The Ackermann function is the simplest example of a well-defined total function which is computable but not primitive recursive, providing a counterexample to the belief in the early 1900s that every computable function was also primitive recursive (Dötzel 1991). It grows faster than an exponential function, or even a multiple exponential function.

The basic idea behind Ackermann's function (which actually comes in many minor variants, some of which obscure this more than others) is actually a very simple one, one that even occurs spontaneously to many school children: **it's just the idea of extrapolating the sequence <addition, multiplication, exponentiation, ...>, where each operation amounts to repeated application of the previous operation.**

That is, recall that, for positive integers x and y, we have that x * y = the result of +ing y many xs, and x^y = the result of *ing y many xs.

Extending this, we might define a new operation x # y = the result of ^ing y many xs. [However, as ^ is not associative, we need to make a convention here as to whether that means "left-associative" ...(((x^x)^x)^x)...^x with y many xs, or "right-associative" x^(x^(x^...x)) with y many xs. But note that the left-associative version is just equal to x^(x^(y-1)), which isn't terribly interesting on top of the ^ operator we already have. So the convention we will pick is to define x # y as the result of ^ing y many xs right-associatively]. So 2 # 4 = 2^(2^(2^2)) = 65536, for example.

And then we can define x $ y as the result of #ing y many xs (right-associatively again, as always), and then define x % y as the result of $ing y many xs, and so on ad infinitum.

Of course, no need to keep coming up with new symbols; instead, we can just describe how many levels of iteration of this process we've gone through: x [1] y will mean x + y, x [2] y will mean x * y, x [3] y will mean x ^ y, x [4] y will mean x # y, etc.

So now we have a function of three arguments, x [N] y. This is essentially the Ackermann function (though, like I said, there are many minor variants on this which are also called Ackermann functions; the specific details that change from author to author don't actually matter terribly much for anything (particularly popular are two-argument versions which essentially fix the x here to always be 2)). As it happens, it grows super-fast; 4 [5] 3, for example, is 4 # (4 # 4) = 4 # (4^4^4^4) = 4^4^4^4^4^...^4, with 4^4^4^4 many 4s = way super-fucking-huge.

The relationship to primitive recursive functions is that it's fairly easy to show inductively that the function f(x, y) = x [N] y "grows faster" than any function defined by less than N levels of nested primitive recursion. But, of course, the Ackermann function itself, with appropriate choice of the middle argument, grows as fast as x [N] y for arbitrarily large values of N; accordingly, the Ackermann function cannot be defined by any finite number of primitive recursions, and thus is not primitive recursive.

That having been said, if one's goal is only to show the existence of total computable functions which are not primitive recursive, this is also even much easier to do without bothering with the Ackermann function in the first place: one can computably enumerate the primitive

recursive functions and then diagonalize against them to obtain a total computable function which is not among them.

(This is essentially the observation that the Halting Problem is trivial for primitive recursive functions (every primitive recursive function halts on every input), and therefore, by the uncomputability of the Halting Problem for general computable functions, one finds a computable function which is not primitive recursive).

The Ackermann function ψ is:

1. Total (terminates for all inputs)
2. Computable
3. Not primitive recursive

Part 1: Proving ψ is Total

The proof uses well-founded induction on (N², ≤lex), where:

```
(x,y) ≤lex (x',y') if:
- x < x', or
- x = x' and y ≤ y'
```

Proof steps:

1. (N², ≤lex) is well-founded (no infinite descending chains)
2. For any (x,y), assuming ψ(x',y') terminates for all (x',y') <lex (x,y):
3. Case analysis:
   a. If x = 0:
   ψ(0,y) = y + 1 clearly terminates

   b. If x > 0, y = 0:
   ψ(x,0) = ψ(x-1,1)
   Terminates by induction since (x-1,1) <lex (x,0)

   c. If x > 0, y > 0:
   ψ(x,y) = ψ(x-1,ψ(x,y-1))
   - ψ(x,y-1) terminates by induction since (x,y-1) <lex (x,y)
   - Let u = ψ(x,y-1)
   - Then ψ(x-1,u) terminates since (x-1,u) <lex (x,y)

Part 2: Proving ψ is Computable

The proof uses the concept of valid sets:

1. Define a valid set S ⊆ N³ as satisfying:

```
a. (0,y,z) ∈ S ⇒ z = y + 1
b. (x+1,0,z) ∈ S ⇒ (x,1,z) ∈ S
```

```
c. (x+1,y+1,z) ∈ S ⇒ ∃u.(x+1,y,u) ∈ S ∧ (x,u,z) ∈ S
```

2. Show that for every (x,y,z), ψ(x,y) = z if and only if there exists a finite valid set S containing (x,y,z)
3. Encode finite sets of triples as natural numbers using prime factorization
4. Show the predicate "n encodes a valid set containing (x,y,z)" is decidable
5. Then ψ(x,y) can be computed by:

```
ψ(x,y) = (μz,w)(Val(w) ∧ Triple(x,y,z) ∈ w)₁
```

where Val checks if w encodes a valid set

Part 3: Proving ψ is Not Primitive Recursive

By contradiction:

1. Assume ψ ∈ PR
2. Show that for any f ∈ PR with program P using j levels of nested FOR loops:

```
f(x) < ψj+1(max{x₁,...,xₖ})
```

3. Apply this to ψ itself with x = y = j+1:

```
ψ(j+1,j+1) < ψj+1(j+1)
```

4. But this contradicts known properties of ψ

Key insights:

- Well-founded induction handles complex recursion
- Valid sets capture computation traces
- Growth rate analysis shows non-primitive recursiveness