# 9c
# Die Ackermannfunktion

Prof. Dr. Jasmin Blanchette

Lehr- und Forschungseinheit für
Theoretische Informatik und Theorembeweisen

Stand: 18. Juni 2024
Basierend auf Folien von PD Dr. David Sabel

# Die Ackermannfunktion

Die Ackermannfunktion ist eine von Wilhelm Ackermann in den 1920er Jahre vorgeschlagene sehr schnell wachsende Funktion.

Wir betrachten eine Variante von Rózsa Péter.

# Die Ackermannfunktion

Die Ackermannfunktion ist eine von Wilhelm Ackermann in den 1920er Jahre vorgeschlagene sehr schnell wachsende Funktion.

Wir betrachten eine Variante von Rózsa Péter.

Die Ackermannfunktion $a : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ ist definiert durch

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

# Die Ackermannfunktion

Die Ackermannfunktion ist eine von Wilhelm Ackermann in den 1920er Jahre vorgeschlagene sehr schnell wachsende Funktion.

Wir betrachten eine Variante von Rózsa Péter.

Die Ackermannfunktion $a : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ ist definiert durch

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

Unser nächstes Ziel:

**Theorem**

Die Ackermannfunktion ist WHILE-berechenbar, aber sie ist nicht LOOP-berechenbar (obwohl sie total ist).

# Einige Werte der Ackermannfunktion

Tabelle mit $a(x, y)$-Einträgen:

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

# Totalität der Ackermannfunktion

**Lemma**

Die Ackermannfunktion ist total.

# Totalität der Ackermannfunktion

**Lemma**

Die Ackermannfunktion ist total.

**Beweis** Mehrmaliges Entfalten der dritten Zeile in

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

zeigt

$$a(x,y) = \underbrace{a(x-1,a(x-1,\ldots,a(x-1,}_{(y+1)\text{-mal}}1)\ldots)) \qquad \text{falls } x > 0 \text{ und } y > 0$$

Alle rekursiven Aufrufe sind nun echt kleiner bezüglich des ersten Arguments.
Daher terminiert die Ackermannfunktion stets. □

# Monotonie-Eigenschaften der Ackermannfunktion

**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 $a(x, y) \leq a(x', y')$.

# Monotonie-Eigenschaften der Ackermannfunktion

**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 $a(x, y) \leq a(x', y')$.

**Beweis** Durch Induktion. Siehe Skript. □

# Berechenbarkeit der Ackermannfunktion

Es ist intuitiv klar, dass die Ackermannfunktion mit jeder modernen Programmiersprache implementierbar ist. Daher ist sie auch intuitiv berechenbar.

# Berechenbarkeit der Ackermannfunktion

Es ist intuitiv klar, dass die Ackermannfunktion mit jeder modernen Programmiersprache implementierbar ist. Daher ist sie auch intuitiv berechenbar.

Zudem:

**Satz**

Die Ackermannfunktion ist WHILE-berechenbar.

# Berechenbarkeit der Ackermannfunktion

Es ist intuitiv klar, dass die Ackermannfunktion mit jeder modernen Programmiersprache implementierbar ist. Daher ist sie auch intuitiv berechenbar.

Zudem:

**Satz**

Die Ackermannfunktion ist WHILE-berechenbar.

**Beweis** Erstelle ein WHILE-Programm, welches die rekursive Berechnung durchführt mit einem Keller.

# Berechenbarkeit der Ackermannfunktion

Es ist intuitiv klar, dass die Ackermannfunktion mit jeder modernen Programmiersprache implementierbar ist. Daher ist sie auch intuitiv berechenbar.

Zudem:

**Satz**

Die Ackermannfunktion ist WHILE-berechenbar.

**Beweis** Erstelle ein WHILE-Programm, welches die rekursive Berechnung durchführt mit einem Keller.

Daher ist der erste Schritt im Beweis:

- Darstellung des Kellers
- Implementierung von Operationen auf dem Keller als WHILE-Programme.

Danach wird das Programm angegeben, das die Kelleroperationen durchführt.

# Darstellung vom Keller

Darstellung vom Keller:

- Folge von Zahlen $\langle n_1, \ldots, n_k \rangle$ sodass $n_1$ ganz oben liegt.
  Leerer Keller: 0.

# Darstellung vom Keller

Darstellung vom Keller:

- Folge von Zahlen $\langle n_1, \ldots, n_k \rangle$ sodass $n_1$ ganz oben liegt.
  Leerer Keller: 0.

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

- $\langle n_1, \ldots, n_k \rangle = c(n_1, c(n_2, (\ldots, c(n_k, 0)\ldots)))$
  wobei $c(x, y)$ eine Bijektion $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ ist.

Wir nehmen an, dass *left* und *right* existieren mit $\mathit{left}(c(x, y)) = x$ und $\mathit{right}(c(x, y)) = y$. Genauere Details zu $c$, *left*, *right* werden später nochmal erörtert.

# Darstellung vom Keller

Darstellung vom Keller:

- Folge von Zahlen $\langle n_1, \ldots, n_k \rangle$ sodass $n_1$ ganz oben liegt.
  Leerer Keller: 0.

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

- $\langle n_1, \ldots, n_k \rangle = c(n_1, c(n_2, (\ldots, c(n_k, 0)\ldots)))$
  wobei $c(x, y)$ eine Bijektion $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ ist.

Wir nehmen an, dass *left* und *right* existieren mit $\mathit{left}(c(x, y)) = x$ und $\mathit{right}(c(x, y)) = y$. Genauere Details zu $c$, *left*, *right* werden später nochmal erörtert.

Darstellung vom Keller im WHILE-Programm:

- Variable *stack*, die $\langle n_1, \ldots, n_k \rangle = c(n_1, c(n_2, (\ldots, c(n_k, 0)\ldots)))$ speichert
- Variable *stacksize*, die die Größe des Kellers speichert
- Invariante: $\langle n_1, \ldots, n_k \rangle$ stellt $a(n_k, a(n_{k-1}, \ldots a(n_2, n_1)\ldots))$ dar.

# Kelleroperationen

- $push(x, stack)$ legt Zahl $x$ oben auf den Keller.
  WHILE-Programm dazu:

  $stack := c(x, stack);$
  $stacksize := stacksize + 1$

- $x := pop(stack)$ entfernt das oberste Element vom Keller und setzt $x$ auf dessen Wert.
  WHILE-Programm dazu:

  $x := left(stack);$
  $stack := right(stack);$
  $stacksize := stacksize - 1$

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

$\langle n_1, \ldots, n_k \rangle$ stellt $a(n_k, a(n_{k-1}, \ldots a(n_2, n_1) \ldots))$ dar

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

*stack* := 0;
*stacksize* := 0;
*push*($x$, *stack*);
*push*($y$, *stack*);
**WHILE** *stacksize* $\neq 1$ **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*)

□

# Maximale LOOP-berechenbare Zahlen

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

# Maximale LOOP-berechenbare Zahlen

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

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

# Maximale LOOP-berechenbare Zahlen

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

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

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

# Beispiel für eine maximale LOOP-berechenbare Zahl

Wir berechnen $f_P(5)$ für das LOOP-Programm $P$:

$x_0 := x_1 + 100$

# Beispiel für eine maximale LOOP-berechenbare Zahl

Wir berechnen $f_P(5)$ für das LOOP-Programm $P$:

$$x_0 := x_1 + 100$$

Wir wählen $\rho(x_1) = 5$ und $\rho(x_i) = 0$ für alle anderen Variablen $x_i$.
Daher $\sum_{i=0}^{k} \rho(x_i) = 5 \leq 5$.

# Beispiel für eine maximale LOOP-berechenbare Zahl

Wir berechnen $f_P(5)$ für das LOOP-Programm $P$:

$$x_0 := x_1 + 100$$

Wir wählen $\rho(x_1) = 5$ und $\rho(x_i) = 0$ für alle anderen Variablen $x_i$.
Daher $\sum_{i=0}^{k} \rho(x_i) = 5 \leq 5$.

Die Ausführung von $P$ führt zu folgender Variablenbelegung $\rho'$:

$$\begin{aligned} \rho'(x_0) &= 105 \\ \rho'(x_1) &= 5 \\ \rho'(x_i) &= 0 \quad \text{für alle anderen Variablen } x_i \end{aligned}$$

Daher $\sum_{i=0}^{k} \rho'(x_i) = 110$.

# Beispiel für eine maximale LOOP-berechenbare Zahl

Wir berechnen $f_P(5)$ für das LOOP-Programm $P$:

$$x_0 := x_1 + 100$$

Wir wählen $\rho(x_1) = 5$ und $\rho(x_i) = 0$ für alle anderen Variablen $x_i$.
Daher $\sum_{i=0}^{k} \rho(x_i) = 5 \leq 5$.

Die Ausführung von $P$ führt zu folgender Variablenbelegung $\rho'$:

$$\begin{aligned} \rho'(x_0) &= 105 \\ \rho'(x_1) &= 5 \\ \rho'(x_i) &= 0 \quad \text{für alle anderen Variablen } x_i \end{aligned}$$

Daher $\sum_{i=0}^{k} \rho'(x_i) = 110$.

Da es keine Weise gibt, einen höheren Wert für $\sum_{i=0}^{k} \rho'(x_i)$ zu erreichen, ist $f_P(5) = 110$.

# Maximale LOOP-berechenbare Zahlen

**Satz**

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

# Maximale LOOP-berechenbare Zahlen

**Satz**

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

**Beweis** Normalisiere $P$ zunächst:

- Ersetze Zuweisungen $x_i := x_j + c$ mit $c > 1$ durch $x_i := x_j + 1; \underbrace{x_i := x_i + 1; \ldots x_i := x_i + 1}_{(c-1)\text{-mal}}$.

# Maximale LOOP-berechenbare Zahlen

**Satz**

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

**Beweis** Normalisiere $P$ zunächst:

- Ersetze Zuweisungen $x_i := x_j + c$ mit $c > 1$ durch $x_i := x_j + 1; \underbrace{x_i := x_i + 1; \ldots x_i := x_i + 1}_{(c-1)\text{-mal}}$.
- Für **LOOP** $x_i$ **DO** $Q$ **END** und $x_i$ kommt in $Q$ vor: Ersetze **LOOP** $x_i$ **DO** $Q$ **END** durch $x_i' := x_i$; **LOOP** $x_i'$ **DO** $Q$ **END**; $x_i' := 0$, wobei $x_i'$ eine neue Variable ist.

# Maximale LOOP-berechenbare Zahlen

**Satz**

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

**Beweis** Normalisiere $P$ zunächst:

- Ersetze Zuweisungen $x_i := x_j + c$ mit $c > 1$ durch $x_i := x_j + 1; \underbrace{x_i := x_i + 1; \ldots x_i := x_i + 1}_{(c-1)\text{-mal}}$.
- Für **LOOP** $x_i$ **DO** $Q$ **END** und $x_i$ kommt in $Q$ vor: Ersetze **LOOP** $x_i$ **DO** $Q$ **END** durch $x_i' := x_i$; **LOOP** $x_i'$ **DO** $Q$ **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

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

# Maximale LOOP-berechenbare Zahlen

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

Durch Induktion über die Größe des normalisierten Programms.

# Maximale LOOP-berechenbare Zahlen

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

Durch Induktion über die Größe des normalisierten Programms.

Wir müssen folgende Fälle betrachten:

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

# Maximale LOOP-berechenbare Zahlen

**Beweis** (Fortsetzung)

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

# Maximale LOOP-berechenbare Zahlen

**Beweis** (Fortsetzung)

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

  Mit $a(2, y) = 2y + 3$ (siehe 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

**Beweis** (Fortsetzung)

- Fall $P_1; P_2$: Die Induktionshypothese liefert $f_{P_i}(n) < a(k_i, n)$ für $i \in \{1, 2\}$.

# Maximale LOOP-berechenbare Zahlen

**Beweis** (Fortsetzung)

- Fall $P_1; P_2$: Die Induktionshypothese liefert $f_{P_i}(n) < a(k_i, n)$ für $i \in \{1, 2\}$.
  Es gilt

$$f_P(n) \leq f_{P_2}(f_{P_1}(n))$$

# Maximale LOOP-berechenbare Zahlen

**Beweis** (Fortsetzung)

- Fall $P_1; P_2$: Die Induktionshypothese liefert $f_{P_i}(n) < a(k_i, n)$ für $i \in \{1, 2\}$.

  Es gilt

$$\begin{aligned} f_P(n) &\leq f_{P_2}(f_{P_1}(n)) \\ &< a(k_2, f_{P_1}(n)) \qquad \text{(IH)} \end{aligned}$$

# Maximale LOOP-berechenbare Zahlen

**Beweis** (Fortsetzung)

- Fall $P_1; P_2$: Die Induktionshypothese liefert $f_{P_i}(n) < a(k_i, n)$ für $i \in \{1, 2\}$.

  Es gilt

$$\begin{aligned} f_P(n) &\leq f_{P_2}(f_{P_1}(n)) \\ &< a(k_2, f_{P_1}(n)) && \text{(IH)} \\ &\leq a(k_2, a(k_1, n)) && \text{(IH, 1)} \end{aligned}$$

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

# Maximale LOOP-berechenbare Zahlen

**Beweis** (Fortsetzung)

- Fall $P_1; P_2$: Die Induktionshypothese liefert $f_{P_i}(n) < a(k_i, n)$ für $i \in \{1, 2\}$.

  Es gilt

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

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

# Maximale LOOP-berechenbare Zahlen

**Beweis** (Fortsetzung)

- Fall $P_1; P_2$: Die Induktionshypothese liefert $f_{P_i}(n) < a(k_i, n)$ für $i \in \{1, 2\}$.

  Es gilt

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

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

(2) Definition von $a$: $a(x, y) = a(x - 1, a(x, y - 1))$ falls $x > 0$ und $y > 0$

# Maximale LOOP-berechenbare Zahlen

**Beweis** (Fortsetzung)

- Fall $P_1; P_2$: Die Induktionshypothese liefert $f_{P_i}(n) < a(k_i, n)$ für $i \in \{1, 2\}$.

  Es gilt

$$\begin{aligned}
f_P(n) &\leq f_{P_2}(f_{P_1}(n)) \\
&< a(k_2, f_{P_1}(n)) && \text{(IH)} \\
&\leq a(k_2, a(k_1, n)) && \text{(IH, 1)} \\
&\leq a(\max\{k_1, k_2\}, a(\max\{k_1, k_2\} + 1, n)) && \text{(1)} \\
&= a(\max\{k_1, k_2\} + 1, n + 1) && \text{(2)} \\
&\leq a(\max\{k_1, k_2\} + 2, n) && \text{(3)}
\end{aligned}$$

  (1) Monotonie: $a(x, y) \leq a(x', y')$ falls $x \leq x'$ und $y \leq y'$
  (2) Definition von $a$: $a(x, y) = a(x - 1, a(x, y - 1))$ falls $x > 0$ und $y > 0$
  (3) Monotonie: $a(x, y + 1) \leq a(x + 1, y)$

# Maximale LOOP-berechenbare Zahlen

**Beweis** (Fortsetzung)

- Fall $P_1; P_2$: Die Induktionshypothese liefert $f_{P_i}(n) < a(k_i, n)$ für $i \in \{1, 2\}$.

  Es gilt

$$\begin{aligned}
f_P(n) &\leq f_{P_2}(f_{P_1}(n)) \\
&< a(k_2, f_{P_1}(n)) && \text{(IH)} \\
&\leq a(k_2, a(k_1, n)) && \text{(IH, 1)} \\
&\leq a(\max\{k_1, k_2\}, a(\max\{k_1, k_2\} + 1, n)) && \text{(1)} \\
&= a(\max\{k_1, k_2\} + 1, n + 1) && \text{(2)} \\
&\leq a(\max\{k_1, k_2\} + 2, n) && \text{(3)}
\end{aligned}$$

  (1) Monotonie: $a(x, y) \leq a(x', y')$ falls $x \leq x'$ und $y \leq y'$
  (2) Definition von $a$: $a(x, y) = a(x - 1, a(x, y - 1))$ falls $x > 0$ und $y > 0$
  (3) 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

**Beweis** (Fortsetzung)

- Fall **LOOP** $x_i$ **DO** $Q$ **END** und $x_i$ kommt nicht in $Q$ vor:
  Siehe Skript. □

# Die Ackermannfunktion ist nicht LOOP-berechenbar

**Theorem**

Die Ackermannfunktion ist nicht LOOP-berechenbar.

# Die Ackermannfunktion ist nicht LOOP-berechenbar

**Theorem**

Die Ackermannfunktion ist nicht LOOP-berechenbar.

**Beweis** Durch Widerspruch.

# Die Ackermannfunktion ist nicht LOOP-berechenbar

**Theorem**

Die Ackermannfunktion ist nicht LOOP-berechenbar.

**Beweis** Durch Widerspruch.

Nehme an, dass $a$ LOOP-berechenbar ist. Dann ist auch $g(x_1) = a(x_1, x_1)$ LOOP-berechenbar.

# Die Ackermannfunktion ist nicht LOOP-berechenbar

**Theorem**

Die Ackermannfunktion ist nicht LOOP-berechenbar.

**Beweis** Durch Widerspruch.

Nehme an, dass $a$ LOOP-berechenbar ist. Dann ist auch $g(x_1) = a(x_1, x_1)$ LOOP-berechenbar.

Sei $P$ ein LOOP-Programm, dass $g$ berechnet.

# Die Ackermannfunktion ist nicht LOOP-berechenbar

**Theorem**

Die Ackermannfunktion ist nicht LOOP-berechenbar.

**Beweis** Durch Widerspruch.

Nehme an, dass $a$ LOOP-berechenbar ist. Dann ist auch $g(x_1) = a(x_1, x_1)$ LOOP-berechenbar.

Sei $P$ ein LOOP-Programm, dass $g$ berechnet.

Dann gilt $g(x_1) \leq f_P(x_1)$, da $\rho'(x_0) = g(x_1)$ nach Ausführung von $P$.

# Die Ackermannfunktion ist nicht LOOP-berechenbar

**Theorem**

Die Ackermannfunktion ist nicht LOOP-berechenbar.

**Beweis** Durch Widerspruch.

Nehme an, dass $a$ LOOP-berechenbar ist. Dann ist auch $g(x_1) = a(x_1, x_1)$ LOOP-berechenbar.

Sei $P$ ein LOOP-Programm, dass $g$ berechnet.

Dann gilt $g(x_1) \leq f_P(x_1)$, da $\rho'(x_0) = g(x_1)$ nach Ausführung von $P$.

Es gibt eine Konstante $k$, sodass $f_P(n) < a(k, n)$.

# Die Ackermannfunktion ist nicht LOOP-berechenbar

**Theorem**

Die Ackermannfunktion ist nicht LOOP-berechenbar.

**Beweis** Durch Widerspruch.

Nehme an, dass $a$ LOOP-berechenbar ist. Dann ist auch $g(x_1) = a(x_1, x_1)$ LOOP-berechenbar.

Sei $P$ ein LOOP-Programm, dass $g$ berechnet.

Dann gilt $g(x_1) \leq f_P(x_1)$, da $\rho'(x_0) = g(x_1)$ nach Ausführung von $P$.

Es gibt eine Konstante $k$, sodass $f_P(n) < a(k, n)$.

Starte $P$ mit $\rho = \{x_1 \mapsto k\}$.

# Die Ackermannfunktion ist nicht LOOP-berechenbar

**Theorem**

Die Ackermannfunktion ist nicht LOOP-berechenbar.

**Beweis** Durch Widerspruch.

Nehme an, dass $a$ LOOP-berechenbar ist. Dann ist auch $g(x_1) = a(x_1, x_1)$ LOOP-berechenbar.

Sei $P$ ein LOOP-Programm, dass $g$ berechnet.

Dann gilt $g(x_1) \leq f_P(x_1)$, da $\rho'(x_0) = g(x_1)$ nach Ausführung von $P$.

Es gibt eine Konstante $k$, sodass $f_P(n) < a(k, n)$.

Starte $P$ mit $\rho = \{x_1 \mapsto k\}$.

Dann gilt $g(k) \leq f_P(k) < a(k, k) = g(k)$. Widerspruch. □

# Fazit

**Theorem**

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