Automata and Formal Languages

October 23, 2019

1 Register Machines and Computability

Books: PTJ (Chapter 4)

NOTE: HERE $\mathbb{N} = \{0, 1, 2, \ldots\}$

A ***register machine (RM)*** consists of:

1. A sequence of ***registers*** $R_1, R_2, R_3, \ldots$ where at discrete time steps $t = 0, 1, 2, \ldots$ have $R_i(t) \in \mathbb{N}$, In fact, we only have finitely many registers, and regard $R_i \equiv 0$ for all $i \geq I$.
2. A finite ***program*** consisting of a fixed number of ***states*** S_0 (HALT), S_1 (START), $S_2, \ldots, S_n$. Each state comes with a fixed instruction performed when in state S_i. When the computer reaches HALT, we get the output from R_1. Otherwise, for $1 \leq i \leq n$ we have 2 types of ***commands***:
   (a) Increment R_j, then move to state S_k. We write this $S_i : (j, +, k)$.
   (b) If $R_j \neq 0$ then decrement R_j, then move to state S_k. Otherwise move to state S_l. We write this $S_i : (j, -, k, l)$.

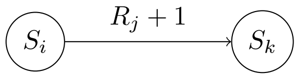
A ***sequence of instructions*** for a RM is the ordered list of the instructions for the program. An ***input*** for a RM is, for some $k \geq 1$, a finite k-tuple $(n_1, \ldots, n_k) \in \mathbb{N}^k$ which are the initial values of $R_1, \ldots, R_k$. The other registers are set to 0.

A ***program diagram*** for a RM is a directed graph with vertices being the states of the machine and the labelled arrows denote the instructions: $S_i : (j, +, k)$

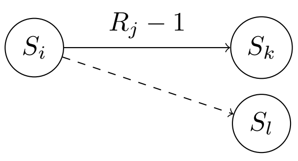
We can then use these to describe programs:

For any $k > 0$ a program P ***halts*** on input $(m_1, m_2, \ldots, m_k) \in \mathbb{N}^k$ if it ever reaches state S_0, written $P(m_1, \ldots, m_k) \downarrow$
The ***halting set*** $\Omega(P)$ is a set of inputs on which P halts.

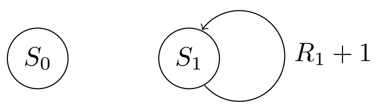
$$\Omega(P) = \cup_{k>0}\{(m_1, \ldots, m_k) : P(m_1, \ldots, m_k) \downarrow\}$$



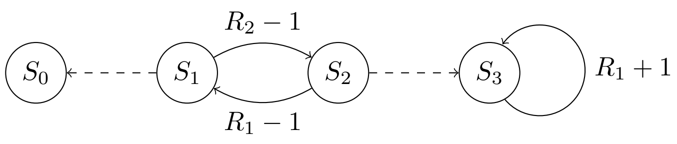
(a) $S_i : (j, +, k)$



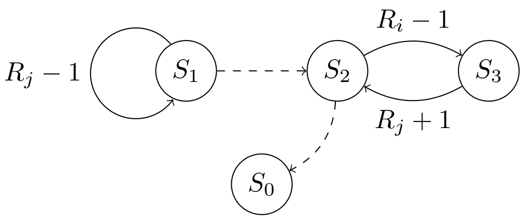
(b) $S_i : (j, -, k, l)$



(a) Repeatedly increment R_1, never halting



(b) For input (n_1, n_2) returns $n_1 - n_2$ if $n_1 \geq n_2$, else never halt



(c) Transfer R_i to R_j, emptying R_i

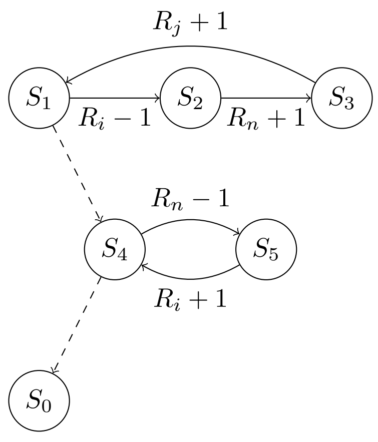
If P does not halt, we write $P(m_1, \dots, m_k) \uparrow$.

For each program P, the ***upper register index*** $\text{Upper}(P)$ is the largest index of a register appearing in the instructions for P. So if $i > \text{Upper}(P)$ then R_i never changes.

A ***partial function*** $f : \mathbb{N}^k \to \mathbb{N}$ is one where the domain of f is a subset of $\mathbb{N}^k$, and undefined otherwise. If f is defined everywhere then we call it a ***total function***. This lets us define these programs as functions - we say f is ***partial computable*** by a program P such that $\forall (m_1, \dots, m_k) \in \text{dom}(f)$ have $P(m_1, \dots, m_k) \downarrow$ with $f(m_1, \dots, m_k) = R_1$ on halting, and $\forall (m_1, \dots, m_k) \notin \text{dom}(f)$ we have $P(m_1, \dots, m_k) \uparrow$. Hence any program P and $k > 0$ gives a partial function $f : \mathbb{N}^k \to \mathbb{N}$.

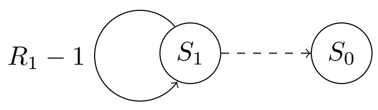
Lemma 1.1. *We can add R_i to R_j leaving R_i unchanged.*

Proof.



Thus by setting $(i = 2, j = 1)$ we see that $(n_1, n_2) \mapsto n_1 + n_2$ is total computable. □

We have already seen that the function $n \mapsto 0$ is also computable. This can be done with the machine:



Corollary 1.2. *There exists a routine which can copy R_i to R_j leaving R_i unchanged.*

Proof. First empty R_j, then use **1.1** to add R_i to R_j. □

We can use these as subroutines to join with other programs P. Use registers R_n s.t. $n >$ Upper(P) and largest input register. Then replace the halt state of P with the start state of the subroutine. In fact we have already done this - if you look carefully at the adding machine, you can see that the middle section is the same as the machine in (c) of the examples - this is the part where we replace the value in R_i from its temporary location in R_n.

Partial Recursive Functions

Partial computable functions have good closure properties.

Theorem 1.3.

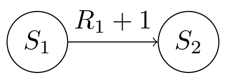
1. *For $i \leq k$, the* **projection function** $(n_1, \dots, n_k) \mapsto n_i$ *is computable.*
2. *The zero function $n \mapsto 0$ and* **successor function** $n \mapsto n+1$ *are computable*
3. *(Composition) If $f : \mathbb{N}^k \to \mathbb{N}$ and $g_1, \dots, g_k : \mathbb{N}^l \to \mathbb{N}$ are all partial computable then so is the composition function $h(n_1, \dots, n_l) = f(g_1(n_1, \dots, n_l), \dots, g_k(n_1, \dots, n_l))$ where defined. If $f, g_1, \dots, g_k$ are total functions, so is h.*
4. *(Recursion) If f on k variables and g on $k+2$ variables are partial computable, then so is the partial function $h : \mathbb{N}^{k+1} \to \mathbb{N}$ defined inductively as:*

$$h(n_1, \dots, n_k, 0) = f(n_1, \dots, n_k)$$
$$h(n_1, \dots, n_k, n_{k+1} + 1) = g(n_1, \dots, n_{k+1}, h(n_1, \dots, n_{k+1}))$$

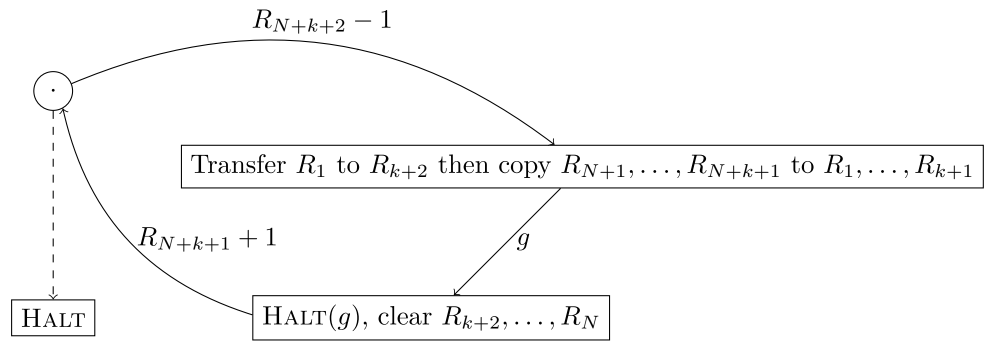
   Moreover, f, g total $\implies$ h total.
5. *(Minimisation) If f on $k+1$ variables is partial computable then so is the partial function $g : \mathbb{N}^k \to \mathbb{N}$ defined by $g(n_1, \dots, n_k) = n$ if $f(n_1, \dots, n_k, n) = 0$ and $f(n_1, \dots, n_k, m) > 0$ for all $m < n$, and is undefined if no zero is ever found. Note that f total $\not\Rightarrow$ g total.*

Proof.

1. We can use the program Transfer R_i to R_1, Halt.
2. Zero function has already been seen. For successor function, use:

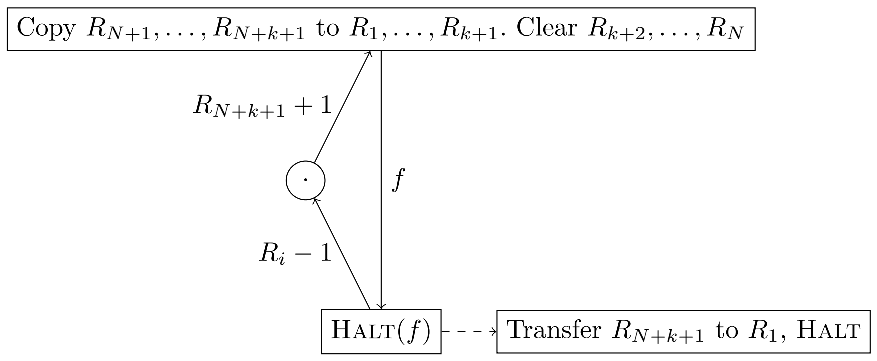


3. First transfer $R_1, \dots, R_l$ to $R_{N+1}, \dots, R_{N+l}$ where N is large enough to not be needed in other subroutines. Then for each $1 \le i \le k$ in turn, copy $R_{N+1}, \dots, R_{N+l}$ to $R_{k+1}, \dots, R_{k+l}$, perform g_i but with all registers shifted up by k and then transfer answer from R_{k+1} to R_i, then clear $R_{k+2}, \dots, R_N$. Finally, apply f.
4. Copy $R_1, \dots, R_k$ to $R_{N+1}, \dots, R_{N+k}$, transfer R_{k+1} to R_{N+k+2} ("counts down"), then do f. Then:



$R_{N+k+2} - 1$
Transfer R_1 to R_{k+2} then copy $R_{N+1}, \dots, R_{N+k+1}$ to $R_1, \dots, R_{k+1}$
g
$R_{N+k+1} + 1$
HALT
HALT(g), clear $R_{k+2}, \dots, R_N$

5. Copy $R_1, \dots, R_k$ to $R_{N+1}, \dots, R_{N+k}$. Then



Copy $R_{N+1}, \dots, R_{N+k+1}$ to $R_1, \dots, R_{k+1}$. Clear $R_{k+2}, \dots, R_N$
$R_{N+k+1} + 1$
f
$R_i - 1$
HALT(f)
Transfer R_{N+k+1} to R_1, HALT

□

The class of ***partial recursive functions*** is the smallest class of partial functions from $\mathbb{N}^k$ to $\mathbb{N}$ over all $k \ge 1$ closed under the operations **1.3** (1) to (5). That is, f can be constructed from basic functions and applications of (3), (4), (5) a finite number of times.

So **1.3** says that partial recursive $\implies$ partial computable.

A partial function is ***primitive recursive*** if we never use **1.3** (5) its construction. Note that primitive recursive $\implies$ total recursive, as (5) was the only construction that breaks the totality of the function. [The converse implication is not true: the Ackermann function.]

Example: $+$ and $\times$ are primitive recursive:

$+$: Let $h(m,0) = m$, $h(m,n+1) = h(m,n) + 1 = g(m,n,h(m,n))$, where $g(x,y,z) = z + 1$.

$\times$: $H(m,0) = 0$, $H(m,n+1) = H(m,n) + m = g(m,n,H(m,n))$ for $g(x,y,z) = x + z$.

Example: $(m,n) \mapsto m^n$ is primitive recursive - left as exercise.

We need to be able to "encode" finite sequences of arbitrary length in $\mathbb{N}$. For $n > 0$ and $i \in \mathbb{N}$, write p_i for the $(i+1)^{\text{th}}$ prime (so $p_0 = 2$). Write $(n)_i$ for the largest power of the prime p_i that divides n.

Lemma 1.4. *For each fixed i, the 1 variable function $(\cdot)_i : \mathbb{N} \to \mathbb{N}$ is primitive recursive.*

Proof. First note that, for any finite sequence $(m_0, m_1, \dots, m_s) \subseteq \mathbb{N}^{s+1}$, the function $f(n) = \begin{cases} m_n & n \leq s \\ 0 & n > s \end{cases}$ is primitive recursive.

By induction on s and recursion from **1.3** *(4)*, for $k = 0$ if c constant and $g : \mathbb{N}^2 \to \mathbb{N}$ is primitive recursive, then so is $h(0) = c, h(n+1) = g(n, h(n))$.

Thus, given $f : \mathbb{N} \to \mathbb{N}$ primitive recursive, let $g(n, m) := f(n)$, which is primitive recursive. So $h(0) = c, h(n+1) = f(n)$ is primitive recursive, and we can repeat this process.

This includes for each <u>fixed</u> k:

1. The step function $\text{Step}_k(n) = \begin{cases} 1 & 0 \leq n \leq k-1 \\ 0 & \text{otherwise} \end{cases}$
2. The delta function $\delta_k(n) = \begin{cases} 1 & n = k \\ 0 & n \neq k \end{cases}$ Let $\epsilon(n) = \delta_0(\delta_0(n)) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \end{cases}$ - this is also primitive recursive.
3. The slope function $\text{Slope}_k(n) = \begin{cases} n+1 & 0 \leq n \leq k-2 \\ 0 & \text{otherwise} \end{cases}$
4. The remainder function $\text{Rem}_k(n) = n \mod k$ use recursion in the form $g(n, m) := f(m)$, so $h(0) = 0$, $h(n+1) = f(h(n))$ primitive recursive if f is. Here, $\text{Rem}_k(n+1) = \text{Slope}_k(\text{Rem}_k(n))$

*5. $\text{Floor}_k(n) = \lfloor \frac{n}{k} \rfloor$

*6. $\text{Divide}_k(n) = \begin{cases} n/k & n \equiv 0 \mod k \\ 0 & \text{otherwise} \end{cases}$

*7. Division by powers $\text{Power}_k(n, m) = \begin{cases} n/k^m & n \equiv 0 \mod k^m \\ 0 & \text{otherwise} \end{cases}$

*8. $\text{Maxpower}_k(n) = \begin{cases} 0 & n = 0 \\ \text{largest power of k dividing n} & n \neq 0 \end{cases}$

Proofs of *ed function are on example sheet 1.

Now define by recursion $h(n, 0) = 0$ and $h(n, m+1) = h(n, m) + \epsilon(\text{Power}_k(n, m+1))$.
$\epsilon(\text{Power}_k(n, j)) = \begin{cases} 1 & k^j \text{ divides } n > 0 \\ 0 & \text{otherwise} \end{cases}$, so is 0 if $j \geq n$

So $h(n, n) = \sum_{i=1}^{n} \epsilon(\text{Power}_k(n, 1)) = \text{Maxpower}_k(n)$, so $h(n, n)$ is primitive recursive. □

Computable = Recursive

We have seen already that partial recursive $\implies$ partial computable.

Theorem 1.5. *Every partial computable function $f : \mathbb{N}^k \to \mathbb{N}$ is partial recursive.*

Proof. From a program P for f, define $g : \mathbb{N}^{k+2} \to \mathbb{N}$, "what actually goes on in P", to be the function:

$$g(n_1, \dots, n_k, 0, t) \text{ is the state of } P \text{ after time } t \text{ with input } (n_1, \dots, n_k)$$

So $t = 0$ gives 1 and if halt at t_0 then gives 0 for all $t \geq t_0$, and:

$$(n_1, \dots, n_k, i, t) \text{ is the contents of } R_i \text{ at time } t$$

So have N (assume $> k$) such that $g(\cdots, i, \cdot) = 0 \forall i > N$. Note that g is a total function.

Suppose that g is recursive and define $q(n_1, \dots, n_k) = \min\{t : g(n_1, \dots, n_k, 0, t) = 0\}$. Then q is partial recursive, and so $f(n_1, \dots, n_k) = g(n_1, \dots, n_k, 1, q(n_1, \dots, n_k))$ is partial recursive.

Proof that g is recursive:
Fix $n_1, \dots, n_k$ and t. For each $0 \leq i \leq N$, g gives $(g_0, \dots, g_N) \in \mathbb{N}^{N+1}$, encode as $c(d_0, \dots, d_N) = 2^{d_0} 3^{d_1} \dots p_N^{d_N} \in \mathbb{N}$ is primitive recursive. Also, $(c(d_0, \dots, d_N))_i = d_i$ is primitive recursive. We will define $h : \mathbb{N}^{k+2} \to \mathbb{N}$ via recursion where $h(n_0, n_1, \dots, n_k, t)$ is the coded integer of state and registers of P at time t for input $n_1, \dots, n_k$ and start state n_0 (here $= 1$).

In particular, for $t = 0, h = 2^{n_0} 3^{n_1} \dots p_k^{n_k}$. For recursion for h, we need $s : \mathbb{N} \to \mathbb{N}$, the "transition function", which computes in coded form the changes at each step. □

Algorithms and Recursive Sets

A function $f : \mathbb{N}^k \to \mathbb{N}$ is ***recursive*** or ***computable*** if it is total. If it s not even partial recursive, then it is ***incomputable***.

A subset $X \subseteq \mathbb{N}^k$, (often $X \subseteq \mathbb{N}$) is ***recursive*** or ***computable*** or ***decidable*** if the characteristic function $\chi_X(n) = \begin{cases} 1 & n \in X \\ 0 & n \notin X \end{cases}$ is computable, i.e. if we can program a computer to tell us if a given number is in it or not.

An ***algorithm*** is any process which takes an input in $\mathbb{N}^k$ for some specified k, or a recursive subset $X \subseteq \mathbb{N}^k$, and returns an output in $\mathbb{N}$ which is simulated by a register machine.

A ***total algorithm*** terminates for all elements in X, whilst a ***partial algorithm*** may fail to terminate for some choices of input.

Lemma 1.6. *For each $k \geq 1$, there is some total function $f : \mathbb{N}^k \to \mathbb{N}$ which is incomputable.*

Proof. Each computable program comes from a finite program with $n+1$ states for some n. Since there are only countably many finite programs, but $\mathcal{P}(\mathbb{N})$ is uncountable, hence one of these sets is not computable. Then its indicator function is not computable. □

For given m, the ***shortlex*** ordering on $\mathbb{N}^m$ is $(n_1, \ldots, n_m) < (n'_1, \ldots, n'_m)$ if $\sum n_i < \sum n'_i$ or $\sum n_i = \sum n_i$ and there is some j with $n_i = n'_i$ for $i < j$, but $n_{j+1} > n'_{j+1}$.

This gives us a bijection to $\mathbb{N}$, as there are only finitely many k-tuples of naturals with sum less than $N \in \mathbb{N}$.

If a register machine P, then for the i^{th} instruction, let $t_i = \begin{cases} 2^j \cdot 5^k & \text{if it is } (j, +, k) \\ 2^j \cdot 3 \cdot 5^k \cdot 7^l & \text{if it is } (j, -, k, l) \end{cases}$.
We can then encode the tuple $(t_1, t_2, \ldots, t_n)$ as $m = 2^n \cdot 3^{t_1} \cdot 5^{t_2} \cdot \ldots \cdot p_n^{t_{n-1}}$. We denote the program encoded by the number m as P_m, if m is a valid encoding of a program. For these m, we say m ***codes*** a program, and P_m is the ***m***

The input for a register machine is a k-tuple for varying k, so we define $f_{n,k}$ for the k-variable function computed by the n^{th} machine if P_n exists.

Here is an explicit total function which is not recursive:

Lemma 1.7. *Consider the following function $g : \mathbb{N} \to \mathbb{N}$ given by:*

$$g(n) := \begin{cases} f_{n,1}(n) + 1 & \textit{if } n \textit{ codes a program and if } f_{n,1}(n) \textit{ is defined} \\ 0 & \textit{else} \end{cases}$$

Then g is not recursive.

Proof. If g is recursive then it is computed by some machine. So there exists an N such that $g = f_{N,1}$ is total. But then $f_{N,1}(N) = g(N) = f_{N,1}(N) + 1$ ↯. □

Church's Thesis

The two key figures in this chapter are Alonzo Church and Alan Turing, doing this work around 1936.

An ***executable process*** is a step-by-step deterministic process with finite description at each step, a finite set of rules, and a finite amount of input and output.

An ***abstract theory of finite computation*** is a theory of computation consisting of these executable processes.

Theorem 1.8 (Church's Thesis)**.**

1. *In any abstract theory of finite computation, $\mathscr{C}$, the $\mathscr{C}$-partial computable function $f : \mathbb{N}^k \to \mathbb{N}$ gives at most the partial recursive functions.*
2. *Any informal description of an executable process starting with input in $\mathbb{N}^k$ and output in $\mathbb{N}$ or never halting is equivalent to a register machine, so we don't need to worry about all the details of the machine.*
3. *There is a total algorithm, that, given the encoding (e.g. shortlex) of a description of an algorithm, returns a code for a register machine that carries out this process.*

This is not so much one theorem as many different independent theorems. It has however been proven that all the following abstract theories of finite computation are equivalent:

- Church's λ-calculus
- Turing machines

- Register machines
- Standard languages
- Quantum/DNA-computers

From now on, we will refer to these three statements as "Church's thesis" or even just "Church".

Lemma 1.9. *Let $h : \mathbb{N} \to \mathbb{N}$ be:*

$$h(n) = \begin{cases} f_{n,1}(n) + 1 & \textit{if } n \textit{ codes a program and } f_{n,1}(n) \textit{ defined} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Then h is partial recursive.

Proof. For input n, check if n codes a program - this is total recursive. If so, run the program with input n. If it then halts, add 1 to R_1 and halt, and so by Church h is partial computable = partial recursive. ☐

Recursively Enumerable Sets

Given a partial recursive function $f : \mathbb{N}^k \to \mathbb{N}$ with domain $X \subseteq \mathbb{N}^k$, suppose we input 7. If $f(7) \downarrow$, then we can run the machine and get the answer within finite time. However, if $f(7) \uparrow$, then we will be waiting forever. By the halting problem, there is no way to know in advance what will happen.

Now suppose we ask: "Does f halt on either 7 or 9?" Then the answer is yes, but if we are unlucky and start with 9, we will never know if we naïvely compute $f(9)$, then $f(7)$.

So instead, we zig-zag: we do one step of each alternately.
Let $P_n^t(x)$ be the t^{th} step of P_n with input x. Then we can do:

$$\begin{tikzcd}
P_n^1(7) \arrow[r] & P_n^1(9) \arrow[dl] \\
P_n^2(7) \arrow[r] & P_n^2(9) \arrow[dl] \\
P_n^3(7) \arrow[r] & P_n^3(9) \arrow[dl, dotted] \\
{} &
\end{tikzcd}$$

We can clearly extend this to any set of finite size. We can even do infinite sets, by following a path similar to in the textbook enumeration of $\mathbb{Q}$:

$$\begin{tikzcd}
P_n^1(1) \arrow[r] & P_n^1(2) \arrow[dl] & P_n^1(3) \arrow[d] \\
P_n^2(1) \arrow[d] & P_n^2(2) \arrow[ur] & P_n^2(3) \arrow[dl, dotted] \\
P_n^3(1) \arrow[ur] & {} &
\end{tikzcd}$$

We can even alternate between different machines, and this process can be extended to any countable set. Then by Church, we can write a program that returns 1 for input $x \in \mathbb{N}^k, k < \infty$ if some partial recursive function f halts on input of x.

We say a set $E \subseteq \mathbb{N}^k$ is ***recursively enumerable*** if the function

$$\phi_E(n) := \begin{cases} 1 & n \in E \\ \uparrow & \text{else} \end{cases}$$

The idea behind this definition is that, compared to a recursive set, here we can only say that x is in E, whereas in a recursive set we can say if $x \in E$ or $x \notin E$. A consequence of this is that, by applying the above process for ϕ_E on all of $\mathbb{N}^k$, we will eventually get out all the elements of E, but we will not know when this has happened (indeed, it might take infinitely long to get all the elements, but we will eventually be notified that any given element is in E), so we can "recursively enumerate" E. Conversely, recursive sets are often called ***decidable*** - we can always decide whether or not $x \in E$. We will cement this discussion into a few theorems now:

Theorem 1.10. *The following are equivalent for any subset $E \subseteq \mathbb{N}$:*

1. *E is the range of some partial recursive function on some number of variables, i.e.* $E = \{f_{n,k}(\mathbf{x}) : \mathbf{x} \in \mathbb{N}^k\}$
2. *E is the domain of definition of some partial recursive function on* 1 *variable, i.e.:* $\exists\, n$ *s.t.* $E = \{m \in \mathbb{N} : f_{n,1}(m) \downarrow\}$
3. *E is recursively enumerable*
4. *The function* $\psi_E(n) = \begin{cases} n & n \in E \\ \uparrow & \textit{otherwise} \end{cases}$ *is partial recursive. i.e. we can enumerate the elements of E in a recursive fashion.*

Proof.

- *2. $\Longrightarrow$ 3.* Given a program P_n for computing $f_{n,1}$, modifying at end so it empties R_1 then adds 1.
- *3. $\Longrightarrow$ 4.* Similarly, we start by copying R_1 to R_N for large enough N, then at the end empty R_1 and copy R_N to R_1.
- *4. $\Longrightarrow$ 1.* Immediate as $f_{n,k} = \psi_E$ for some choice of n, $k = 1$.
- *1. $\Longrightarrow$ 2.* Given a program P_n for computing $f_{n,k} : \mathbb{N}^k \to \mathbb{N}$ with range E, do the following process Q by Church:

  Given an input $x \in \mathbb{N}$ for Q, run a zig-zag procedure having ordered $\mathbb{N}^k$ to compute $f_{n,k}(\cdot)$ for each input $\cdot \in \mathbb{N}^k$. Each time, $f_{n,k}(m_1, \ldots, m_k)$ halts, run the total subroutine comparing the output to x wand halt with output 1 if equal. If not, return to the zig-zag process. So Q will return 1 if x is in the range $f_{n,k}$, and if not will run forever, because we would never get an output equalling x.

$\square$

Properties of Recursively Enumerable Sets

In **1.10**(*1.*), can we have a total recursive function?

Theorem 1.11. *If $E \neq \emptyset$ and $E \subseteq \mathbb{N}$, then E is recursively enumerable if and only if E is the range of a total recursive function $f : \mathbb{N} \to \mathbb{N}$.*

Proof. For $E = \{e_0, \dots, e_{k-1}\}$ finite, define $f(n) := \begin{cases} e_n & n < k \\ e_{k-1} & \text{otherwise} \end{cases}$. This is total recursive.

In the case of infinite $E \subseteq \mathbb{N}$ which is the domain of a partial recursive function $g : \mathbb{N} \to \mathbb{N}$, run a zig-zag process to perform the steps of computing $g(0), g(1), \dots$. Define a total function $f : \mathbb{N} \to \mathbb{N}$ as $f(0)$ is the first input number i for which the computation of $g(i)$ halts in this process (not necessarily the smallest), $f(1)$ the second, and so on. Thus the range of f is E and f is recursive by Church. $\square$

So each program P_n corresponds to a recursively enumerable set which is the domain of $f_{n,1}$. Thus let $W_n = \{x \in \mathbb{N} : f_{n,1}(x) \downarrow\}$ if n encodes a program, and the empty set otherwise. We say W_n is the ***n*^th^ *recursively enumerable set***, though the labelling is not unique, i.e. it could be that $W_n = W_m$ for some $n \neq m$.

Every recursive subset is recursively enumerable by definition. However:

Theorem 1.12. *For any $E \subseteq \mathbb{N}$, E is recursive if and only if E and $\mathbb{N} \setminus E$ is recursively enumerable.*

Proof.

- $\underline{\implies}$ As χ_E is total recursive, ϕ_E is partial recursive because when χ_E halts, if the output is 1 then halt, otherwise enter an infinite loop. We can do the same for $\mathbb{N} \setminus E$, to get $\phi_E, \phi_{\mathbb{N}\setminus E}$.
- $\underline{\impliedby}$ As ϕ_E and $\phi_{\mathbb{N}\setminus E}$ are both partial recursive, we can do a zig-zag process between the two. Then, if $\phi_E(x)$ halts, return 1, otherwise if $\phi_{\mathbb{N}\setminus E}(x)$ halts, return 0. Since $E \cup \mathbb{N} \setminus E = \mathbb{N}$, exactly one of these will halt, and so this gives χ_E which is total recursive.

$\square$

Theorem 1.13.

1. *If $I \subseteq \mathbb{N}$ is recursively enumerable, then the union $\cup_{n\in I} W_n$ is recursively enumerable.*
2. *If $J \subseteq \mathbb{N}$ is finite, then $\cap_{n\in J} W_n$ is recursively enumerable.*

Proof.

1. We want a process that, given an input $x \in \mathbb{N}$, halts if $x \in \cup_{n\in I} W_n$, and not if $x \notin \cup_I W_n$.

   By a big zig-zag process, start outputting elements n of I, test if n codes a program P_n, move on if not but if so, start to run P_n on this input x. One of these subprocesses halts if and only if $x \in W_n$ for some $n \in I$, in which case halt the whole process. However, if $x \notin \cup_I W_n$, then the process will run forever.
2. First check all the $j \in J$ code a program, else we have $\emptyset$. Then run these $|J|$ programs in parallel, each with input x. If/when all of these halt, then halt. Then we halt if and only if $x \in \cap_J W_n$.

$\square$