Ackerman function is not P.t.

\[ A(n) = \begin{cases} n \cdot n & \text{if } n \leq 1 \\ 1 & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ 4 & \text{if } n = 2 \\ 27 & \text{if } n = 3 \\ \text{some big } n \end{cases} \]

Idea of the proof:
- If we don't have for-loops at all, then all we can do is apply \( f \) several times.

\[ f(n) = n + \text{const} \]

at most

we have one for-loop:

\[ \leq n + c \]

for \( \{ i = 1; i < n; i++ \} \} \] \( \leq c \cdot n \)

\[ \Rightarrow \leq n + c \]

\[ f_0(a, b) = a + 1 \]
\[ f_1(a, b) = a + b \]
\[ f_2(a, b) = a \times b \]
\[ f_3(a, b) = a^b \]

\[ f(n) \leq f_2(n, c) \]

\[ \underbrace{n + c + \ldots + c}_{n \text{ times}} \]

2 for-loops:

\[ \leq c^2 \]

\[ f(n) \leq f_3(c, n) \]

If we have \( k \) for-loops then \( f(n) \leq f_{k+1}(c, n) \)

\[ f_4(a, b) = a^b \] 5 times
Ackerman function will never be less than this and can not be expressed using for-loops. Thus Ackerman function is not \( \text{PR} \).

Every \( \text{PR} \) function is bounded by \( f_{\text{PR}}(c, n) \) where \( k \) is \# of for-loops inside of it, Ackerman function \( A(c, n) \) grows faster than any of them.

**Pitagora's theorem**

\[
(a + b)^2 = a^2 + 2ab + b^2 = a^2 + b^2 = c^2
\]

\[
\begin{align*}
L &= \text{PR}(\emptyset, n, \emptyset) \\
\emptyset(n_1, ..., n_k, \emptyset) &= g(n_1, ..., n_k) \\
f(n_1, ..., n_k, m) &= h(n_1, ..., n_k, m, f(n_1, ..., n_k, m))
\end{align*}
\]

**Example**

\[
L = \text{PR}(\emptyset, n, \emptyset) \\
\emptyset(n_1, ..., n_k, \emptyset) = \emptyset \\
f(n_1, ..., n_k, m) = 1
\]
\[
\begin{align*}
\text{if } f(3,4) &= 9 \\
\text{this is } m
\end{align*}
\]

\[
\begin{align*}
f(3,0) &= 1 \\
f(3,1) &= f(3,0) + 2 = 3 \\
f(3,2) &= f(3,1) + 2 = 5 \\
f(3,3) &= f(3,2) + 2 = 7 \\
f(3,4) &= f(3,3) + 2 = 9
\end{align*}
\]

---

Formalizing while-loops.

Normally used for efficiency; example search
but we have shown that while-loops are
needed to compute some computable functions

- A while-loop is like a for-loop, but instead of
  fixed # of iterations, the # of iterations is the
  smallest # at which some condition is satisfied.

\[
\text{Pl}(a_1, \ldots, n, m) \text{- condition} \\
\text{true or false, 1 or 0)} \]

\[
\text{nu-minimum, the smallest} \]

\[
\text{Smallest natural number } m \text{ for which} \]

\[
\text{this condition is true} \]

\[
\text{p}(a_1, m)
\]

nu-recursion (going back)

Definition. A function is called \( \mu \)-recursive if
it can be obtained from \( \emptyset, 0, \Pi_1^k \) by using \( , \Pi \), and \( \mu \)-recursion

One feature: We have functions which are sometimes
not defined

\[
\mu \emptyset \text{ } \mu \emptyset
\]
not defined
μm. Ø

```c
int m=Ø;
while (i < p)
m += 1;
```

not well defined!

\[ f(n) = \begin{cases} 3 & \text{if } n = 2 \\ 5 & \text{if } n = 0 \\ \text{undefined otherwise} \end{cases} \]

Stop the program when \( n \geq 2 \) or \( n = 0 \)

\[ μm ((a = 2) \land (m = 3)) \lor ((a = 5)\land (b = m = 5)) \]

```c
int m=Ø;
while (! ( (a = 2) \land (m = 3)) \lor ((a = 5)\land (b = m = 5)))
m += 1;
```

Condition is satisfied at \( n = 2 \) and \( n = 3 \)

Show that the following function is \( μ \)-recursive

\[ f(a, b) = a \land b \]

- \( f(a, b) \) is undefined
- \( f(a, b) \) is undefined otherwise

\[ a \land b = μm (b \land m = a) \]

\[ a \land b = μm (b + m = a) \]

\[ e \land 2 \]

\[ μm (z + m \geq s) = 3 \]

\[ 2 < 5 \]
\[ \mu m (s + n \geq 2) = \emptyset \]

```c
int m = 0;
while(!(m + b > a))
    m++;
```

\[ \begin{array}{c}
\frac{2}{a} \quad \frac{1}{b} \quad \frac{5}{m} \quad \text{end of program}
\end{array} \]

**Extra Credit**

**Division by recursion**

It requires:

\[ a/b \]

\[ \mu n (m \times b > a) \]

```
int m = 0;
if (m \times b > a)
    int m = 0;
while(!(m \times b > a))
    m++;
```

\[ \begin{array}{c}
\frac{5}{2} \quad \frac{2}{a} \times \frac{3}{b} \quad \frac{a}{b} \quad \text{This has to be 2!}
\end{array} \]