Kapitola: The connection between the iterated logarithm and the inverse Ackermann function

This chapter is about two functions: The iterated logarithm function $\log^* n$, and the inverse Ackermann function $\alpha(n)$.

Both $\log^* n$ and $\alpha(n)$ are very slowly growing functions: for any number n that fits into the memory of your computer, $\log^* n \leq 6$ and $\alpha(n) \leq 4$.

Historically, both functions have been used in upper bounds for the time complexity of a particular implementation of the Disjoint-set data structure: Union-Find algorithm with both union by rank and path compression.

One particular insight I have been missing for a long time is the connection between the two functions. Here it is: for large enough n,

$$\alpha(n) = \min\{i : \log^{\overbrace{* \cdots *}^{i-3}}(n) \leq i\}$$

Okay, you got me, I made that scary on purpose :) After the few seconds of stunned silence, it's now time for a decent explanation.

The iterated logarithm

The function iteration (the asterisk operator *) can be seen as a functional (a higher-order function): You can take any function f and define a new function f^* as follows:

$$f^*(n) = \begin{cases} 0 & \leftarrow n \leq 1 \\ 1 + f^*(f(n)) & \leftarrow n > 1 \end{cases}$$

In human words: Imagine that you have a calculator with a button that applies f to the currently displayed value. Then, $f^*(n)$ is the number of times you need to push that button in order to make the value decrease to 1 or less.

Here are a few examples of what the iteration produces:

f	approximate f^*
$f(n) = n - 1$	$f^*(n) = n - 1$
$f(n) = n - 2$	$f^*(n) = n/2$
$f(n) = n/2$	$f^*(n) = \log(n)$
$f(n) = \sqrt{n}$	$f^*(n) = \log(\log(n))$
$f(n) = \log n$	$f^*(n) = \log^* n$

We don't have any simpler expression for $\log^*$.

Just like a logarithm is the inverse of exponentiation, $\log^*$ can be seen as the inverse of a particular quickly growing function: tetration (with the same base).

Hyperoperations

Tetration is actually just a special case of a hyperoperation. What does that mean?

Imagine that the only function in the world is the successor function s defined as follows: $\forall n : s(n) = n + 1$.

Each function you can now compute in constant time is kind of boring. Basically, you can only do constant functions and functions like "add 7".

Now let's assume I let you use one for-cycle. What new interesting functions can you compute now? One such notable function is addition. Addition is just iterated incrementation. We can compute $x + y$ as follows:

```
define add(x,y):
    result = x
    for i in 0..y-1:
        result = s(result)
```

What can you do with two nested for-cycles? Multiplication! Multiplication is just iterated addition.

```
define mul(x,y):
    result = 0
    for i in 0..y-1:
        result = add(result,x)
```

Here is the same thing, but this time with two nested cycles:

```
define mul(x,y):
    result = 0
    for i in 0..y-1:
        for j in 0..x-1:
            result = s(result)
```

What can you do with three nested for-cycles? This should now be obvious: we can iterate multiplication, which gives us exponentiation. What about four nested for-cycles? Precisely, tetration, also known as "power towers". And clearly we can continue in this fashion forever, creating functions that grow faster and faster.

(The entire set of functions obtained in this way basically spans the entire set of primitive recursive functions.)

The Ackermann function

The Ackermann function $A(m,n)$ is a recursive binary function that is specially constructed to grow quickly (namely, faster than any primitive recursive function can). We can then define a unary Ackermann function a as $\forall n : a(n) = A(n,n)$. Then, the inverse Ackermann function can be defined as the inverse to this function.

How is the Ackermann function related to the functions we mentioned above? Let's look at rows of this function: unary functions a_k such that $\forall n : a_k(n) = A(k,n)$. Here is what we see:

- The function a_0 is your plain successor function.
- The function a_1 is the "plus 2" function, representing addition. Formally, $\forall n : a_1(n) = n + 2$.
- The function a_2 roughly corresponds to multiplication: $\forall n : a_2(n) = 2n + 3$.
- The function a_3 roughly corresponds to exponentiation: $\forall n : a_3(n) = 2^{n+3} - 3$.
- The function a_4 roughly corresponds to tetration: $\forall n : a_4(n) = 2 \uparrow\uparrow (n + 3) - 3$ (see Knuth's up-arrow notation)
- … and so on.

This directly follows from the recursive definition of the binary Ackermann function. Note that each row function a_k is primitive recursive. However, the "diagonal" unary Ackermann function a grows faster than each of the a_k functions – and, in fact, faster than any primitive recursive function.

(The proof is by contradiction. Here is its essence: Suppose there is a program that computes a using for-cycles only. Let k be the number of for-cycles used in the program. Then, already the function in row (k plus some small constant) grows faster than anything your program can possibly compute – and the diagonal function grows even faster, which is the contradiction.)

Now we can derive the first result: we can compare our two slowly growing functions. The iterated logarithm is the inverse of the function a_4, the inverse Ackermann function α is the inverse of the much-faster-growing function a. Hence, the inverse Ackermann function must grow much slower than the iterated logarithm.

And it gets even better. Remember how log was the inverse to a_3 (exponentiation) and $\log^*$ was the inverse to a_4? Well, we can now easily show that $\log^{**}$ (that is, $(\log^*)^*$) is the inverse to a_5, and so on.

We can even start sooner. Let p be the predecessor function – the inverse to the successor function. As we saw above, p^* is the function "minus two", p^{**} is the function "divide by two", and p^{***} is the binary logarithm. Thus, the function $p^{\overbrace{*\cdots*}^{k}}$ is the inverse to the function a_k, for each k.

How can we now define the inverse Ackermann function? Given n, the value $\alpha(n)$ is at most equal to the smallest x such that $A(x,x) \geq n$. In other words, such that $a_x(x) \geq n$.

And we already know that the inverse function to a_x is $p^{\overbrace{*\cdots*}^{x}}$. Hence, we are looking for the smallest x such that $p^{\overbrace{*\cdots*}^{x}}(n) \leq x$.

And that's our main conclusion for the day. The inverse Ackermann function goes one level of abstraction deeper:

- With the iterated logarithm, you are applying one particular function (the logarithm) and you are counting the steps until the result becomes small.
- With the inverse Ackermann, you are also counting the steps until the result becomes small. However, now in each step you apply the iteration functional that makes the previous function grow slower by an order of magnitude.