berechenbare Zahlen (7)

Fall: **LOOP**  $x_i$  **DO**  $Q$  **END** und  $x_i$  kommt nicht in  $Q$  vor

- Sei  $\rho(x_i) \geq 2$ . ... Da  $x_i$  nicht in  $Q$  vorkommt, gilt:

$$\begin{aligned} f_P(n) &= \underbrace{f_Q(\dots (f_Q(n - \rho(x_i)) \dots))}_{\rho(x_i) \text{ mal}} + \rho(x_i) \quad (\text{da } x_i \notin Q, f_Q \text{ unabh. von } \rho(x_i)) \\ &\leq a(k_1, \dots, a(k_1, n - \rho(x_i)) \dots) \quad (\rho(x_i)\text{-mal gilt } < \text{ (für jedes } f_Q). \\ &\quad \text{Daher } \leq \text{ und } \rho(x_i) \text{ fällt weg)} \\ &< \underbrace{a(k_1, \dots, a(k_1, a(k_1 + 1, n - \rho(x_i)) \dots))}_{\rho(x_i) - 1 \text{ mal}} \quad (1) \\ &= a(k_1 + 1, n - \rho(x_i) + \rho(x_i) - 1) \quad (\text{Definition von } a) \\ &= a(k_1 + 1, n - 1) \leq a(k_1 + 1, n) \quad (2) \end{aligned}$$

(1) Monotonie:  $a(x, y) < a(x + 1, y)$  (2) Monotonie:  $a(x, y) < a(x, y + 1)$

Daher gilt  $f_P(n) < a(k, n)$  für  $k = k_1 + 1$

## Theorem

Die Ackermannfunktion ist nicht LOOP-berechenbar.

Beweis:

- Annahme:  $a$  ist LOOP-berechenbar.
- Dann ist auch  $f(x_1) = a(x_1, x_1)$  LOOP-berechenbar.
- Sei  $P$  LOOP-Programm, das  $f$  berechnet.
- Dann gilt  $f(x_1) \leq f_P(x_1)$  (da  $\rho'(x_0) = f(x_1)$  nach Ausführung von  $P$ )
- Es gibt Konstante  $k$ , sodass  $f_P(n) < a(k, n)$
- Starte  $P$  mit  $\rho = \{x_1 \mapsto k\}$ .
- Dann gilt  $f(k) \leq f_P(k) < a(k, k) = f(k)$
- Widerspruch!
- $a$  ist nicht LOOP-berechenbar.

## Theorem

Es gibt totale WHILE-berechenbare (bzw. GOTO-berechenbare, Turingberechenbare) Funktionen, die nicht LOOP-berechenbar sind.