Computability, Decidability, and Complexity

Lecture Notes #1: Computability

Prof.Dr. Ferucio Laurențiu Țiplea

“Al. I. Cuza” University of Iași
Department of Computer Science
Iasi 740083, Romania
E-mail: `fltiplea@mail.dntis.ro`
URL: `http://www.infoiasi.ro/~fltiplea`

Computability

1. Introduction to computability

2. Basic models of computation

    2.1. Recursive functions

    2.2. Turing machines

    2.3. Properties of recursive functions

3. Other models of computation

1. Introduction to Computability

- Questions and problems
- Computation (What-) problems and decision problems
- Problems and algorithms
- Hilbert's problems
- Hilbert's program
- Algorithms and functions
- The theory of computation

Questions and Problems

Questions:

- For what real number x is $2x^2 + 36x + 25 = 0$?
- What is the smallest prime number greater than 20?
- Is 257 a prime number?

A problem is a class of questions. Each question in the class is an instance of the problem. Examples of problems:

- Find the real roots of the equation $ax^2+bx+c = 0$, where a, b, and c are real numbers. If we substitute a, b, and c by real values, we get an instance of this problem;
- Does the equation $ax^2 + bx + c = 0$, where a, b, and c are real numbers, have positive (real) roots? If we substitute a, b, and c by real values, we get an instance of this problem.

Computation and Decision Problems

Most problems in mathematical sciences are of two kinds:

- computation (what-) problems - such a problem is to obtain the value of a function for a given argument.

  Example: What is the square root of a given positive integer x to nearest thousandth?

- decision problems (yes/no problems) - these are problems whose answer is yes or no.

  Example: Does a given vector satisfy a given set of linear equations?

Decision problems could be considered computation problems but it is worthwhile to make the distinction.

Problems and Algorithms

An algorithm for a problem is an organized set of commands for answering on demand any question that is an instance of the problem.

A problem is solvable if there exists an algorithm for it. Otherwise it is called unsolvable.

A solvable decision problem is also called decidable; an unsolvable decision problem is also called undecidable.

An algorithm for a decision problem is usually called a decision algorithm (procedure).

Hilbert's Problems

Hilbert's problems are a set of (originally) unsolved problems in mathematics proposed by Hilbert. Of the 23 total appearing in the printed address, 10 were actually presented at the Second International Congress in Paris on August 8, 1900. In particular, the problems presented by Hilbert were 1, 2, 6, 7, 8, 13, 16, 19, 21, and 22 (Derbyshire 2004, p. 377). Furthermore, the final list of 23 problems omitted one additional problem on proof theory (Thiele 2001).

Hilbert's problems were designed to serve as examples for the kinds of problems whose solutions would lead to the furthering of disciplines in mathematics. As such, some were areas for investigation and therefore not strictly "problems".

Hilbert's Problems (cont'd)

The tenth problem in Hilbert's list was that of finding a decision procedure to tell whether a diophantine equation has a solution. Such an equation is of the form

$$P(x_1, \ldots, x_n) = 0,$$

where P is a polynomial in the integer unknowns $x_1, \ldots, x_n$ with integer coefficients.

At the time Hilbert posed this problem, mathematicians did not consider the possibility of proving it unsolvable. Nevertheless, when undecidability results began to appear in the 1930s and early 1940s, many researchers began to suspect that the problem was unsolvable. Much work has been done, but it was not until 1972 that the proof of the undecidability of Hilbert's tenth problem was completed by Y. Matiyacevic.

Hilbert's Program

In the early 1920s, David Hilbert put forward a new proposal for the foundation of classical mathematics which has come to be known as Hilbert's Program. It calls for a formalization of all of mathematics in axiomatic form, together with a proof that this axiomatization of mathematics is consistent. The consistency proof itself was to be carried out using only what Hilbert called "finitary" methods. Although Hilbert proposed his program in this form only in 1921, various facets of it are rooted in foundational work of his going back until around 1900.

In September 1930, Kurt Gödel announced his first incompleteness theorem at a conference in Königsberg, showing that Hilbert's program is not realizable.

Algoritms and Functions

Every algorithm computes a function:

$$\text{input } (x) \rightarrow \text{algorithm} \rightarrow \text{output } (y)$$
$$f(x) = y$$

A function computable by some algorithm is called a computable function.

The theory of computation is concerned with the following main question: What functions are computable? Equivalently, What problems are solvable?

The Theory of Computation

- The theory of computation originated in the 1930s, in the work of the logicians Church, Gödel, Kleene, Post, and Turing;
- The theory of computation provides models of computation together with their basic properties;
- The concept of computation is related to that of an algorithm;
- Computation: numerical or non-numerical.

2. Basic Models of Computation

2.1. Recursive functions

2.2. Turing machines

2.3. Properties of recursive functions

2.1. Recursive Functions

2.1.1. Notation and terminology

2.1.2. Primitive recursive functions

2.1.2.1. Definitions and examples

2.1.2.2. Algebraic properties

2.1.2.3. Primitive recursive enumerations

2.1.2.4. The Ackermann-Peter function

2.1.3. Recursive functions

2.1.3.1. Definitions and examples

2.1.3.2. The Ackermann-Peter function is recursive

2.1.3.3. Limits of Algorithmic Computability

2.1.3.4. Recursive Sets. Rice's Theorem

A Bit of History

Recursive functions emerge from logic, and so are very useful for formalizing algorithms of which we have intuitively natural descriptions.

A bit of history:

- Richard Dedekind, in 1888, and Giuseppe Peano, in 1889, were the first who used inductive definitions to define addition, multiplication, and exponentiation on natural numbers;
- Thoralf Skolem, in 1923, and Kurt Gödel, in 1931, developed the theory of recursive functions.

2.1.1. Notation and Terminology

- $f : A \rightsquigarrow B$ means that f is a partial function from A into B. That is, f might not be defined for some input values $a \in A$. We will write $f(a) = \perp$ or $f(a)\uparrow$ whenever f is not defined in a ($\perp$ means undefined);
- When a function $f : A \rightsquigarrow B$ is defined for all $a \in A$, we say that f is a total function and also write $f : A \rightarrow B$;
- Let $g_i : A \rightsquigarrow B_i$, $1 \leq i \leq n$, and $h : B_1 \times \cdots \times B_n \rightsquigarrow C$. Then, applying function composition to the functions $h, g_1, \ldots, g_n$ yields the function $f : A \rightsquigarrow C$ given by

$$f(a) = h(g_1(a), \ldots, g_n(a)),$$

for all $a \in A$. We usually write $f = h \circ (g_1, \ldots, g_n)$.

f is undefined for all values a for which $g_i(a)\uparrow$ for some i, or $h(g_1(a), \ldots, g_n(a))\uparrow$.

2.1.1. Notation and Terminology

The theory of recursive functions is primarily concerned with arithmetic functions, i.e., functions from $\mathbf{N}^n$ into $\mathbf{N}$, for some $n \geq 1$.

We adopt the following notation:

- $AF^{(n)} = \{f | f : \mathbf{N}^n \rightsquigarrow \mathbf{N}\}$, $n \geq 1$;
- $AF = \bigcup_{n\geq 1} AF^{(n)}$;
- $TAF^{(n)} = \{f | f : \mathbf{N}^n \rightarrow \mathbf{N}\}$, $n \geq 1$;
- $TAF = \bigcup_{n\geq 1} TAF^{(n)}$;
- $S : \mathbf{N} \rightarrow \mathbf{N}$, $S(x) = x + 1$ for all $x \in \mathbf{N}$ (the successor function).

2.1.2. Primitive Recursive Functions

2.1.2.1. Definitions and examples

2.1.2.2. Algebraic properties

2.1.2.3. Primitive recursive enumerations

2.1.2.4. The Ackermann-Peter function

2.1.2.1. Definitions and Examples

Let $g \in AF^{(n)}$ and $h \in AF^{(n+2)}$, where $n \geq 0$. We say that a function $f \in AF^{(n+1)}$ is obtained from g and h by primitive recursion if the following hold true:

- $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$;
- $f(x_1, \ldots, x_n, y+1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$,

for all $x_1, \ldots, x_n, y \in \mathbf{N}$.

Remark: For $n = 0$ the primitive recursion scheme is:

- $f(0) = a$, where $a \in \mathbf{N}$;
- $f(y+1) = h(y, f(y))$,

for all $y \in \mathbf{N}$.

2.1.2.1. Definitions and Examples

Remark: If f is obtained by primitive recursion from the total functions g and h, then f ia a total function.

Proof By mathematical induction on y. □

2.1.2.1. Definitions and Examples

In order to define the class of primitive recursive functions we need some functions on which to get started. These will be:

- the successor function $S : \mathbf{N} \rightarrow \mathbf{N}$;
- the constant 0 function $C_0^{(1)} : \mathbf{N} \rightarrow \mathbf{N}$ given by

  $$C_0^{(1)}(x) = 0,$$

  for all $x \in \mathbf{N}$;
- the projection function $P_i^{(n)} : \mathbf{N}^n \rightarrow \mathbf{N}$ given by

  $$P_i^{(n)}(x_1, \ldots, x_n) = x_i,$$

  for all $x_1, \ldots, x_n \in \mathbf{N}$, $n \geq 1$, and $1 \leq i \leq n$.

The functions S, $C_0^{(1)}$ and $P_i^{(n)}$ are called initial functions.

2.1.2.1. Definitions and Examples

The class of primitive recursive functions is the smallest class PRF of arithmetic functions which satisfies:

- includes all the initial functions;
- it is closed under composition (i.e., $f \circ (g_1, \ldots, g_n) \in PRF$ whenever $f, g_1, \ldots, g_n \in PRF$ and the composition is defined);
- it is closed under primitive recursion (i.e., $f \in PRF$ whenever f is obtained from g and h by primitive recursion and $g, h \in PRF$).

Remark: PRF contains only total functions.

Proof By structural induction. □

2.1.2.1. Definitions and Examples

Let $m \in \mathbf{N}$ and $n \geq 1$. The constant m function $C_m^{(n)} : \mathbf{N}^n \to \mathbf{N}$ is defined by

$$C_m^{(n)}(x_1, \ldots, x_n) = m,$$

for all $x_1, \ldots, x_n \in \mathbf{N}$.

Proposition 1 $C_m^{(n)}$ is a primitive recursive function, for all $m, n \in \mathbf{N}$.

Proof Case 1. $m = 0$ and $n = 1$: from definitions;

Case 2. $m > 0$ and $n = 1$: $C_m^{(1)}(x) = \underbrace{S(\cdots S}_{m\ times}(C_0^{(1)}(x))\cdots)$

Case 3. $m > 0$ and $n > 1$: (prove it!) □

2.1.2.1. Definitions and Examples

Example 1 The following functions are primitive recursive:

1. the addition, multiplication, and exponentiation functions;
2. the factorial function;
3. the predecessor function $Pd : \mathbf{N} \rightarrow \mathbf{N}$ defined by

$$Pd(x) = \begin{cases} x-1, & \text{if } x > 0 \\ 0, & \text{otherwise,} \end{cases}$$

   for all $x \in \mathbf{N}$;

4. the arithmetic (recursive) difference function $\dot{-}$ defined by

$$x \dot{-} y = \begin{cases} x-y, & \text{if } x \geq y \\ 0, & \text{otherwise,} \end{cases}$$

   for all $x, y \in \mathbf{N}$;

5. the absolute difference function $|x-y|$;

2.1.2.1. Definitions and Examples

6. the sign functions sg and $\bar{sg}$ given by

$$sg(x) = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{otherwise} \end{cases} \qquad \bar{sg}(x) = \begin{cases} 0, & \text{if } x \neq 0 \\ 1, & \text{otherwise} \end{cases}$$

   for all $x \in \mathbf{N}$;

7. the comparison functions ls (less than), gr (greater than), eq (is equal to);

8. the function $[\sqrt{\ }]$;

9. the function E given by $E(x) = x \dot{-} [\sqrt{x}]^2$, for all $x \in \mathbf{N}$;

10. the maximum and minimum functions max and min given by:

$$max(x,y) = \begin{cases} x, & \text{if } x \geq y \\ y, & \text{otherwise} \end{cases} \qquad min(x,y) = \begin{cases} x, & \text{if } x \leq y \\ y, & \text{otherwise} \end{cases}$$

   for all $x, y \in \mathbf{N}$.

2.1.2.1. Definitions and Examples

Remark: As we can see, many of the functions normally studied in number theory, and approximations to real-valued functions, are primitive recursive.

Remark: sg, $\bar{sg}$, ls, gr, and eq are primitive recursive predicates.

We will dnote by PRP the class of primitive recursive predicates.

2.1.2.1. Definitions and Examples

Primitive recursive functions and programs:

- We want to compute $z = C_0^{(1)}(x)$. The corresponding program is:

```
input:    $x$;
output:   $z = C_0^{(1)}(x)$;
begin
  $z := 0$;
end.
```

- We want to compute $z = S(x)$. The corresponding program is:

```
input:    $x$;
output:   $z = S(x)$;
begin
  $z := x + 1$;
end.
```

2.1.2.1. Definitions and Examples

- We want to compute $z = P_i(n)(x_1, \ldots, x_n)$. The corresponding program is:

```
input:    $x = (x_1, \ldots, x_n)$;
output:   $z = P_i(n)(x_1, \ldots, x_n)$;
begin
  $z := x[i]$;
end.
```

- We want to compute $z = (h \circ (g_1, \ldots, g_n))(x)$. The corresponding program is:

```
input:    $x$;
output:   $z = (h \circ (g_1, \ldots, g_n))(x)$;
begin
  for $i := 1$ to $n$ do $y_i := g_i(x)$;
  $z := h(y_1, \ldots, y_n)$;
end.
```

2.1.2.1. Definitions and Examples

- We want to compute $z = f(x, y)$, where f is obtained from g and h by primitive recursion. The corresponding program is:

```
input:    $g$, $h$, and $x, y \in \mathbf{N}$;
output:   $z = f(x, y)$;
begin
  $z := g(0)$;
  for $i := 1$ to $y$ do $z := h(x, i-1, z)$;
end.
```

2.1.2.2. Algebraic Properties

A few basic algebraic properties of primitive recursive functions are in order.

Proposition 2 If $f(x_1, \ldots, x_n)$ is primitive recursive and φ is a permutation of the set $\{1, \ldots, n\}$, then the function $g(x_1, \ldots, x_n)$ given by

$$g(x_1, \ldots, x_n) = f(x_{\varphi(1)}, \ldots, x_{\varphi(n)}),$$

for all $x_1, \ldots, x_n \in \mathbf{N}$, is primitive recursive.

Proposition 3 If $f(x_1, \ldots, x_n)$ is primitive recursive, then the function $g(x_1, \ldots, x_n, y_1, \ldots, y_m)$ given by

$$g(x_1, \ldots, x_n, y_1, \ldots, y_m) = f(x_1, \ldots, x_n),$$

for all $x_1, \ldots, x_n, y_1, \ldots, y_m \in \mathbf{N}$, is primitive recursive.

2.1.2.2. Algebraic Properties

Proposition 4 If $f(x_1, \ldots, x_n)$ is primitive recursive and $i_1, \ldots, i_k \in \{1, \ldots, n\}$, then the function $g(x_1, \ldots, x_n)$ given by

$$g(x_1, \ldots, x_n) = f(y_1, \ldots, y_n),$$

where

$$y_i = \begin{cases} x_{i_1}, & \text{if } i \in \{i_1, \ldots, i_k\} \\ x_i, & \text{otherwise} \end{cases}$$

for all i and $x_1, \ldots, x_n \in \mathbf{N}$, is primitive recursive.

We say that the function $f \in AF^{(n)}$ is 0 almost everywhere if $f(x_1, \ldots, x_n) \neq 0$, for finitely many inputs $(x_1, \ldots, x_n) \in \mathbf{N}^n$.

Proposition 5 Every arithmetic function which is 0 almost everywhere, is primitive recursive.

2.1.2.2. Algebraic Properties

A function $f \in AF^{(n+1)}$ is obtained from $g \in AF^{(n+1)}$ by bounded summation if

$$f(x_1, \ldots, x_n, y) = \sum_{i=0}^{y} g(x_1, \ldots, x_n, i),$$

for all $x_1, \ldots, x_n, y \in \mathbf{N}$.

Proposition 6 PRF is closed under bounded summation.

2.1.2.2. Algebraic Properties

A function $f \in AF^{(n)}$ is obtained from $g \in AF^{(n+1)}$ and $u, v \in AF^{(n)}$ by primitive recursive summation if

$$f(x_1, \ldots, x_n) = \sum_{i=u(x_1,\ldots,x_n)}^{v(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i),$$

for all $x_1, \ldots, x_n \in \mathbf{N}$.

Proposition 7 PRF is closed under primitive recursive summation.

2.1.2.2. Algebraic Properties

A function $f \in AF^{(n+1)}$ is obtained from $g \in AF^{(n+1)}$ by bounded product if

$$f(x_1, \ldots, x_n, y) = \prod_{i=0}^{y} g(x_1, \ldots, x_n, i),$$

for all $x_1, \ldots, x_n, y \in \mathbf{N}$.

Proposition 8 PRF is closed under bounded product.

2.1.2.2. Algebraic Properties

A function $f \in AF^{(n)}$ is obtained from $g \in AF^{(n+1)}$ and $u, v \in AF^{(n)}$ by primitive recursive product if

$$f(x_1, \ldots, x_n) = \prod_{i=u(x_1,\ldots,x_n)}^{v(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i),$$

for all $x_1, \ldots, x_n \in \mathbf{N}$.

Proposition 9 PRF is closed under primitive recursive product.

2.1.2.2. Algebraic Properties

A function $f \in AF^{(n+1)}$ is obtained from $g \in AF^{(n+1)}$ by bounded minimization if

- $f(x_1, \ldots, x_n, z) =$ the least $y \leq z$ for which $g(x_1, \ldots, x_n, y) = 0$, if such an y exists;
- $f(x_1, \ldots, x_n, z) = z + 1$, otherwise,

for all $x_1, \ldots, x_n, z \in \mathbf{N}$.

If f is obtained from g by bounded minimization then we write

$$f(x_1, \ldots, x_n, z) = \mu(y \leq z)[g(x_1, \ldots, x_n, y) = 0]$$

Proposition 10 PRF is closed under bounded minimization.

2.1.2.2. Algebraic Properties

A function $f \in AF^{(n)}$ is obtained from $g \in AF^{(n+1)}$ and $u \in AF^{(n)}$ by primitive recursive minimization if

- $f(x_1, \ldots, x_n) =$ the least $y \leq u(x_1, \ldots, x_n)$ for which $g(x_1, \ldots, x_n, y) = 0$, if such an y exists;
- $f(x_1, \ldots, x_n) = z + 1$, otherwise,

for all $x_1, \ldots, x_n \in \mathbf{N}$.

If f is obtained from g by primitive recursive minimization then we write

$$f(x_1, \ldots, x_n) = \mu(y \leq u(x_1, \ldots, x_n))[g(x_1, \ldots, x_n, y) = 0]$$

Proposition 11 PRF is closed under primitive recursive minimization.

2.1.2.2. Algebraic Properties

A predicate $f \in AP^{(n+1)}$ is obtained from $g \in AP^{(n+1)}$ by bounded existential quantification if

- $f(x_1, \ldots, x_n, y) = 1$, if there exists $0 \leq i \leq y$ such that $g(x_1, \ldots, x_n, i) = 1$;
- $f(x_1, \ldots, x_n, y) = 0$, otherwise,

for all $x_1, \ldots, x_n, y \in \mathbf{N}$.

If f is obtained from g by bounded existential quantification then we write

$$f(x_1, \ldots, x_n, y) = \exists_{i=0}^{y} g(x_1, \ldots, x_n, i)$$

Proposition 12 PRP is closed under bounded existential quantification.

2.1.2.2. Algebraic Properties

A predicate $f \in AP^{(n)}$ is obtained from $g \in AP^{(n+1)}$ and $u, v \in AF^{(n)}$ by primitive recursive existential quantification if

- $f(x_1, \ldots, x_n) = 1$, if there exists $u(x_1, \ldots, x_n) \leq i \leq v(x_1, \ldots, x_n)$ such that $g(x_1, \ldots, x_n, i) = 1$;
- $f(x_1, \ldots, x_n) = 0$, otherwise,

for all $x_1, \ldots, x_n \in \mathbf{N}$.

If f is obtained from g by bounded existential quantification then we write

$$f(x_1, \ldots, x_n) = \exists_{i=u(x_1,\ldots,x_n)}^{v(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i)$$

Proposition 13 PRP is closed under primitive recursive existential quantification.

2.1.2.2. Algebraic Properties

A predicate $f \in AP^{(n+1)}$ is obtained from $g \in AP^{(n+1)}$ by bounded universal quantification if

- $f(x_1, \ldots, x_n, y) = 1$, if $g(x_1, \ldots, x_n, i) = 1$ for all $0 \leq i \leq y$;
- $f(x_1, \ldots, x_n, y) = 0$, otherwise,

for all $x_1, \ldots, x_n \in \mathbf{N}$.

If f is obtained from g by bounded universal quantification then we write

$$f(x_1, \ldots, x_n, y) = \forall_{i=0}^{y} g(x_1, \ldots, x_n, i)$$

Proposition 14 PRP is closed under bounded universal quantification.

2.1.2.2. Algebraic Properties

A predicate $f \in AP^{(n)}$ is obtained from $g \in AP^{(n+1)}$ and $u, v \in AF^{(n)}$ by primitive recursive universal quantification if

- $f(x_1, \ldots, x_n) = 1$, if $g(x_1, \ldots, x_n, i) = 1$ for all $u(x_1, \ldots, x_n) \leq i \leq v(x_1, \ldots, x_n)$;
- $f(x_1, \ldots, x_n) = 0$, otherwise,

for all $x_1, \ldots, x_n, y \in \mathbf{N}$.

If f is obtained from g by primitive recursive universal quantification then we write

$$f(x_1, \ldots, x_n) = \forall_{i=u(x_1,\ldots,x_n)}^{v(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i)$$

Proposition 15 PRP is closed under primitive recursive universal quantification.

2.1.2.2. Algebraic Properties

Proposition 16 PRP is closed under $\neg$, $\vee$, and $\wedge$.

The properties established so far allow us to prove the primitive recursiveness of the following important functions:

Example 2 The following functions are primitive recursive:

1. (integer division) $quo : \mathbf{N}^2 \rightarrow \mathbf{N}$ given by
   - $quo(x, y) =$ the largest z such that $z \cdot y \leq x$, if $y \neq 0$;
   - $quo(x, y) = 0$, otherwise,

   for all $x, y \in \mathbf{N}$;

2.1.2.2. Algebraic Properties

2. (remainder) $rm : \mathbf{N}^2 \to \mathbf{N}$ given by

$$rm(x, y) = x \dot{-} y \cdot quo(x, y),$$

   for all $x, y \in \mathbf{N}$;

3. (division test) $div : \mathbf{N}^2 \to \mathbf{N}$ given by

$$div(x, y) = \begin{cases} 1, & \text{if } xy > 0 \text{ and } x|y \\ 0, & \text{otherwise,} \end{cases}$$

   for all $x, y \in \mathbf{N}$;

4. (number of divisors) $ndiv : \mathbf{N} \to \mathbf{N}$ given by

$$ndiv(x) = \text{number of divisors of } x,$$

   for all $x \in \mathbf{N}$;

2.1.2.2. Algebraic Properties

5. (primality test) $pr : \mathbf{N} \to \mathbf{N}$ given by

$$pr(x) = \begin{cases} 1, & \text{if } x \text{ is prime} \\ 0, & \text{otherwise,} \end{cases}$$

for all $x \in \mathbf{N}$;

6. (n^{th} prime number) $pn : \mathbf{N} \to \mathbf{N}$ given by

$$pn(n) = \text{the } n^{\text{th}} \text{ prime number,}$$

for all $n \in \mathbf{N}$ (2 is the 0^{th} prime number).

2.1.2.3. Primitive Recursive Enumerations

Primitive recursive enumerations are primitive recursive encodings of tuples of natural numbers. We will study:

- Cantor's enumeration – used to encode fixed length sequences of natural numbers;
- Gödel's numbering – used to encode arbitrary length sequences of natural numbers.

2.1.2.3. Primitive Recursive Enumerations

Cantor's enumeration

Cantor's enumeration, also called Cantor's diagonal method, is a clever technique used to show that the set $\mathbf{N}^2$ can be put into a bijective correspondence with $\mathbf{N}$.

The technique is quite general and applies in many circumstances.

2.1.2.3. Primitive Recursive Enumerations

2.1.2.3. Primitive Recursive Enumerations

Cantor's enumeration is the function $J : \mathbf{N}^2 \rightarrow \mathbf{N}$ given by:

$$J(x, y) = \frac{(x+y)(x+y+1)}{2} + x,$$

for all $(x, y) \in \mathbf{N}^2$.

Lemma 1 Cantor's enumeration is a primitive recursive bijection.

Proof

- J is bijective (prove it!);
- J is primitive recursive because it is a composition of primitive recursive functions.

$\square$

2.1.2.3. Primitive Recursive Enumerations

The inverse of J is the function $J^{-1} : \mathbf{N}\to\mathbf{N}^2$ given by:

$$J^{-1}(z) = \left(z \dot{-} \frac{k(k+1)}{2}, \frac{k(k+3)}{2} \dot{-} z \right),$$

where

$$k = \left\lfloor \frac{\lfloor\sqrt{8z+1}\rfloor \dot{-} 1}{2} \right\rfloor$$

for all $z \in \mathbf{N}$.

Let $K, L : \mathbf{N}\to\mathbf{N}$ be the functions given by

$$K(z) = z \dot{-} \frac{k(k+1)}{2} \quad \text{and} \quad L(z) = \frac{k(k+3)}{2} \dot{-} z,$$

for all $z \in \mathbf{N}$.

2.1.2.3. Primitive Recursive Enumerations

The functions J, K, and L satisfy:

$$J^{-1}(z) = (K(z), L(z)),$$

for all $z \in \mathbf{N}$.

Lemma 2 The functions K and L are primitive recursive.

Proof K and L are compositions of primitive recursive functions. $\square$

The functions K and L are called the inverses of J.

2.1.2.3. Primitive Recursive Enumerations

Cantor's enumeration can be generalized as follows:

- $J^{(2)} = J$;
- $J^{(n)}(x_1, \ldots, x_n) = J^{(2)}(J^{(n-1)}(x_1, \ldots, x_{n-1}), x_n)$,
  for all $n > 2$ and $x_1, \ldots, x_n \in \mathbf{N}$.

If we denote by $I_1^{(n)}, \ldots, I_n^{(n)}$ the inverses of $J^{(n)}$, then:

- $I_1^{(2)} = K$ and $I_2^{(2)} = L$;
- $I_1^{(n)} = I_1^{(n-1)} \circ I_1^{(2)}$,
  $\ldots$
  $I_{n-1}^{(n)} = I_{n-1}^{(n-1)} \circ I_1^{(2)}$,
  $I_n^{(n)} = I_2^{(2)}$,
  for all $n > 2$.

2.1.2.3. Primitive Recursive Enumerations

The functions $J^{(n)}$, $I_1^{(n)}, \ldots, I_n^{(n)}$ satisfy:

$$(J^{(n)})^{-1}(z) = (I_1^{(n)}(z), \ldots, I_n^{(n)}(z)),$$

for all $z \in \mathbf{N}$.

Lemma 3 The functions $J^{(n)}$, $I_1^{(n)}, \ldots, I_n^{(n)}$ are primitive recursive.

Proof $J^{(n)}$, $I_1^{(n)}, \ldots, I_n^{(n)}$ are compositions of primitive recursive functions. $\square$

2.1.2.3. Primitive Recursive Enumerations

Application: closure under simultaneous recursion

Two functions $f_1, f_2 \in AF^{(n+1)}$ are obtained from $g_1, g_2 \in AF^{(n)}$ and $h_1, h_2 \in AF^{(n+3)}$ by simultaneous recursion if the following hold true:

- $f_1(x_1, \ldots, x_n, 0) = g_1(x_1, \ldots, x_n)$,
- $f_2(x_1, \ldots, x_n, 0) = g_2(x_1, \ldots, x_n)$,
- $f_1(x_1, \ldots, x_n, y+1) = h_1(x_1, \ldots, x_n, y, f_1(x_1, \ldots, x_n, y), f_2(x_1, \ldots, x_n, y))$,
- $f_2(x_1, \ldots, x_n, y+1) = h_2(x_1, \ldots, x_n, y, f_1(x_1, \ldots, x_n, y), f_2(x_1, \ldots, x_n, y))$,

for all $x_1, \ldots, x_n, y \in \mathbf{N}$.

2.1.2.3. Primitive Recursive Enumerations

Proposition 17 PRF is closed under simultaneous recursion.

Proof Hint: Use Cantor's enumeration

- assume that f_1 and f_2 are functions obtained by simultaneous recursion over primitive recursive functions;
- define

$$t(x_1, \ldots, x_n, y) = J^{(2)}(f_1(x_1, \ldots, x_n, y), f_2(x_1, \ldots, x_n, y));$$

- show that t is primitive recursive;
- obtain $f_1 = K(t)$ and $f_2 = L(t)$ and conclude the lemma.

$\square$

2.1.2.3. Primitive Recursive Enumerations

Gödel's numbering

The Gödel number of a sequence of natural numbers $(x_0, \ldots, x_{k-1})$, denoted by $\langle x_0, \ldots, x_{k-1} \rangle$, is

$$pn(0)^{x_0} \cdots pn(k-1)^{x_{k-1}}$$

By convention, the Gödel number of the empty sequence, denoted by $\langle\rangle$, is 0.

Example 3

1. $\langle 3,1,2,1 \rangle = 2^3 \cdot 3^1 \cdot 5^2 \cdot 7^1 = 4200$;
2. $\langle 3,1,2,1,0 \rangle = \langle 3,1,2,1,0,0 \rangle = 4200$.

2.1.2.3. Primitive Recursive Enumerations

Remark:

1. Gödel numbers get quite large even when the terms of the sequences are small. For instance:

$$\langle 1,1,1,1,1,1,1,1,1\rangle = 223,092,870$$

2. We do not have to compute Gödel numbers !
3. Distinct sequences might have the same Gödel number (for instance, (1) and (1,0));
4. Every natural number is the Gödel number of some sequence (this is a consequence of the fundamental theorem of arithmetic). Therefore, the Gödel numbering induces a surjective function from the set of all sequences into the set $\mathbf{N}$.

2.1.2.3. Primitive Recursive Enumerations

The following two primitive recursive functions are important in order to deal with Gödel numbers:

Example 4 The following two functions are primitive recursive:

1. $exp : \mathbf{N}^2 \rightarrow \mathbf{N}$ given by

$$exp(i,z) = \begin{cases} \text{the greatest } x \text{ s.t. } pn(i)^x | z, & \text{if } z > 0 \\ 0, & \text{otherwise} \end{cases}$$

2. $long : \mathbf{N} \rightarrow \mathbf{N}$ given by

$$long(z) = \begin{cases} \text{the greatest } i \text{ s.t. } exp(i,z) > 0, & \text{if } z > 1 \\ 0, & \text{otherwise} \end{cases}$$

2.1.2.3. Primitive Recursive Enumerations

Example 5

1. $exp(i, 0) = 0$;
2. $exp(i, 1) = 0$ because $pn(i)^0 = 1|1$ and $pn(i)^1 \not| 1$;
3. $exp(0, 15) = 0$ because $pn(0) = 2 \not| 15$;
4. $long(0) = 0 = long(1)$;
5. $long(2) = 0$ because $2 = pn(0)^1$;
6. $long(15) = 2$ because $15 = pn(0)^0 \cdot pn(1)^1 \cdot pn(2)^1$;
7. if $z > 1$ then

$$z = \langle exp(0, z), \ldots, exp(long(z), z)\rangle$$

2.1.2.3. Primitive Recursive Enumerations

Application: closure under hereditary recursion

A functions $f \in AF^{(n+1)}$ is obtained from $g \in AF^{(n)}$ and $h \in AF^{(n+2)}$ by hereditary recursion if the following hold true:

- $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$,
- $f(x_1, \ldots, x_n, y+1) = h(x_1, \ldots, x_n, y, \langle f(x_1, \ldots, x_n, 0), \ldots, f(x_1, \ldots, x_n, y)\rangle)$,

for all $x_1, \ldots, x_n, y \in \mathbf{N}$.

2.1.2.3. Primitive Recursive Enumerations

Proposition 18 PRF is closed under hereditary recursion.

Proof Hint: Use simultaneous recursion

- assume that f is obtained by hereditary recursion over primitive recursive functions;
- define

$$t(x_1,\ldots,x_n,y) = \langle f(x_1,\ldots,x_n,0),\ldots,f(x_1,\ldots,x_n,y)\rangle;$$

- show that t and f can be defined by simultaneous recursion, and conclude the proposition.

$\square$

2.1.2.4. The Ackermann-Peter Function

The Ackermann-Peter function is a simple example of a total function which is computable but not primitive recursive, providing a counterexample to the belief in the early 1900s that every computable function was also primitive recursive.

History – In 1928, Wilhelm Ackermann, a mathematician studying the foundations of computation, originally considered the following sequence of operations:

2.1.2.4. The Ackermann-Peter Function

$$S(x) = x + 1 \qquad \Big|\ \varphi_0$$

$$\left\{\begin{array}{lcl} +(x,0) & = & x \\ +(x,y+1) & = & S(+(x,y)) \end{array}\right. \qquad \Big|\ \varphi_1$$

$$\left\{\begin{array}{lcl} \cdot(x,0) & = & 0 \\ \cdot(x,y+1) & = & +(x,\cdot(x,y)) \end{array}\right. \qquad \Big|\ \varphi_2$$

$$\left\{\begin{array}{lcl} x\,\hat{}\,0 & = & 1 \\ x\,\hat{}\,(y+1) & = & \cdot(x, x\,\hat{}\,y) \end{array}\right. \qquad \Big|\ \varphi_3$$

$$\left\{\begin{array}{lcl} \varphi_4(x,0) & = & 1 \\ \varphi_4(x,y+1) & = & \varphi_3(x,\varphi_4(x,y)) \end{array}\right.$$

2.1.2.4. The Ackermann-Peter Function

In general:

$$\begin{cases} \varphi_{i+1}(x,0) & = & 1 \\ \varphi_{i+1}(x,y+1) & = & \varphi_i(x,\varphi_{i+1}(x,y)) \end{cases}$$

for all $i \geq 3$ ($\varphi_0 = S$, $\varphi_1 = +$, $\varphi_2 = \cdot$, and $\varphi_3 = \hat{\ }$).

A few values of φ_4 are in order:

- $\varphi_4(x,1) = x$,
- $\varphi_4(x,2) = x^x$,
- $\varphi_4(x,3) = x^{x^x}$,
- and so on.

2.1.2.4. The Ackermann-Peter Function

In 1928, W. Ackermann considered the function

$$f(i,x,y) = \varphi_i(x,y),$$

for all $i, x, y \in \mathbf{N}$, and proved that:

- f is a recursive function;
- $h(x) = f(x,x,x)$ grows faster than any primitive recursive function. Therefore, it is not primitive recursive.

2.1.2.4. The Ackermann-Peter Function

In 1935, Rosza Peter, defined a similar function of only two variables. Nowadays, it is called the Ackermann-Peter function.

The Ackerman-Peter function is the function $A : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ given by:

$$\text{(R1)}\ \ A(0, y) = y + 1$$

$$\text{(R2)}\ \ A(x + 1, 0) = A(x, 1)$$

$$\text{(R3)}\ \ A(x + 1, y + 1) = A(x, A(x + 1, y))$$

for all $x, y \in \mathbf{N}$.

2.1.2.4. The Ackermann-Peter Function

Example of computation:

$$
\begin{aligned}
A(2,1) &\overset{(R3),\ x\to 1,\ y\to 0}{=} A(1,A(2,0))\\
& A(1,A(2,0))\\
&\overset{(R2),\ x\to 1}{=} A(1,A(1,1))\\
& A(1,A(1,1))\\
&\overset{(R3),\ x\to 0,\ y\to 0}{=} A(1,A(0,A(1,0)))\\
& A(1,A(0,A(1,0)))\\
&\overset{(R2),\ x\to 0}{=} A(1,A(0,A(0,1)))\\
& A(1,A(0,A(0,1)))\\
&\overset{(R1),\ y\to 1}{=} A(1,A(0,2))\\
& A(1,A(0,2))\\
&\overset{(R1),\ y\to 2}{=} A(1,3)\\
&\ldots
\end{aligned}
$$

2.1.2.4. The Ackermann-Peter Function

Conclusions:

- each element of this computation is a term
- each computation step is based on applying one of the equations (R1), (R2) or (R3)
- each equation is used in one direction only (“from left to right”)
- each equation is based on a substitution (“$x \to 1$, $y \to 0$”) which matches the left hand side of the equation to some subterm of the current term
- the immediate successor of a term t is obtained by replacing a subterm of t by an instance of the right hand side of some equation

2.1.2.4. The Ackermann-Peter Function

Do the equations $R1 - R3$ define a function?

Lemma 4 For any $x, y \in \mathbf{N}$ there exists an unique $z \in \mathbf{N}$ such that $z = A(x, y)$.

Proof by mathematical induction on x, followed by mathematical induction on y. $\square$

Therefore, the equations $R1 - R3$ do define a function.

2.1.2.4. The Ackermann-Peter Function

The Ackermann-Peter function can be calculated by a simple algorithm based directly on the definition:

```
Function $A(x, y)$
 begin
   if $x = 0$ then return $y + 1$
     else if $x > 0$ and $y = 0$ then return $A(x - 1, 1)$
       else return $A(x - 1, A(x, y - 1))$
 end.
```

At each step either y decreases, or y increases and x decreases. Each time that y reaches zero, x must decrease, so x must eventually reach zero as well. Note, however, that when x decreases there is no upper bound on how much y can increase – and it will often increase greatly.

2.1.2.4. The Ackermann-Peter Function

x/y	0	1	2	3	4	y
0	1	2	3	4	5	$y+1$
1	2	3	4	5	6	$y+2$
2	3	3	7	9	11	$2y+3$
3	5	13	29	61	125	$2^{y+3}-3$
4	13	65533	$2^{65533}-3$	$A(3, 2^{65533}-3)$	$A(3, A(4,3))$	$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{y+3\ twos} -3$

One surprising aspect of the Ackermann-Peter 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.

2.1.2.4. The Ackermann-Peter Function

Some astonishing facts about the Ackermann-Peter function:

1. $A(4,2)$ is greater than the number of particles in the universe raised to the power 200;
2. $A(5,2)$ cannot be written as a decimal expansion in the physical universe;
3. Beyond row 4 and column 1, the values can no longer be feasibly written with any standard notation other than the Ackermann-Peter function itself – writing them as decimal expansions, or even as references to rows with lower x, is not possible;
4. Despite the inconceivably large values occurring in this early section of the table, some even larger numbers have been defined, such as Graham's number. This number is constructed with a technique similar to applying the Ackermann-Peter function to itself recursively.

2.1.2.4. The Ackermann-Peter Function

The Ackermann-Peter function enjoy many interesting and important properties.

Proposition 19 The following properties hold true:

1. $A(x, y) > y$;
2. $A(x, y + 1) > A(x, y)$;
3. $A(x + 1, y) \geq A(x, y + 1)$;
4. $A(x, y) > x$;
5. $A(x + 1, y) > A(x, y)$;
6. $A(x + 2, y) > A(x, 2y)$,

for all $x, y \in \mathbf{N}$.

2.1.2.4. The Ackermann-Peter Function

The following theorem is crucial in order to prove that the Ackermann-Peter function is not primitive recursive.

Theorem 1 For any $n \geq 1$ and any primitive recursive function $f \in PRF^{(n)}$, there exists $k \in \mathbf{N}$, which depends on f and A, such that

$$f(x_1, \ldots, x_n) < A(k, ||(x_1, \ldots, x_n)||),$$

for all $(x_1, \ldots, x_n) \in \mathbf{N}^n$, where

$$||(x_1, \ldots, x_n)|| = max\{x_1, \ldots, x_n\}.$$

This theorem shows that the Ackermann-Peter function grows faster than any primitive recursive function.

2.1.2.4. The Ackermann-Peter Function

Now we are able to prove the main result regarding the Ackermann-Peter function.

Corollary 1 The Ackermann-Peter function is not primitive recursive. Moreover, the function $f(x) = A(x, x)$ is not primitive recursive too.

Proof If we assume that $A(x, y)$ is primitive recursive, then

$$f(x) = A(x, x)$$

is a primitive recursive function. By Theorem 1, there exists k such that $f(x) < A(k, x)$, for any x. Therefore,

$$A(x, x) < A(k, x)$$

for any x, which is a contradiction. □

2.1.2.4. The Ackermann-Peter Function

Since the function $f(x) = A(x,x)$ considered above grows very rapidly, its inverse function, f^{-1}, grows very slowly. In fact, it is less than 5 for any conceivable input size x, since $A(4,4)$ has a number of digits that cannot itself be written in binary in the physical universe. For all practical purposes, $f^{-1}(x)$ can be regarded as being a constant.

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.

2.1.2.4. The Ackermann-Peter Function

A two-parameter variation of the inverse Ackermann-Peter function can be defined as follows:

$$\alpha(x, y) = \min\{i \geq 1 | A(i, \lfloor x/y \rfloor) \geq \log_2 y\}$$

This function arises in more precise analysis of the algorithms mentioned above, and gives a more refined time bound. In the disjoint-set data structure, x represents the number of operations while y represents the number of elements; in the minimum spanning tree algorithm, x represents the number of edges while y represents the number of vertices.

2.1.2.4. The Ackermann-Peter Function

The Ackermann-Peter function, due to its definition in terms of extremely deep recursion, can be used as a benchmark of a compiler's ability to optimize recursion. For example, a compiler which, in analyzing the computation of $A(3, 30)$, is able to save intermediate values like the $A(3, y)$ and $A(2, y)$ in that calculation rather than recomputing them, can speed up computation of $A(3, 30)$ by a factor of hundreds of thousands. Also, if $A(2, y)$ is computed directly rather than as a recursive expansion of the form $A(1, A(1, A(1, ...A(1, 0)...)))$, this will save significant amounts of time.

Computing $A(1, y)$ takes linear time in y. Computing $A(2, y)$ requires quadratic time, since it expands to $\mathcal{O}(y)$ nested calls to $A(1, i)$ for various i. Computing $A(3, y)$ requires time proportionate to 4^{y+1}.

2.1.3. Recursive Functions

Recursive functions:

- Definitions and examples
- The Ackermann-Peter function is recursive
- Reducibility
- Recursively enumerable sets. Rice's theorem

2.1.3.1. Definitions and Examples

Let $f : \mathbf{N} \rightarrow \mathbf{N}$ be the function given by

$$f(x) = \begin{cases} \lfloor x/3 \rfloor, & \text{if } 3|x \\ \uparrow, & \text{otherwise,} \end{cases}$$

for all $x \in \mathbf{N}$.

f is not primitive recursive but it is effectively computable:

```
Function $f(x)$
  input:  $x \in \mathbf{N}$;
  output: "$x/3$", if $3|x$, and "undefined", otherwise;
  begin
    if $3|x$ then return $x/3$ else return "undefined"
  end.
```

2.1.3.1. Definitions and Examples

Remarks:

1. The function f, defined as above, is an effectively computable partial function, but it is not primitive recursive;
2. The Ackermann-Peter function is an effectively computable total function, but it is not primitive recursive.

Conclusion: The class of primitive recursive functions needs to be extended!

2.1.3.1. Definitions and Examples

The function f given above can be defined by

$$f(x) = \begin{cases} \text{the least } y \text{ s.t. } g(x,y)=0, & \text{if such an } y \text{ exists} \\ \uparrow, & \text{otherwise,} \end{cases}$$

for all $x \in \mathbf{N}$, where $g(x,y) = |x - 3y|$.

A function $f \in AF^{(n)}$ is obtained from $g \in AF^{(n+1)}$ by minimization if

- $f(x_1,\ldots,x_n) =$ the least y for which $g(x_1,\ldots,x_n,y) = 0$, if such an y exists;
- $f(x_1,\ldots,x_n) = \uparrow$, otherwise,

for all $x_1,\ldots,x_n \in \mathbf{N}$.

2.1.3.1. Definitions and Examples

If f is obtained from g by minimization then we write

$$f(x_1, \ldots, x_n) = \mu(y)[g(x_1, \ldots, x_n, y) = 0]$$

The class of recursive functions is the smallest class RF of arithmetic functions which satisfies:

- includes all the initial functions;
- it is closed under composition;
- it is closed under primitive recursion;
- it is closed under minimization.

Remark: RF includes PRF and contains partial functions too.

2.1.3.1. Definitions and Examples

Example 6

1. The function given by $f(x) = \mu(y)[|x - 3y| = 0]$, for all $x \in \mathbf{N}$, is a recursive function (but not primitive recursive);
2. The function given by

$$\Omega^{(n)}(x_1, \ldots, x_n) = \uparrow,$$

for all $x_1, \ldots, x_n \in \mathbf{N}$, is a recursive function (but not primitive recursive), for all $n \geq 1$. It is the totally undefined function.

```
Function $\Omega(x_1, \ldots, x_n)$
  input:  $x_1, \ldots, x_n \in \mathbf{N}$;
  output: "undefined";
  begin
    return "undefined"
  end.
```

2.1.3.1. Definitions and Examples

Remark: The class RF contains:

- recursive functions which are totally defined, called total recursive functions;
- recursive functions which are partially defined, called partial recursive functions.

2.1.3.2. The Ackermann-Peter Function is Recursive

Inner-most computation of the Ackermann-Peter function:

no.	configuration	code
0	$A(x,y)$	$c(x,y,0)$
$\dots$		
m	$A(x_1, A(x_2, \dots, A(x_k, y') \cdots))$	$c(x,y,m)$
$\dots$		
m_f	z	$c(x,y,m_f)$

Encoding the computation steps:

- $c(x,y,0) = 2^2 3^{x+1} 5^{y+1}$
- $c(x,y,m) = pn(0)^{k+1} pn(1)^{x_1+1} \dots pn(k)^{x_k+1} pn(k+1)^{y'+1}$
- $c(x,y,m_f) = 2^1 3^{z+1}$
- $c(x,y,k) = c(x,y,k-1)$, if $k > m_f$

2.1.3.2. The Ackermann-Peter Function is Recursive

Lemma 5 The function $c(x, y, m)$ is primitive recursive.

Proof (sketch)

Define the predicate G by:

$$G(\alpha) = 1 \quad \Leftrightarrow \quad \alpha \text{ is the code of some configuration}$$

G is primitive recursive:

- $G(\alpha) = eq(exp(0, \alpha), long(\alpha)) \cdot \prod_{i=0}^{long(\alpha)} gr(exp(i, \alpha), 0)$

2.1.3.2. The Ackermann-Peter Function is Recursive

Define the predicate $R_i(\alpha)$, for $i = 1, 2, 3$, by:

$$R_i(\alpha) = 1 \;\Leftrightarrow\; G(\alpha) = 1 \text{ and } (R_i) \text{ can be applied to the configuration encoded by } \alpha$$

R_i is primitive recursive, for all $i = 1, 2, 3$:

- $R_1(\alpha) = G(\alpha) \cdot eq(exp(long(\alpha) \dot{-} 1, \alpha), 1)$
- $R_2(\alpha) = G(\alpha) \cdot gr(exp(long(\alpha) \dot{-} 1, \alpha), 1) \cdot eq(exp(long(\alpha), \alpha), 1)$
- $R_3(\alpha) = G(\alpha) \cdot gr(exp(long(\alpha) \dot{-} 1, \alpha), 1) \cdot gr(exp(long(\alpha), \alpha), 1)$

Define the predicate R_4 by:

- $R_4(\alpha) = G(\alpha) \cdot \neg(R_1(\alpha) + R_2(\alpha) + R_3(\alpha))$

R_4 is primitive recursive

2.1.3.2. The Ackermann-Peter Function is Recursive

Define the function h_i, $i = 1, 2, 3, 4$, by

$$h_i(\alpha) = \text{the new code obtained by applying } R_i \text{ to } \alpha$$

The function h_i is primitive recursive, for all $i = 1, 2, 3, 4$:

- $h_1(\alpha) = R_1(\alpha) \cdot (\alpha / (pn(0) \cdot pn(long(\alpha) \dot{-} 1) \cdot pn(long(\alpha))^{exp(long(\alpha),\alpha)})) \cdot pn(long(\alpha) \dot{-} 1)^{exp(long(\alpha),\alpha)+1}$
- $h_2(\alpha) = R_2(\alpha) \cdot (\alpha / pn(long(\alpha) \dot{-} 1)) \cdot pn(long(\alpha))$
- $h_3(\alpha) = R_3(\alpha) \cdot (\alpha / (pn(long(\alpha) \dot{-} 1) \cdot pn(long(\alpha))^{exp(long(\alpha),\alpha)})) \cdot pn(long(\alpha))^{exp(long(\alpha) \dot{-} 1,\alpha)} \cdot pn(long(\alpha) + 1)^{exp(long(\alpha),\alpha) \dot{-} 1} \cdot pn(0)$
- $h_4(\alpha) = \alpha$

2.1.3.2. The Ackermann-Peter Function is Recursive

The function c can be written as:

- $c(x, y, 0) = 2^2 3^{x+1} 5^{y+1}$;
- $c(x, y, m+1) = \sum_{i=1}^{4} R_i(c(x, y, m)) \cdot h_i(c(x, y, m))$,

for all $x, y, m \in \mathbf{N}$.

Therefore, the function c is primitive recursive, and the proof is completed. $\square$.

2.1.3.2. The Ackermann-Peter Function is Recursive

Lemma 6 The function $t : \mathbf{N}^2 \to \mathbf{N}$ given by:

$$t(x,y) = \text{the number of the final step in the inner-most computation of } A(x,y),$$

for all $x, y \in \mathbf{N}$, is a total recursive function.

Proof

$$t(x,y) = \mu(m)[exp(0, c(x,y,m)) = 1]$$

□

2.1.3.2. The Ackermann-Peter Function is Recursive

Theorem 2 The Ackermann-Peter function is recursive.

Proof

$$A(x,y) = exp(1, c(x,y,t(x,y))) \dot{-} 1$$

□

Remark: The function $t(x,y)$ is not primitive recursive (otherwise, A would be primitive recursive).

2.1.3.2. The Ackermann-Peter Function is Recursive

Notation:

- AF – arithmetic functions;
- TAF – total arithmetic functions;
- PRF – primitive recursive functions;
- RF – recursive functions;
- TRF - total recursive functions.

Remark:

- AF is uncountable: $|AF| = 2^{\aleph_0}$
- PRF, TRF, and RF is countable:

$$|PRF| = |TRF| = |RF| = \aleph_0$$

2.1.3.2. The Ackermann-Peter Function is Recursive

Theorem 3 The following relationships between these classes of functions hold:

2.1.3.2. The Ackermann-Peter Function is Recursive

Proof The following inclusions hold:

- $PRF \subset TRF$: the Ackermann-Peter function is a total recursive function but not primitive recursive;
- $TRF \subset RF$: the totally undefined function is recursive but it is not a total function;
- $RF \subset AF$: $|RF| < |AF|$;
- RF and TAF are incomparable : RF contains partial functions, and $|TAF| > |RF|$.

$\square$

2.1.3.3. Limits of Algorithmic Computability

Church's Thesis: formulatd by the American logician Alonzo Church in 1935, states that the recursive functions are the only functions that can be mechanically calculated.

Originally, Alonzo Church (1932, 1936, 1941) and Stephen Kleene (1935) introduced lambda-definable functions, and Kurt Gödel (1934) and Jacques Herbrand (1932) introduced recursive functions. These two formalisms describe the same set of functions, as was shown in the case of functions of positive integers by Church (1936) and Kleene (1936).

2.1.3.3. Limits of Algorithmic Computability

Terminology:

- decision problem
  - solvable (computable)
    - decidable = total recursive function
    - semi-decidable = (partial) recursive function
  - unsolvable (undecidable) = non-recursive function
- computation problem
  - solvable (computable) = recursive function
  - unsolvable (non-computable) = non-recursive function

How does a problem (function) that is not computable (recursive) look like?

2.1.3.3. Limits of Algorithmic Computability

The class of recursive functions is countable. Therefore, we may enumerate all the n-ary recursive functions:

$$f_0^{(n)}, f_1^{(n)}, \ldots$$

(the arity will be omitted when it is clear from the context).

Assumption (Wagner-Strong axiom): There exists a recursive function $u^{(n+1)}$ such that

$$u^{(n+1)}(i, x_1, \ldots, x_n) = f_i^{(n)}(x_1, \ldots, x_n),$$

for all $i, x_1, \ldots, x_n \in \mathbf{N}$. u is called an universal function (later we will prove that such a function exists but, for the time being we accept it as an axiom).

An universal function compresses an infinite sequence of functions (as the (original) Ackermann function does).

2.1.3.3. Limits of Algorithmic Computability

Denote by $d : \mathbf{N}^4 \to \mathbf{N}$ the function given by:

$$d(x, y, u, v) = \quad \texttt{if } x = y \texttt{ then } u \texttt{ else } v,$$

for all $x, y, u, v \in \mathbf{N}$.

d is primitive recursive (prove it!)

Let $diag : \mathbf{N} \to \mathbf{N}$ be the predicate given by:

$$diag(x) = \begin{cases} 1, & f_x(x)\downarrow \\ 0, & \text{otherwise,} \end{cases}$$

for all $x \in \mathbf{N}$.

This predicate describes, in a functional way, a particular halting test.

2.1.3.3. Limits of Algorithmic Computability

Theorem 4 $diag$ is not recursive.

Proof (sketch)

- assume, by contradiction, that $diag$ is recursive;
- define $g(x) = u(d(diag(x), 0, i, j), x)$, for some i and j;
- g is recursive and, therefore, there exists k such that $g = f_k$;
- derive a contradiction.

□

2.1.3.3. Limits of Algorithmic Computability

Once some function f has been shown to be non-recursive, we can use that function to give other examples of non-recursive functions by way of the reducibility method.

A function f is recursively reducible (or r-reducible) to g if there exists a recursive function h such that $g \circ h = f$. The function h is called a reducibility function.

If f is recursively reducible to g by means of h then we will write $f \prec_h g$ or even $f \prec g$.

Remark: If $f \prec_h g$ and f is not recursive, then g is not recursive.

2.1.3.3. Limits of Algorithmic Computability

Let $halt^{(n+1)} : \mathbf{N}^{(n+1)} \to \mathbf{N}$ be the predicate given by:

$$halt^{(n+1)}(i, x_1, \ldots, x_n) = \begin{cases} 1, & f_i(x_1, \ldots, x_n)\downarrow \\ 0, & \text{otherwise,} \end{cases}$$

for all $i, x_1, \ldots, x_n \in \mathbf{N}$.

Corollary 2 $halt^{(2)}$ is not recursive.

Proof Hint: use reducibility. □

2.1.3.3. Limits of Algorithmic Computability

Assumption (Wagner-Strong axiom): For any $m \geq 0$ and $n > 0$ there exists a total recursive function $s_{m,n}^{(m+1)}$ such that

$$f_i^{(m+n)}(x_1,\ldots,x_m,y_1,\ldots,y_n) = f^{(n)}_{s_{m,n}^{(m+1)}(i,x_1,\ldots,x_m)}(y_1,\ldots,y_n),$$

for all $i,x_1,\ldots,x_m,y_1,\ldots,y_n \in \mathbf{N}$. $s_{m,n}$ is called an s-m-n function (later we will prove that such a function exists but, for the time being we accept it as an axiom).

An s-m-n function reduces by m the number of arguments of $(m+n)$-ary functions.

2.1.3.3. Limits of Algorithmic Computability

Let $t : \mathbf{N} \to \mathbf{N}$ be the predicate given by:

$$t(x) = \begin{cases} 1, & \text{if } f_x \text{ is total} \\ 0, & \text{otherwise,} \end{cases}$$

for all $x \in \mathbf{N}$.

Corollary 3 t is not recursive.

Proof Hint: use reducibility and the s-m-n property. □

This predicate describes, in a functional way, a totality test.

2.1.3.4. Recursive Sets. Rice's Theorem

An arithmetic predicate $f : \mathbf{N}^n \rightarrow \mathbf{N}$ can be regarded as a subset of $\mathbf{N}$:

$$A_f = \{x \in \mathbf{N}^n | f(x) = 1\}$$

The characteristic function of a subset $A \subseteq \mathbf{N}^n$ is the function $\chi_A : \mathbf{N}^n \rightarrow \mathbf{N}$ given by $\chi_A(x) = 1$ iff $x \in A$, for all $x \in \mathbf{N}^n$.

Given a predicate f, $f = \chi_{A_f}$.

2.1.3.4. Recursive Sets. Rice's Theorem

A set $A \subseteq \mathbf{N}^n$ is called primitiv recursive if its characteristic function χ_A is a <u>primitive recursive</u> function.

A set $A \subseteq \mathbf{N}^n$ is called recursive if its characteristic function χ_A is a <u>total recursive</u> function.

Proposition 20

(1) $\emptyset$ and $\mathbf{N}$ are primitive recursive sets.

(2) If $A, B \subseteq \mathbf{N}^n$ are (primitive) recursive sets, then $A \cup B$, $A \cap B$, and $A - B$ are (primitive) recursive sets too.

2.1.3.4. Recursive Sets. Rice's Theorem

Example 7 Not every set $A \subseteq \mathbf{N}^n$ is recursive.

1. The set $A = \{x \in \mathbf{N} | f_x(x)\downarrow\}$ is not recursive because its characteristic set is $\chi_A = diag$;
2. The set $B = \{x \in \mathbf{N} | f_x$ total$\}$ is not recursive because its characteristic set is $\chi_B = t$.

The sets in the example above are referring to the class of all unary recursive functions. There is a general result due to Rice (1953).

2.1.3.4. Recursive Sets. Rice's Theorem

Theorem 5 (Rice's Theorem)
Let $A \subseteq \{f_i^{(1)} | i \geq 0\}$ and $Ind(A) = \{i | f_i \in A\}$. Then, $Ind(A)$ is recursive iff $Ind(A) = \emptyset$ or $Ind(A) = \mathbf{N}$.

Proof $\Rightarrow$: (sketch)

- Assume that $Ind(A)$ is recursive but $Ind(A) \neq \emptyset$ and $Ind(A \neq \mathbf{N}$;
- reduce $diag$ to $\chi_{Ind(A)}$ and get a contradiction.

$\Leftarrow$: straightforward. □

Rice's theorem states that, no non-trivial property of unary recursive functions is decidable.

2.1.3.4. Recursive Sets. Rice's Theorem

Corollary 4 The following sets are not recursive:

1. $\{i \in \mathbf{N} | f_i \text{ total}\}$;
2. $\{i \in \mathbf{N} | dom(f_i) = \emptyset\}$;
3. $\{i \in \mathbf{N} | dom(f_i) \text{ infinite}\}$;
4. $\{i \in \mathbf{N} | f_i(x_0) = y_0\}$, where x_0 and y_0 are given;
5. $\{i \in \mathbf{N} | f_i = f_m\}$, where m is given;
6. $\{i \in \mathbf{N} | y_0 \in cod(f_i)\}$, where y_0 is given.

Proof Use Rice's Theorem. □

2.2. Turing Machines

2.2.1. Turing machines

2.2.2. Techniques for Turing machine construction

2.2.3. Recursive functions are Turing-computability

2.2.4. Turing-computable functions are recursive

2.2.5. The Church-Turing Thesis

2.2.1. Turing Machines

A formal model for an effective procedure should possess certain properties:

- it should be finitely describable;
- the procedure should consist of discrete steps, each of which can be carried out mechanically.

In 1936, Alan Turing proposed such a model. His model, called today Turing machine, has become the accepted formalization of an effective procedure.

2.2.1. Turing Machines

The basic model has a finite control, an input tape that is divided into cells, and a tape head that scans one cell of the tape at a time.

The tape has a leftmost cell but it is infinite to the right. Each cell may hold exactly one of a finite number of tape symbols. Initially, the n leftmost cells, for some $n \geq 0$, hold the input; the remaining infinity of cells each hold the blank, which is a special tape symbol that is not an input symbol. The finite control can be in one of a finite number of internal states.

2.2.1. Turing Machines

In one move, the machine, depending upon the symbol scanned by the tape head and the state of the finite control, does the followings:

- changes state,
- replaces the current symbol on the tape cell by a new symbol, and
- moves its tape head one cell to the left or to the right.

2.2.1. Turing Machines

A deterministic Turing machine, abbreviated DTM, is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- Q is a finite non-empty set of states;
- Σ is a finite non-empty set of input symbols, called the input alphabet of M;
- Γ is a finite non-empty set of tape symbols, called the tape alphabet of M. It is assumed that $\Sigma \subseteq \Gamma$;
- $\delta : Q \times \Gamma \rightsquigarrow Q \times \Gamma \times \{L, R\}$ is the transition relation/function of M;
- $q_0 \in Q$ is the initial/start state;
- $B \in \Gamma - \Sigma$ is the blank symbol;
- $F \subseteq Q$ is the set of final states.

2.2.1. Turing Machines

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a DTM.

- A configuration (instantaneous description) of M is any pair $(q, u|av) \in Q \times \Gamma^*\{|\}\Gamma^+$. Its meaning is:
  - q denotes the current state;
  - u consists of the symbols from the first cell up to, but not including, the symbol under the tape head;
  - the vertical bar "$|$" denotes the position of the tape head;
  - a is the symbol under the tape head;
  - v is either the empty string, if all symbols from the tape head, including a, are B, or the string of tape symbols for which all the other symbols on the rest of the tape are blank;

2.2.1. Turing Machines

- A configuration $(q, u|av)$ for which $q = q_0$, $u = \lambda$, and $v = \lambda$ whenever $a = B$, is called an initial configuration;
- $Config(M)$ stands for the set of all configurations of M;
- $C_0(w)$ stands for $(q_0, |w)$, if $w \neq \lambda$, and $(q_0, |B)$, otherwise.

Example 8 The following pairs are configurations of some Turing machine:

- $(q, aaBa|aBBBaab)$
- $(q, |BaBBab)$

2.2.1. Turing Machines

Let M be a DTM. Define the transition relation of M,

$$\vdash_M \subseteq Config(M) \times Config(M),$$

as follows:

$$(q, u|av) \vdash_M (q', u'|a'v')$$

iff one of the following two properties holds:

- (move/step to the right)
  $\delta(q,a) = (q', \bar{a}, R)$, $u' = u\bar{a}$, and

  $$\texttt{if } v = \lambda \texttt{ then } a' = B \textit{ and } v' = \lambda \texttt{ else } v = a'v';$$

- (move/step to the left)
  $\delta(q,a) = (q', \bar{a}, L)$, $u \neq \lambda$, $u = u'a'$, $v' = \bar{a}v$.

2.2.1. Turing Machines

Let M be a DTM.

- (computation step/move/transition)

$$(q, u|av) \vdash_M (q', u'|a'v')$$

- (computation)

$$(q_0, u_0|a_0v_0) \vdash_M \cdots \vdash_M (q_n, u_n|a_nv_n),$$

  where $(q_0, u_0|a_0v_0)$ is an initial configuration.

- a configuration C is called a blocking configuration if there is no configuration C' such that $C \vdash_M C'$;
- a configuration C' is called reachable from C if $C \overset{*}{\vdash}_M C'$;

2.2.1. Turing Machines

Remark: Let M be a DTM. Then:

1. In any configuration, the machine can perform at most one move (because the machine is deterministic);
2. Blocking configurations are not necessarily final configurations, and vice-versa;
3. A configuration $(q, u|av)$ is a blocking configuration if either $\delta(q,a)\uparrow$ or $u = \lambda$ and $\delta(q,a) = (q', \bar{a}, L)$.

2.2.1. Turing Machines

Let M be a DTM and w and input for M.

- M halts on w if there exists a blocking or final configuration C such that $(q_0, |w) \overset{*}{\vdash} C$. Two cases are to be considered:
  - M accepts w if C is a final configuration. Denote by $accept(M)$ the set of all w accepted by M;
  - M rejects w if C is a blocking configuration. Denote by $reject(M)$ the set of all w rejected by M;
- M loops on w if it does not halt on w.

$$accept(M) \cup reject(M) \cup loop(M) = \Sigma^*$$

$accept(M)$ is also called the language accepted by M, denoted $L(M)$.

2.2.1. Turing Machines

Example 9 The following Turing machine accepts

$$L = \{0^n 1^n | n \geq 1\}.$$

	0	1	X	Y	B
q_0	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	(q_4, B, R)
q_4					

q_4 is the only final state.

2.2.1. Turing Machines

Turing machines may be viewed as computers of arithmetic functions. The traditional approach is to represent integers in unary. The integer $x \geq 0$ is represented by 0^x.

$$(x_1, \ldots, x_k) \rightsquigarrow 0^{x_1}1 \cdots 10^{x_k}$$

Example 10

- $(1, 2, 1) \rightsquigarrow 010010$
- $(2, 0, 0, 3) \rightsquigarrow 00111000$

Remark:

- Three tape symbols are needed: 0, 1, and B;
- 0 and 1 may be replaced by any two distinct symbols a and b.

2.2.1. Turing Machines

Let M be a DTM and $f : \mathbf{N}^k \to \mathbf{N}$. M computes f if, for any $x_1, \ldots, x_k \in \mathbf{N}$, one of the following two properties holds true:

- if $f(x_1, \ldots, x_k)\downarrow$ then M halts on $(q_0, |0^{x_1}1 \cdots 10^{x_k})$ with a tape consisting of $0^{f(x_1,\ldots,x_k)}$;
- if $f(x_1, \ldots, x_k)\uparrow$ then M does not halt on $(q_0, |0^{x_1}1 \cdots 10^{x_k})$.

An arithmetic function is Turing-computable if there exists a Turing machine which computes it.

We denote by TCF the set of all Turing-computable functions.

2.2.1. Turing Machines

Remark:

- All natural numbers are Turing-computable;
- If the unary notation for natural numbers is adopted as being

$$x \rightsquigarrow 0^{x+1}$$

  then Turing machines with only two tape symbols are sufficient to compute arithmetic functions (prove it!).

2.2.1. Turing Machines

Example 11 The following Turing machine computes $x \dot{-} y$.

	0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6			

The machine has no final state.

2.2.2. Techniques for Turing Machine Construction

Storage in the finite control

The finite control can be used to hold a finite amount of information. To do so, the state is written as a pair of elements, one exercising control and the other one storing a symbol.

Example 12 Design a Turing machine that looks at the first input symbol, records it in its finite control, accepts if the first symbol does not appear elsewhere on its input, and rejects, otherwise.

2.2.2. Techniques for Turing Machine Construction

Shifting over

A Turing machine can make space on its tape by shifting blocks of symbols a finite number k of cells to the right. To do so,

- the machine uses its finite control to store k symbols;
- initially, the state is $(q, B, \ldots, B)$;
- if the current state is $(q, X_1, \ldots, X_k)$ and the symbol scanned on the tape is X, the machine prints X_1, and enters the state $(q, X_2, \ldots, X_k, X)$. This process is repeated until the block in question is exhausted.

If space is available, it can push blocks of symbols left in a similar manner.

2.2.2. Techniques for Turing Machine Construction

Remark:

- If k is large, the number of states could became very large. A different solution will be proposed after the "checking off symbols" technique is discussed;
- by this technique, blocks of symbols can be surrounded by two special symbols, \$ and #, for instance. The block in question is shifted one cell to the right by printing \$ first and # at the end.

2.2.2. Techniques for Turing Machine Construction

Multiple tracks

We can view the tape of the Turing machine as being divided into k tracks, for any $k \geq 2$. To do so, the tape symbols are considered k-tuples, one component for each track.

Example 13 Design a Turing machine that takes a binary input greater than 2 and determines whether it is a prime.

2.2.2. Techniques for Turing Machine Construction

Checking off symbols

Turing machines can check off symbols on their tapes. To do so, the tape is divided into 2 tracks, one holding the input and the other one being used to check symbols.

Example 14 Design a Turing machine that accepts the language

$$L = \{wcw | w \in \{a,b\}^*\},$$

where $c \notin \{a,b\}$.

2.2.2. Techniques for Turing Machine Construction

Using this technique, a Turing machine can shift blocks of symbols a finite number k of cells to the right, as follows:

- check (if it is necessarily) the first symbol of the block;
- repeatedly do
  - store and erase the rightmost symbol of the block;
  - move right k cells and print the symbol;

  until the checked symbol is encounter (and shift it too).

If k is large, this solution might be more convenient than the previous one.

2.2.2. Techniques for Turing Machine Construction

Subroutines

As with programs, Turing machine can be designed in a “to-down” manner, by using “subroutines”. The general idea is to write part of a Turing machine program to serve as a subroutine, with an initial state and several return states. The call of this subroutine is effected by entering in its initial state, and the return is effected by the move from a return state.

Example 15 Design a Turing machine which computes the multiplication function.

2.2.3. Recursive Functions are Turing-computable

Lemma 7 Initial functions are Turing-computable.

Proof

- $S(x) = x + 1$
  start off with 0^x on the tape, skip over 0's until the leftmost B is encountered, change B to 0, and halt;
- $C_m^{(n)}(x_1, \ldots, x_n) = m$
  start off with $0^{x_1}1 \cdots 10^{x_n}$ on the tape, change all 0s and 1s to B's and, on encountering B, print m 0s;
- $P_i^{(n)}(x_1, \ldots, x_i, \ldots, x_n) = x_i$
  start off with $0^{x_1}1 \cdots 10^{x_i}1 \cdots 10^{x_n}$ on the tape, change all 0s and 1s to B's until the $(i-1)$st 1 is encountered, skip over 0's until the leftmost 1 or B is encountered, and erase the rest of the tape.

□

2.2.3. Recursive Functions are Turing-computable

Lemma 8 Function composition of Turing-computable functions is Turing-computable.

Proof Let $g_1, \ldots, g_m \in AF^{(n)}$ and $f \in AF^{(m)}$ be Turing-computable functions. Assume that M_i computes g_i, $1 \leq i \leq m$, and M_0 computes f. Define M as follows:

- start off with $0^{x_1}1\cdots 10^{x_n}$;
- surround the input by \$ and #,

$$\$0^{x_1}1\cdots 10^{x_n}\#$$

- copy the input to the right of the first #,

$$\$0^{x_1}1\cdots 10^{x_n}\#0^{x_1}1\cdots 10^{x_n}$$

2.2.3. Recursive Functions are Turing-computable

- call M_1 on the second input's copy

$$\$0^{x_1}1\cdots10^{x_n}\#0^{g_1(x_1,\ldots,x_n)}$$

- repeat the cycle and call $M_2,\ldots,M_m$

$$\$0^{x_1}1\cdots10^{x_n}\#0^{g_1(x_1,\ldots,x_n)}\#\cdots\#0^{g_m(x_1,\ldots,x_n)}$$

- change all #'s, except for the first one, to 1's

$$\$0^{x_1}1\cdots10^{x_n}\#0^{g_1(x_1,\ldots,x_n)}1\cdots10^{g_m(x_1,\ldots,x_n)}$$

- call M_0

$$\$0^{x_1}1\cdots10^{x_n}\#0^{f(g_1(x_1,\ldots,x_n),\ldots,g_m(x_1,\ldots,x_n))}$$

- erase the block $\$0^{x_1}1\cdots10^{x_n}\#$ and halt.

$\square$

2.2.3. Recursive Functions are Turing-computable

Lemma 9 If f is obtained by primitive recursion over g and h, and g and h are total Turing-computable functions, then f is Turing-computable.

Proof Assume that M_1 computes g and M_2 computes h. For the sake of simplicity we will consider that f is a 2-ary function. Define M as follows:

- start off with 0^x10^y
- generates

$$\$0^x10^y\$0^x10^{y-1}\$\cdots\$0^x10\$0^x$$

- call M_1 on the last block and compute $f(x,0) = g(x)$

$$\$0^x10^y\$0^x10^{y-1}\$\cdots\$0^x10\$0^{f(x,0)}$$

2.2.3. Recursive Functions are Turing-computable

- replace the last block 0\$ by 1

$$\$0^x10^y\$0^x10^{y-1}\$\cdots\$0^x110^{f(x,0)}$$

- call M_2 on the last block and compute $f(x,1) = h(x,0,f(x,0))$

$$\$0^x10^y\$0^x10^{y-1}\$\cdots\$0^{f(x,1)}$$

- repeat the cycle $y-1$ times and get

$$\$0^{f(x,y)}$$

- erase the symbol \$ and halt.

□

2.2.3. Recursive Functions are Turing-computable

Lemma 10 If f is obtained by minimization over a total Turing-computable function g, then f is Turing-computable.

Proof For the sake of simplicity we assume that f is a unary function. Let M be a Turing machine which computes g. Define M' as follows:

1. start off with 0^x
2. generate $\$0^x1\$$
3. copy the first block after the second \$, $\$0^x1\$0^x1$, and call M on the second block in order to compute $g(x, 0)$
4. if $g(x, 0) = 0$ then halt; otherwise, increment the second argument, $\$0^x10\$$, and go to step 3.

□

2.2.3. Recursive Functions are Turing-computable

Theorem 6 Recursive functions are Turing-computable.

Proof The class of Turing-computable functions includes the initial functions and it is closed under function composition, primitive recursion, and minimization. □

2.2.4. Turing-computable Functions are Recursive

Let $M = (Q, \Sigma, \Gamma, \delta, q_1, B, F)$ be a Turing machine. If it is used to compute functions then we may assume that:

- $\Sigma = \{0, 1\}$;
- $F = \emptyset$;
- $Q = \{q_1, \ldots, q_n\}$, $n \geq 1$;
- $\Gamma = \{X_1, \ldots, X_m\}$, where $X_1 = 0$, $X_2 = 1$, $X_3 = B$, and $m \geq 3$.

Therefore, M halts only if a blocking configuration is reached.

2.2.4. Turing-computable Functions are Recursive

Configurations $(q, u|av)$ of M are encoded as follows:

- $(q, u|av) \rightsquigarrow (i, |u|, G(uav)) \rightsquigarrow J^{(3)}(i, |u|, G(uav)) \in \mathbf{N}$;
- $G(uav)$ is the Gödel number of the sequence uav, i.e.,

$$G(uav) = pn(0)^{i_1} \cdots pn(k-1)^{i_k}$$

  assuming $uav = X_{i_1} \cdots X_{i_k}$.

Example 16

- $(q_3, |010) \rightsquigarrow (3, 0, 2^1 \cdot 3^2 \cdot 5^1) \rightsquigarrow J^{(3)}(3, 0, 90) = 4959$
- $(q_2, 100|B01) \rightsquigarrow (2, 3, 2^2 \cdot 3^1 \cdot 5^1 \cdot 7^3 \cdot 11^1 \cdot 13^2)$
  $\rightsquigarrow J^{(3)}(2, 3, 38258220) = J^{(2)}(17, 38258220)$
  $= 12,441,388,261,154,451$

2.2.4. Turing-computable Functions are Recursive

Given a configuration C, denote by $JG(C)$ its code (defined as above).

Define $c_M : \mathbf{N}^2 \to \mathbf{N}$ by

$$c_M(x,n) = \begin{cases} JG(C_n), & \text{if there exists a computation } C_0(x), \ldots, C_n \\ JG(C_{n_0}), & \text{if there exists a computation } C_0(x), \ldots, C_{n_0},\ C_{n_0} \text{ is a halting configuration, and } n > n_0, \end{cases}$$

for all $x, n \in \mathbf{N}$.

Lemma 11 The function c_M is primitive recursive.

2.2.4. Turing-computable Functions are Recursive

Define $nr_M : \mathbf{N} \to \mathbf{N}$ by

$$nr_M(x) = \begin{cases} n, & \text{if there exists a computation} \\ & C_0(x), \ldots, C_n, \text{ and } C_n \text{ is a} \\ & \text{halting configuration} \\ \uparrow, & \text{otherwise,} \end{cases}$$

for all $x \in \mathbf{N}$.

Lemma 12 The function nr_M is recursive.

Proof $nr_M(x) = \mu(y)[c_M(x, y) = c_M(x, y + 1)]$. □

2.2.4. Turing-computable Functions are Recursive

Without loss of generality we may assume that any blocking configuration of M is of the form $(q, |0^m)$, for some $m \geq 0$ (assume $0^m = B$ whenever $m = 0$).

Define $decode : \mathbf{N} \rightarrow \mathbf{N}$ by

$$\begin{aligned} decode(z) &= (1 \dot{-} eq(I_3^{(3)}(z), pn(0)^3)) \\ &\quad (\mu(y \leq z)[I_3^{(3)}(z) = \textstyle\prod_{i=0}^{y-1} pn(i)]), \end{aligned}$$

for all $z \in \mathbf{N}$.

Lemma 13 $decode$ is primitive recursive.

2.2.4. Turing-computable Functions are Recursive

Lemma 14 The unary function $f : \mathbf{N} \rightarrow \mathbf{N}$ computed by a Turing machine M is given by

$$f(x) = decode(c_M(x, nr_M(x))),$$

for all $x \in \mathbf{N}$.

Theorem 7 Any Turing-computable function is recursive.

Corollary 5 $RF = TCF$.

2.3. Properties of Recursive Functions

2.3.1. Universal functions

2.3.2. The s-m-n theorem

2.3.3. Recursively enumerable sets

2.3.1. Universal Functions

Recall that, to compute functions, Turing machines can be considered in the form $M = (Q, \Sigma, \Gamma, \delta, q_1, B, F)$, where:

- $Q = \{q_1, \ldots, q_n\}$, $n \geq 1$;
- $\Sigma = \{0, 1\}$;
- $\Gamma = \{0, 1, B\}$;
- $F = \emptyset$.

Moreover,

- denote $0 = X_1$, $1 = X_2$, $B = X_3$, and
- denote $L = D_1$, $R = D_2$, and
- assume that for any state q there exists at least one transition $\delta(q_i, X_j) = (q_k, X_l, D_m)$ such that $q = q_i$ or $q = q_k$.

2.3.1. Universal Functions

Let $M = (Q, \Sigma, \Gamma, \delta, q_1, B, F)$ a Turing machine as given above. We encode M as follows:

- encode each transition $\delta(q_i, X_j) = (q_k, X_l, D_m)$ by
$$0^i 1 0^j 1 0^k 1 0^l 1 0^m$$
- assume that M has r transitions and their codes are $code_1$,. . . , $code_r$;
- encode M by:
$$111 code_1 11 code_2 11 \cdots 11 code_{r-1} 11 code_r 111$$

2.3.1. Universal Functions

Example 17 Let $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \emptyset)$, where:

- $\delta(q_1, 1) = (q_3, 0, R)$,
- $\delta(q_3, 0) = (q_1, 1, R)$,
- $\delta(q_3, 1) = (q_2, 0, R)$, and
- $\delta(q_3, B) = (q_3, 1, L)$.

The following binary sequence is a code of this machine:

111010010001010011000101010010011

000100100101001100010001000100101111

2.3.1. Universal Functions

Remark:

1. Except for the totally undefined Turing machine or Turing machines with exactly one transition, all the other Turing machines have more than one code, but finitely many.
2. If we order lexicographically the transitions of Turing machines, then each Turing machine has exactly one code. The lexicographic order is given by:
   - order first by states, $q_1 < q_2 < \cdots < q_n$,
   - order then by input symbols, $0 < 1 < B$.

   The unique code such obtained is denoted by $\langle M \rangle$.
3. Distinct Turing machines have distinct codes.

2.3.1. Universal Functions

- The code of a Turing machine is a binary sequence. If we order lexicographically these binary sequences, we get the standard enumeration of Turing machines

$$M_0, M_1, \ldots$$

- The standard enumeration of Turing machines induces a standard enumeration of all n-ary recursive functions:

$$f_0^{(n)}, f_1^{(n)}, \ldots$$

  where $f_i^{(n)}$ is the n-ary function computed by M_i;
- As there are infinitely many Turing machines computing the same n-ary function, each function $f_i^{(n)}$ appears infinitely many times in the standard enumeration of all n-ary recursive functions.

2.3.1. Universal Functions

Given M_i and $x_1, \ldots, x_n \in \mathbf{N}$, we denote by $\langle M, x_1, \ldots, x_n \rangle$ the binary string

$$\langle M_i \rangle 0^{x_1} 1 \cdots 1 0^{x_n}$$

Theorem 8 There exists a Turing machine M_u that, on input $\langle M_i, x_1, \ldots, x_n \rangle$, computes $f_i^{(n)}(x_1, \ldots, x_n)$, for all i and $x_1, \ldots, x_n$.

Proof For the sake of simplicity we will consider only the case $n = 1$.

We shall exhibit a 3-track Turing machine M_u:

- the first track holds the input $\langle M_i, x \rangle$;

2.3.1. Universal Functions

- the second track is used to simulate the tape of M_i;
- the third track holds the current state of M_i, with q_i represented by 0^i. Initially, it holds 0^1 (i.e., q_1).

The behavior of M_u is as follows:

- mark the first cell so the head can find its way back;
- M_u checks the input to see that $\langle M_i \rangle$ is the code of some Turing machine;
- copy 0^x on the second track and initialize track 3 to hold 0 (q_1);

2.3.1. Universal Functions

- let 0^i (q_i) be the current content of track 3 and X_j be the current symbol scanned on tracked 2.
  - M_u scans the first track (from left to right) looking for a substring which begins with 110^i10^j1;
  - If no such string is found, M_u halts (in a blocking configuration);
  - If such a code is found, let it be $0^i10^j10^k10^l10^m$. Then, it puts 0^k on track 3, replaces X_j by X_l on the second track, and moves in the direction pointed by D_m;
- When M halts, M_u halts.

□

2.3.1. Universal Functions

A Turing machine as the one in the proof above is called an universal Turing machine.

Corollary 6 There exist n-ary universal functions, for any $n \geq 2$.

Proof Modify an arbitrary universal Turing machine M_u as follows:

- start off with $0^i 1 0^{x_1} 1 \ldots 1 0^{x_n}$ on the tape;
- generate the code of M_i (use a new track if needed);
- call M_u on $\langle M_i, x_1, \ldots, x_n \rangle$.

The machine such obtained computes an universal function $u^{(n+1)}$, for all $n \geq 1$. □

2.3.2. The s-m-n Theorem

Theorem 9 (s-m-n theorem)
For any recursive function $f_i^{(m+n)}$ there exists a total recursive function $s_{m,n}^{(m+1)}$ such that:

$$f_i^{(m+n)}(x_1, \ldots, x_m, y_1, \ldots, y_n) = f^{(n)}_{s_{m,n}^{(m+1)}(i,x_1,\ldots,x_m)}(y_1, \ldots, y_n),$$

for all $x_1, \ldots, x_m, y_1, \ldots, y_n \in \mathbf{N}$.

Proof use Turing machines $\square$

2.3.3. Recursively Enumerable Sets

A set $A \subseteq \mathbf{N}$ is called recursively enumerable, abbreviated r.e., if $A = \emptyset$ or A is the range of an unary total recursive function.

Remark: If $A \subseteq \mathbf{N}$ is a non-empty r.e. set and f is an unary total recursive function such that $A = cod(f)$, then we will say that f enumerates A. Why?

- $0 \rightsquigarrow f(0) \in A$;
- $1 \rightsquigarrow f(1) \in A$;
- $\cdots$
- $A = \{f(x) | x \geq 0\}$.

2.3.3. Recursively Enumerable Sets

Remark: $A \subseteq \mathbf{N}$ is an r.e. set if $A = \emptyset$ or there exists a Turing machine M such that:

- M halts on all inputs $x \in \mathbf{N}$;
- for each input x, when M halts on x then the output $M(x)$ is in A;
- for any $a \in A$ there exists $x \in \mathbf{N}$ such that $a = M(x)$

(the machine M enumerates A).

We may say that:

> A is an r.e. set if $A = \emptyset$ or there exists an algorithm $\mathcal{A}$ that enumerates all the elements of A.

2.3.3. Recursively Enumerable Sets

Notation:

- PRS = the set of all primitive recursive sets $A \subseteq \mathbf{N}$;
- RS = the set of all recursive sets $A \subseteq \mathbf{N}$;
- RES = the set of all r.e. sets $A \subseteq \mathbf{N}$;

Proposition 21

$$PRS \subseteq RS \subseteq RES$$

2.3.3. Recursively Enumerable Sets

Theorem 10

1. A set is r.e. iff it is the range of an unary recursive function.
2. A set is r.e. iff it is the domain of an unary recursive function.

Remark: The above theorem can be used to show that certain sets are/are not r.e.:

- use (1.) to show that a given set is r.e.;
- use (2.) to show that a given set is not r.e.

2.3.3. Recursively Enumerable Sets

Example 18

1. The set $A = \{x | f_x(x)\downarrow\}$ is r.e.
2. The set $\mathbf{N} - A$, where A is the set from (1.), is not r.e.
3. The set $B = \{x | f_x \text{ is total}\}$ is not r.e.
4. The set $\mathbf{N} - B$, where B is the set from (3.), is not r.e.

Proposition 22 The set RES is closed under union and intersection but not under complementation.

From this proposition follows that $RS \subset RES$. Moreover,

$$PRS \subset RS \subset RES$$

2.3.3. Recursively Enumerable Sets

Theorem 11 A set A is recursive iff A and $\mathbf{N} - A$ are r.e.

Proof $\underline{\Rightarrow:}$ straightforward.

$\underline{\Leftarrow:}$ If $A = \emptyset$ or $\mathbf{N} - A = \emptyset$, then A is recursive.

Assume that both A and $\mathbf{N} - A$ are non-empty. Let g_1 and g_2 be two total recursive functions such that $A = cod(g_1)$ and $\mathbf{N} - A = cod(g_2)$.

Let $h(x) = \mu(y)[g_1(y) = x \ \vee \ g_2(y) = x]$. h is recursive and

$$\chi_A(x) = \begin{cases} 1, & g_1(h(x)) = x \\ 0, & \text{otherwise} \end{cases}$$

Therefore, A is reursive. $\square$

2.3.3. Recursively Enumerable Sets

Remark: Given a subset $A \subseteq \mathbf{N}$, define the language of A as being the set $L_A = \{0^x | x \in A\} \subseteq \{0,1\}^*$. Then:

$$\begin{array}{lll} A \subseteq \mathbf{N} \text{ is recursive} & \Leftrightarrow & \chi_A \text{ is total recursive} \\ & \Leftrightarrow & (\exists \text{ a Turing machine } M) \\ & & (M \text{ computes } \chi_A) \\ & \Leftrightarrow & (\exists \text{ a Turing machine } M') \\ & & (L_A = accept(M') \ \wedge \ \overline{L_A} = reject(M')) \end{array}$$

The second equivalence is based on:

- Given M which computes χ_A, design M' which accepts when M halts in a configuration $(q, |0)$, and rejects when M halts in a configuration $(q, |B)$.

2.3.3. Recursively Enumerable Sets

Remark:

$$\begin{aligned} A \subseteq \mathbf{N} \text{ is r.e.} \quad &\Leftrightarrow \quad A = dom(f) \text{ for some recursive function} \\ &\Leftrightarrow \quad (\exists \text{ a Turing machine } M) \\ & \quad (L_A = \{0^x | M \text{ halts on } 0^x\}) \\ &\Leftrightarrow \quad (\exists \text{ a Turing machine } M') \\ & \quad (L_A = accept(M')) \end{aligned}$$

The second equivalence is based on:

- Given M which computes f, design M' which accepts 0^x if and only if M halts on 0^x.

2.3.3. Recursively Enumerable Sets

Extension to languages

Let Σ be an alphabet and $L \subseteq \Sigma^*$.

We say that L is recursive if there exists a Turing machine M such that $L = accept(M)$ and $\overline{L} = reject(M)$.

We say that L is recursively enumerable, abbreviated r.e., if there exists a Turing machine M such that $L = accept(M)$.

Denote:

- $RS(\Sigma)$ = the set of all recursive sets $L \subseteq \Sigma^*$;
- $RES(\Sigma)$ = the set of all r.e. sets $L \subseteq \Sigma^*$;
- $\mathcal{L}_{rec}$ = the class of all recursive languages;
- $\mathcal{L}_{re}$ = the class of all r.e. languages.

2.3.3. Recursively Enumerable Sets

Remark: The main difference between recursive and r.e. sets:

- If L is recursive $\Rightarrow (\exists M)(L = acccept(M) \wedge \overline{L} = reject(M))$
  - given $w \in \Sigma^*$, if $w \in L$ then M accepts w and, therefore, we know that w is a member of L;
  - given $w \in \Sigma^*$, if $w \notin L$ then M rejects w and, therefore, we know that w is not a member of L;
- If L is r.e. $\Rightarrow (\exists M)(L = acccept(M))$
  - given $w \in \Sigma^*$, if $w \in L$ then M accepts w and, therefore, we know that w is a member of L;
  - given $w \in \Sigma^*$, if $w \notin L$ then M either rejects w or loops. In this case we might not know that w is not a member of L.

$\boxed{\text{Recursive sets = membership problem is decidable}}$

$\boxed{\text{r.e. sets = membership problem is semi-decidable}}$

2.3.3. Recursively Enumerable Sets

Theorem 12

1. $RS(\Sigma)$ is closed under union, intersection, and complementation;
2. $RES(\Sigma)$ is closed under union and intersection, but not under complementation;
3. $RS(\Sigma) \subseteq RES(\Sigma)$;
4. $L \subseteq \Sigma^*$ is recursive iff L and $\Sigma^* - L$ are r.e.
5. $\mathcal{L}_{rec} \subset \mathcal{L}_{re} = \mathcal{L}_0$ ($\mathcal{L}_0$ is the class of type 0 Chomsky languages).