This function is just a for loop, initialized at \( n \), then sets the iterator variable \( m \) each time. At the end of the for loop (PR), we add one (sigma).

\[
\begin{align*}
  f & \equiv 6 \left( \mathbb{P}(\mathbb{R}, \mathbb{R}) \right) \\
(3,3,0) & \equiv g(n) \\
& \equiv r \equiv m \\
\{(3,3,3) & \equiv 3 \\
\{(3,3,2) & \equiv 2 \\
\{(3,3,1) & \equiv 1 \\
\{(3,3,0) & \equiv 0 \\
& \equiv m+1 \\
& \equiv m = 0.
\end{align*}
\]

We still need to add the sigma,

\[
\sum_{(3,3,1)} = 44
\]

\( \leq, =, \) and \( \leq \) are primitive recursive.

We prove this by inducting on the definition of a primitive recursive function. A primitive recursive function is any function which can be obtained from \( 0, \neg, \mathbb{T} \) by using composition "0" and Primitive Recursion (PR).

By showing that relations are compositions of PR functions, by definition we prove that they themselves are PR functions.

\[
A \leq B \equiv a = b = 0
\]

We must show "different" is PR

\[
\begin{align*}
diff(a, 0) & \equiv a \\
diff(a, m+1) & \equiv prev(diff(a, m))
\end{align*}
\]

We must show "previous" is PR

\[
\begin{align*}
prev(0) & \equiv 0 \\
prev(m+1) & \equiv m
\end{align*}
\]

\[A \leq B \equiv B : A > 0\]

We already showed \( = \) is PR

We must show Greater than 0 is PR

\[
\begin{align*}
greaterThan(0) & \equiv m \\
greaterThan(0, m+1) & \equiv \neg prev(m+1)
\end{align*}
\]

We must show "not" is PR

\[
\begin{align*}
\neg(0) & \equiv 1 \\
\neg(m+1) & \equiv 0
\end{align*}
\]

We must show "and" is PR

Just logical disjunction.

\[
\begin{align*}
0 \cdot 0 & \equiv 0 \\
0 \cdot 1 & \equiv 0 \\
1 \cdot 0 & \equiv 0 \\
1 \cdot 1 & \equiv 1
\end{align*}
\]

We must show "multiplication" is PR

\[
\begin{align*}
mult(0,0) & \equiv 0 \\
mult(0, m+1) & \equiv \neg (m+1) \\
mult(m+1, 0) & \equiv 0 \\
mult(m+1, m+1) & \equiv mult(m, b) + a
\end{align*}
\]

We must show "addition" is PR

\[
\begin{align*}
\add(0,0) & \equiv a \\
\add(a, b+1) & \equiv si(a, b)
\end{align*}
\]

We must show "subtracting" is PR

\[
\begin{align*}
\subtract(0, a) & \equiv a \\
\subtract(m+1, a) & \equiv \neg a
\end{align*}
\]
4) Every P.R. function is by definition μ-recursive.

A μ-recursive function is any function which can be obtained from 0, O, and T1, by using composition, ε, P.R. and μ-recursion.

By definition, P.R. functions are μ-recursive functions.

Not every μ-recursive function is Primitive Recursive.

Church-Turing Thesis: Anything that can be computed on a physical device can also be computed by a μ-recursive function

In essence, any computable function is μ-recursive.

To provide a counter example, we consider a while loop.

Primitive Recursive functions are always defined.

Consider the function

\[ f(n) = \begin{cases} 3 & \text{if } n = 2 \\ 2 & \text{if } n = 5 \\ \text{undefined otherwise} \end{cases} \]

This is clearly not a P R function, since it can be undefined for some inputs.

This is a μ recursive function, but not a P.R. function.

\[ f(n) = \mu m. \ (m=2 \land m=3) \lor (n=5 \land m=2) \]
\[ a - b = a - b - 0 \]
\[ = \text{eqTo0}(\text{sub}(a, b)) \]
\[ \text{PR} \quad \text{PR} \]

So \( a = b \) is \( \text{PR} \)

\[ a \leq b \equiv \neg(\text{gt0}0(\text{sub}(a, b))) \]
\[ \text{PR} \quad \text{PR} \quad \text{PR} \]

So less than or equal is \( \text{PR} \)

4. **Every PR function is effectively recursive**

By definition

Every function which can be computed from \( \emptyset, 0, \text{PR} \)
by using \( \emptyset, \text{PR} \) and recursion.

So every PR function is effectively recursive.

*Not every effectively recursive function is PR.*

- We introduce the notion of a code for a PR function.
  - A natural \# can be assigned to every PR function.

Every PR function is represented by an expression
- Then using PSLIL code to translate it into 0s & 1s
- In order to make the expression of the PR unique we code 1 to the front.

There exists an algorithm that given a code \( c \)
- Checks whether \( c \) is a valid code for a PR function.
- If yes produces a Java code \( f_c \) which computes this function.

**Proof:** Let us define a function \( f \) as follows:

In particular

For \( n = c \) \( f(0) = ff \)

By definition of \( f \),

Since \( c \) is a valid code
\[ f(c) = f_c(c) + 1 \]

\( 0 = 1 \)
\[
\frac{1}{2} (a, b, c) = \min \left\{ a = b = c \land m = 1 \right\} \lor \left( a \neq b \land b + c \land a + c \lor a \neq a \land m = 0 \right) \]

\[
(f(a, b, c)) = \begin{cases} 
1 & \text{if } a = b = c \\
0 & \text{if } a \neq b, b + c, a + c \\
\text{undefined} & \text{elsewhere} 
\end{cases}
\]

6) The union of two R.E. sets A and B is R.E. i.e. \( A \cup B \) is R.E.

**Proof:** We construct an algorithm to print every element in \( A \cup B \)

Since A is R.E., there exists an algorithm that eventually prints all members of A. (alg A)

Since B

Algorithm:

Run alg A for 1 hr
Run alg B for 1 hr
Run alg A for 2 hr
Run alg B for 2 hr

We can see that their union will be printed, assuming we don’t care for repeats

<table>
<thead>
<tr>
<th>Alg</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alg A</td>
<td>1</td>
</tr>
<tr>
<td>Alg B</td>
<td>1</td>
</tr>
<tr>
<td>Alg A</td>
<td>2</td>
</tr>
<tr>
<td>Alg B</td>
<td>2</td>
</tr>
<tr>
<td>Alg A</td>
<td>3</td>
</tr>
<tr>
<td>Alg B</td>
<td>3</td>
</tr>
<tr>
<td>Alg A</td>
<td>4</td>
</tr>
<tr>
<td>Alg B</td>
<td>4</td>
</tr>
</tbody>
</table>

It will be printed at moment 8, corresponding to the union algorithm already at moment 5

5 printed
5 printed
set $S \cup \overline{S}$ are both RE. Then $S$ is decidable.

This means there exists an algorithm that prints all elements of $S \cup \overline{S}$ and sense $\overline{S}$ and prints all elements not in $S$. This means a number $n$ is either in $S$ or $\overline{S}$.

We can find this in $S$ or not this way.

Run also to print elements from $S$ and from $\overline{S}$.

If $n$ has been printed? repeat & print more if it does, if $S$ is infinite by $\overline{S}$.

<table>
<thead>
<tr>
<th>number</th>
<th>7</th>
<th>guardian</th>
<th>7</th>
<th>guardian</th>
</tr>
</thead>
<tbody>
<tr>
<td>S set</td>
<td>S</td>
<td>S</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

This algorithm will think at 7 does not belong.?
we can show this by first deciding program $p \leq 1$, data $d \leq 1$ for time $t = 1$ if a Lea$t$ combination halts we print it, we then continue for $p \leq 2$, $d \leq 2$, $t = 2$ testing all combinations added at 2 line steps. Continue in this fashion.

$\begin{align*}
\forall 1 \leq i, d \leq 1, t = 1 \\
(0,0) &\quad \text{first line} \\
(0,1) &\quad \text{next line} \\
(1,0) &\quad \text{third line} \\
(1,1) &\quad \text{last line} \\
\end{align*}$

$\begin{align*}
\forall 2 \leq i, d \leq 2, t = 2 \\
(0,0) &\quad \text{17 newest} \\
(0,1) &\quad \text{12 newest} \\
(1,0) &\quad \text{2 newest} \\
(1,1) &\quad \text{1 newest} \\
(2,0) &\quad \text{2 newest} \\
(2,1) &\quad \text{2 newest} \\
(2,2) &\quad \text{2 newest} \\
\end{align*}$

The pair $(2,2,1)$ will be printed after 17 times.
7) \( S \) is R.E. 
- \( \overline{S} \) is R.E.

Then \( S \) is decidable

To prove we build an algorithm that computes \( x_S \) characteristic function of \( S \)

To run Alg-5 for \( \overline{3} \) if \( n \) appeared we know if \( n \in \overline{S} \) or \( n \in S \)

\[
\begin{array}{c|c|c}
\text{Alg-5} & 1 & 2 \\
\hline
\text{Alg-5} & 1 & 2 \\
\text{Alg-5} & 2 & 2 \\
\text{Alg-5} & 3 & 3 \\
\text{Alg-5} & 3 & 3 \\
\end{array}
\]

The algorithm will decide whether \( 7 \) belongs to \( S \) just after moment 6.

8) \( H \) is the halting set

The halting set is R.E. meaning there is an algorithm that each element of the halting set will eventually be printed and only elements of the halting set are printed

We denote the halting set as

\[ H = \{ p \mid p \text{ halts on input } d \} \]

To prove \( H \) is R.E. we build an algorithm that prints all elements of \( H \)

Take all \( p \leq 1, d \leq 1, t = 1 \), run for 1hr. If any of them halts, print (p,d)

Take all \( p \leq 2, d \leq 2, t = 2 \), run for 2hrs.

Take all \( p \leq 3, d \leq 3, t = 3 \), run for 3hrs.

\[
\begin{array}{c|c|c}
\text{p} & \text{d} & \text{t} \\
\hline
1 & 1 & 10 \text{ hrs} \\
0 & 0 & 0 \text{ hrs} \\
0 & 1 & 0 \text{ hrs} \\
0 & 1 & 1 \text{ hrs} \\
1 & 0 & 1 \text{ hrs} \\
1 & 1 & 0 \text{ hrs} \\
2 & 1 & 1 \text{ hrs} \\
2 & 1 & 0 \text{ hrs} \\
2 & 1 & 1 \text{ hrs} \\
2 & 1 & 0 \text{ hrs} \\
\end{array}
\]

\[ 18 \text{ hrs} \quad \text{It will be printed at moment of time 18} \]
4. A \rightarrow \text{decidable} \quad B \rightarrow \text{RE} 

By theorem 11: every decidable set is RE.
So A is RE.

Then by Theorem 12: union of RE sets is RE.

A \cup B \text{ is RE.}

If B = \emptyset \text{ and } A = \emptyset

then \[ B \text{ is RE but not decidable. (By theorem 15).} \]
It is a counter example because we are giving a decidable set \( A = \emptyset \) (by theorem 4 it is decidable) and a RE but not decidable set \( B = \emptyset \) therefore the union of A-\text{decidable and B-RE sets} is not always decidable.

10. Theorem: no algorithm is possible that given a program \( P \) checks whether \( P \) always returns the \( n \)-th prime.

\( f_c(p) = \begin{cases} 1 & \text{if for every } n \text{ } P(n) \text{ halts and } P(n) = n^{\text{th prime}} \\ 0 & \text{if for every } n \text{ } P(n) \text{ halts and for some } n \text{ } P(n) \neq n^{\text{th prime}} \end{cases} \)

Proof: we can prove that if \( f \) exists we can use \( f_c \) to design a zero checker.

\[ \text{P(n) = } \emptyset \iff q(n) = n^{\text{th prime}} \]

\[ \begin{array}{c}
\text{P} \\
\text{\quad} \\
\downarrow \\
\text{q}
\end{array} \quad \begin{array}{c}
\forall n (P(n) = \emptyset) \\
\quad \quad \\
\forall n (q(n) = n^{\text{th prime}})
\end{array} \]

\[ \forall n (P(n) + n^{\text{th prime}} = q) \]

\[ n^{\text{th prime}} \text{ checker}(q) = \exists \\
\forall n (q(n) = n^{\text{th prime}}) \iff \forall n (P(n) + n^{\text{th prime}} = n^{\text{th prime}}) \]

\[ \exists \forall n (P(n) = \emptyset) \iff \forall n (P(n) + n^{\text{th prime}}) = n^{\text{th prime}} \]

\[ \text{zerochecker}(P) = n^{\text{th prime}} \text{ checker}(P + n^{\text{th prime}}) \]
9) The union of a decidable set and a R.E. set is always R.E.

Proof
Let set A be decidable.
Let set B be R.E.

According to Theorem 12, the union of R.E. sets is R.E. and we proved this in problem 6 of this test.

A, being decidable, is also R.E. (*Theorem 1*)

Proof: Easiest algorithm

```
int n = 0;
while (true) {
    if (n E A) {
        print n;
    }
    n++;
}
```

Thus A and B are R.E. By Theorem AUB is R.E.

Such a union is. AUB is not always decidable, To show this we provide a counter example.

By theorem, \( \emptyset \) is decidable. Let \( A = \emptyset \)

We showed that \( H \) is R.E. (problem 8), let \( B = H \)

By Theorem 4, we know the union of decidable sets is decidable

\[ n E AUB \iff n E A \lor n E B \]

So for AUB to be decidable, we would need B to be decidable.

We know is impossible \( (H \) is not decidable).)

Thus, \( AUB \) is not always decidable.
It is not possible to have a prime checker, a program that checks whether an input program always computes the n-th prime number.

\[ \text{prime checker}(p) = \begin{cases} 1 & \text{if } \text{pen} \text{ halts and } p(n) = n \text{-th prime number} \\ 0 & \text{if } \text{pen} \text{ halts and } \exists n : p(n) \neq n \text{-th prime number} \\ \text{P-halts otherwise} \end{cases} \]

We will prove prime checkers don't exist by applying problem reduction and building a zero checker out of it.

**Lemma:** Zero checkers are not possible.

Let's assume that a prime checker exists. Let's build a zero checker out of it.

\[ P \xrightarrow{\text{prime checker}} H(n(p(n) = 0)) \]

To build this, we want a function q, which, given input n, its output is the n-th prime number and also that pen = 0, such that \( p(n) = 0 \Rightarrow q(n) = n \)-th prime number.

We define a specific auxiliary program q:

\[ q(n) = p(n) + n \text{-th prime number} \]

Using our "prime checker" on q, we can see that:

\[ \text{prime checker}(q) = 1 \iff H(n(q(n) = n \text{-th prime number}) \iff H(n(p(n) + n \text{-th prime number}) = n \text{-th prime number}) \]

\[ \iff H(p(n) = 0) \]

\[ \text{prime checker}(q) = 1 \iff H(n(p(n) = 0)) \]

\[ \text{prime checker}(q) = 0 \iff \exists n : p(n) \neq 0 \]

We have effectively built a zero checker out of a prime checker. We know that zero checkers don't exist, so this leads to a contradiction, and the assumption is false.
\[ f(n) = \begin{cases} n+1 & \text{if } n = 0 \\ f(n-1) + 1 & \text{if } n > 0 \end{cases} \]

We have:

\[ g(n) = \begin{cases} 0 & \text{if } n = 0 \\ n-1 & \text{if } n > 0 \end{cases} \]

We have either:

1. \[ \#\# \ldots \# \]
2. \[ \#\# \ldots \# \]

or

\[ \#\# \ldots \# \]

We want:

\[ \#\# \ldots \# \]

Where

\[ f(g(1)) = \begin{cases} 1 & \text{in add} \\ \#\# \# \ldots \# & \text{in subadd} \end{cases} \]

\[ \#\# \# \ldots \# \]

\[ \text{check}_2 R \]

\[ \text{check}_2 L \]

\[ \text{sub}_2 R \]

\[ \text{case}_2 L \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]

\[ \text{gohome}_2 \# \]

\[ \text{gohome}_2 L \]

\[ \text{gohome}_2 R \]
For general composition, instead of halting, jump to state 12, (change state names as well).

Could be

But odd.