Total recursive functions that are not primitive recursive

Contains transcriptions...

For my personal use only

Armando B. Matos,

June 21, 2016

Contents

Abstract

It is well known that not all total recursive functions are primitive recursive. A well known example is the Ackermann function. Other examples include a function related with the Paris-Harrington Theorem, the Sudan functions, and the Goodstein function. All these functions grow extremely fast; however, we also describe another example of such a function that only takes the values 0 and 1. Some background material on primitive recursive functions is also included.
This text is for my personal study only. A lot of typos and a few references are included.

We transcribe[1] and comment several texts that describe several total recursive, but not primitive recursive (TRNPR).

The functions mentioned in this work are

(i) the Ackermann function (page 4),

(ii) a function related with the Paris-Harrington Theorem (page 11),

(iii) the Sudan functions (page 13),

(iv) the Goodstein function (page 15 and 26),

(v) a function with codomain $\{0, 1\}$ (obtained by diagonalisation, page 33).

Except for the function v, the information was basically obtained from the literature. The example v shows that the TRNPR functions are not necessarily "large"[2].

Some characteristics of primitive recursive functions, including their time of computation, are reviewed in Appendices A (page 34) and B (page 42).

Note. There is some overlap on the material on Goodstein sequences (different transcriptions...) but this may be useful for the reader (me).

1 The Ackermann function

In computability theory, the Ackermann function, named after Wilhelm Ackermann, is one of the simplest [1] and earliest-discovered examples of a total computable function that is not primitive recursive. All primitive recursive functions are total and computable, but the Ackermann function illustrates that not all total computable functions are primitive recursive.

After Ackermann's publication [2] of his function (which had three nonnegative integer arguments), many authors modified it to suit various purposes, so that today "the Ackermann function" may refer to any of numerous variants of the original function. One common version, the two-argument Ackermann–Péter function, is defined as follows for nonnegative integers m and n:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Its value grows rapidly, even for small inputs. For example, $A(4,2)$ is an integer of 19,729 decimal digits [3].

[1] Essentially from the Wikipedia.

[2] However, every such function takes a very very long time to be computed.

1.1 History

In the late 1920s, the mathematicians Gabriel Sudan and Wilhelm Ackermann, students of David Hilbert, were studying the foundations of computation. Both Sudan and Ackermann are credited [4] with discovering total computable functions (termed simply "recursive" in some references) that are not primitive recursive. Sudan published the lesser-known Sudan function, then shortly afterwards and independently, in 1928, Ackermann published his function φ. Ackermann's three-argument function, $\varphi(m, n, p)$, is defined such that for $p = 0$, 1, 2, it reproduces the basic operations of addition, multiplication, and exponentiation as

$$\begin{aligned} \varphi(m, n, 0) &= m + n, \\ \varphi(m, n, 1) &= m \cdot n, \\ \varphi(m, n, 2) &= m^n, \end{aligned}$$

and for $p > 2$ it extends these basic operations in a way that can be compared to the hyper-operations: (aside from its historic role as a total-computable-but-not-primitive-recursive function, Ackermann's original function is seen to extend the basic arithmetic operations beyond exponentiation, although not as seamlessly as do variants of Ackermann's function that are specifically designed for that purpose –such as Goodstein's hyper-operation sequence.)

In "On the Infinite", David Hilbert hypothesised that the Ackermann function was not primitive recursive, but it was Ackermann, Hilbert's personal secretary and former student, who actually proved the hypothesis in his paper On Hilbert's Construction of the Real Numbers ([2,5]).

Rózsa Péter and Raphael Robinson later developed a two-variable version of the Ackermann function that became preferred by many authors [6].

1.2 Definition and properties

Ackermann's original three-argument function $\varphi(m, n, p)$ is defined recursively as follows for nonnegative integers m, n, and p:

$$\varphi(m, n, p) = \begin{cases} \varphi(m, n, 0) = m + n & \\ \varphi(m, 0, 1) = 0 & \\ \varphi(m, 0, 2) = 1 & \\ \varphi(m, 0, p) = m & \text{for } p > 2 \\ \varphi(m, n, p) = \varphi(m, \varphi(m, n-1, p), p-1) & \text{for } n > 0 \text{ and } p > 0. \end{cases}$$

Of the various two-argument versions, the one developed by Péter and Robinson (called "the" Ackermann function by some authors) is defined for nonnegative

integers m and n as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

It may not be immediately obvious that the evaluation of $A(m,n)$ always terminates. However, the recursion is bounded because in each recursive application either m decreases, or m remains the same and n decreases. Each time that n reaches zero, m decreases, so m eventually reaches zero as well. (Expressed more technically, in each case the pair (m,n) decreases in the lexicographic order on pairs, which is a well-ordering, just like the ordering of single non-negative integers; this means one cannot go down in the ordering infinitely many times in succession.) However, when m decreases there is no upper bound on how much n can increase – and it will often increase greatly.

The Péter-Ackermann function can also be expressed in terms of various other versions of the Ackermann function:

- the indexed version of Knuth's up-arrow notation (extended to integer indices ≥ 2):

  $$A(m,n) = 2 \uparrow^{m-2} (n+3) - 3.$$

  The part of the definition $A(m,0) = A(m-1,1)$ corresponds to $2 \uparrow^{m+1} 3 = 2 \uparrow^m 4$.

- Conway chained arrow notation:

  $$A(m,n) = (2 \to (n+3) \to (m-2)) - 3 \text{ for } m \geq 3.$$

  hence

  $$2 \to n \to m = A(m+2, n-3) + 3 \text{ for } n > 2.$$

  ($n = 1$ and $n = 2$ would correspond with $A(m,-2) = -1$ and $A(m,-1) = 1$, which could logically be added.)

For small values of m like 1, 2, or 3, the Ackermann function grows relatively slowly with respect to n (at most exponentially). For $m \geq 4$, however, it grows much more quickly; even $A(4,2)$ is about 2×10^{19728}, and the decimal expansion of $A(4,3)$ is very large by any typical measure.

Logician Harvey Friedman defines a version of the Ackermann function as follows ($A(m,n)$ is defined for $n \geq 0$ and $m \geq 1$):

$$\begin{cases} A(m,0) &= 1 & \\ A(1,n) &= 2n & (\text{for } n \geq 1) \\ A(m,n) &= A(m-1, A(m,n-1)) & (\text{for } n \geq 1,\ m \geq 2). \end{cases}$$

He also defines a single-argument version $A(n) = A(n,n)$ (see [7])

A single-argument version $A(k) = A(k, k)$ that increases both m and n at the same time dwarfs every primitive recursive function, including very fast-growing functions such as the exponential function, the factorial function, multi- and superfactorial functions, and even functions defined using Knuth's up-arrow notation (except when the indexed up-arrow is used). It can be seen that $A(n)$ is roughly comparable to $f_\omega(n)$ in the fast-growing hierarchy.

This extreme growth can be exploited to show that f, which is obviously computable on a machine with infinite memory such as a Turing machine and so is a computable function, grows faster than any primitive recursive function and is therefore not primitive recursive. In a category with exponentials, using the isomorphism

$$(A \times B) \to C \;\cong\; A \to (B \to C)$$

(in computer science, this is called currying), the Ackermann function may be defined via primitive recursion over higher-order functionals as follows:

$$\begin{aligned} \mathsf{Ack}(0) &= \mathrm{Succ} \\ \mathsf{Ack}(m+1) &= \mathsf{Iter}(\mathsf{Ack}(m)) \end{aligned}$$

where Succ is the usual successor function and Iter is defined by primitive recursion as well:

$$\begin{aligned} \mathsf{Iter}(f)(0) &= f(1) \\ \mathsf{Iter}(f)(n+1) &= f(\mathsf{Iter}(f)(n)). \end{aligned}$$

One interesting aspect of the Ackermann function is that the only arithmetic operations it ever uses are addition and subtraction of 1. Its properties come solely from the power of unlimited recursion. This also implies that its running time is at least proportional to its output, and so is also extremely huge. In actuality, for most cases the running time is far larger than the output; see below.

1.3 Table of values

Computing the Ackermann function can be restated in terms of an infinite table. We place the natural numbers along the top row. To determine a number in the table, take the number immediately to the left, then look up the required number in the previous row, at the position given by the number just taken. If there is no number to its left, simply look at the column headed "1" in the previous row. Here is a small upper-left portion of the table:

Values of $A(m, n)$

$m\backslash n$	0	1	2	3	n
0	1	2	3	4	$n+1$
1	2	3	4	5	$n+2 = 2+(n+3)-3$
2	3	5	7	9	$2n+3 = 2(n+3)-3$
3	5	13	29	61	$2^{n+3}-3$
4	$13 = 2^{2^2}- 3$	$65533 = 2^{2^{2^2}} - 3$	$2^{65536} - 3 = 2^{2^{2^{2^2}}} -3$	$2^{2^{65536}} - 3 = 2^{2^{2^{2^{2^2}}}} - 3$	...
5	$2\uparrow\uparrow\uparrow 3-3$	$2\uparrow\uparrow\uparrow 4-3$	$2\uparrow\uparrow\uparrow 5-3$	$2\uparrow\uparrow\uparrow 6-3$	$2\uparrow\uparrow\uparrow (n+3)-3$
6	$2\uparrow\uparrow\uparrow\uparrow 3-3$	$2\uparrow\uparrow\uparrow\uparrow 4-3$	$2\uparrow\uparrow\uparrow\uparrow 5-3$	$2\uparrow\uparrow\uparrow\uparrow 6-3$	$2\uparrow\uparrow\uparrow\uparrow (n+3)-3$

The numbers here which are only expressed with recursive exponentiation or Knuth arrows are very large and would take up too much space to notate in plain decimal digits.

Despite the large values occurring in this early section of the table, some even larger numbers have been defined, such as Graham's number, which cannot be written with any small number of Knuth arrows. This number is constructed with a technique similar to applying the Ackermann function to itself recursively.

This is a repeat of the above table, but with the values replaced by the relevant expression from the function definition to show the pattern clearly:

Values of $A(m, n)$

$m\backslash n$	0	1	2	3	4	n
0	$0+1$	$1+1$	$2+1$	$3+1$	$4+1$	n+1
1	$A(0,1)$	$A(0,A(1,0))$ $= A(0,2)$	$A(0,A(1,1))$ $= A(0,3)$	$A(0,A(1,2))$ $= A(0,4)$	$A(0,A(1,3))$ $= A(0,5)$	$A(0,A(1,n-1))$
2	$A(1,1)$	$A(1,A(2,0))$ $= A(1,3)$	$A(1,A(2,1))$ $= A(1,5)$	$A(1,A(2,2))$ $= A(1,7)$	$A(1,A(2,3))$ $= A(1,9)$	$A(1,A(2,n-1))$
3	$A(2,1)$	$A(2,A(3,0))$ $= A(2,5)$	$A(2,A(3,1))$ $= A(2,13)$	$A(2,A(3,2))$ $= A(2,29)$	$A(2,A(3,3))$ $= A(2,61)$	$A(2,A(3,n-1))$
4	$A(3,1)$	$A(3,A(4,0))$ $= A(3,13)$	$A(3,A(4,1))$ $= A(3,65533)$	$A(3,A(4,2))$	$A(3,A(4,3))$	$A(3,A(4,n-1))$
5	$A(4,1)$	$A(4,A(5,0))$	$A(4,A(5,1))$	$A(4,A(5,2))$	$A(4,A(5,3))$	$A(4,A(5,n-1))$
6	$A(5,1)$	$A(5,A(6,0))$	$A(5,A(6,1))$	$A(5,A(6,2))$	$A(5,A(6,3))$	$A(5,A(6,n-1))$

1.4 Expansion

To see how the Ackermann function grows so quickly, it helps to expand out some simple expressions using the rules in the original definition. For example, we can fully evaluate $A(1, 2)$ in the following way:

$$\begin{aligned} A(1,2) &= A(0, A(1,1)) \\ &= A(0, A(0, A(1,0))) \\ &= A(0, A(0, A(0,1))) \\ &= A(0, A(0,2)) \\ &= A(0,3) \\ &= 4. \end{aligned}$$

To demonstrate how $A(4, 3)$'s computation results in many steps and in a large

number:

$$
\begin{aligned}
A(4,3) &= A(3, A(4,2)) \\
&= A(3, A(3, A(4,1))) \\
&= A(3, A(3, A(3, A(4,0)))) \\
&= A(3, A(3, A(3, A(3,1)))) \\
&= A(3, A(3, A(3, A(2, A(3,0))))) \\
&= A(3, A(3, A(3, A(2, A(2,1))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(2,0)))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(1,1)))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(0, A(1,0))))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(0, A(0,1))))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(0,2)))))) \\
&= A(3, A(3, A(3, A(2, A(1,3))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(1,2)))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(1,1))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(0, A(1,0)))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(0, A(0,1)))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(0,2))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0,3)))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0,4)))))) \\
&= A(3, A(3, A(3, A(2,5)))) \\
&= \ldots \\
&= A(3, A(3, A(3,13))) \\
&= \ldots \\
&= A(3, A(3,65533)) \\
&= \ldots \\
&= A(3, 2^{65536} - 3) \\
&= \ldots \\
&= 2^{2^{65536}} - 3.
\end{aligned}
$$

1.5 Functions that satisfy the general recurrence

In [MP16] a closed form of $a(m,n)$, which uses the Knuth superpower notation, namely $a(m,n) = 2 \overset{m-2}{\uparrow} (n+3) - 3$. Generalised Ackermann functions, that is functions satisfying only the general recurrence and one of the boundary conditions are also studied. In particular it is shown that the function $2 \overset{m-2}{\uparrow} (n+2) - 2$ also belongs to the "Ackermann class".

1.6 Inverse

Since the function $f(n) = A(n,n)$ considered above grows very rapidly, its inverse function, f^{-1}, grows very slowly. This inverse Ackermann function f^{-1} is usually denoted by α. In fact, $\alpha(n)$ is less than 5 for any practical input size n, since $A(4,4)$ is on the order of $2^{2^{2^{2^{16}}}}$.

This inverse appears in the time complexity of some algorithms, such as the disjoint-set data structure and Chazelle's algorithm for minimum spanning trees. Sometimes Ackermann's original function or other variations are used in these settings, but they all grow at similarly high rates. In particular, some modified functions simplify the expression by eliminating the -3 and similar terms.

A two-parameter variation of the inverse Ackermann function can be defined as follows, where $\lfloor x \rfloor$ is the floor function:

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n\}.$$

This function arises in more precise analyses of the algorithms mentioned above, and gives a more refined time bound. In the disjoint-set data structure, m represents the number of operations while n represents the number of elements; in the minimum spanning tree algorithm, m represents the number of edges while n represents the number of vertices. Several slightly different definitions of $\alpha(m,n)$ exist; for example, $\log_2 n$ is sometimes replaced by n, and the floor function is sometimes replaced by a ceiling.

Other studies might define an inverse function of one where m is set to a constant, such that the inverse applies to a particular row [8].

2 A function related with the Paris–Harrington theorem

In mathematical logic, the Paris–Harrington theorem states that a certain combinatorial principle in Ramsey theory, namely the strengthened finite Ramsey theorem, is true, but not provable in Peano arithmetic. This was the first "natural" example of a true statement about the integers that could be stated in the language of arithmetic, but not proved in Peano arithmetic; it was already

known that such statements existed by Gödel's first incompleteness theorem.

2.1 The strengthened finite Ramsey theorem

The strengthened finite Ramsey theorem is a statement about colorings and natural numbers and states that:

> For any positive integers n, k, m we can find N with the following property: if we color each of the n-element subsets of $S = \{1, 2, 3, \dots N\}$ with one of k colors, then we can find a subset Y of S with at least m elements, such that all n element subsets of Y have the same color, and the number of elements of Y is at least the smallest element of Y.

Without the condition that the number of elements of Y is at least the smallest element of Y, this is a corollary of the finite Ramsey theorem in $K_{\mathcal{P}_n(S)}$, with N given by:

$$\binom{N}{n} = |\mathcal{P}_n(S)| \geq R(\underbrace{m, m, \dots, m}_{k}).$$

Moreover, the strengthened finite Ramsey theorem can be deduced from the infinite Ramsey theorem in almost exactly the same way that the finite Ramsey theorem can be deduced from it, using a compactness argument (see the article on Ramsey's theorem for details). This proof can be carried out in second-order arithmetic.

The Paris–Harrington theorem states that the strengthened finite Ramsey theorem is not provable in Peano arithmetic.

2.2 The Paris–Harrington theorem

Roughly speaking, Jeff Paris and Leo Harrington showed that the strengthened finite Ramsey theorem is unprovable in Peano arithmetic by showing that in Peano arithmetic it implies the consistency of Peano arithmetic itself. Since Peano arithmetic cannot prove its own consistency by Gödel's theorem, this shows that Peano arithmetic cannot prove the strengthened finite Ramsey theorem.

The smallest number N that satisfies the strengthened finite Ramsey theorem is a computable function of n, m, k, but grows extremely fast. In particular it is not primitive recursive, but it is also far larger than standard examples of non primitive recursive functions such as the Ackermann function. Its growth is so large that Peano arithmetic cannot prove it is defined everywhere, although Peano arithmetic easily proves that the Ackermann function is well defined.

3 The Sudan functions

In the theory of computation, the Sudan function is an example of a function that is recursive, but not primitive recursive. This is also true of the better-known Ackermann function. The Sudan function was the first function having this property to be published.

It was discovered (and published [1]) in 1927 by Gabriel Sudan, a Romanian mathematician who was a student of David Hilbert. Definition

$$\begin{cases} F_0(x,y) & = & x+y,, \\ F_{n+1}(x,0) & = & x, \qquad (n \geq 0) \\ F_{n+1}(x,y+1) & = & F_n(F_{n+1}(x,y), F_{n+1}(x,y)+y+1), \qquad (n \geq 0) \end{cases}$$

Value Tables

Values of $F_0(x, y)$

$y\backslash x$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	6
2	2	3	4	5	6	7
3	3	4	5	6	7	8
4	4	5	6	7	8	9
5	5	6	7	8	9	10
6	6	7	8	9	10	11

Values of $F_1(x, y)$

$y\backslash x$	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	3	5	7	9	11	13
2	4	8	12	16	20	24	28
3	11	19	27	35	43	51	59
4	26	42	58	74	90	106	122
5	57	89	121	153	185	217	249
6	120	184	248	312	376	440	504

In general, $F_1(x, y)$ is equal to $F_1(0, y) + 2^y x$.

Values of $F_2(x, y)$

$y\backslash x$	0	1	2	3	4
0	0	1	2	3	4
1	1	8	27	74	185
2	19	$F_1(8, 10) =$ 10228	$F_1(27, 29) \approx 1.55 \times 10^{10}$	$F_1(74, 76) \approx 5.74 \times 10^{24}$	$F_1(185, 187) \approx 3.67 \times 10^{58}$

A Haskell program that computes $F_n(x, y)$ (denoted by `f n x y`).)

```
f n x y
  | n==0     =  x+y
```

```
| y==0     =  x
| otherwise=  f (n-1) (f n x (y-1)) (f n x (y-1)+(y-1)+1)
```

It may be interesting to compare the general recursion schema for: the Ackermann function, the function g defined in [MP16], and for the Sudan functions:

$$\begin{aligned} A(m,n) &= A(m-1, A(m,n-1)) && \text{Ackermann function} \\ g(m,n) &= g(m-1, g(m,n-1)) && g \text{ function of [MP16]} \\ F_p(m,n) &= F_{p-1}(F_p(m,n-1), F_p(m,n-1), F_p(m-1,n-1)+n) && \text{Sudan functions} \end{aligned}$$

Reference

Cristian Calude, Solomon Marcus, Ionel Tevy,
The first example of a recursive function which is not
primitive recursive, Historia Mathematica 6 (1979), no. 4,
pages 380–384.

4 The Goodstein function

In mathematical logic, Goodstein's theorem is a statement about the natural numbers, proved by Reuben Goodstein in 1944, which states that every Goodstein sequence eventually terminates at 0. Kirby and Paris [1] showed that it is unprovable in Peano arithmetic (but it can be proven in stronger systems, such as second order arithmetic). This was the third example of a true statement that is unprovable in Peano arithmetic, after Gödel's incompleteness theorem and Gerhard Gentzen's 1943 direct proof of the unprovability of ε_0-induction in Peano arithmetic. The Paris–Harrington theorem was a later example.

Laurence Kirby and Jeff Paris introduced a graph theoretic hydra game with behaviour similar to that of Goodstein sequences: the "Hydra" is a rooted tree, and a move consists of cutting off one of its "heads" (a branch of the tree), to which the hydra responds by growing a finite number of new heads according to certain rules. Kirby and Paris proved that the Hydra will eventually be killed, regardless of the strategy that Hercules uses to chop off its heads, though this may take a very long time [1].

4.1 Hereditary base-n notation

Goodstein sequences are defined in terms of a concept called "hereditary base-n notation". This notation is very similar to usual base-n positional notation, but the usual notation does not suffice for the purposes of Goodstein's theorem.

In ordinary base-n notation, where n is a natural number greater than 1, an

arbitrary natural number m is written as a sum of multiples of powers of n:

$$m = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0,$$

where each coefficient a_i satisfies $0 \leq a_i < n$, and $a_k \neq 0$. For example[3], in base 2,

$$35 = 32 + 2 + 1 = 2^5 + 2^1 + 2^0.$$

Thus the base 2 representation of 35 is 100011, which means 2^5+2+1. Similarly, 100 represented in base 3 is 10201:

$$100 = 81 + 18 + 1 = 3^4 + 2 \cdot 3^2 + 3^0.$$

Note that the exponents themselves are not written in base-n notation. For example, the expressions above include 2^5 and 3^4.

To convert a base-n representation to hereditary base n notation, first rewrite all of the exponents in base-n notation. Then rewrite any exponents inside the exponents, and continue in this way until every number appearing in the expression has been converted to base-n notation.

For example, while 35 in ordinary base-2 notation is $2^5 + 2 + 1$, it is written in hereditary base-2 notation as

$$35 = 2^{2^2+1} + 2 + 1,$$

using the fact that $5 = 2^2 + 1$. Similarly, 100 in hereditary base 3 notation is

$$100 = 3^{3+1} + 2 \cdot 3^2 + 1.$$

4.2 Goodstein sequences

The Goodstein sequence $G(m)$ of a number m is a sequence of natural numbers. The first element in the sequence $G(m)$ is m itself. To get the second, $G(m)(2)$, write m in hereditary base 2 notation, change all the 2's to 3's, and then subtract 1 from the result. In general, the $(n+1)$st term $G(m)(n+1)$ of the Goodstein sequence of m is as follows:

> – take the hereditary base $n+1$ representation of $G(m)(n)$, and replace each occurrence of the base $n+1$ with $n+2$ and then subtract one. Note that the next term depends both on the previous term and on the index n.
> – Continue until the result is zero, at which point the sequence

[3]Most of the examples in Sections 4 and 5 are from: the Wikipedia entry on "Goodstein sequences", the paper [Goo44], and the URLs `http://www.xamuel.com/goodstein-sequences`, and `http://blog.kleinproject.org/?p=674`

Base	Hereditary notation	Value	Notes
2	$2^1 + 1$	3	Write 3 in base 2 notation
3	$3^1 + 1 - 1 = 3^1$	3	Switch the 2 to a 3, then subtract 1
4	$4^1 - 1 = 3$	3	Switch the 3 to a 4, then subtract 1 (now there are no more 4s left)
5	$3 - 1 = 2$	2	No 4s left to switch to 5s. Just subtract 1
6	$2 - 1 = 1$	1	No 5s left to switch to 6s. Just subtract 1
7	$1 - 1 = 0$	0	No 6s left to switch to 7s. Just subtract 1

Figure 1: The sequence $G(3)$. The rightmost column contains the corresponding ordinals.

terminates.

Early Goodstein sequences terminate quickly. For example, $G(3)$ terminates at the sixth step:

Later Goodstein sequences increase for a very large number of steps. For example, $G(4)$ starts as seen in Figure 4.2, page 18.

Elements of $G(4)$ continue to increase for a while, but at base $3 \cdot 2^{402653209}$, they reach the maximum of $3 \cdot 2^{402653210} - 1$, stay there for the next $3 \cdot 2^{402653209}$ steps, and then begin their first and final descent.

The value 0 is reached at base $3 \cdot 2^{402653211} - 1$ (curiously, this is a Woodall number: $3 \cdot 2^{402653211} - 1 = 402653184 \cdot 2^{402653184} - 1$. This is also the case with all other final bases for starting values greater than 4).

However, even $G(4)$ doesn't give a good idea of just how quickly the elements of a Goodstein sequence can increase. $G(19)$ increases much more rapidly, and starts as follows:

Value	Hereditary notation	Ordinal
4	2^2	ω^ω
26	$3^3 - 1 = 2 \cdot 3^2 + 2 \cdot 3 + 2$	$2 \cdot \omega^2 + 2 \cdot \omega + 2$
41	$2 \cdot 4^2 + 2 \cdot 4 + 1$	$2 \cdot \omega^2 + 2 \cdot \omega + 1$
60	$2 \cdot 5^2 + 2 \cdot 5$	$2 \cdot \omega^2 + 2 \cdot \omega$
83	$2 \cdot 6^2 + 2 \cdot 6 - 1 = 2 \cdot 6^2 + 6 + 5$	$2 \cdot \omega^2 + \omega + 5$
109	$2 \cdot 7^2 + 7 + 4$	$2 \cdot \omega^2 + \omega + 4$
...	...	...
253	$2 \cdot 11^2 + 11$	$2 \cdot \omega^2 + \omega$
299	$2 \cdot 12^2 + 12 - 1 = 2 \cdot 12^2 + 11$	$2 \cdot \omega^2 + 11$
...	...	...

Figure 2: Successive values of $G(4)$. The first line corresponds to $G(1)(1)$.The "Ordinal" column contains the ordinals used in the proof of Goodstein Theorem.

Hereditary notation	Value
$2^{2^2} + 2 + 1$	19
$3^{3^3} + 3$	7 625 597 484 990
$4^{4^4} + 3$	$\approx 1.3 \times 10^{154}$
$5^{5^5} + 2$	$\approx 1.8 \times 10^{2184}$
$6^{6^6} + 1$	$\approx 2.6 \times 10^{36,305}$
7^{7^7}	$\approx 3.8 \times 10^{695,974}$
$8^{8^8} - 1 = 7 \cdot 8^{(7\cdot8^7+7\cdot8^6+7\cdot8^5+7\cdot8^4+7\cdot8^3+7\cdot8^2+7\cdot8+7)}+$	
$+7 \cdot 8^{(7\cdot8^7+7\cdot8^6+7\cdot8^5+7\cdot8^4+7\cdot8^3+7\cdot8^2+7\cdot8+6)}+$	
$\cdots + 7 \cdot 8^{(8+2)} + 7 \cdot 8^{(8+1)} + 7 \cdot 8^8 + 7 \cdot 8^7 + 7 \cdot 8^6+$	
$+7 \cdot 8^5 + 7 \cdot 8^4 + 7 \cdot 8^3 + 7 \cdot 8^2 + 7 \cdot 8 + 7$	$\approx 6 \times 10^{15,151,335}$
$\vdots$	$\vdots$

In spite of this rapid growth, Goodstein's theorem states that every Goodstein sequence eventually terminates at 0, no matter what the starting value is.

Proof of Goodstein's theorem

Goodstein's theorem can be proved (using techniques outside Peano arithmetic, see below) as follows: Given a Goodstein sequence $G(m)$, we construct a parallel sequence $P(m)$ of ordinal numbers which is strictly decreasing and [thus] terminates. Then $G(m)$ must terminate too, and it can terminate only when it goes to 0. A common misunderstanding of this proof is to believe that $G(m)$ goes to 0 because it is dominated by $P(m)$. In fact, the fact that $P(m)$ dominates $G(m)$ plays no role at all. The important point is: $G(m)(k)$ exists if and only if $P(m)(k)$ exists (parallelism). Then if $P(m)$ terminates, so does $G(m)$. And $G(m)$ can terminate only when it comes to 0.

More precisely, each term $P(m)(n)$ of the sequence $P(m)$ is obtained by applying a function f on the term $G(m)(n)$ of the Goodstein sequence of m as follows: take the hereditary base $n+1$ representation of $G(m)(n)$, and replace each occurrence of the base $n+1$ with the first infinite ordinal number ω. For example $G(3)(1) = 3 = 2^1 + 2^0$ and $P(3)(1) = f(G(3)(1)) = \omega^1 + \omega^0 = \omega + 1$. Addition, multiplication and exponentiation of ordinal numbers are well defined.

> The base-changing operation of the Goodstein sequence when going from $G(m)(n)$ to $G(m)(n+1)$ does not change the value of f (that's the main point of the construction), thus after the minus 1 operation, $P(m)(n+1)$ will be strictly smaller than $P(m)(n)$. For example, $f(3\cdot 4^{4^4}+4) = 3\omega^{\omega^\omega} + \omega = f(3\cdot 5^{5^5}+5)$, hence $f(3\cdot 4^{4^4}+4)$ is strictly greater than $f((3\cdot 5^{5^5}+5)-1)$.

If the sequence $G(m)$ did not go to 0, it would not terminate and would be infinite (since $G(m)(k+1)$ would always exist). Consequently, $P(m)$ also would be infinite (since in its turn $P(m)(k+1)$ would always exist too). But $P(m)$ is strictly decreasing and the standard order $<$ on ordinals is well-founded, therefore an infinite strictly decreasing sequence cannot exist, or equivalently, every strictly decreasing sequence of ordinals does terminate (and cannot be infinite). This contradiction shows that $G(m)$ terminates, and since it terminates, goes to 0 (by the way, since there exists a natural number k such that $G(m)(k) = 0$, by construction of $P(m)$ we have that $P(m)(k) = 0$).

While this proof of Goodstein's theorem is fairly easy, the Kirby–Paris theorem, [1] which shows that Goodstein's theorem is not a theorem of Peano arithmetic, is technical and considerably more difficult. It makes use of countable nonstandard models of Peano arithmetic. What Kirby showed is that Goodstein's theorem leads to Gentzen's theorem, i.e. it can substitute for induction up to ε_0.

Extended Goodstein's theorem

Suppose the definition Goodstein sequence is changed so that instead of replacing each occurrence of the base b with $b+1$ it was replaces it with $b+2$. Would the sequence still terminate? More generally, let b_1, b_2, b_3,... be any sequences of integers. Then let the $(n+1)$st term $G(m)(n+1)$ of the extended Goodstein sequence of m be as follows: take the hereditary base b_n representation of $G(m)(n)$, and replace each occurrence of the base b_n with b_n+1 and then subtract one. The claim is that this sequence still terminates. The extended proof defines $P(m)(n) = f(G(m)(n), n)$ as follows: take the hereditary base b_n representation of $G(m)(n)$, and replace each occurrence of the bn with the first infinite ordinal number ω. The base-changing operation of the Goodstein sequence when going from $G(m)(n)$ to $G(m)(n+1)$ still does not change the value of f. For example, if $b_n = 4$ and if $bn+1 = 9$, then $f(3\cdot 4^{4^4}+4, 4) = 3\,\omega^{\omega^\omega}+\omega = f(3\cdot 9^{9^9}+9, 9)$, hence the ordinal $f(3\cdot 4^{4^4}+4, 4)$ is strictly greater than the ordinal $f((3\cdot 9^{9^9}+9)-1, 9)$.

Sequence length as a function of the starting value

The Goodstein function, $\mathcal{G} : \mathbb{N} \to \mathbb{N}$, is defined such that $\mathcal{G}(n)$ is the length of the Goodstein sequence that starts with n. (This is a total function since every Goodstein sequence terminates.) The extreme growth-rate of $\mathcal{G}$ can be calibrated by relating it to various standard ordinal-indexed hierarchies of functions, such as the functions H_α in the Hardy hierarchy, and the functions f_α in the fast-growing hierarchy of Löb and Wainer:

- Kirby and Paris (1982) proved that
  $\mathcal{G}$ has approximately the same growth-rate as H_{ϵ_0} (which is the same as that of f_{ϵ_0}); more precisely, $\mathcal{G}$ dominates H_α for every $\alpha < \epsilon_0$, and H_{ϵ_0} dominates $\mathcal{G}$.
  (For any two functions $f, g : \mathbb{N} \to \mathbb{N}$, f is said to dominate g if $f(n) > g(n)$ for all sufficiently large n.)

- Cichon (1983) showed that
  $\mathcal{G}(n) = H_{R_2^\omega(n+1)}(1) - 1$, where $R_2^\omega(n)$ is the result of putting n in hereditary base-2 notation and then replacing all 2s with ω (as was done in the proof of Goodstein's theorem).

- Caicedo (2007) showed that if $n = 2^{m_1} + 2^{m_2} + \cdots + 2^{m_k}$ with $m_1 > m_2 > \cdots > m_k$, then
  $\mathcal{G}(n) = f_{R_2^\omega(m_1)}(f_{R_2^\omega(m_2)}(\cdots(f_{R_2^\omega(m_k)}(3))\cdots)) - 2$.

Some examples are shown in Figure 3, page 21.

Application to computable functions

Goodstein's theorem can be used to construct a total computable function that Peano arithmetic cannot prove to be total. The Goodstein sequence of a num-

n			$\mathcal{G}(n)$		
1	2^0	$2-1$	$H_\omega(1)-1$	$f_0(3)-2$	2
2	2^1	2^1+1-1	$H_{\omega+1}(1)-1$	$f_1(3)-2$	4
3	2^1+2^0	2^2-1	$H_{\omega^\omega}(1)-1$	$f_1(f_0(3))-2$	6
4	2^2	2^2+1-1	$H_{\omega^\omega+1}(1)-1$	$f_\omega(3)-2$	[1]
5	2^2+2^0	2^2+2-1	$H_{\omega^\omega+\omega}(1)-1$	$f_\omega(f_0(3))-2$	$>A(4,4)$
6	2^2+2^1	$2^2+2+1-1$	$H_{\omega^\omega+\omega+1}(1)-1$	$f_\omega(f_1(3))-2$	$>A(6,6)$
7	$2^2+2^1+2^0$	$2^{2+1}-1$	$H_{\omega^{\omega+1}}(1)-1$	$f_\omega(f_1(f_0(3)))-2$	$>A(8,8)$
8	2^{2+1}	$2^{2+1}+1-1$	$H_{\omega^{\omega+1}+1}(1)-1$	$f_{\omega+1}(3)-2$	[2]
⋮	⋮	⋮	⋮	⋮	⋮
12	$2^{2+1}+2^2$	$2^{2+1}+2^2+1-1$	$H_{\omega^{\omega+1}+\omega^\omega+1}(1)-1$	$f_{\omega+1}(f_\omega(3))-2$	[3]
⋮	⋮	⋮	⋮	⋮	⋮

Figure 3: Some values of $\mathcal{G}(n)$. Notes: [1] equal to $3\cdot 2^{402653211}-2$, [2] greater than $A^3(3,3)=A(A(61,61),A(61,61))$, [3] greater than $f\omega{+}1(64)$, which is greater than Graham's number.

ber can be effectively enumerated by a Turing machine; thus the function which maps n to the number of steps required for the Goodstein sequence of n to terminate is computable by a particular Turing machine. This machine merely enumerates the Goodstein sequence of n and, when the sequence reaches 0, returns the length of the sequence. Because every Goodstein sequence eventually terminates, this function is total. But because Peano arithmetic does not prove that every Goodstein sequence terminates, Peano arithmetic does not prove that this Turing machine computes a total function.

Here is a Haskell program due to John Tromp that computes the function described above

```
g b 0 = b
g b n = g c ((s 0 n)-1) where
        s _ 0 = 0
        s e n = (n `mod` b) * c^(s 0 e) + (s (e + 1) (n `div` b))
        c = b+1
```

In Figure 4 (page 22) we list the first 3 values of the weak and of the general Goodstein sequence that starts in 266, base 2; see the program in Figure 5 (page 23).

```
Base   Goodstein sequence                Weak Goodstein sequence
-------------------------------------------------------------------
  2    266                                 266

  3    443426488243037769948249630 61\           6590
       9149892886

  4    323170060713110073007148766 88\          65601
       669951960444102669715484032 13\
       034542752465513886789089319 72\
       014115229134636887179609218 98\
       019494119559150490921095088 15\
       238644828312063087736730099 60\
       917501977503896521067960576 38\
       384067568276792218642619756 16\
       183809433847617047058164585 20\
       363050428875758915410658086 07\
       552399123930385521914333389 66\
       834242068497478656456949485 61\
       760353263220580778056593310 26\
       192708460314150258592864177 11\
       672594360371846185735759835 11\
       523016459044036976132332872 31\
       227125684710820209725157101 72\
       693132346967854258065669793 50\
       459972683529986382155251663 89\
       437335543602135433229604645 31\
       847860495214819355585361105 95\
       96231681
```

Figure 4: The first 3 values of the Goodstein sequence (second column) and of the weak Goodstein sequence (third column). In both cases the first value is 266. The symbol "\" indicates that the number continues in the following line.

```
-- nton b x: convert x to base b
ntob b 0 = []
ntob b x = (x `mod` b):(ntob b (x `div` b))

-- next b x next element of Goodstein sequence
--    (current base is b)
-- 1) express 'x' and recursively all exponents in base 'b'
-- 2) replace everywhere 'b' by 'b+1'
-- 3) subtract 1
next b x = next' b x - 1

next' b x = (p_next b 0 xs)
  where  xs = ntob b x

p_next b p [] = 0
p_next b p (x:xs) = x*((b+1)^(next' b p)) + (p_next b (p+1) xs)

-- st x number: list of the first 'number' elements
st x n = (x:start 2 x n)
   where
   start _ _ 0 = []
   start b x n = nbx:(start (b+1) nbx (n-1))
     where nbx = next b x

-- The example used in the text: st 266 2
```

Figure 5: A Haskell program for the computation of the (general) Goodstein sequence.

4.3 More on weak sequence termination

Ordinals can be either *successor ordinals*, say $\alpha + 1$, or *limit ordinals*, say $\lim_{i\to\infty} \alpha_i$, where $\alpha_{i+1} > \alpha_i$ for every $i \in \mathbb{N}$. A limit ordinal has no predecessor.

The set of ordinals is totally ordered and well ordered. Well ordering means that every set of ordinals has a least element. Or equivalently that there is no infinite sequence $\alpha_1 > \alpha_2 > \alpha_3 \ldots$ The equivalence is easy to prove.

The ordinal ω^ω can be seen as the set of ordinals of the form

$$d_1\omega^{e_1} + d_2\omega^{e_2} + \ldots + d_k\omega^{e_k} \tag{1}$$

where $k \geq 0$, $d_1, \ldots, d_n, e_1, \ldots, e_k \in \mathbb{N}$, $d_1 > 0$ (if $k = 0$, the expression is null and represents 0), $e_1 > e_2 > \ldots > e_k$. We simplify the notation, writing for instance $\omega^2 + 5\omega + 3$ instead of $1\omega^2 + 5\omega^1 + 3\omega^0$.

Two sums of the form 1 can be compared in the usual way. That is, let

$$\begin{aligned} \alpha &= d_1\omega^{e_1} + \ldots + d_k\omega^{e_k} \\ \alpha' &= d'_1\omega^{e'_1} + \ldots + d'_k\omega^{e'_k}. \end{aligned}$$

We have $\alpha > \alpha'$ iff

$e_1 > e'_1$ or
$e_1 = e'_1$ and $d_1 > d'_1$ or
$e_1 = e'_1$ and $d_1 = d'_1$ and $e_2 > e'_2$ or
...

Relatively to the weak Goodstein sequence, the function f that maps an integer written in some base b into the corresponding ordinal simply replaces every b^n in the expression by ω^n. For instance, 8043 written in base 10 is $8\cdot10^3+4\cdot10+3$, so that it is transformed in

$$8\omega^3 + 4\omega + 3.$$

Note that if 8043 was a representation of any number n_b written in some base $b \geq 9$ (there is a digit 8!), we would get exactly *the same ordinal*.

When we subtract 1 from a number, the corresponding ordinal is also smaller. An example using the first element $n = 2$ and the initial base $b = 2$ follows.

The elements of the sequence are underlined.

i	(comment)	b	n	n_b	ordinal
1	in base 2:	2	$\underline{5}$	101	ω^2+1
	to base 3:	3	10	101	ω^2+1
2	$n \leftarrow n-1$:	3	$\underline{9}$	100	ω^2
	to base 4:	4	16	100	ω^2
3	$n \leftarrow n-1$:	4	$\underline{15}$	33	$3\omega+3$
	to base 5:	5	18	33	$3\omega+3$
4	$n \leftarrow n-1$:	5	$\underline{17}$	32	$3\omega+2$
...	...	...	...	...	
62			$\underline{0}$		

Note that the sequence finishes with $n = 0$. It is interesting to display the entire sequence:

5	9	15	17	19	21	23	24	25	26
27	28	29	30	31	31	31	31	31	31
31	31	31	31	31	31	31	31	31	31
31	30	29	28	27	26	25	24	23	22
21	20	19	18	17	16	15	14	13	12
11	10	9	8	7	6	5	4	3	2
1	0								

Bibliography

Goodstein, R. (1944), "On the restricted ordinal theorem", Journal of Symbolic Logic 9: 33–41, doi:10.2307/2268019, JSTOR 2268019.

Cichon, E. (1983), "A Short Proof of Two Recently Discovered Independence Results Using Recursive Theoretic Methods", Proceedings of the American Mathematical Society 87: 704–706, doi:10.2307/2043364, JSTOR 2043364.

Caicedo, A. (2007), "Goodstein's function" (PDF), Revista Colombiana de Matemáticas 41 (2): 381–391.

5 Goodstein sequences again

Note. This section was transcribed from

`www.xamuel.com/goodstein-sequences/`

The best math is the kind that makes you do a double-take, when you read a theorem and can't believe it's true. That's how Goodstein Sequences struck me when I was introduced to them. These number-sequences have the property that they grow for awhile but eventually stop growing and shrink down to zero. The way they're defined, at first glance you'd think, "no way, they'll grow forever".

Number systems with different bases

We use decimal, which is the base 10 number system. Other popular systems are binary (base 2) and hexadecimal (base 16). In general, for any base $b > l$, there's a base-b number system. You can write every number in "base b representation" as a string of digits where each digit ranges from 0 to $b-1$ (for example, the decimal digits range from 0 to 9). What's actually going on when a number n has representation, say, "5761" in base 10, that really means $n = 5 * 10^3 + 7 * 10^2 + 6 * 10^1 + 1 * 10^0$.

For the (weak) Goodstein sequences, we'll start with a number written in one base, and then increase the base, without changing the digits.

For example, start with the binary number 1101_2, which is $2^3 + 2^2 + 2^0 = 13$. Now, leaving the digits the same, 1101, change the base to 3. The number becomes 1101_3, which is $3^3 + 3^2 + 3^0 = 37$. As you can see, performing this operation usually makes the end number increase.

Intuitively, if you don't pick some stupid number like 1 at the beginning, one expects that "raising the base" should raise the number rather dramatically. Raise the base repeatedly, and the number should explode toward infinity. But what if every time we raise the base, we subtract 1? At first glance, subtracting 1 should be a drop in the bucket, compared to the huge growth which comes from changing the base. But things are not how they seem...

Weak Goodstein Sequences

The "(weak) Goodstein Sequence starting with n" is defined as follows. The first number in the sequence is n itself. If $n = 0$, the sequence stops there. Otherwise, write n in binary, and then raise the base while keeping the digits the same. Subtract 1 from whatever the result is. This gives you the next term in the series. If it's 0, stop. Otherwise, write this 2nd term in base 3, pump up the base to 4, and subtract 1.

That's the next number. Keep going like this forever (or until you hit 0).

Amazingly, whatever initial n you choose, you'll always reach 0 eventually.

Example: $n = 3$
Let's look at the $n = 3$ weak Goodstein sequence.

- First term: 3.
  - Writing 3 in binary we get 11_2.
  - Raising the base, we get 11_3 (which is 3+1=4).
  - Subtracting 1, we get 10_3 (which is 3).
- Second term: 3.
- Writing 3 in base 3 now, we get 10_3.
- Raising the base, we get 10_4 (which is 4).
- Subtracting 1, we get 3_4 (which is 3).
- Third term: 2.
- Writing 3 in base 4, we get 3_4.
- Raising the base gives 3_5. Minus 1 makes 2_5 (which is 2).
- Fourth term: 2.
- Similarly the next terms are 1 and then 0.

The sequence goes 3, 3, 3, 2, 1, 0. But surely this is just because we chose the initial n too small. Surely if we took something bigger, like $n = 10$, the sequence would be less well-behaved.

Example: n=10

- First term: 10.
  In binary, 10 is 1010_2. So we compute $1010_3 - 1 = 1002_3 = 29$.
- Second term: 29.
  In ternary, 29 is 1002_3. Compute $1002_4 - 1 = 1001_4 = 65$.
- Third term: 65.
  In base 4, 65 is 1001_4. Compute $1001_5 - 1 = 1000_5 = 125$.
- Third term: 125.
  125 is 1000_5. Compute $1000_6 - 1 = 555_6 = 215$.
- Fourth term: 215.

So far the sequence shows no sign of slowing down: it goes 10, 29, 65, 125, 215. The next few terms are 284, 363, 452, 551, and 660. To compute the next term after 660, you have to start using some new terminology for digits, because digits above 9 start popping out. For digits above 9, one solution is to wrap them in parentheses, so for example, 54(11)12 stands for 5*122+4*121+11. Which, incidentally, is 779, the next number in the sequence. Using this digit terminology, the sequence continues:

$54(11)_{12} = 779$

$54(10)_{13} = 907$

$549_{14} = 1045$

$548_{15} = 1193$

(And so on, eventually reaching...)

$540_{23} = 2737$, which gets followed by

$53(23)_{24} = 2975$

$53(22)_{25} = 3222$

(And so on for an awful long while, until...)

$530_{47} = 11186$

$52(47)_{48} = 11663$

(And much, much later on...)

$500_{383} = 733445$

$4(383)(383)_{384} = 737279$

If you're only watching the numbers in decimal form, on the right, it seems hopeless that they'll ever stop growing, and absurd that they'll ever reach zero. But as you follow the structure on the left, you should start to notice what's going on. Although the base is getting larger and larger, the digits are slowly wrapping down. But when we reach a 0 in the l's place [...], we get $b - 1$ where b is the appropriate base. Thus, it takes a really long time before the digits further to the left ever wrap down. Meanwhile, as the base increases, the numbers on the right merrily increase with no sign of slowing down.

But sooner or later, that 4 in the "100's place" is going to become a 3, and then even later that will become a 2... and eventually, this sequence will reach 0. Though, it'll sure take a long time.

Formal proof ordinal arithmetic

In order to formally prove that the (weak) Goodstein Sequences eventually shrink to 0, one uses what's called "the well-foundedness of the ordinal ω^ω". ω^ω is a set, and its elements are ordered, and "ω^ω is well-founded" is the statement that there is no sequence of elements of ω^ω which decreases forever. If you take any sequence of elements of ω^ω and they seem to shrink and shrink, they have to eventually stop shrinking. Another way of saying it is, every subset of ω^ω has a least element.

So what are the elements of ω^ω? They can be very loosely thought of as numbers in "base infinity". Formally, they are (isomorphically, see the "Notes" below) formal strings of the form (1), page 24.

[...]Given a number written in a (finite) base b, you can always map it to an element of ω^ω by "replacing the b's with ω's". For example, $101_2 = 2^2 + 2^0$, gets mapped to $\omega^2 + \omega^0$, and $8043_{10} = 8 \cdot 10^3 + 4 \cdot 10^1 + 3$ gets mapped to $8\omega^3 + 4\omega^1 + 3$.

The key is this. Given a number in a certain base, if you map it into ω^ω, you get the same thing as if you raise the base first. For example, 101_2 maps to $\omega^2 + 1$, but so do 101_3 and 101_4 and even $101_{1000000}$. The numbers themselves differ but they map to the same element of ω^ω.

But if you subtract 1 from a number before mapping it to ω^ω, that will result in a smaller end result. For example, 101_2 maps to $\omega^2 + 1$, but if we subtract 1 first, we get 100_2, which maps to ω^2. A *smaller* result.

To a monster living if "ω^ω space", the Goodstein sequence doesn't seem to grow at all. It seems to *shrink*. That's because the monster cannot "see" the bases. But it can see the results of all those 1-subtractions! And, because cow is well-founded, sequences in cow cannot decrease forever. The Goodstein Sequences always decrease in ω^ω-land, but they can't decrease forever, so they must somehow stop – the only way for that to happen is if they reach 0 eventually.

A slightly stronger Goodstein sequence

In the standard (weak) Goodstein Sequences, we raised the base by 1 each time. This fact is never really used in the proof. All the proof uses is that the digits are the same before we subtract 1. So you could modify the Goodstein Sequence and increase the bases at any arbitrary speed, and still the sequences would go to zero. For example, you could start in base 2, then double the bases every time, going from binary to base 4 to base 8 to base 16. This would cause the initial terms of the sequence to blow up super-exponentially fast – but still the sequences would have to go to zero.

Goodstein sequences

The "weak" Goodstein Sequences I just introduced are a watered down version of the traditional sequences.

To get the "strong" Goodstein Sequences, instead of using base-b representation, one uses a modified version of "Cantor Normal Form". In short, for "base" b, you start by writing your number as a sum of powers of b times coefficients between 0 and $b - 1$. But then, recursively, you do the same for all the exponents – and all their exponents, and so on until eventually you have a "tree" of b's with 0's as leaves.

Example Take $b = 3$, $n = 1210000000000000022_3$.

- Start by writing $n = 1 \cdot 3^{18} + 2 \cdot 3^{17} + 1 \cdot 3^{16} + 2 \cdot 3^1 + 2 \cdot 3^0$.
- The exponents are 18, 17, 16, 1, and 0.

– Recursively, convert the exponents (besides 0) in the same way:

$$18 = 2 \cdot 3^2$$
$$17 = 1 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0$$
$$16 = 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0$$

– The exponents of the exponents are 2, 1, and 0.

– Recursively, convert them (besides 0) in the same way:

$$2 = 2 \cdot 3^0$$
$$1 = 1 \cdot 3^0$$

– All the exponents-of-exponents-of-exponents are 0 and the recursion ends.

– The final result is:
$n = 1 \cdot 3^{2 \cdot 3^{2 \cdot 3^0}} + 2 \cdot 3^{1 \cdot 3^{2 \cdot 3^0} + 2 \cdot 3^{1 \cdot 3^0}} + 1 \cdot 3^{1 \cdot 3^{2 \cdot 3^0} + 2 \cdot 3^{1 \cdot 3^0} + 1 \cdot 3^0} + 2 \cdot 3^{1 \cdot 3^0} + 2 \cdot 3^0$

– This is the (modified) base-3 Cantor Normal Form of 1210000000000000022_3.

The idea behind the strong Goodstein Sequence is as follows. Start with an initial number n. The first term is n. If $n = 0$, stop. Otherwise, to get the 2nd term, write n in base-2 Cantor Normal Form as above. Then go through and replace all the 2's with 3's. For small numbers, this is the same exact thing as "raising the base", but for larger numbers, "raising the Cantor-base" can make them blow up in a cosmically large way.

Anyway, after replacing all the 2's with 3's, that gives you a new number; subtract 1 to get the 2nd element of the sequence. If this 2nd element is 0, stop. Otherwise, write this 2nd element in base-3 Cantor Normal Form, replace the 3's with 4's, and subtract one, and so on.

Again, the resulting sequence is guaranteed to shrink to zero and stop.

Some mathematicians have been known to shake their heads in outright disbelief after being told this fact. It seems like upping the Cantor- base should cause the number to blow up in such a giant way that the 1-subtractions can never hope to catch up. But sure enough, those 1- subtractions eventually kill off the whole sequence.

The formal proof for the stronger Goodstein sequences reaching zero uses the well-foundedness of the ordinal ε_0 ("epsilon naught"). ε_0 is defined as the smallest ordinal x such that $x = \omega^x$. Just as the elements of ω^ω are the formal sums of things of the form ω^n where n is in ω (the set of natural numbers), ω^{ω^ω} is the set of formal sums of things of the form ω^n where n is in ω^ω. And $\omega^{\omega^{\omega^\omega}}$ is the set of sums with summands that look like ω^n where n is in ω^{ω^ω}. This process continues; ε_0 is the union of all of them: the union of ω, ω^ω, ω^{ω^ω}, and so on forever. Some example elements of ε_0:

1

$4\omega + 3$

$8\omega^{2\omega+1} + 73\omega^{\omega+7} + \omega^{\omega+2} + 5\omega^{43} + 8\omega$

$3\omega^{4\omega^{2\omega+9}+12\omega^{\omega+1}} + 834\omega^{52\omega} + 2\omega^8 + 4$

And basically any other "tree" of ω's you can think of...

Just as numbers in base b map nicely into ω^ω, numbers in Cantor Normal Form (base b) map nicely into ε_0, in such a way that changing the b doesn't change the result at all: informally, replace all the b's with ω's. To a monster living in "ε_0 space", the strong Goodstein Sequences don't appear to grow at all, but to shrink – and, like every other ordinal, ε_0 is well-founded, so the Goodstein Sequences are forced to terminate.

Independence results

The theorem, "(strong) Goodstein Sequences always shrink to 0", is very interesting to logicians. The proof uses the fact that ε_0 is well-founded. Coincidentally, ε_0 is "the ordinal" of certain important proof systems including Peano Arithmetic (PA). What this means is that ε_0 is the first ordinal x such that PA cannot prove x is well-founded. (Every ordinal is well-founded, but there are only countably many possible proofs in PA, and there are uncountably many ordinals, so PA can't hope to have an individual well-foundedness proof for each of them. And ε_0 is the first one for which PA has no proof)

So, if you wanted to prove "Goodstein Sequences go to 0" using only the axioms of Peano Arithmetic, you'd have to come up with some clever alternate proof: you can't use the fact that ε_0 is well-founded, because that can't be proven in PA. This suggests that maybe you can't prove the theorem at all... and sure enough, you can't. With Peano Arithmetic alone, it's impossible to prove "Goodstein Sequences go to 0". (Of course, the proof-of-no-proof is too advanced for this essay...) This is one of the most important and surprising nonprovability results of elementary arithmetic.

Notes

In order to clarify the connection between numbers in base b and ordinals, I implicitly reversed the order of ordinal multiplication. Technically, the ordinal 2ω is the same as ω, and what I have written here as 2ω, is actually usually written $\omega \cdot 2$ (in ordinal arithmetic, $a \times b$ is not usually equal to $b \times a$). But I think you'll agree that for the sake of demonstrating that Goodstein Sequences die, the modified notation is far clearer.

6 Goodstein paper: definitions and notation

Note. The "Goodstein paper" is [Goo44] ⊠

Expressing a positive integer:

$$c_k \cdot s^{a_k} + c_{k-1} \cdot s^{a_{k-1}} + \ldots + c_2 \cdot s^{a_2} + c_1 \cdot s^{a_1} + c_0 \tag{2}$$

Note. The base of the representation is denoted by s, instead of b. ⊠

In (2):

$s \geq 2$ is the base, here called "scale symbol".

$0 \leq c_0 < s$.

$0 < c_1, c_2, c_3, \ldots, c_k < s$ (null terms, $c_i = 0$, are not displayed).

$k \geq 0$, $0 < a_1 < a_2 < a_3 < \ldots < a_k$ (exponents decreasing).

Each a_i, $1 \leq i \leq k$, has the form (2). That is, exponents, their exponents and so on, are also expressed in base s.

The form (2) is called the representation of n with digits 0, 1, 2,..., $k-1$ and base (or "scale symbol") s.

$\phi_s(n)$: abbreviation of (2) ⊠

Following (2) we can define

representation of n: $cs^{\phi_s(a)} + \phi_s(n - cs^a)$.

where

$a = \max\{e : s^e \leq n\}$
$cs^a = \max_{c'}\{c's^a : c's^a \leq n\}$

Define also

- $\mathrm{Sb}_y^x\alpha$: the result of replacing x by y in α.
- $\mathrm{T}_x^m(n)$: $\mathrm{Sb}_x^m\phi_m(n)$, that is, apply Sb_x^m to the entire expression $\phi_m(n)$ (so that $\mathrm{Sb}_x^m\phi_m(n)$ is not $\phi_x(n)$).
  Example. $n = 106$. In base 3,
  $n = 10220_3 = 3^4 + 2 \cdot 3^2 + 2 \cdot 3 = 3^{3+1} + 2 \cdot 3^2 + 2 \cdot 3$.
  Now replace 3 by ω: $\mathrm{T}_\omega^3(106) = \omega^{\omega+1} + 2\omega^2 + 2\omega$. ⊠
- Ordinal $< \varepsilon_0$: an element of $\mathrm{T}_\omega^m(n)$ with $m, n \geq 2$.
- $\mathrm{S}_y^x(n)$: $\mathrm{Sb}_y^x\phi_x(n)$ (base y).
  Example: $\mathrm{S}_4^3(34) = \mathrm{S}_4^3(3^3 + 2 \cdot 3 + 1) = 4^4 + 2 \cdot 4 + 1 = 265$. ⊠
  Example: $\mathrm{S}_4^2(16) = \mathrm{S}_4^2(2^{2^2}) = 4^{4^4} = 4^{256}$. ⊠

Working in the other direction, and using Cantor Theory, every ordinal $\alpha < \varepsilon_0$ can be expressed as $\mathrm{T}_\omega^m(n)$ where

- m is greater than every coefficient or exponent in the expression of α as a sum of powers of ω.

– n is uniquely defined by α and m.

Example. Using the previous example, with $\alpha = \omega^{\omega+1} + 2\omega^2 + 2\omega$, let $m = 5$ (or any $m \geq 3$). We have $\mathrm{T}^5_\omega(10220_5) = \alpha$. ⊠

Comparison of $\mathrm{T}^m_\omega(n)$ ordinals.
Ordinals in $\varepsilon_0$can be compared in the same way of the corresponding integers, when we replace a base by a not smaller one. More precisely, let $m_1 \geq m_2 > 1$. Then

$$\mathrm{T}^{m_1}_\omega(n_1) > \mathrm{T}^{m_2}_\omega(n_2) \text{ if } n_1 > \mathrm{S}^{m_1}_{m_2}(n_2).$$

$$\mathrm{T}^{m_1}_\omega(n_1) = \mathrm{T}^{m_2}_\omega(n_2) \text{ if } n_1 = \mathrm{S}^{m_1}_{m_2}(n_2).$$

$$\mathrm{T}^{m_1}_\omega(n_1) < \mathrm{T}^{m_2}_\omega(n_2) \text{ if } n_1 < \mathrm{S}^{m_1}_{m_2}(n_2).$$

This agrees to the usual ordinal comparison. ⊠

Then, a decreasing sequence of ordinals is

$$\mathrm{T}^{m_1}_\omega(n_1),\ \mathrm{T}^{m_2}_\omega(n_2),\ \mathrm{T}^{m_3}_\omega(n_3), \ldots,\ \mathrm{T}^{m_r}_\omega(n_r), \ldots$$

where, for every r, we have
$m_{r+1} \geq m_r$ and $n_{r+1} < \mathrm{S}^{m_r}_{m_{r+1}}(n_r)$.
If the sequence of ordinals is constructive, the sequence m_r is recursive (but possibly not primitive recursive).

For a given sequence m_r, the longest decreasing sequence of integers is $n_{r+1} = \mathrm{S}^{m_r}_{m_{r+1}}(n_r)$, because
$\mathrm{T}^m_\omega(n) = 0$ iff $n = 0$, and
$n < n_r$ implies $\mathrm{S}^{m_r}_{m_{r+1}}(n) < \mathrm{S}^{m_r}_{m_{r+1}}(n_r)$.

Restricted ordinal Theorem. For every non-decreasing function p_r with $p_0 \geq 2$, and for n_r defined by $n_{r+1} = \mathrm{S}^{p_r}_{p_{r+1}}(n_r) \dot{-} 1$, there is an integer r such that $\mathrm{T}^{p_r}_\omega(n_r) = 0$. ⊠

This is equivalent to the number-theoretic proposition

> $P^\star$: For every non-decreasing function p_r with $p_0 \geq 2$, every n_0 and the function n_r defined by $n_{r+1} = \mathrm{S}^{p_r}_{p_{r+1}}(n_r) \dot{-} 1$, there is an integer r such that $n_r = 0$. ⊠

It makes no essential difference if we interchange the operations "change the base $p_r \to p_{r+1}$" and "reduce by 1".

7 A 0-1 function obtained by diagonalisation

The example given in the section proves that there exist *small* total recursive functions that are not primitive recursive.

Theorem 1 *There is a total recursive function f with domain $\{0,1\}$ that is not primitive recursive.*

Proof. Consider the representation of primitive recursive function by Loop programs and an index method for primitive recursive functions such that the transformation of an index into a Loop program is primitive recursive; this transformation does not need to be injective. Denote by L_i the Loop program associated to the index i. Define the function $f(n)$ as

$$f(n) = \begin{cases} 1 & \text{if } L_n(n) = 0 \\ 0 & \text{otherwise.} \end{cases}$$

The function f is total recursive but it is obviously not implementable by any Loop program. Otherwise, let L_m implement f, that is, $f(x) = L_m(x)$ for every x, where $L_m(x)$ denotes the output of the corresponding computation (that always halts). We would have $L_m(m) = 0$ iff $L_m(m) \neq 0$, a contradiction. □

8 A few values of some total non primitive recursive functions

A Primitive recursive functions (from the Wikipedia)

The primitive recursive functions are among the number-theoretic functions, which are functions from the natural numbers (nonnegative integers) $\{0, 1, 2, \ldots\}$ to the natural numbers. These functions take n arguments for some natural number n and are called n-ary.

The basic primitive recursive functions are given by these axioms:

1. Constant function: The 0-ary constant function 0 is primitive recursive.
2. Successor function: The 1-ary successor function S, which returns the successor of its argument (see Peano postulates), is primitive recursive. That is, $S(k) = k + 1$.
3. Projection function: For every $n \geq 1$ and each i with $1 \leq 1 \leq n$, the n-ary projection function P_i^n, which returns its i-th argument, is primitive recursive.

More complex primitive recursive functions can be obtained by applying the operations given by these axioms:

1. Composition: Given f, a k-ary primitive recursive function, and k m-ary primitive recursive functions $g_1, \ldots, g_k$, the composition of f with $g_1, \ldots,$

m	n	$A(m,n)$	$g(m,n)$	$F_0(n,m)$	$F_1(n,m)$	$\mathcal{G}(m,n)$
0	0	1	1	0	0	
0	1	2	2	1	1	
0	2	3	3	2	4	
0	3	4	4	3	11	
0	4	5	5	4	26	
1	0	2	3	1	1	
1	1	3	4	2	3	
1	2	4	5	3	8	
1	3	5	6	4	19	
1	4	6	7	5	42	
2	0	3	4	2	2	
2	1	5	7	3	5	
2	2	6	10	4	12	
2	3	9	13	5	27	
2	4	11	16	6	58	
3	0	5	7	3	3	
3	1	13	25	4	7	
3	2	29	79	5	16	
3	3	61	241	6	35	
3	4	125	727	7	74	
...	...	...	...	...	...	...
m	n	$2 \overset{m-2}{\uparrow} (n+3)-3$	$3 \overset{m-2}{\uparrow} (n+2)-2$			

Figure 6: First values of some functions that are not primitive recursive: $A(m,n)$, (i) the Ackermann function, (ii) $g(m,n)$ defined in [], (iii) ...

g_k, i.e. the m-ary function

$$h(x_1, \ldots, x_m) = f(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))$$

is primitive recursive.

2. Primitive recursion: Given f, a k-ary primitive recursive function, and g, a $(k+2)$-ary primitive recursive function, the $(k+1)$-ary function h is defined as the primitive recursion of f and g, i.e. the function h is primitive recursive when

$$\begin{aligned} h(0, x_1, \ldots, x_k) &= f(x_1, \ldots, x_k) && \text{and} \\ h(S(y), x_1, \ldots, x_k) &= g(y, h(y, x_1, \ldots, x_k), x_1, \ldots, x_k). \end{aligned}$$

The primitive recursive functions are the basic functions and those obtained from the basic functions by applying these operations a finite number of times.

Role of the projection functions

The projection functions can be used to avoid the apparent rigidity in terms of the arity of the functions above; by using compositions with various projection functions, it is possible to pass a subset of the arguments of one function to another function. For example, if g and h are 2-ary primitive recursive functions then

$$f(a, b, c) = g(h(c, a), h(a, b))$$

is also primitive recursive. One formal definition using projection functions is

$$f(a, b, c) = g(h(P_3^3(a, b, c), P_1^3(a, b, c)), h(P_1^3(a, b, c), P_2^3(a, b, c))).$$

Converting predicates to numeric functions

In some settings it is natural to consider primitive recursive functions that take as inputs tuples that mix numbers with truth values $\{t = \mathsf{true}, f = \mathsf{false}\}$, or that produce truth values as outputs (see Kleene [1952 pp. 226–227]). This can be accomplished by identifying the truth values with numbers in any fixed manner.

For example, it is common to identify the truth value t with the number 1 and the truth value f with the number 0. Once this identification has been made, the characteristic function of a set A, which literally returns 1 or 0, can be viewed as a predicate that tells whether a number is in the set A. Such an identification of predicates with numeric functions will be assumed for the remainder of this article.

Computer language definition

An example of a primitive recursive programming language is one that contains basic arithmetic operators (e.g. + and −, or ADD and SUBTRACT), conditionals and comparison (IF-THEN, EQUALS, LESS-THAN), and bounded loops, such as the basic for loop, where there is a known or calculable upper bound to all loops (FOR i FROM 1 to n, with neither i nor n modifiable by the loop body).

No control structures of greater generality, such as while loops or IF-THEN plus GOTO, are admitted in a primitive recursive language. Douglas Hofstadter's Bloop in "Gödel, Escher, Bach" is one such. Adding unbounded loops (WHILE, GOTO) makes the language partial recursive, or Turing-complete; Floop is such, as are almost all real-world computer languages.

Arbitrary computer programs, or Turing machines, cannot in general be analysed to see if they halt or not (the halting problem). However, all primitive recursive functions halt. This is not a contradiction; primitive recursive programs are a non-arbitrary subset of all possible programs, constructed specifically to be analysable.

A.1 Examples

Most number-theoretic functions definable using recursion on a single variable are primitive recursive. Basic examples include the addition and truncated subtraction functions. A

Addition

Intuitively, addition can be recursively defined with the rules:

$$\begin{aligned} \text{add}(0, x) &= x, \\ \text{add}(n+1, x) &= \text{add}(n, x) + 1. \end{aligned}$$

To fit this into a strict primitive recursive definition, define:

$$\begin{aligned} \text{add}(0, x) &= P_1^1(x), \\ \text{add}(S(n), x) &= S(P_2^3(n, \text{add}(n, x), x)). \end{aligned}$$

Here $S(n)$ is "the successor of n" (i.e., $n+1$), P_1^1 is the identity function, and P_2^3 is the projection function that takes 3 arguments and returns the second one. Functions f and g required by the above definition of the primitive recursion operation are respectively played by P_1^1 and the composition of S and P_2^3.

Subtraction

Because primitive recursive functions use natural numbers rather than integers, and the natural numbers are not closed under subtraction, a truncated subtraction function (also called "proper subtraction") is studied in this context. This limited subtraction function $\text{sub}(a, b)$ (or $b \dot{-} a$) returns $b - a$ if this is nonnegative and returns 0 otherwise.

The predecessor function acts as the opposite of the successor function and is recursively defined by the rules:

$$\begin{aligned} \text{pred}(0) &= 0, \\ \text{pred}(n+1) &= n. \end{aligned}$$

These rules can be converted into a more formal definition by primitive recursion:

$$\begin{aligned} \text{pred}(0) &= 0, \\ \text{pred}(S(n)) &= P_1^2(n, \text{pred}(n)). \end{aligned}$$

The limited subtraction function is definable from the predecessor function in a manner analogous to the way addition is defined from successor:

$$\begin{aligned} \text{sub}(0, x) &= P_1^1(x), \\ \text{sub}(S(n), x) &= \text{pred}(P_2^3(n, \text{sub}(n, x), x)). \end{aligned}$$

Here $\text{sub}(a, b)$ corresponds to $b \dot{-} a$; for the sake of simplicity, the order of the arguments has been switched from the "standard" definition to fit the requirements of primitive recursion. This could easily be rectified using composition with suitable projections.

Other operations on natural numbers

Exponentiation and primality testing are primitive recursive. Given primitive recursive functions e, f, g, and h, a function that returns the value of g when $e \leq f$ and the value of h otherwise is primitive recursive.

Operations on integers and rational numbers

By using Gödel numberings, the primitive recursive functions can be extended to operate on other objects such as integers and rational numbers. If integers are encoded by Gödel numbers in a standard way, the arithmetic operations including addition, subtraction, and multiplication are all primitive recursive. Similarly, if the rationals are represented by Gödel numbers then the field operations are all primitive recursive.

A.2 Relationship to recursive functions

The broader class of partial recursive functions is defined by introducing an unbounded search operator. The use of this operator may result in a partial function, that is, a relation with at most one value for each argument, but does not necessarily have any value for any argument (see domain). An equivalent definition states that a partial recursive function is one that can be computed by a Turing machine. A total recursive function is a partial recursive function that is defined for every input.

Every primitive recursive function is total recursive, but not all total recursive functions are primitive recursive. The Ackermann function $A(m, n)$ is a well-known example of a total recursive function that is not primitive recursive. There is a characterisation of the primitive recursive functions as a subset of the total recursive functions using the Ackermann function. This characterisation states that a function is primitive recursive if and only if there is a natural

number m such that the function can be computed by a Turing machine that always halts within $A(m, n)$ or fewer steps, where n is the sum of the arguments of the primitive recursive function.[4]

An important property of the primitive recursive functions is that they are a recursively enumerable subset of the set of all total recursive functions (which is not itself recursively enumerable). This means that there is a single computable function f(e,n) such that:

- For every primitive recursive function g, there is an e such that $g(n) = f(e, n)$ for all n, and
- For every e, the function $h(n) = f(e, n)$ is primitive recursive.

However, the primitive recursive functions are not the largest recursively enumerable set of total computable functions.

Limitations

Primitive recursive functions tend to correspond very closely with our intuition of what a computable function must be. Certainly the initial functions are intuitively computable (in their very simplicity), and the two operations by which one can create new primitive recursive functions are also very straightforward. However the set of primitive recursive functions does not include every possible total computable function – this can be seen with a variant of Cantor's diagonal argument. This argument provides a total computable function that is not primitive recursive. A sketch of the proof is as follows:

> The primitive recursive functions of one argument (i.e., unary functions) can be computably enumerated. This enumeration uses the definitions of the primitive recursive functions (which are essentially just expressions with the composition and primitive recursion operations as operators and the basic primitive recursive functions as atoms), and can be assumed to contain every definition once, even though a same function will occur many times on the list (since many definitions define the same function; indeed simply composing by the identity function generates infinitely many definitions of any one primitive recursive function). This means that the n-th definition of a primitive recursive function in this enumeration can be effectively determined from n. Indeed if one uses some Gödel

[4] This follows from the facts that the functions of this form are the most quickly growing primitive recursive functions, and that a function is primitive recursive if and only if its time complexity is bounded by a primitive recursive function. For the former, see Linz, Peter (2011), "An Introduction to Formal Languages and Automata", p. 332. For the latter, see Moore, Cristopher; Mertens, Stephan (2011), "The Nature of Computation, Oxford University Press, p. 287

numbering to encode definitions as numbers, then this n-th definition in the list is computed by a primitive recursive function of n. Let f_n denote the unary primitive recursive function given by this definition.

Now define the "evaluator function" ev with two arguments, by $\mathsf{ev}(i, j) = f_i(j)$. Clearly ev is total and computable, since one can effectively determine the definition of f_i, and being a primitive recursive function f_i is itself total and computable, so $f_i(j)$ is always defined and effectively computable. However a diagonal argument will show that the function ev of two arguments is not primitive recursive.

Suppose ev were primitive recursive, then the unary function g defined by $g(i) = S(\mathsf{ev}(i, i))$ would also be primitive recursive, as it is defined by composition from the successor function and ev. But then g occurs in the enumeration, so there is some number n such that $g = f_n$. But now $g(n) = S(\mathsf{ev}(n, n)) = S(f_n(n)) = S(g(n))$ gives a contradiction.

This argument can be applied to any class of computable (total) functions that can be enumerated in this way, as explained in the article "Machines that always halt". Note however that the *partial* computable functions (those that need not be defined for all arguments) can be explicitly enumerated, for instance by enumerating Turing machine encodings.

Other examples of total recursive but not primitive recursive functions are known:

- The function that takes m to $\mathsf{Ackermann}(m, m)$ is a unary total recursive function that is not primitive recursive.
- The Paris–Harrington theorem involves a total recursive function that is not primitive recursive. Because this function is motivated by Ramsey theory, it is sometimes considered more "natural" than the Ackermann function.
- The Sudan function.
- The Goodstein function.
- The 0/1 diagonal function.

A.3 Some common primitive recursive functions

The following examples and definitions are from Kleene (1952) pp. 223-231 – many appear with proofs. Most also appear with similar names, either as proofs or as examples, in Boolos-Burgess-Jeffrey 2002 pp. 63-70.

In the following we observe that primitive recursive functions can be of four types, where n is the arity of the function:

- *functions* for short: "number-theoretic functions" from $\{0, 1, 2, \ldots\}^n$ to $\{0, 1, 2, \ldots\}$,
- *predicates*: from predicates: from $\{0, 1, 2, \ldots\}^n$ to truth values $\{t = \mathsf{true}, f = \mathsf{false}\}$,
- *propositional connectives*: from truth values $\{t, f\}^n$ to truth values $\{t, f\}$,
- *representing functions*: from truth values $\{t, f\}$ to $\{0, 1, 2, \ldots\}$. Many times a predicate requires a representing function to convert the predicate's output $\{t, f\}$ to $\{0, 1\}$ (note the order "t" to "0" and "f" to "1" matches with $\sim \mathsf{sg}()$ defined below). By definition a function $\phi(x)$ is a "representing function" of the predicate $P(x)$ if ϕ takes only values 0 and 1 and produces 0 when P is true".

In the following the mark "′", e.g. a', is the primitive mark meaning "the successor of", usually thought of as "+1", e.g. $a + 1 \stackrel{\text{def}}{=} a'$. The functions 16-20 and $\#G$ are of particular interest with respect to converting primitive recursive predicates to, and extracting them from, their "arithmetical" form expressed as Gödel numbers.

Addition: $a + b$
Multiplication: $a \times b$
Exponentiation: a^b
Factorial $a!$: $0! = 1$, $a'! = a! \times a'$
pred(a): (Predecessor or decrement): If $a > 0$ then $a - 1$ else 0
Proper subtraction $a \dot{-} b$: if $a \geq b$ then $a - b$ else 0
$\min(a_1, \ldots, a_n)$
$\max(a_1, \ldots, a_n)$
Absolute difference: $|a - b| \stackrel{\text{def}}{=} (a \dot{-} b) + (b \dot{-} a)$
$\mathsf{sg}(a)$: $\mathsf{NOT}[\mathsf{signum}(a)]$: if $a = 0$ then 1 else 0
$\mathsf{sg}(\mathsf{a})$: $\mathsf{signum}(a)$: if $a = 0$ then 0 else 1
$a \mid b$: (a divides b): if $b = k \times a$ for some k then 0 else 1
$\mathsf{remainder}(a, b)$: the leftover if b does not divide a "evenly"
(also called $\mathsf{MOD}(a, b)$)
$a = b$: $\mathsf{sg}|a - b|$[5] $a < b$: $\mathsf{sg}(a' \dot{-} b)$

[5]Kleene's convention was to represent true by 0 and false by 1; presently, especially in computers, the most common convention is the reverse, namely to represent true by 1 and false by 0, which amounts to changing sg into ∼sg here and in the next item.

$\mathsf{Pr}(a)$: a is a prime number.
$\mathsf{Pr}(a) \stackrel{\text{def}}{=} [a > 1] \wedge [\mathsf{NOT}(\exists c)_{1<c<a} : c \mid a]$
p_i: the $i+1$-st prime number
$(a)_i$: exponent of p_i in a: the unique x such that $[p_i^x \mid a] \wedge [\mathsf{NOT}(p_i^{x'} \mid a)]$
$\mathsf{lh}(a)$: the "length" or number of non-vanishing exponents in a
$\mathsf{lo}(a, b)$: logarithm of a to the base b

In the following, the abbreviation $\boldsymbol{x} \stackrel{\text{def}}{=} x_1, \ldots x_n$; subscripts may be applied if the meaning requires.

B Primitive recursive functions: computation time

We collect here some results related to the computation time of primitive recursive functions.

B.1 Loop programs: computation time lower bound

Here we follow essentially the argument of Meyer and Ritchie in [MR67]. Denote $(x_1, \ldots, x_n)$ by $\mathbf{x}$. Suppose that $f(\mathbf{x})$ is computed by some Loop program L. The registers used in L include $\mathbf{x}$ and possibly other registers initialised with 0.

Following [MR67] we assume that the following Loop instructions take 1 unit of time

(a) `inc` x,

(b) `dec` x,

(c) $x \leftarrow 0$,

(d) $x \leftarrow y$.

In a loop instruction `for` n $\langle P \rangle$ the execution time is only assigned to P, and not to the overhead time needed to implement the 'for' instruction, say to translate it in something like

$$
\begin{array}{ll}
 & c \leftarrow n \\
\mathtt{L}: & \mathtt{if}\ c == 0\ \mathtt{then\ goto}\ \mathtt{L}' \\
 & \quad \langle P \rangle \\
 & \mathtt{dec}\ c \\
 & \mathtt{goto}\ \mathtt{L} \\
\mathtt{L}' & \ldots
\end{array}
$$

Let m be the largest value contained in any register, Thus, m is a function of time, say $m(t)$. Initially m is the largest argument of f. When any of the instructions (a) to (d), is executed, the value of m increases by at most 1 (this

increase happens only with the instruction (a)). As the output of the program is one of the registers, say y, we have at the end of the computation,

$$y(t_f) \leq m(t_f) \leq \max(\mathbf{x}(0)) + t(\mathbf{x}(0))$$

where the initial and final values of time are 0 and t_f respectively. For instance, the computation of $f(0, 2, 5) = 1000$ takes at least 995 units of time.

Theorem 2 *Let f be a primitive recursive function. The time needed to compute $f(\mathbf{x})$ by a Loop program is at least $f(\mathbf{x}) - \max(\mathbf{x})$.*

Note that in this result $\mathbf{x}$ denotes a tuple of *mathematical variables*, not a tuple of registers.

B.2 The computation time is also primitive recursive

Consider the computation of a primitive recursive function f by a Loop program L. The computation time of $f(\mathbf{x})$ can be obtained by a Loop program L' that is similar to L, except that

(a) There an additional register t (initialised with 0).

(b) Immediately after any instruction of the forms (a), (b), (c), or (d) (see page 42) a new instruction `inc` t is inserted.

(c) When the program halts the output is the contents of t.

We see that the computation time of $f(\mathbf{x})$ can be computed by a Loop program, so that $t(\mathbf{x})$ is primitive recursive.

B.3 Computation time of total recursive functions that are not primitive recursive

Consider any function $f(x)$ that is total recursive but not primitive recursive. The computation *time* (by a Turing machines) of $f(x)$ grows extremely fast with x, in fact it grows faster that any PR function $t(x)$.

This statement can be proved by contradiction. Suppose that $t(x) \leq g(x)$, where $g(x)$ is PR, and consider the computation of $f(x)$ by a Turing machine. Use the following variant of the Kleene Normal Form Theorem ([Odi89, BBJ07, Kle52]):

$$f(x) = U(e, \mathbf{x}, \mu y : [T(e, \mathbf{x}, y) = 1])$$

where

$f \equiv \varphi_e$ (e is an index of f),
$T(e, \mathbf{x}, y) = 0$ iff the computation $\varphi_e(\mathbf{x})$ halts in $\leq y$ steps,
$U(e, \mathbf{x}, y)$ "runs" y steps the computation $\varphi_e(x)$

Once f is total recursive, the value of $\mu y : [T(e, \mathbf{x}, y) = 1]$ is defined; denote it by y_f. Thus $f(\mathbf{x}) = U(e, \mathbf{x}, y_f)$.

Consider a Loop program $L_e(\mathbf{x})$ that computes $f(\mathbf{x})$ in a minimum possible time.

Let e be an index for f such that $L_e(\mathbf{x})$ is a "fastest time" computation of $L_e(\mathbf{x})$. Denote this time by $t(\mathbf{x})$ (we should of course measure this time in terms of some convenient "order of magnitude"). We can interpret $t(\mathbf{x})$, the running time of $f(\mathbf{x})$, as $\mu y : T(e, \mathbf{x}, y) = 1$.

As above let $y_f = \mu y : [T(e, \mathbf{x}, y) = 1]$. We assume that for every $y \geq y_f$ we have $T(e, \mathbf{x}, y) = 1$, and that $U(e, \mathbf{x}, y) = U(e, \mathbf{x}, y_f)$.

Let $g(\mathbf{x})$ be an upper bound of the computation time. Consider the following computation of $f(\mathbf{x})$, where $\{\dots\}$ denotes a programming comment.

Input: $\mathbf{x}$, output $f(\mathbf{x})$.

1. 1. Compute $z = g(\mathbf{x})$. $\{$we know that $T(e, x, z) = 1\}\}$
2. Run as $T(e, x, z)$ $\{$which halts$\}$ and output the corresponding result.

By assumption, g is primitive recursive; the Normal Form Theorem states that U and T are also primitive recursive. It follows that f is primitive recursive, a contradiction.

References

[BBJ07] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and Logic*. Cambridge University Press, 2007. Fifth Edition.

[Goo44] R. L. Goodstein. On the restricted ordinal theorem. *The Journal of Symbolic Logic*, 9(2):33–41, 1944.

[Kle52] Stephan Cole Kleene. *Introduction to Metamathematics*. North-Holland, 1952. Reprinted by Ishi press, 2009.

[MP16] Armando B. Matos and António Porto. Ackermann and the superpowers (revised in 2011 and 2016). *ACM SIGACT*, 12(Fall 1980), 1980/1991/2016.

[MR67] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proceedings of 22nd National Conference of the ACM*, pages 465–469. Association for Computing Machinery, 1967.

[Odi89] Piergiorgio Odifreddi. *Classical Recursion Theory – The Theory of Functions and Sets of Natural Numbers*, volume I. Studies in Logic and the Foundations of Mathematics. Elsevier North Holland, 1989.