# Computability Basics

Last Updated January 16th, 2024

## 1 Computational Problems

Informally, when we think of a problem, we think of a situation that needs to be resolved. Moreover, in computer science we think of a computing problem as not just one situation, but rather a collection of situations that share a common underlying theme. Each situation is referred to as a **problem instance**, and represents a concrete example of the general problem.

**Definition 1.1.** A **functional problem** is one for which, for each problem instance $x$, there is a unique solution to $x$. Letting $I$ denote the set of problem instances, and $S$ the set of possible solutions, we may think of a functional problem as a function $\text{sol} : I \to S$, such that, for each problem instance $x \in I$, $\text{sol}(x)$ equals its solution. Not surprisingly, we call function `sol` the **solution function** for the given problem.

**Definition 1.2.** A **decision problem** is a functional problem for which $S = \{0, 1\}$. This means that sol is a **predicate function** since its codomain is $\{0, 1\}$, and it is also called the **indicator function** for the given problem. In other words, we may think of each problem instance $x$ as either having or not having some property and $\text{sol}(x)$ indicates whether or not $x$ has the property. Finally, we call $x$ a **positive** (respectively, **negative**) instance iff $\text{sol}(x) = 1$ (respectively, $\text{sol}(x) = 0$).

**Definition 1.3.** A **discrete computational problem** is a functional problem such that, for both the set $I$ of instances and the set $S$ of solutions, each member of the set can be represented as a word (i.e. a sequence of characters) over some finite alphabet $\Sigma$. Thus, we may think of $I$ as a subset of words, i.e. a **language**, over alphabet $\Sigma$.

**Example 1.4.** Consider the problem called `Prime`, where a problem instance is a positive integer $n \geq 2$, and the solution we seek is an answer, `yes` or `no` (or 1 or 0), to the question of whether $n$ is a prime number, i.e. a number that is only divisible by 1 and itself. Then `Prime` is a decision problem as well as a discrete computational problem, since each integer may be written as a word over $\Sigma = \{0, 1, \ldots, 9\}$ while 0 and 1 are single-bit words over $\Sigma = \{0, 1\}$. □

**Example 1.5.** Consider the problem called `Sort`, where a problem instance is an array $a$ of integers, and the solution we seek is another array $b$ whose members are the same members of $a$, but in sorted order. `Sort` is certainly a functional problem since, for every integer array $a$, there is exactly one array, call it sort($a$), whose members are the same as those of $a$ and written in sorted order. Finally, `Sort` is a discrete computational problem since every integer array may be written as a word over the alphabet

$$\Sigma = \{0, 1, \ldots, 9, [, ], \text{“,”}\},$$

where the left and right bracket symbols are symbols of the alphabet, while the double quotes surrounding the comma are used as delimiters and are *not* part of the “comma” symbol. □

# 2 Models of Computation

Chances are you've already encountered several models of computation. Indeed, programming languages (such as Python, C, Java, Haskell, and PROLOG) are models of computation, as are digital circuits and CPU's. However, the study of theoretical computer science tends to benefit from models of computation that are as simple as possible. This is because theoretical computing problems often require reasoning about computations of arbitrary programs and so the computation of any program on some input should seem relatively easy to express. Thus, we desire a model of computation to be as simple as possoble, yet still meet the computing requirements that are assumed by the problem under investigation. The following are some approaches to achieving a minimalist-style computing model.

**Automata** Here, a computation is viewed as a sequence of state changes. A computation begins with an initial state and a machine has a finite-state controller that determines the next based on the current state and the current data that is being read. Examples: Finite Automata, Pushdown Automata, Turing Machines.

**Register Machines** These models are inspired by the architecture of a CPU where the machine consists of a finite number of registers along with the ability to perform basic logical and arithmetic operations on the words stored in each register. Examples: Random-Access Machines (RAM's), Unlimited Register Machines (URM's).

**Function Families** This approach views the function as the basis for computation and defines rules for constructing functions that are deemed "computable". To be in the function family means to be definable based on the provided rules of construction. Examples: Primitive and General Recursice Functions, Church's Lambda Calculus.

**Rewriting Systems** These models are similar to automata but with both data and state being combined into a single string of symbols. Examples: Markov Normal Algorithms, Post Production Systems.

**Concurrency** These models allow for multiple computation threads to simultaneously occur. Examples: Boolean and quantum Circuits, Petri Nets, Cellular Automata.

## 2.1 Uniform versus Non-Uniform Models of Computation

**Definition 2.1.** A computational problem is said to be $\mathcal{M}$**-computable** iff there is a model of computation $\mathcal{M}$ and some instance(s) of $\mathcal{M}$ that, for each instance $x \in I$, it accepts $x$ as input and, after a finite number of computation steps, outputs $\text{sol}(x)$.

**Definition 2.2.** A model of computation $\mathcal{M}$ is said to be **uniform** iff, when using the model to solve a given computational problem $\mathcal{P}$, a single instance of $\mathcal{M}$ is capable of solving *all* the instances of $\mathcal{P}$. Otherwise we say it is a **non-uniform** model.

Examples of uniform models include general-purpose programming languages, such as C and Python. An example of a non-uniform model is a Boolean circuit, since it can handle only a finite number of different inputs. But computing problems usually have an infinite number of instances, and so an infinite number of circuits are required to solve all problem instances.

This lecture introduces the URM register-machine model. URM's find use in computability theory because URM programs are readily encodable as a single integer. Such an encoding is called a **Gödel number** and is fundamental to both the study of computability and complexity theory. The URM is an example of what is called a **general model of computation**, meaning that it is a model that is capable of computing any process whose output is obtained in a deterministic step-by-step fashion with respect to one or more inputs being fed into the process. Most programming languages, such as C, Python, and Java, are also considered general models of computation.

Regardless of what computing model is being considered, in this lecture we assume that the purpose of a URM program is to compute a function that maps one or more nonnegative integers to a nonnegative integer. In other words, a problem instance is a vector of natural numbers, while the solution is also a natural number. By making this assumption, we do not lose any generality since any instance of any discrete computational problem domain can be encoded with one or more nonnegative integers. Moreover, the natural number output that is computed can then be decoded back to the original problem domain.

**Definition 2.3.** $\mathcal{N} = \{0, 1, 2, \ldots\}$ denote the set of nonnegative integers.

**Unary Function** $f : \mathcal{N} \to \mathcal{N}$ means that, for any input $x \in \mathcal{N}$, $f$ assigns $x$ to some value $f(x) \in \mathcal{N}$.

**Multivariate Function** For $m \geq 1$, $f : \mathcal{N}^m \to \mathcal{N}$ means that for any input vector $(x_1, \ldots, x_m) \in \mathcal{N}^m$, $f$ assigns it to some value $f(x_1, \ldots, x_m) \in \mathcal{N}$.

In computability theory it's important to allow for functions that may not be defined on all inputs.

**Definition 2.4.** A **partial function** is one that is undefined on zero or more of its inputs. A function that is defined on all of its inputs is said to be a **total** function. Note: all total functions are (technically speaking) partial since they are undefined on zero of their inputs.

**Example 2.5.** The function $f : \mathcal{N} \to \mathcal{N}$ defined by $f(n)$ equals the value $m$ for which $m^2 = n$ is only defined for $n = 1, 4, 9, 16, 25, \ldots$ and is undefined for all other values of $n$ that are not perfect squares.

# 3 The Unlimited Register Machine

The **Unlimited Register Machine (URM)** first introduced by Shepherdson and Sturgis (See Chapter 2 of Nigel Cutland's "Computability"). The purpose of a URM is to compute an $m$-ary function $f : \mathcal{N}^m \to \mathcal{N}$, from the set of $m$-tuples of nonnegative integers to nonnegative integers.

To begin, a **register** is a memory component that is capable of storing a nonnegative integer of arbitrary size. Registers form the basis of URM's. Indeed, a URM $M$ consists of

1. $r$ registers $R_1, \ldots, R_r$,
2. a finite program $P = I_1, \ldots, I_s$ consisting of $s$ instructions that are used for step-by-step manipulation of the registers, and
3. a **program counter**, denoted `pc`, that stores the index of the next program instruction to be executed.

A URM $M$ takes as input $m$ nonnegative integers $\vec{x} = x_1, \ldots, x_m$, performs a computation on this input, and outputs a nonnegative integer, denoted $M(\vec{x})$, that is ultimately stored in register 1.

**Definition 3.1.** A **machine configuration** for an $r$-register URM is an $(r+1)$-dimensional tuple whose first $r$ components equal the integers currently stored in registers $R_1, \ldots, R_r$, and whose final component, called the **program counter (`pc`)**, is the index of the next instruction.

**Initial Configuration** The **initial configuration** is

$$\sigma_0 = (x_1, \ldots, x_m, \underbrace{0, \ldots, 0}_{r-m}, 1),$$

where $(x_1, \ldots, x_m)$ is the URM input vector.

**Final Configuration** A **final configuration** is any configuration whose program counter exceeds $s = |P|$.

**Computation** A **computation** of $M$ on input $\vec{x}$ is a (possibly infinte) sequence of machine configurations $\sigma_0, \sigma_1, \ldots$ for which

1. $\sigma_0$ is the initial configuration
2. $\sigma_{k+1}$ is obtained from $\sigma_k$ by executing instruction $I_i$, where $i$ is the value of $\sigma_k$'s program counter `pc`, and updating the value of $M$'s registers accordingly.

We write $M(\vec{x}) \downarrow$ (respectively, $M(\vec{x}) \uparrow$) in case the computation of $M$ on input $\vec{x}$ is finite (respectively infinite).

## 3.1 URM Instruction Set

The following is a description of the different types of URM instructions, and how each affects the current machine configuration.

**Zero** $\mathrm{Z}(i)$, $1 \leq i \leq r$, assigns 0 to register $R_i$: $R_i \leftarrow 0$.

**Sum** $\mathrm{S}(i)$, $1 \leq i \leq r$, increments by 1 the value stored in $R_i$: $R_i \leftarrow R_i + 1$.

**Transfer** $\mathrm{T}(i, j)$, $1 \leq i, j \leq r$, assigns to $R_j$ the value stored in $R_i$: $R_j \leftarrow R_i$.

**Jump** $\mathrm{J}(i, j, k)$, $1 \leq i, j \leq r$, $1 \leq k \leq s$, has the effect of setting `pc` to $k$ in case $R_i$ and $R_j$ store the same integer. Otherwise `pc` is incremented by one.

**Example 3.2.** Consider a URM $M$ with $r = 3$ registers and the following program.

I1. $J(1, 2, 6)$

I2. $S(2)$

I3. $S(3)$

I4. $J(1, 2, 6)$

I5. $J(1, 1, 2)$

I6. $T(3, 1)$

The following is the sequence of configurations produced by the computation $M(9, 7)$.

| $\sigma_i$ | $R_1$ | $R_2$ | $R_3$ | **pc** | **Instruction** |
|---|---|---|---|---|---|
| 0 | 9 | 7 | 0 | 1 | J(1,2,6) |
| 1 | 9 | 7 | 0 | 2 | S(2) |
| 2 | 9 | 8 | 0 | 3 | S(3) |
| 3 | 9 | 8 | 1 | 4 | J(1,2,6) |
| 4 | 9 | 8 | 1 | 5 | J(1,1,2) |
| 5 | 9 | 8 | 1 | 2 | S(2) |
| 6 | 9 | 9 | 1 | 3 | S(3) |
| 7 | 9 | 9 | 2 | 4 | J(1,2,6) |
| 8 | 9 | 9 | 2 | 6 | T(3,1) |
| 9 | 2 | 9 | 2 | 7 | n/a |

What function is being computed? It is worth noting that the above program is said to be **standard form** since since the computation will always terminate with the program counter at $s + 1$, where $s$ is the number of instructions. A program is not in standard form in case the program counter can ever be assigned a value that exceeds $s + 1$.

**Definition 3.3.** An $m$-ary function $f : \mathcal{N}^m \to \mathcal{N}$ is **URM-computable** iff there exists a URM $M$ for which, for all $\vec{x} \in \mathcal{N}^m$,

1. if $f(\vec{x})$ is defined, then $M(\vec{x}) = f(\vec{x})$, and
2. if $f(\vec{x})$ is undefined, then $M(\vec{x}) \uparrow$.

If $f$ is defined on all inputs, then it is called **total URM-computable**. Otherwise, it is called **partially URM-computable**. Note: when we say a function is partially computable, it still may be possible that it is total computable. In other words, totally computable implies partially computable, but the converse is not necessarily true.

**Example 3.4.** Show that the function $f(x, y) = x + y$ is URM-computable.

**Solution.**

**Example 3.5.** By designing an appropriate URM $M$, show that the function

$$f(x) = \begin{cases} \lfloor x/2 \rfloor & \text{if } x \text{ is even} \\ \uparrow & \text{otherwise} \end{cases}$$

is URM-computable. Show the computations $M(2)$ and $M(3)$.

**Solution.**

**Definition 3.6.** We have the following definitions.

1. A **predicate** function is any function $f : \mathcal{N}^m \to \{0, 1\}$ whose output values are either 0 or 1.
2. A total predicate function is said to be **URM-decidable** iff there is a URM program that computes (i.e. **decides**) $f$.
3. A total unary predicate function is often referred to as a "property of the nonnegative integers".

**Example 3.7.** The property of being even can be represented by the function

$$\text{Even}(x) = \begin{cases} 1 & \text{if } x \bmod 2 = 0 \\ 0 & \text{otherwise} \end{cases}$$

**Example 3.8.** Provide a URM $M$ that proves that the predicate function Even$(x)$ from the previous example is URM-decidable.

**Solution.**

# 4 The Church-Turing Thesis

Notice that, in theory, a human is capable of simulating a URM program. This is because, for any program $P$ and input $y$ to $P$ a human is capable of maintaining a configuration vector for the computation of $P$ on input $y$ by successively updating the vector in accordance with the demands of the current instruction. For example, to perform an update on the configuration vector $\vec{c}$, the human first checks if the value $i$ stored in the program-counter component exceeds the number of instructions of $P$. If yes, then the computation has ended and $\vec{c}$ is the final configuration. In this case the human outputs the value currently stored in $\vec{c}$'s first component. On the other hand, if $\vec{c}$ is not a final configuration, the human inspects instruction $I_i$ and updates $\vec{c}$ accordingly. For example, suppose the instruction is $S(5)$. Then the human adds 1 to register $R_5$ (i.e. the fifth component of $\vec{c}$) and increments the program counter by adding 1 to the final component of $\vec{c}$.

The observation that a human is capable of simulating a URM program and the fact that mathematicians, such as Alan Turing and Alonzo Church, discovered that seemingly disparate models of computation were yielding the same set of computable functions, it led to the following famous informal thesis.

**Church-Turing Thesis.** Any function $f : \mathcal{N}^m \to \mathcal{N}$ that can in theory be computed in a deterministic step-by-step manner by a human using pencil and paper is URM computable.

Although the Church-Turing Thesis cannot be formally proved, evidence that supports it includes the fact that several different models of computation (e.g. URM's, Turing machines, Church's Lambda Calculus, all procedural programming languages used in practice, to name a few) have all proven to be equivalent, in that they compute the same set of functions.

The thesis allows us to let our creative imaginations run wild when devising an algorithm and, so long as we are able to describe how the algorithm works in a deterministic step-by-step manner, we may assume that it can be programmed by a URM or any other equivalent model of computation.

# 5 Encoding and Decoding URM Programs

**Definition 5.1.** The function $\mathrm{LP}(x, y)$ outputs the largest power of $y$ that divides evenly into $x > 0$, and equals 0 in case $x = 0$.

**Example 5.2.** We have $\mathrm{LP}(28, 2) = 2$ since $2^2 \mid 28$, but $2^3 \nmid 28$. □

**Theorem 5.3.** The following statements are all true.

a. **Binary encoder** $\pi(x, y) = 2^x(2y+1)-1$ is a bijection (i.e. one-to-one correspondence) between $\mathcal{N}^2$ and $\mathcal{N}$.

b. **Binary decoders** $\pi_1(z)$ and $\pi_2(z)$, where $\pi_1$ and $\pi_2$ satisfy

$$\pi(\pi_1(z), \pi_2(z)) = z$$

for all $z \in \mathcal{N}$, where $\pi_1(z) = \mathrm{LP}(z + 1, 2)$, and $\pi_2(z) = ((z + 1)/2^{\pi_1(z)} - 1)/2$.

c. **Ternary encoder** $\xi(x, y, z) = \pi(\pi(x, y), z)$ is a bijection between $\mathcal{N}^3$ and $\mathcal{N}$.

d. **Ternary decoders** $\xi_1(w)$, $\xi_2(w)$, and $\xi_3(w)$ where $\xi_1 = \pi_1(\pi_1(w))$, $\xi_2 = \pi_2(\pi_1(w))$, and $\xi_3 = \pi_2(w)$ satisfy

$$\xi(\xi_1(w), \xi_2(w), \xi_3(w)) = w$$

for all $w \in \mathcal{N}$.

e. **$k$-tuple encoder** $\tau : \bigcup_{k\geq 1} \mathcal{N}^k \to \mathcal{N}$ is a bijection between the set of all tuples of natural numbers and $\mathcal{N}$, where

$$\tau(a_1, a_2 \ldots, a_k) = 2^{a_1} + 2^{a_1+a_2+1} + \cdots + 2^{a_1+a_2+\cdots+a_k+k-1} - 1.$$

Notice that $\tau$ encodes tuples of all sizes.

f. **$k$-tuple decoder** $k(x)$ gives the dimension of the tuple $t$ for which $\tau(t) = x$. In particular,

$$k(x) = \sum_{i=0}^{\lfloor \log x \rfloor + 1} [(x + 1)/2^i \bmod 2].$$

In words, it's the number of nonzero bits of $x + 1$, when expressed in binary.

g. **$k$-tuple decoder** $c(x, i)$ gives the $i$th exponent of the powers-of-two expansion of $x + 1$. In particular, for $1 \leq i \leq k(x)$,

$$c(x, i) = \min_{j\geq 0}\Big(\sum_{r=0}^{j} [\lfloor (x + 1)/2^r \rfloor \bmod 2] = i\Big).$$

h. **$k$-tuple decoder** $a(x, i)$ gives the $i$th component of the tuple $t$ for which $\tau(t) = x$. In particular, $a(x, 1) = c(x, 1)$ and, for $2 \leq i \leq k(x)$, $a(x, i) = c(x, i) - c(x, i - 1) - 1$.

## 5.1 Decoding tips

Most practitioners will agree that decoding can seem somewhat more challenging than encoding. Whether performing a binary, ternary, or general tuple decoding of a number. The following are some useful tips.

**Add 1** Remember to first add 1 to the number being decoded.

**Binary Decoding of $z$** Write $z+1$ as

$$z + 1 = 2^x \cdot (2y + 1).$$

Then $\pi_1(z) = x$ and $\pi_2(z) = y$.

**Ternary Decoding of $w$** Follow these steps.

1. Perform binary decoding of $w$ (see above). In this case $\xi_3(w) = y$.
2. Binary decode $x$ to obtain $\xi_1(w) = \pi_1(x)$ and $\xi_2(w) = \pi_2(x)$.

**General Tuple Decoding** Follow these steps.

1. Write $x + 1$ as a power-of-two sum:

$$x + 1 = 2^{c(x,1)} + \cdots + 2^{c(x,k)},$$

   where $0 \leq c(x, 1) < \cdots < c(x, k)$ and $k = k(x)$ equals the number of terms in the sum.
2. Then $a(x, 1) = c(x, 1)$ and $a(x, i) = c(x, i) - c(x, i - 1) - 1$, for all $i = 2, \ldots, k$.

**Example 5.4.** Compute $\pi(3,4)$, $\pi_1(29)$, $\xi(3,2,1)$, $\xi_1(30)$, $\xi_2(30)$, $\xi_3(30)$, $\tau(1,4,0,2)$, $\tau^{-1}(49)$, $k(42)$, $c(2,42)$, and $a(2,42)$.

**Solution.**

a.
$$\pi(3,4) = 2^3(2(4)+1) - 1 = 71.$$

b. For $\pi_1(29)$ we have
$$29 + 1 = 30 = 2^1 \cdot (2(7)+1),$$
which yields $\pi_1(29) = 1$ and $\pi_2(29) = 7$.

c.
$$\xi(3,2,1) = \pi(\pi(3,2),1) = \pi(39,1) = 2^{39}(2(1)+1) - 1 = 2^{39} \cdot 3 - 1.$$

d. Since $\xi_3(30) = \pi_2(30)$ we first binary-decode 30 as
$$30 + 1 = 31 = 2^0 \cdot (2(15)+1,$$
which gives $\xi_3(30) = 15$. Now, to obtain $\xi_1(30)$ and $\xi_2(30)$, we must binary decode 0 as
$$0 + 1 = 2^0(2(0)+1),$$
which gives $\xi_1(30) = 0$ and $\xi_2(30) = 0$.

e.
$$\tau(1,4,0,2) = 2^1 + 2^{1+4+1} + 2^{(1+4+1)+0+1} + 2^{(1+4+1+0+1)+2+1} - 1 = 2^1 + 2^6 + 2^7 + 2^{10} - 1 = 1217.$$

f. For $\tau^{-1}(49)$,
$$49 + 1 = 50 = 2^5 + 2^4 + 2^1 = 2^1 + 2^4 + 2^5$$
which means $\tau^{-1}(49) = (1,2,0)$. This is because $a(49,1) = c(49,1) = 1$ is the first power-of-two exponent, $a(49,2) = c(49,2) - c(49,1) - 1 = 4 - 1 - 1 = 2$, and $a(49,3) = c(49,3) - c(49,2) - 1 = 5 - 4 - 1 = 0$.

g. For the final three parts, first compute $\tau^{-1}(42)$,
$$42 + 1 = 43 = 2^5 + 2^3 + 2^1 + 2^0 = 2^0 + 2^1 + 2^3 + 2^5$$
which means $\tau^{-1}(42) = (0,0,1,1)$. Since,
$$a(42,1) = c(42,1) = 0,\ a(42,2) = c(42,2) - c(42,1) - 1 = 1 - 0 - 1 = 0,$$
$a(42,3) = c(42,3) - c(42,2) - 1 = 3 - 1 - 1 = 1,$ and $a(42,4) = c(42,4) - c(42,3) - 1 = 5 - 3 - 1 = 1.$

Therefore, $k(42) = 4$, $c(42,2) = 1$, and $a(42,2) = 0$.

□

# 6 Encoding and Decoding URM Programs

We now use Theorem 5.3 to show a bijection $\beta : \mathcal{I} \to \mathcal{N}$ between the set $\mathcal{I}$ of all possible URM instructions and $\mathcal{N}$. Indeed, $\beta$ is defined by the following rules.

$$\beta(Z(n)) = 4(n-1),$$
$$\beta(S(n)) = 4(n-1)+1,$$
$$\beta(T(m,n)) = 4\pi(m-1,n-1)+2,$$

and

$$\beta(J(m,n,q)) = 4\xi(m-1,n-1,q-1)+3.$$

**Example 6.1.** Compute $\beta(Z(3))$, $\beta(S(4))$, $\beta(T(1,2))$, and $\beta(J(1,2,7))$.

**Solution.**

$$\beta(Z(3)) = 4(2) = 8.$$

$$\beta(S(4)) = 4(3)+1 = 13.$$

$$\beta(T(1,2)) = 4\pi(0,1)+2 = 4(2^0(2(1)+1)-1)+2 = 10.$$

$$\beta(J(1,2,7)) = 4\xi(0,1,6)+3 = 4\pi(\pi(0,1),6)+3 = 4\pi(2,6)+3 = 4(2^2(2(6)+1)-1)+3 = 207.$$

□

We now use $\beta$ above and $\tau$ from Theorem 5.3 to define bijection $\gamma : \mathcal{P} \to \mathcal{N}$ between the set of all URM programs $\mathcal{P}$ and $\mathcal{N}$. Indeed, given URM program $P = I_1, \ldots, I_s$, then

$$\gamma(P) = \tau(\beta(I_1), \ldots, \beta(I_s)).$$

**Example 6.2.** Calculate $\gamma(P)$, for $P = T(1,3), S(4), Z(6)$.

**Solution.**

We have $\beta(T(1,3)) = 4\pi(0,2) + 2 = 18$, $\beta(S(4))) = 4(3) + 1 = 13$, and $\beta(Z(6)) = 4(5) = 20$. Hence,

$$\gamma(P) = \tau(18, 13, 20) = 2^{18} + 2^{32} + 2^{53} - 1 = 9007203549970431.$$

□

**Example 6.3.** Determine the program $P$ for which $\gamma(P) = 4127$. In other words, compute $\gamma^{-1}(4127)$.

**Solution.** We have

$$(4128)_2 = 1000000100000 = 2^5 + 2^{12}$$

which implies $P$ has two instructions $I_1$ and $I_2$, where $\beta(I_1) = 5$ and $\beta(I_2) = 6$. Moreover, since $5 \bmod 4 = 1$, we have $I_1 = S(2)$, and since $6 \bmod 4 = 2$, $I_2$ is a transfer function $T(m, n)$ where $4\pi(m-1, n-1) + 2 = 6$, which means $\pi(m-1, n-1) = 1$. Thus, $m = 2$, $n = 1$. Thefore, $P = S(2), T(2, 1)$. □

## 6.1 Gödel numbers and indices

The following notation will prove useful for the remaining lectures on computability.

$P_a$ denotes the program $P$ for which $\gamma(P) = a$. Number $a$ is called the **Gödel number** of $P$.

$\phi_a^{(n)}$ denotes the $n$-ary function computed by URM program $P_a$.

$W_a^{(n)}$ denotes the domain of $\phi_a^{(n)}$, i.e.

$$W_a^{(n)} = \{(x_1, \ldots, x_n) | P_a(x_1, \ldots, x_n) \downarrow\}.$$

$E_a^{(n)}$ denotes the range of $\phi_a^{(n)}$, i.e.

$$E_a^{(n)} = \{y | \phi_a^{(n)}(x_1, \ldots, x_n) = y \text{ for some tuple input } (x_1, \ldots, x_n)\}.$$

Note that the superscripts in the above definitions are dropped in case $n = 1$. For example, $\phi_{10}$ denotes the unary function computed by URM program $P_{10}$. In such cases, $W_a$ will denote a set of natural numbers rather than 1-tuples.

**Example 6.4.** Use the previous example and the above definitions to describe $\phi_{4127}$, $W_{4127}$, and $E_{4127}$.

**Solution.**

We say that $a \in \mathcal{N}$ is an **index** for computable function $f(x)$ iff $f(x) = \phi_a(x)$. Moreover, since an infinite number of programs can be defined for computing the same function, it follows that every computable function $f$ has infinitely many indices, and that $f$ will appear infinitely many times within the sequence $\phi_0, \phi_1, \ldots$.

# 7 The S-M-N Theorem

Given a computable function $f(x, y)$, each fixed integer $x_0$ induces a computable unary function $g(y) = f(x_0, y)$. Thus, $f(x, y)$ spawns an infinite number of computable functions by fixing its first input in different ways. Moreover, the s-m-n theorem states that these functions are related in a uniform way.

**Simplified S-M-N Theorem.** Given a computable function $f(x, y)$ there exists a total URM computable function $h(x)$ for which

$$f(x, y) = \phi_{h(x)}(y).$$

In other words, when $x = x_0$ is fixed, $h(x_0)$ represents the Gödel number of a program that computes $g(y) = f(x_0, y)$.

**Proof.** Let $x$ be given. Let $P = I_1, \ldots, I_s$ be a URM program for computing $f(x, y)$. Letting $x \in \mathcal{N}$ be arbitrary, consider the following URM program $P_x$ for which $P_x(y) = f(x, y)$.

1. $T(1, 2)$ //transfer input $y$ to $R_2$
2. $Z(1)$
3. $\underbrace{S(1), \ldots, S(1)}_{x \text{ times}}$ //place $x$ in $R_1$
4. $I_1, \ldots, I_s$ //compute $f(x, y)$

Letting $h(x)$ denote the Gödel number of this program, it is an exercise to show that $h(x)$ can be written as a primitive recursive function using the encoding functions from Sections 1 and 2. Moreover, we have

$$f(x, y) = \phi_{h(x)}(y),$$

and the theorem is proved. □

**General S-M-N Theorem.** Given $\phi_s^{(m+n)}(\vec{x}, \vec{y})$, where $\vec{x}$ and $\vec{y}$ are respectively $m$ and $n$-dimensional, there exists a total computable function $h(s, \vec{x})$ for which

$$\phi_s^{(m+n)}(\vec{x}, \vec{y}) = \phi_{h(s,\vec{x})}^{(n)}(\vec{y}).$$

**Example 7.1.** Consider the function $f(x, y) = x + y$ which is computable via the following program $P$.

1. $J(2, 3, 5)$
2. $S(1)$
3. $S(3)$
4. $J(1, 1, 1)$

The following table shows $g(y)$ and $h(x)$ for the first few values of $x$. Here, $h(x)$ provides the index of a function $\phi_{h(x)}(y)$ that computes $g(y)$. Moreover $h(x)$ is obtained by computing the Gödel number of the following program.

1. $T(1, 2)$
2. $Z(1)$
3. $\underbrace{S(1), \ldots, S(1)}_{x \text{ times}}$
4. $J(2, 3, 8)$
5. $S(1)$
6. $S(3)$
7. $J(1, 1, 4)$

The Gödel number of the above program is

$$h(x) = \tau(10, 0, \underbrace{1, \ldots, 1}_{x \text{ times}}, 30719, 1, 9, 0) =$$

$$2^{10} + 2^{12} + \cdots + 2^{10+2x} + 2^{30730+2x} + 2^{30732+2x} + 2^{30742+2x} + 2^{30743+2x} - 1.$$

| $x$ | $g(y)$ | Index $h(x)$ for $g(y)$ |
|---|---|---|
| 0 | $g(y) = 0 + y = y$ | $h(0) = 2^{10} + 2^{30730} + 2^{30732} + 2^{30742} + 2^{30743} - 1$ |
| 1 | $g(y) = 1 + y$ | $h(1) = 2^{10} + 2^{12} + 2^{30732} + 2^{30734} + 2^{30744} + 2^{30745} - 1$ |
| 2 | $g(y) = 2 + y$ | $h(2) = 2^{10} + 2^{12} + 2^{14} + 2^{30734} + 2^{30736} + 2^{30746} + 2^{30747} - 1$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $x$ | $g(y) = x + y$ | $h(x) = 2^{10} + 2^{12} + \cdots + 2^{10+2x} + 2^{30730+2x} + 2^{30732+2x} + 2^{30742+2x} + 2^{30743+2x} - 1$ |
| $\vdots$ | $\vdots$ | $\vdots$ |

Notice that $h(x)$ is primitive recursive, and thus total computable as is stated in the s-m-n theorem.

# 8 Universal Programs

Consider the function $\psi_U(x, y) = \phi_x(y)$. This function takes as input a program number $x$, along with a natural number $y$, and returns the output that program $P_x$ produces when supplied input $y$. Thus, we see that $\psi_U(x, y)$ embodies *all* possible computations of all possible programs, since its first input $x$ can represent any program and its second input $y$ can represent any input to the program.

We now establish that universal programs exist by invoking the Church-Turing thesis. Indeed, to compute $P_U(x, y)$ we execute the following steps.

1. Decode $x$ to obtain URM program $P_x$ and examine the instructions to obtain the maximum register index $n$ that is used by $P_x$, as well as the number of instructions $s$ possessed by $P_x$.
2. Let $c_0 = (y, 0_1, \ldots, 0_{n-1}, \text{pc} = 1)$ denote the initial configuration of $P_x(y)$.
3. While $\text{pc} \leq s$,
   (a) Execute $I_{\text{pc}}$ and update the current configuration of $P_x(y)$.
4. Return the value of the first register of the current configuration.

Note that implementing the above algorithm with a URM requires moving from one configuration *encoding* to the next configuration encoding, since the URM that computes $P_U$ only has a finite number of registers, while the program $P_x$ it simulates can possess an arbitrarily large number of registers.

**Example 8.1.** A universal program $P_U$ is simulating a program that has 205 instructions and whose Gödel number is

$$x = 2^3 + 2^{45} + 2^{67} + 2^{78} + 2^{117} + 2^{141} + \cdots + 2^{c_{205}} - 1.$$

If the current configuration of the computation of $P_x$ on some input has encoding

$$\sigma = 2^2 + 2^6 + 2^8 + 2^{14} - 1,$$

then provide the next configuration of the computation *and* its encoding.

**Solution.**

# Exercises

Note: assume that "computable" means the same thing as "URM-computable". By the Church-Turing thesis, we may assume that any arithmetic-related function is URM-computable.

For exercises 1-5 you may find it useful and fun to test your solutions with an online URM simulator:

`https://sites.oxy.edu/rnaimi/home/URMsim.htm`

1. Provide URM-programs that compute the following functions.

   a.
   $$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } x \neq 0 \end{cases}$$

   b. $f(x) = 4$

   c.
   $$f(x, y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{if } x > y \end{cases}$$

2. Show that the function
   $$f(x, y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$$
   is URM-computable.

3. Show that the function $f(x, y) = \min(x, y)$ is URM-computable.

4. Suppose $f(x)$ and $g(x)$ are both URM-computable via programs $P_1$ and $P_2$ respectively. Provide an outline of a URM program that computes $f(g(x))$.

5. Suppose $P_1$ and $P_2$ are two programs, and we desire to make a third program $P_3$ whose behavior can be described as "Run $P_1$ until it halts. Then run $P_2$ on the final register configuration produced by $P_1$." Explain why $P_1P_2$ may not have the desired effect, where $P_1P_2$ means list the instructions of $P_2$ immediately after those of $P_1$. Explain the alterations that may need to be made in order for $P_1P_2$ to work as desired.

6. If $f(x)$ is URM-computable via a program that has no jump instructions, then prove that $f(x) = C$ or $f(x) = x + C$, for some constant $C \in \mathcal{N}$. In other words, $f(x)$ must either be a constant function or a linear function with slope equal to 1.

7. Prove that $\pi(x, y) = 2^x(2y + 1) - 1$ is a bijection from $\mathcal{N}^2$ to $\mathcal{N}$.

8. Compute the following: $\pi(3, 8)$, $\pi^{-1}(117)$, $\xi(3, 4, 2)$, $\xi^{-1}(563)$, $\tau(5, 8, 4, 2, 4)$, $\tau^{-1}(5387)$.

9. Use the $\beta$ function to encode the following URM instructions: $Z(6)$, $S(17)$, $T(5, 8)$, and $J(4, 6, 3)$. Also, determine $\beta^{-1}(99)$, $\beta^{-1}(108)$, $\beta^{-1}(129)$ and $\beta^{-1}(150)$.

10. Provide the Gödel number of the program $P = S(1), S(1), T(1,2), J(1,1,1)$.

11. Provide the instructions for $P_{100}$.

12. Suppose that $f(x,y)$ is a total computable function. For each $m \in \mathcal{N}$, let $g_m(y)$ denote the total computable function defined by $g_m(y) = f(m,y)$. Provide a total computable function $h$ such that, for each $m \in \mathcal{N}$, $h \neq g_m$.

13. Show that there is a total computable function $k(n)$ such that $k(n)$ is an index of the function $\lfloor \sqrt[n]{x} \rfloor$.

14. A universal program $P_U$ is simulating a program that has 754 instructions and whose Gödel number is
$$x = 2^7 + 2^{23} + 2^{63} + 2^{71} + 2^{105} + 2^{141} + \cdots + 2^{c_{754}} - 1.$$
If the current configuration of the computation of $P_x$ on some input has encoding
$$\sigma = 2^3 + 2^5 + 2^{10} + 2^{13} + 2^{16} - 1,$$
then provide the next configuration of the computation *and* its encoding.

15. Consider the function CurInsZero$(x,i)$ which evaluates to 1 iff the $i$th instruction of URM program $P_x$ is a `Zero` instruction. Use the encoding and decoding functions from Section 5 to provide an arithmetic formula for computing CurInsZero.

16. Consider the function IncrementComponent$(x,i)$ which returns the encoding of a tuple that equals $\tau^{-1}(x)$, but with 1 added to component $i$. Use the encoding and decoding functions from Section 5 to provide an arithmetic formula for computing IncrementComponent.

# Exercise Solutions

1. Provide URM-programs that compute the following functions.

   a. $J(1,2,3), S(2), T(2,1)$

   b. $Z(1), S(1), S(1), S(1), S(1)$.

   c. $J(1,3,5), J(2,3,6), S(3), J(1,1,1), S(4), T(4,1)$

2. 1. $J(1,2,10)$, 2. $T(1,3)$, 3. $T(2,4)$, 4. $S(3)$, 5. $J(2,3,10)$, 6. $S(4)$, 7. $S(5)$, 8. $J(1,4,12)$, 9. $J(1,1,4)$, 10. $Z(1)$, 11. $J(1,1,15)$, 12. $T(5,1)$, 13. $J(1,1,14)$

3. 1. $J(1,2,10)$, 2. $T(1,3)$, 3. $T(2,4)$, 4. $S(3)$, 5. $J(2,3,10)$, 6. $S(4)$, 7. $J(1,4,9)$, 8. $J(1,1,4)$, 9. $T(2,1)$,

4. First execute the instructions of $P_2$. Let $m$ be the index of the maximum register used by $P_2$. Next, perform the instructions $Z(2), \ldots, Z(m)$. Finally, execute the instructions of $P_1$.

5. Suppose $P_1$ has $k$ instructions, then any jump instruction of $P_1$ that jumps to a value $v > k$, should now jump to $k+1$, so that the first instruction of $P_2$ executes next. Furthermore, each jump instruction of $P_2$ should have its jump address incremented by $k$ so that jumps do not accidentally land back in $P_1$.

6. Case 1: register $R_1$ is written over via either a $Z(1)$ or $T(m,1)$ instruction, for some $m > 1$. In case of a $Z(1)$ instruction, $R_1$ can hold at most a constant $C$ which equals the number of $S(1)$ instructions that follow the final $Z(1)$ instruction. In case $R_1$ was written over via a transfer from register $R_m$, $R_1$ equals $C$, where $C$ is the number of $S(m)$ instructions that precede the final $T(m,1)$ instruction, plus the number of $S(1)$ instructions that follow the final $T(m,1)$ instruction.

   Case 2: register $R_1$ is never written over. Then $R_1$ will hold the value $x + C$, where $C$ is the number of $S(1)$ program instructions.

   If $f(x)$ is URM-computable via a program that has no jump instructions, then prove that $f(x) = C$ of $f(x) = x + C$, for some constant $C \in \mathcal{N}$.

7. $\pi$ maps onto the set of natural numbers since. for any natural number $z$, $z+1$ can always be written in the form $2^x J$, where $x \geq 0$ and $J \geq 1$ is odd. Thus, letting $y = (J-1)/2$, we have $\pi(x,y) = 2^x(2y+1) - 1 = z$.

   We now show that $\pi$ is one-to-one. Suppose $2^{x_1}(2y_1+1) - 1 = 2^{x_2}(2y_2+1) - 1$. This implies

   $$2^{x_1}(2y_1+1) = 2^{x_2}(2y_2+1),$$

   which is a positive integer. But any positive integer has a unique prime factorization. Hence, we must have $x_1 = x_2$, which in turn implies $2y_1 + 1 = 2y_2 + 1$, and so $y_1 = y_2$. Therefore, $\pi$ is one-to-one.

8. We have $\pi(3,8) = 135$, $\pi^{-1}(117) = (1,29)$, $\xi(3,4,2) = 5 \cdot 2^{71} - 1$, $\xi^{-1}(563) = (0,1,70)$, $\tau(5,8,4,2,4) = 138952735$, $\tau^{-1}(5387) = (2,0,4,1,1)$.

9. We have $\beta(Z(6)) = 20$, $\beta(S(17)) = 65$, $\beta(T(5,8)) = 958$, $\beta(J(4,6,3)) = 20 \cdot 2^{87} - 1$, $\beta^{-1}(99) = J(1,1,13)$, $\beta^{-1}(108) = Z(28)$, $\beta^{-1}(129) = S(33)$ and $\beta^{-1}(150) = T(2,10)$.

10. $\gamma(P) = 278537$.

11. $Z(1), S(1), T(1,1), Z(1)$.

12. By the s-m-n theorem, there is a total computable function $k(m)$ for which $\phi_{k(m)}(y) = g_m(y) = f(m,y)$. Now define $h(y) = \phi_{k(y)}(y) + 1$. Now let $m \in \mathcal{N}$ be given. Then

$$h(m) = \phi_{k(m)}(m) + 1 = g_m(m) + 1 \neq g_m(m).$$

Thus, $h$ disagrees with $g_m$ on input $m$, and so $h \neq g_m$.

13. The function $f(n,x) = \lfloor \sqrt[n]{x} \rfloor$ is certainly computable (exercise: show that it is actually primitive recursive). Thus, by the s-m-n theorem, there is a total computable function $k(n)$ for which

$$\phi_{k(n)}(x) = f(n,x) = \lfloor \sqrt[n]{x} \rfloor.$$

Therefore, $k(n)$ is an index for the "$n$ th root" function.

14. We have

$$c = \tau^{-1}(\sigma) = (3,1,4,2,2).$$

Also, $\beta(I_2) = 15$ and $15 \bmod 4 = 3$ implies that $I_2$ is a jump instruction $J(i,j,k)$, where $\xi(i-1, j-1, k-1) = 3 = (15-3)/4$. Finally, to get $\xi^{-1}(3)$ we see that

$$3 + 1 = 4 = 2^2(2(0) + 1),$$

and $\pi^{-1}(2) = (0,1)$ to give $\beta^{-1}(15) = J(1,2,1)$. Therefore,

$$c_{\text{next}} = (3,1,4,2,3)$$

and

$$\tau(c_{\text{next}}) = 2^3 + 2^5 + 2^{10} + 2^{13} + 2^{17} - 1.$$

15. $\text{CurInsZero}(x,i) = (a(x,i) \bmod 4 = 0)$

16. Incrementing the $i$ th component has the effect of adding a 1 to the power-of-two exponents $i, \ldots, k(x)$, which yields

$$\text{IncrementComponent}(x,i) = x + \sum_{j=i}^{k(x)} 2^{c(x,j)}.$$