Computability, Definition

We need a formal definition of computability that

- is easy to understand and apply, and
- matches our intuitive notion of computability.

There are many plausible approaches, we'll start with a model for arithmetic functions that dates back to the 19th century and is exceedingly straightforward.

Knee-Jerk Reaction

Computable means: can be done, in principle, by a standard digital computer.

This sounds good to anyone who has ever written and executed a program; sadly, there are lots of problems with this approach.

First, the hedge “in principle” means you really have to abstract away from a concrete physical device (time, space, mass, energy, ...).

Then there is the question which operating system, which programming language, which compiler? These typically have no clear semantics, so what exactly are we defining?

Math versus CS

Aka computability versus algorithms.

Implementation details are usually of little interest in mathematics, it only matters whether a function is computable or not. Computability is a central foundational issue, but does not require detailed analysis.

CS is a bit different here, computability alone is typically of interest only as a very first step (e.g., when one establishes the decidability of some problem), to be followed by a careful effort to streamline the computations (so as to keep resource bounds low).

This second step leads into the realm of algorithms, which should not be confused with computable functions. Arguably, algorithms are a much more complicated and currently only ill-defined notion.

Computable Arithmetic Functions

For the time being we consider only one data type: the natural numbers \( \mathbb{N} \). The corresponding functions are called arithmetic functions or number theoretic functions: Some examples are familiar to any kindergartener: addition, multiplication, squaring, roots, exponentiation and so on.

\[ f : \mathbb{N}^n \rightarrow \mathbb{N} \]

We introduce a model of computation that is designed to work particularly well with these, no input/output coding is required.

For the time being, all our functions will be total.
**Primitive Recursion**

The main idea behind our first model is quite straightforward: we will define a function \( f : \mathbb{N} \times \mathbb{N}^m \rightarrow \mathbb{N} \) by

- defining \( f(0, y) \) explicitly, and
- defining \( f(x + 1, y) \) in terms of \( f(x, y) \).

This should produce computable functions: we can either compute \( f(n, y), f(n - 1, y), f(n - 2, y) \ldots \) top-down (slightly complicated, requires a recursion stack), or we can compute bottom-up \( f(0, y), f(1, y), f(2, y) \ldots \) This requires no more than a loop.

Later we will see more complicated forms of recursion.

---

**Details: Primitive Recursive Functions**

Interestingly, Gödel encountered the problem of defining computable functions working on his seminal incompleteness theorem. He introduced a class of "very simple," easily describable functions, that are now called **primitive recursive functions**.

Recursion is the key idea, but we need a few more ingredients such as composition and projections.

It will always be crystal clear that our functions are intuitively computable.

---

**Composition**

Given functions \( g_i : \mathbb{N}^m \rightarrow \mathbb{N} \) for \( i = 1, \ldots, n \), \( h : \mathbb{N}^m \rightarrow \mathbb{N} \), we define a new function \( f : \mathbb{N}^m \rightarrow \mathbb{N} \) by composition as follows:

\[
 f(x) = h(g_1(x), \ldots, g_n(x))
\]

Notation: we write \( \text{Comp}[h, g_1, \ldots, g_n] \) or simply \( h \circ (g_1, \ldots, g_n) \) inspired by the the well-known special case \( m = 1 \):

\[
 (h \circ g)(x) = h(g(x)).
\]

It is clear that computability is closed wrt composition: output can be re-used as input.

---

**Bureaucracy: Projections**

Unfortunately, composition by itself is not quite enough.

Suppose we have a binary version \( \text{add} \) of addition, and want to define a ternary version. No problem:

\[
 \text{add}_3(x, y, z) = \text{add}(x, \text{add}(y, z))
\]

But, this is not allowed according to our definition of composition; just try.

We need a simple auxiliary tool, so-called **projections**:

\[
 P^n_m : \mathbb{N}^m \rightarrow \mathbb{N} \quad P^n_m(x_1, \ldots, x_n) = x_i
\]

where \( 1 \leq i \leq n \) for the projections.

---

**Describing Functions**

Now we can write

\[
 \text{add}_3 = \text{add} \circ (P^3_1, \text{add} \circ (P^3_2, P^3_3))
\]

Note that no variables are needed in this notation system.

In general, we will prefer the informal notation, but you should know how to use projections to write formally correct terms.

---

**Clones**

A **clone** is a collection of arithmetic functions that contains all projections and is closed under composition.

For example, all projections form a clone, as do all arithmetic functions.

To get something more interesting, we need to consider clones that are generated by

- given basic functions, and/or
- closed under additional operations.
Basic Arithmetic Functions

When dealing with natural numbers, it is natural (duh) to have

- Constant zero
  \[0 : \mathbb{N}\]
- Successor function
  \[S : \mathbb{N} \to \mathbb{N}, S(x) = x + 1\]

This is a rather spartan set of built-in functions, but as we will see it’s all we need. Needless to say, these functions are trivially computable.

In fact, it is hard to give a reasonable description of the natural numbers without them.

Example: Factorials

The standard definition of the factorial function uses recursion like so:

\[
f(0) = 1 \\
f(x+1) = (x+1) \cdot f(x)
\]

To write the factorial function in the form \(f = \text{Prec}[h,g]\) we need

\[
g : \mathbb{N}^0 \to \mathbb{N}, \quad g() = 1 \\
h : \mathbb{N}^2 \to \mathbb{N}, \quad h(u,v) = (u+1) \cdot v
\]

\(g\) is none other than \(S \circ 0\) and \(h\) is multiplication combined with the successor function:

\[f = \text{Prec}[\text{mult} \circ (S \circ P_1^2, P_1^2), S \circ 0]\]

Digging Down

To get multiplication we use another recursion:

\[
mult(0,y) = 0 \\
mult(x+1,y) = \text{add}(\text{mult}(x,y), y)
\]

Here we use addition, which can in turn be defined by yet another recursion:

\[
\text{add}(0,y) = y \\
\text{add}(x+1,y) = S(\text{add}(x,y))
\]

Since \(S\) is a basic function, this is a complete, inductive proof that factorial is primitive recursive.

R. Dedekind

These equational, inductive definitions of basic arithmetic functions date back to Dedekind’s 1888 paper “Was sind und was sollen die Zahlen?”

Arithmetic

It is a good idea to go through the definitions of all the standard basic arithmetic functions from the p.r. point of view.

\[
\text{add} = \text{Prec}[S \circ P_1^2, P_1^1] \\
\text{mult} = \text{Prec}[\text{add} \circ (P_2^2, P_1^2), 0] \\
\text{pred} = \text{Prec}[P_1^1, 0] \\
\text{sub} = \text{sub} \circ (P_2^2, P_1^1)
\]

Since we are dealing with \(\mathbb{N}\) rather than \(\mathbb{Z}\), sub here is proper subtraction: \(x - y = x - y\) whenever \(x \geq y\), and 0 otherwise.

Exercise

Show that all these functions behave as expected.
Informal versus Formal

Strictly speaking, in order to exhibit a p.r. function, we should write down a term in the corresponding programming language. For example, the following expression shows that the factorial function is p.r.

\[
\text{Prec}[\text{Prec}[\text{S} \circ P_3, P_1^2] \circ (P_3^1, P_3^3), 1] \circ (S \circ P_1, P_2^2), 1]
\]

where we have written 1 for \(S \circ 0\) for legibility. The innermost \(\text{Prec}\) yields addition, the next multiplication and the last factorial.

This is an instance of the old battle between formal and informal proofs. If you are a theorem prover, the formal version is far better. But it is very hard on the human eye: we will usually prefer the informal descriptions from above.

A Programming Language

So we really have a simple programming language \(\text{PR}\), consisting of just 3 categories:

- \text{bureaucracy} projections, composition
- \text{basics} constants zero, successor function
- \text{constructor} primitive recursion

Any well-formed term \(\tau\) (aka program) in this language describes an arithmetic function \(\tau^*\).

Evaluation

Note that it would be quite straightforward to program out an evaluation operator \(\text{eval}\) that takes as input any well-formed term \(\tau\) of arity \(n\) and input \(x = x_1, \ldots, x_n \in \mathbb{N}\):

\[
\text{eval}(\tau, x) = \text{value of } \tau^* \text{ on arguments } x
\]

Exercise

Write a compiler that given any string \(\tau\) checks whether it is a well-formed expression denoting a primitive recursive function.

Exercise

Write an interpreter for primitive recursive functions (i.e., implement \(\text{eval}\)).

A Primitive Recursive Zoo

We have seen that basic arithmetic functions such as addition, multiplication and proper subtraction are all primitive recursive.

In fact, it is quite difficult to come up with an arithmetic function that fails to be primitive recursive, yet is somehow intuitively computable. Go through any basic book on number theory, everything will be p.r.

To show that lots of functions are primitive recursive we need two tools:

- A pool of known p.r. functions, and
- strong closure properties.

Admissibility

Here is an example of a closure property that is not obvious from the definitions. Apparently, we lack a mechanism for definition-by-cases:

\[
f(x) = \begin{cases} 3 & \text{if } x < 5, \\ x^2 & \text{otherwise.} \end{cases}
\]

We know that \(x \mapsto 3\) and \(x \mapsto x^2\) are p.r., but is \(f\) also p.r.?

We want to show that definition by cases is admissible in the sense that when applied to primitive recursive functions/relations we obtain another primitive recursive function. Having lots of admissible operations around makes it easier to show that some functions are primitive recursive.
Definition by Cases 25

**Definition**

Let \( g, h : \mathbb{N}^n \to \mathbb{N} \) and \( R \subseteq \mathbb{N}^n \).
Define \( f = \text{DC}[g, h, R] \) by

\[
f(x) = \begin{cases} 
  g(x) & \text{if } x \in R, \\
  h(x) & \text{otherwise.}
\end{cases}
\]

We need to explain what it means for the relation \( R \) to be primitive recursive, we’ll do that in a minute.

Sign and Inverted Sign 26

The first step towards implementing definition-by-cases is a bit strange, but we will see that the next function is actually quite useful.

The \textit{sign} function is defined by

\[
\text{sign}(x) = \min(1, x)
\]

so that \( \text{sign}(0) = 0 \) and \( \text{sign}(x) = 1 \) for all \( x \geq 1 \). Sign is primitive recursive: \( \text{Prec}[\text{min}(1, \_)] \) in sloppy notation.

Similarly the \textit{inverted sign} function is primitive recursive:

\[
\text{sign}^{-1}(x) = 1 - \text{sign}(x)
\]

Relations 27

As usual, define the characteristic function of a relation \( R \)

\[
\text{char}_R(x) = \begin{cases} 
  1 & x \in R, \\
  0 & \text{otherwise.}
\end{cases}
\]

to translate relations into functions.

**Definition**

A relation is \textit{primitive recursive} if its characteristic function is primitive recursive.

We will use analogous definitions later for all kinds of other types of computable functions: Turing, polynomial time, polynomial space, whatever.

Equality and Order 28

Define \( E : \mathbb{N}^2 \to \mathbb{N} \) by

\[
E = \text{sign} \circ \text{add} \circ (\text{sub} \circ (P^2_1, P^2_2), \text{sub} \circ (P^2_2, P^2_1))
\]

Or, less formally, but more intelligible:

\[
E(x, y) = \text{sign}((x \cdot y) + (y \cdot x))
\]

Then \( E(x, y) = 1 \) iff \( x = y \), and 0 otherwise. Hence equality is primitive recursive. Even better, all standard order relations such as

\[
\neq, \leq, <, \geq, ...
\]

are primitive recursive (so we can use them e.g. in definitions by cases).

Closure Properties 29

**Proposition**

The primitive recursive relations are closed under intersection, union and complement.

**Proof.**

\[
\begin{align*}
\text{char}_{R \cap S} &= \text{mult} \circ (\text{char}_R, \text{char}_S) \\
\text{char}_{R \cup S} &= \text{sign} \circ \text{add} \circ (\text{char}_R, \text{char}_S) \\
\text{char}_{-R} &= \text{sub} \circ (S \circ 0, \text{char}_R)
\end{align*}
\]

In other words, primitive recursive relations form a Boolean algebra, and even an effective one: we can compute the Boolean operations.

Arithmetic and Logic 30

Note what is really going on here: we are using arithmetic to express logical concepts such as disjunction.

The fact that this translation is possible, and requires very little on the side of arithmetic, is a central reason for the algorithmic difficulty of many arithmetic problems: logic is hard, by implication arithmetic is also difficult.

For example, finding solutions of Diophantine equations is hard.

**Exercise**

Show that every finite set is primitive recursive. Show that the even numbers are primitive recursive.
**DC is Admissible**

**Proposition**

If \( g, h, R \) are primitive recursive, then \( f = DC[g, h, R] \) is also primitive recursive.

**Proof.**

\[
f = \text{add} \circ (\text{mult} \circ (\text{char}_R, g), \text{mult} \circ (\overline{\text{char}}_R, h))
\]

Less cryptically

\[
f(x) = \text{char}_R(x) \cdot g(x) + \overline{\text{char}}_R(x) \cdot h(x)
\]

Since either \( \text{char}_R(x) = 0 \) and \( \overline{\text{char}}_R(x) = 1 \), or the other way around, we get the desired behavior. \( \square \)

**More Bounded Sum**

Also, abusing notation ever so slightly, we have written \( 0^n \) to indicate an \( n \)-ary function that is constant 0. Hence, by definition \( 0^0 \) is primitive recursive \( ; - \)

**Exercise**

Show that \( 0^n \) is primitive recursive.

**Exercise**

Repeat the proof for products.

**Exercise**

Show that \( f(x, y) = \sum_{z < x} g(z, y) \) is primitive recursive when \( h \) is primitive recursive and strictly monotonic.

**BS is Admissible**

One can show that bounded search is also admissible, it adds nothing to the class of p.r. functions.

**Proposition**

If \( g \) is primitive recursive, then so is \( BS[g] \).

**Exercise**

Show that bounded search is indeed admissible ("primitive recursive functions are closed under bounded search").

**Bounded Sum**

**Proposition**

Let \( g : \mathbb{N}^{n+1} \to \mathbb{N} \) be primitive recursive, and define

\[
f(x, y) = \sum_{z < x} g(z, y)
\]

Then \( f : \mathbb{N}^{n+1} \to \mathbb{N} \) is again primitive recursive. The same holds for products.

**Proof.**

\[
\text{Prec} \circ \text{add} \circ (g \circ (P_{n+2}, P_{n+2}, \ldots, P_{n+2}, 0^n), 0^n)
\]

Less formally,

\[
\begin{align*}
    f(0, y) &= 0 \\
    f(x^+, y) &= f(x, y) + g(x, y)
\end{align*}
\]

Here we have written \( x^+ \) instead of \( x + 1 \). Yes, that helps.

**Bounded Search**

A particularly important algorithmic technique is search over some finite domain.

For example, in brute-force factoring \( n \) we are searching over an interval \([2, n - 1]\) for a number that divides \( n \). Or in a chess program we search for the optimal next move over a space of possible next moves.

We can model search in the realm of p.r. functions as follows.

**Definition (Bounded Search)**

Let \( g : \mathbb{N}^{n+1} \to \mathbb{N} \). Then \( f = BS[g] : \mathbb{N}^{n+1} \to \mathbb{N} \) is the function defined by

\[
f(x, y) = \begin{cases} 
    \min \{ z < x \mid g(z, y) = 0 \} & \text{if } z \text{ exists}, \\
    x & \text{otherwise}.
\end{cases}
\]

**Dire Warning:**

But we have to have a p.r. bound, unbounded search as in

\[
f(y) := \min \{ z \mid g(z, y) = 0 \}
\]

is not an admissible operation; not even when there is a suitable witness \( z \) for each \( y \).

**Bounded Search II**

This can be pushed a little further: the search does not have to end at \( x \). Instead, we can search up to a primitive recursive function of \( x \) and \( y \).

\[
f(x, y) = \begin{cases} 
    \min \{ z < h(x, y) \mid g(z, y) = 0 \} & \text{if } z \text{ exists}, \\
    h(x, y) & \text{otherwise}.
\end{cases}
\]
**Example: Primality**

Claim (1)

The divisibility relation $\text{div}(x,y)$ is primitive recursive.

Note that

$$\text{div}(x,y) \iff \exists z \leq y (x \times z = y)$$

so that bounded search intuitively should suffice to obtain divisibility.

Formally, we have already seen that the characteristic function $M(z,x,y)$ of $x \times z = y$ is p.r. But then

$$\text{sign} \left( \sum_{z \leq y} M(z,x,y) \right)$$

is the p.r. characteristic function of $\text{div}$.

---

**Yet More Logic**

Arguments like the ones for basic number theory suggest another type of closure properties, with a more logical flavor.

**Definition (Bounded Quantifiers)**

$$P_\forall(x,y) \iff \forall z < x P(z,y) \quad \text{and} \quad P_\exists(x,y) \iff \exists z < x P(z,y).$$

Note that $P_\forall(0,y) = \text{true}$ and $P_\exists(0,y) = \text{false}$.

Informally,

$$P_\forall(x,y) \iff P(0,y) \land P(1,y) \land \ldots \land P(x-1,y)$$

and likewise for $P_\exists$.

---

**Next Prime**

Claim (3)

The next prime function $f(x) = \min \{ z > x \mid z \text{ prime} \}$ is p.r.

This follows from the fact that we can bound the search for the next prime by a p.r. function:

$$f(x) \leq 2x \quad \text{for } x \geq 1.$$  

This bounding argument requires a little number theory (a white lie).

---

**Enumerating Primes**

Claim (4)

The function $n \mapsto p_n$, where $p_n$ is the $n$th prime, is primitive recursive.

To see this we can iterate the “next prime” function from the last claim:

$$p(0) = 2$$

$$p(n+1) = f(p(n))$$

---

**Bounded Quantification**

Bounded quantification is really just a special case of bounded search: for $P_\exists(x,y)$ we search for a witness $z < x$ such that $P(z,y)$ holds. Generalizes to $\exists z < h(x,y) P(z,y)$ and $\forall z < h(x,y) P(z,y)$.

**Proposition**

**Primitive recursive relations are closed under bounded quantification.**

**Proof.**

$$\text{char}_{P_\forall}(x,y) = \prod_{z < x} \text{char}_{P}(z,y)$$

$$\text{char}_{P_\exists}(x,y) = \text{sign} \left( \sum_{z < x} \text{char}_{P}(z,y) \right)$$

---

**Primality**

Claim (2)

The primality relation is primitive recursive.

To see why, note that $x$ is prime iff

$$1 < x \land \forall z < x (\text{div}(z,x) \Rightarrow z = 1).$$

The building blocks $1 < x$, div and $z = 1$ are all p.r., and we can combine things by $\land$ and $\Rightarrow$. The only potential problem is the (bounded) universal quantifier.

But this is quite similar to the situation with div from the last slide.

Time for a general solution.
Exercises

Exercise
Give a proof that primitive recursive functions are closed under definition by multiple cases.

Exercise
Show in detail that the function \( n \mapsto p_n \) where \( p_n \) is the \( n \)th prime is primitive recursive. How large is the p.r. expression defining the function?

Faking Data structures

Our primitive recursive programming language has one glaring defect: it only supports one data type, \( \mathbb{N} \). There are no lists, trees, graphs, hash tables and so on, only natural numbers.

As it turns out, all these discrete structures can be obtained from just integers if we are able to express sequences \( a_0, a_1, \ldots, a_{n-1} \) of numbers as a single number \( \langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle \).

This is obviously not meant as a practical programming idea, it is purely conceptual: natural numbers already suffice in principle, and the ability to compute with them means that other computation involving, say, list, are also possible.

Coding

Write \( \mathbb{N}^* \) for the set of all finite sequences of natural numbers and nil for the empty sequence.

We would like to express a sequence \( a_0, a_1, \ldots, a_{n-1} \in \mathbb{N}^* \) as a single number \( \langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle \). So we need a coding function, a polyadic map of the form

\[
(\cdot): \mathbb{N}^* \to \mathbb{N}
\]

that allows us to decode: from \( b = \langle a_0, a_1, \ldots, a_{n-1} \rangle \) we can recover \( n \) as well as all the \( a_i \).

Note that the coding function must necessarily be injective. Moreover, both the coding and decoding operations should be computationally cheap, at least primitive recursive.

Decoding

Suppose \( b = \langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle \) is some code number. Note that we have used 0-indexing to simplify notation below.

We want a unary length function \( \text{len} : \mathbb{N} \to \mathbb{N} \) that determines the length of the coded sequence

\( \text{len}(b) = n \)

and a binary decoding function \( \text{dec} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \) that extracts the components:

\( \text{dec}(b, i) = a_i \)

for all \( i = 0, \ldots, n-1 \). Traditionally, \( \text{dec}(b, i) \) is written \( (b)_i \).
### In Binary

The first step is to select a **pairing function**, an injective map
\( \pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \).

Equivalently, we are looking for 3 functions \( \pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \), \( \pi_i : \mathbb{N} \rightarrow \mathbb{N} \),
\( i = 1, 2 \), such that
\[
\pi_i(\pi(x_1, x_2)) = x_i
\]

There are many possibilities, the following choice arguably yields one of
the most intuitive coding functions.

\[
\pi(x, y) = 2^x(2y + 1)
\]

For example
\[
\pi(5, 27) = 32 \cdot 55 = 1760 = 11011100000_2
\]

### Sequence Numbers

The numbers of the form \( \langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle \) that appear as codes of
sequences are called **sequence numbers**.

Note that a priori \( \text{len}(x) \) need not be defined when \( x \) is not a sequence number. The same is true for \( \text{dec}(x, i) \), plus \( i \) may be too large to make sense. Still, one usually insists that both decoding functions are total and return some default value like 0 for meaningless arguments.

**Exercise**

Show how to check if a number is a sequence number given \( \text{dec} \) and \( \text{len} \).

### Pairs

The first step is to select a **pairing function**, an injective map
\( \pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \).

Equivalently, we are looking for 3 functions \( \pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \), \( \pi_i : \mathbb{N} \rightarrow \mathbb{N} \),
\( i = 1, 2 \), such that
\[
\pi_i(\pi(x_1, x_2)) = x_i
\]

There are many possibilities, the following choice arguably yields one of
the most intuitive coding functions.

\[
\pi(x, y) = 2^x(2y + 1)
\]

For example
\[
\pi(5, 27) = 32 \cdot 55 = 1760 = 11011100000_2
\]

### Aside: Fueter-Pólya

Another popular pairing function is the quadratic polynomial due to
Cantor:
\[
p(x, y) = ((x + y)^2 + 3x + y)/2
\]

Note that this function is a bijection (unlike our exponential pairing
function which misses 0).

A surprising theorem by Fueter and Pólya from 1923 states that, up to a
swap of variables, this is the only quadratic polynomial that defines a
bijection \( \mathbb{N}^2 \leftrightarrow \mathbb{N} \).

The proof is rather difficult and uses the fact that \( e^a \) is transcendental
for algebraic \( a \neq 0 \).

It is an open problem whether there are other bijections for higher degree
polynomials. Extra Credit.

### Extending to Sequences

Here are some sequence numbers for this particular coding function:
\[
\langle \text{nil} \rangle := 0
\]
\[
\langle a_0, \ldots, a_{n-1} \rangle := \pi(\langle a_0 \rangle, \langle a_1, \ldots, a_{n-1} \rangle)
\]

\[
\langle 10 \rangle = 1024
\]
\[
\langle 0, 0, 0 \rangle = 7
\]
\[
\langle 1, 2, 3, 4, 5 \rangle = 532754
\]
It's a Bijection 55

Lemma
⟨ , ⟩ : N∗ → N is a bijection.

Proof. Suppose
⟨ a0, . . . , an−1 ⟩ = ⟨ b0, . . . , bm−1 ⟩
We may safely assume 0 < n ≤ m (why?). Since π is a pairing function, we get a0 = b0 and
⟨ a1, . . . , an−1 ⟩ = ⟨ b1, . . . , bm−1 ⟩ .
By induction, ai = bi for all i = 1, . . . , n − 1 and
0 = ⟨ nil ⟩ = ⟨ bn, . . . , bm−1 ⟩ . Hence n = m and our map is injective.

Exercise
Prove that the function is surjective.

Less formally . . . 56

Here is a sequence number and its binary expansion:
⟨ 2, 3, 5, 1 ⟩ = 20548
= 1 0 0 0 0 1 0 0 0 1 0 0

So the number of 1's (the digitsum) is just the length of the sequence, and the spacing between the 1's indicates the actual numerical values.

It follows that the coding function is injective and surjective, right?

That's It! 57

Here is a sequence number and its binary expansion:
⟨ 2,3,5,1 ⟩ = 20548
= 1 0 0 0 0 1 0 0 0 1 0 0

We can now code any discrete structure as an integer by expressing it as
a nested list of natural numbers, and then applying the coding function.

Exercise
Show that the pairing function π and both unpairing functions
x = π1(π(x, y)) and y = π2(π(x, y)) are primitive recursive.

Exercise
Show that the length and decoding functions len and dec are primitive recursive.

Exercise
Show that the coding function ⟨ ⟩ is primitive recursive when restricted
to inputs of fixed length.

Everything is PR 58

Exercise
Show that the value at x depends directly on all the
previous values of f. Now consider the following function associated with f:

\[ \mathcal{F}(x, y) := \langle f(0, y), f(1, y), \ldots, f(x, y) \rangle \]

Lemma
f is primitive recursive iff \( \mathcal{F} \) is primitive recursive.

Course of Value Recursion 60

Thus, it is natural to generalize the primitive recursion scheme slightly by
defining functions so that the value at x depends directly on all the
previous values.

\[ f(0, y) = g(y) \]
\[ f(x^+, y) = H(x, \mathcal{F}(x, y), y) \]

Lemma
If g and H are primitive recursive then f is also primitive recursive.

Exercise
Prove the last two lemmata. You may safely assume that standard
sequence operations such as append are primitive recursive.
*Digression: Gödel’s β Function

There is more elegant way to code sequence numbers due to Gödel that he used in his famous incompleteness theorem.

For the sake of completeness, here is a brief description of Gödel’s method.

---

Gödel’s Trick

To deal with sequences of arbitrary length one can use a clever divisibility argument.

Lemma

Gödel

There exists a primitive recursive function $\text{dec} : \mathbb{N}^2 \rightarrow \mathbb{N}$ such that

$$\forall a_0, \ldots, a_{n-1} \exists a \forall i < n \ (a_i = \text{dec}(a, i)).$$

So $a$ is a potential code number for $a_0, \ldots, a_{n-1}$

Proof. Set

$$\text{dec}(a, i) = \min \{ x < a \ | \ ((\pi(x, i) + 1)C + 1) \text{ divides } \pi_1(a) \}$$

The idea is that the factors of $\pi_1(a)$ contain information about the $a_i$.

We need to establish the existence of the witness $a$.

---

Algorithms in the RealWorld

We claim that any algorithm you will ever see, outside of a class dealing directly with logic and computability, is always primitive recursive. And, in fact, trivially so.

There are two parts to this claim:

- All these algorithms operate on finitary data structures that can be coded naturally as sequence numbers, and
- given this natural coding, for input as well as output, the corresponding functions are always primitive recursive.

Of course, there is no actual theorem here, just an observation. I’d be most curious to hear about anything that might contradict this claim.

---

Sequence Numbers

Definition

Define a coding function $\langle \cdot \rangle$ by

$$\langle x \rangle = \min \{ a \ | \ \text{dec}(a, 0) = n \land \forall i \in [n] \ (\text{dec}(a, i) = a_i) \}$$

Also set $\text{lh}(a) = \text{dec}(a, 0)$ and $(a)_i := \text{dec}(a, i)$.

Again, $\langle \cdot \rangle$ is not primitive recursive, but we have:

- Seq $= \{ \langle x \rangle \ | \ x \in \mathbb{N}^* \} \subseteq \mathbb{N}$ is primitive recursive.
- The restriction to $\mathbb{N}^n$ is primitive recursive.
- dec is primitive recursive.

Exercise

Prove this claim in detail.

---

Sequence Operations

As always, having a data structure by itself is not particularly interesting, we need to be able to implement operations. In our case, one can show that the following operations on sequences are primitive recursive.

- head, tail
- concatenate
- reverse
- sort
- map
- sum, product

In fact, it would be quite difficult to come up with any example of an operation used in a real program that fails to be primitive recursive.
Exercise
Prove that all these functions are indeed primitive recursive.

Exercise
Explain how to implement search in binary search trees as a primitive recursive operation.

Exercise
Come up with yet another coding function based on repeated application of a pairing function (make sure your method really works).