Lessons touched by this meeting according to schedule

- 6. 28/10/2024
    - Generation of computable functions
        - Primitive recursion and examples [§2.4]
        - Definition by cases. Algebra of Decidability. Bounded sums and products.
        - Bounded quantification [§2.4.6, §2.4.7, §2.4.10]
        - Bounded minimalisation [§2.4.12, §2.4.13, §2.4.14, §2.4.15]
- 7. 29/10/2024
    - Generation of computable functions
        - Unbounded minimalisation [§2.5]
        - Computability of the inverse function. Finite functions and their computability.
    - Partial recursive functions [§3.1, §3.2, §3.7]
        - Definition
        - The class of partial recursive functions coincide with the class of URM-computable functions [statement and some ideas]

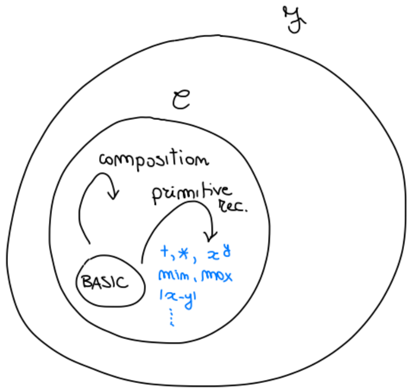
Class $\mathcal{C}$ of URM-computable functions

* contains the BASIC FUNCTIONS

(a) zero
(b) successor
(c) projections

* closed under

(1) (generalised) composition
(2) primitive recursion
(3) (unbounded) minimalisation

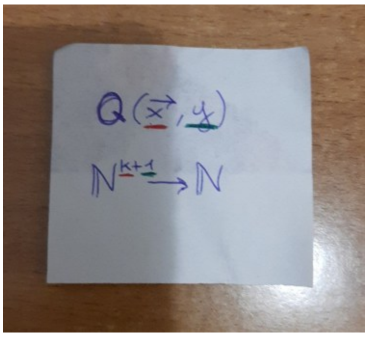


What we learn as important concepts here:

- If a function is defined by cases, then every part can be also seen as decidable predicate, so everything is computable
    - Meaning of vectors defined functions
    - Meaning of characteristic functions

- Algebra of decidable predicates (important when you will see Rice-Shapiro)
    - $\neg Q \rightarrow negated\ sign\ of\ X_Q$
    - $Q_1 \wedge Q_2 \rightarrow product$
    - $Q_1 \vee Q_2 \rightarrow sum$

Based on the notation shown in the image, this appears to be representing:



1. Q(x̄,y) - A predicate or relation Q taking a vector x̄ (representing multiple inputs x1,...,xk) and y as arguments
2. Nk+1 → N - A function mapping from k+1-dimensional natural numbers to natural numbers

Exercises on the predicates become important ahead:

a. Provide the definition of a decidable predicate.

b. Provide the definition of a semi-decidable predicate.

c. Is it true that if the predicates $P(\vec{x}), Q(\vec{x}) \subseteq \mathbb{N}^k$ are decidable then also their logical implication $R(\vec{x}) \equiv P(\vec{x}) \Rightarrow Q(\vec{x})$ is decidable? Does the same result hold for semi-decidability, i.e., if $P(\vec{x}), Q(\vec{x})$ are semi-decidable then $R(\vec{x}) \equiv P(\vec{x}) \Rightarrow Q(\vec{x})$ is semi-decidable?

3. Assume that $P(\vec{x}), Q(\vec{x}) \subseteq \mathbb{N}^k$ are decidable. Then also their logical implication $R(\vec{x}) \equiv P(\vec{x}) \Rightarrow Q(\vec{x})$ is decidable. In fact, $P(\vec{x}) \Rightarrow Q(\vec{x})$ is logically equivalent to $\neg P(\vec{x}) \vee Q(\vec{x})$ and we know that decidable predicates are closed by negation and disjunction.

For the second part of the question, since in particular, if we take $Q(\vec{x}) \equiv \mathit{false}$ and $P(\vec{x})$, semi-decidable then $R(\vec{x}) \equiv P(\vec{x}) \Rightarrow \mathit{false} \equiv \neg P(\vec{x})$. Hence, since semi-decidable predicates are not closed by negation, the property does not generalise to semi-decidability (for an example, take $P(x) =$ "$x \in K$" which is decidable, but $Q(x) \equiv \mathit{false}$ and $P(x) \equiv$ "$x \notin K$" is not semi-decidable).

Sum and product need to be <u>bounded</u>, which means it has to hold for all subpredicates. Clarification on notation being used:

(1) For $g(\vec{x},y) = \Sigma z<y\ f(\vec{x},y)$: The function can be defined by primitive recursion as:

$g(\vec{x},0) = 0$
$g(\vec{x},y+1) = g(\vec{x},y) + f(\vec{x},y)$

Both the base case and step case use computable functions:

- The constant 0 function is computable (basic function)
- f is computable by hypothesis
- Addition is computable
- The composition of computable functions is computable

Therefore g is computable by primitive recursion.

(2) For $h(\vec{x},y) = \Pi z<y\ f(\vec{x},y)$: Similarly, this function can be defined by primitive recursion as:

$h(\vec{x},1) = 1$
$h(\vec{x},y+1) = h(\vec{x},y) \cdot f(\vec{x},y)$

The components are computable:

- The constant 1 function is computable (basic function)
- f is computable by hypothesis
- Multiplication is computable
- The composition of computable functions is computable

Exercise (2024-02-16)

Define the class of primitive recursive functions. Using only the definition show that the function $a : \mathbb{N}^2 \to \mathbb{N}$ defined below is primitive recursive

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

Solution: The class of primitive recursive functions is the least class of functions $\mathcal{PR} \subseteq \bigcup_k(\mathbb{N}^k \to \mathbb{N})$ containing the base functions (zero, successor, projections) and and closed under composition and primitive recursion.

In order to show that f is primitive recursive observe that it can be defined as

$$\begin{cases} a(x,0) & = 0 \\ a(x,y+1) & = sg(x) \end{cases}$$

where $sg : \mathbb{N} \to \mathbb{N}$, is the sign function, which can be defined by primitive recursion as

$$\begin{cases} sg(0) & = 0 \\ sg(y+1) & = 1 \end{cases}$$

Exercise 2

Define the class of primitive recursive functions. Using only the definition show that the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(y) = 2y + 1$ is primitive recursive.

The class of primitive recursive functions (PR) is the least class of functions containing:

(a) The zero function: z : N → N, z(x) = 0

(b) The successor function: s : N → N, s(x) = x + 1

(c) The projection functions: U_i^k : N^k → N, U_i^k(x_1,...,x_k) = x_i

and closed under:

1. Composition
2. Primitive recursion

Let's define it using composition of primitive recursive functions:

1. First define `double(y) = 2y` by primitive recursion:

`double(0) = 0`

`double(y+1) = double(y) + 2 = (double(y) + 1) + 1`

1. This function is primitive recursive as it uses only composition and primitive recursion from basic functions.
2. Then we can express `f(y) = 2y + 1` as: `f(y) = s(double(y))` Where s is the successor function (basic function) and double(y) is primitive recursive as shown above.

Since f is obtained by composition of primitive recursive functions (s and double), and composition preserves primitive recursion, we conclude that `f(y) = 2y + 1` is primitive recursive.

Esercizio 1

Dare la definizione dell'insieme $\mathcal{PR}$ delle funzioni primitive ricorsive e dimostrare che è primitiva ricorsiva la funzione $cpr : \mathbb{N}^2 \to \mathbb{N}$ definita come

$$cpr(x, y) = |\{p \mid x \leqslant p < y \ \wedge \ p \text{ primo}\}|,$$

ovvero $cpr(x, y)$ è il numero di primi nell'intervallo $[x, y)$ (si può assumere che somma $+$ e differenza $\dot{-}$ tra numeri naturali, nonché la funzione caratteristica dell'insieme dei numeri primi χ_{Pr} siano primitive ricorsive, senza provarlo). [Suggerimento: Può essere conveniente considerare inizialmente la funzione $cpr'(x, k) = |\{p \mid x \leqslant p < x + k \ \wedge \ p \text{ primo}\}|$]

Official solution:

Soluzione: Si definisce $cpr' : \mathbb{N}^2 \to \mathbb{N}$, tale che $cpr'(x, k) = |\{p \mid x \leqslant p < x+k \wedge p \text{ primo}\}|$, per ricorsione primitiva, utilizzando solo funzioni primitive e loro composizioni:

$$\begin{cases} cpr(x, 0) = 0 \\ cpr(x, k+1) = cpr(x, k) + \chi_{Pr}(x + k) \end{cases}$$

quindi si osserva che $cpr(x, y) = cpr'(x, y \dot{-} x)$, composizione di funzioni primitive ricorsive è ancora primitiva ricorsiva. □

Alternatively:

We can define `cpr(x,y)` by adapting the counting approach:

1. First define an auxiliary function count: $N^3 \to N$ that counts primes in a range while building it:

`count(x,y,0) = 0 count(x,y,z+1) = count(x,y,z) + χ_Pr(z) · χ_LE(x,z) · χ_LT(z,y)`

where:

- `χ_Pr` is characteristic function of primes (given)
- `χ_LE(a,b) = 1 if a ≤ b, 0` otherwise (primitive recursive)
- `χ_LT(a,b) = 1 if a < b, 0` otherwise (primitive recursive)

count is primitive recursive because:

- Base case uses zero function
- Recursive step uses only primitive recursive functions and operations

2. Then `cpr(x,y) = count(x,y,y)`

This is primitive recursive as:

- count is primitive recursive (shown above)
- It uses only composition with the projection function U_2^2 for y

Therefore `cpr(x,y)` is primitive recursive.

We define a URM-Back machine as a URM-like machine with the usual $Z(n)$, $S(n)$, $T(n,m)$ and $J(n,m,t)$ instructions, but the $J(n,m,t)$ instruction is restricted to only backward jumps.

We want to show that the set $\mathcal{C}^B$ of URM-Back computable functions is equal to the set $\mathcal{C}$ of computable functions.

Solution:

1. First, let's prove CB ⊆ C: This is straightforward since URM-Back is a restricted version of URM. Any URM-Back program is already a valid URM program with the same semantics, therefore every URM-Back computable function is URM computable.
2. Now, let's prove C ⊆ CB: We need to show that any URM program can be transformed into an equivalent URM-Back program. The key is to simulate forward jumps using backward jumps.

Given a URM program P of length n, we can construct a URM-Back program P' as follows:

a) Transform each forward jump instruction J(m,n,t) into:

- Initialize a counter register Rc to n-t (where n is program length)
- Use a loop with backward jumps that:
  - Decrements Rc
  - Jumps back to decrement again if Rc > 0
  - When Rc = 0, we've effectively moved forward t steps

b) Specifically, for a forward jump J(m,n,t), replace it with:

```
T(m,k)       // Save comparison values

T(n,k+1)

Z(k+2)       // Initialize counter

T(1,k+3)     // Save original contents

...

J(k,k+1,p)   // Compare original values

J(1,1,q)     // Backward jump to decrement counter
```

where p points to the start of the counter decrement sequence and q points to the corresponding target instruction.

This transformation preserves the semantics of the original program, uses only backward jumps and is clearly effective (can be done algorithmically). Therefore C = CB.

Now, for an important concept: bounded minimalization.

Given a total function $f : \mathbb{N}^{k+1} \to \mathbb{N}$, we define a function $h : \mathbb{N}^{k+1} \to \mathbb{N}$ as follows:

$$h(\vec{x}, y) = \mu z < y . f(\vec{x}, z) = \begin{cases} \text{miniumum } z < y \text{ such that } f(\vec{x}, z) = 0 & \text{if it exists} \\ y & \text{otherwise} \end{cases}$$

In plain English, this means we are only using bounded operations (sums, products):

1. We're looking for the smallest value of z that is less than y where $f(\vec{x},z) = 0$
2. If no such z exists (i.e., $f(\vec{x},z) \neq 0$ for all $z < y$), then h returns y

The key differences from unbounded minimalization are:

- The search is bounded by y
- The function h is always total because:
  - Either we find a $z < y$ where $f(\vec{x},z) = 0$
  - Or we hit the bound y and return it
- We don't have the issue of divergence that exists with unbounded minimalization

a) $D(x) = \textit{number of divisors of } x$

b) $Pr(x) = \begin{cases} 1 & x \textit{ is prime} \\ 0 & \textit{otherwise} \end{cases}$ *(x prime is decidable)*

PROOF. a) $D(x) = \sum_{y \leqslant x} div(y,x)$

b) $Pr(x)$ is 1 if $x > 1$ and is divided only by 1 and itself

$$Pr(x) = \begin{cases} 1 & D(x) = 2 \\ 0 & \text{otherwise} \end{cases} = \bar{sg}(|D(x) - 2|)$$

Fibonacci function introduces the concept of π → pair encoding, basically at an inductive step defining recursion on a couple of previously defined values

Unbounded minimalization ($\mu y.f(\bar{x},y)$) searches for the smallest y where $f(\bar{x},y) = 0$, but unlike bounded minimalization, this search has no upper limit. This is equivalent to a while loop:

```
y = 0
while (f(x̄,y) ≠ 0) {
    y++
}
return y
```

The key difference from bounded minimalization is that this search might:

- Never find a solution (no y makes $f(\bar{x},y) = 0$)
- Never terminate (hit a value where $f(\bar{x},y)$ is undefined)

$$f(x) = \begin{cases} \sqrt{x} \text{ se } x \text{ è un quadrato} \\ \uparrow \text{ altrimenti} \end{cases}$$

```
f(x,y) = |x - y²|
μy.f(x,y) = {
    √x   if x is a perfect square
    ↑    otherwise
}
```

This works because:

- If x is a perfect square (like 16), it will find y (4) where $y^2 = x$
- If x is not a perfect square (like 5), it will never find such a y
- The function continues searching indefinitely until it either finds a solution or determines none exists - it captures the idea of "searching until a condition is met" without a known upper bound

Comment on:

DEFINITION 7.1 (Partially recursive functions). The class $\mathcal{R}$ of **partially recursive functions** is the least class of partial functions on the natural numbers which contains

(a) zero function;

(b) successor;

(c) projections

and **closed** under

(1) composition;

(2) primitive recursion;

(3) minimalisation.

We argue that the above is a well given definition.

DEFINITION 7.2 (Rich class). A class of functions $\mathcal{A}$ is said to be **rich** if it includes (a),(b) and (c) and it is closed under (1), (2) and (3).

DEFINITION 8.1 (Primitive recursive functions). The class of *primitive recursive functions* is the smallest class of functions $\mathcal{PR}$ containing

(a) zero function

(b) successor

(c) projections

and closed under

(1) composition

(2) primitive recursion

Exercise 2

Give the definition of the class $\mathcal{PR}$ of primitive recursive functions. Show that the following functions are in $\mathcal{PR}$

1. $isqrt : \mathbb{N} \to \mathbb{N}$ such that $isqrt(x) = \lfloor \sqrt{x} \rfloor$;
2. $lp : \mathbb{N} \to \mathbb{N}$ such that $lp(x)$ is the largest prime divisor of x (Conventionally, $lp(0) = lp(1) = 1$.)

You can assume primitive recursiveness of the basic arithmetic functions seen in the course.

Solution:

1. The basic observation is that $isqrt(x)$ is largest y such that $y^2 \leqslant x$ and in turn this is the smallest y such that $y^2 > x$. In addition, it is immediate to realise that such a y is bounded by x, hence we get

$$isqrt(x) = \mu y < x + 1.\, ((y+1)^2 > x) = \mu y < x + 1.\, \overline{sg}(((y+1)^2 \dot{-} x))$$

2. Observe that, for $x > 1$, $lp(x)$ is surely smaller or equal to p_x. Hence one can count the prime divisors of x, restricting the search to $p_1, \ldots, p_x$:

$$count(x) = \Sigma_{i=1}^{x} div(p_i, x)$$

and then $lp(x) = p_{count(x)}$. The function needs to be adjusted for $x = 0$ and $x = 1$, where $count(x) = 0$ and thus $p_{count(x)} = 0$ while $lp(x) = 1$. This is easily done as follows:

$$lp(x) = p_{count(x)} + \overline{sg}(x \dot{-} 1).$$

Since we use only known primitive recursive functions, bounded sum and composition we conclude that lp is primitive recursive.

Alternatively, the idea can be to check explicitly the prime divisors of x, starting from p_x, then p_{x-1} and so on, stopping at the first. In detail, look for the smaller y, call it $i(x)$, such that p_{x-y} is a divisor of x.

$$i(x) = \mu y \leqslant x. \overline{sg}(div(p_{x-y}, x))$$

Then, whenever $x > 1$, $lp(x)$ $p_{x-i(x)}$ and the cases $x \leqslant 1$ must be treated separately as before:

$$lp(x) = p_{x-i(x)} \cdot sg(x \dot{-} 1) + \overline{sg}(x \dot{-} 1).$$

Taken from an Italian exam (2018-11-20-parziale):

Si consideri una variante della macchina URM, che comprende le istruzioni di salto, trasferimento e due nuove istruzioni

- *A(m,n)* che scrive nel registro m la somma dei registri m e n, ovvero $r_m \leftarrow r_m + r_n$;
- *C(n)* che scrive nel registro n il valore del suo segno negato, ovvero $r_n \leftarrow \overline{sg}(r_n)$.

Dire quale relazione sussiste tra l'insieme $\mathcal{C}'$ delle funzioni calcolabili con la nuova macchina e l'insieme $\mathcal{C}$ delle funzioni calcolabili con la macchina URM. Sono uno contenuto nell'altro? L'inclusione è stretta? Motivare le risposte.

Soluzione: Indichiamo con URM* la macchina modificata. Osserviamo che le istruzioni della macchina URM* sono codificabili nella macchina URM.

L'istruzione $I_j : A(m,n)$ si può sostituire con un salto alla seguente routine: detto q l'indice del primo registro non usato dal programma (quindi inizialmente a 0)

$$\begin{array}{lcl} SUB & : & J(n,q,j+1) \\ & & S(m) \\ & & S(q) \\ & & J(1,1,SUB) \end{array}$$

Allo stesso modo, indicando ancora con q l'indice di un registro non utilizzato, un'istruzione $I_j : C(m)$ si può sostuire con un salto alla subroutine

$$\begin{array}{lcl} SUB & : & J(n,q,ZERO) \\ & & Z(n) \\ & & J(1,1,j+1) \\ ZERO & : & S(n) \\ & & J(1,1,j+1) \end{array}$$

Più formalmente, si prova che $\mathcal{C}^* \subseteq \mathcal{C}$ dimostrando che, per ogni numero di argomenti k per ogni un programma P che utilizzi entrambi i set di istruzioni si può ottenere un programma URM P' che calcola la stessa funzione ovvero tale che $f_{P'}^{(k)} = f_P^{(k)}$.

La prova procede per induzione sul numero h di istruzioni A e C nel programma. Il caso base $h = 0$ è banale, dato che P con 0 istruzioni A e C è già un programma URM. Supposto vero il risultato per h dimostriamolo per $h+1$. Il programma P contiene certamente almeno una istruzione A o una istruzione C. Supponiamo che sia l'istruzione di indice j e sia una istruzione A.

$$\begin{array}{lcl} 1 & : & I_1 \\ & & \dots \\ j & & A(m,n) \\ & & \dots \\ \ell(P) & : & I_{\ell(P)} \end{array}$$

Costruiamo un programma P'', utilizzando un registro non riferito da P, $q = \max\{\rho(P), k\} + 1$

$$\begin{array}{lcl} 1 & : & I_1 \\ & & \dots \\ j & & J(1,1,SUB) \\ & & \dots \\ \ell(P) & : & I_{\ell(P)} \\ & & J(1,1,END) \\ SUB & : & J(n,q,ZERO) \\ & & Z(n) \\ & & J(1,1,j+1) \\ ZERO & : & S(n) \\ & & J(1,1,j+1) \\ END & : & \end{array}$$

Il programma P'' è tale che $f_{P''}^{(k)} = f_{P}^{(k)}$ e contiene h istruzioni di tipo A o C. Per ipotesi induttiva, esiste un programma URM P' tale che $f_{P'}^{(k)} = f_{P''}^{(k)}$ che quindi è il programma desiderato.

Se l'istruzione I_j è di tipo C si procede in modo completamente analogo, rimpiazzando l'istruzione con la sua codifica e usando l'ipotesi induttiva.

Per l'inclusione opposta $\mathcal{C} \subseteq \mathcal{C}^*$ si osserva, analogamente, che le istruzioni $Z(n)$ e $S(n)$ sono codificabili nella macchina modificata.

più precisamente, dato un programma P e detti q_1 e q_2 indici di registri non usati dal programma (quindi inizialmente a 0), si considera il programma

$C(q_1)$ `// fa in modo che` q_1 `contenga 1`
P'

dove P' è ottenuto da P sostituendo ogni istruzione $Z(m)$ con $T(q_2, m)$ e ogni istruzione $S(m)$ con $A(m, q_1)$. □

Esercizio 1

Si consideri una variante URM^p della macchina URM nella quale l'istruzione di azzeramento $Z(n)$ è sostituita dell'istruzione di predecessore $P(n)$ che decrementa il contenuto del registro n, ovvero $r_n \leftarrow r_n \dot{-} 1$. Indicato con $\mathcal{C}^p$ l'insieme delle funzioni calcolabili dalla macchina URM^p, stabilire la relazione tra $\mathcal{C}^P$ e l'insieme delle funzioni URM-calcolabili $\mathcal{C}$. Sono uno contenuto nell'altro? L'inclusione è stretta? Motivare le risposte.

Soluzione: Dato che l'istruzione $P(n)$ è codificabile nella macchina URM, chiaramente $\mathcal{C}^P \subseteq \mathcal{C}$. Più precisamente l'istruzione $I_j : P(n)$ si può sostituire con un salto alla seguente routine. Si indichi con q l'indice del primo registro non usato dal programma (quindi inizialmente a 0)

$$
\begin{array}{lll}
SUB & : & J(n, q, RIS) \\
 & & S(q+1) \\
LOOP & : & J(n, q+1, RIS) \\
 & & S(q) \\
 & & S(q+1) \\
 & & J(1, 1, LOOP) \\
RIS & : & T(q, n) \\
 & & J(1, 1, j+1)
\end{array}
$$

La routine controlla se il registro n contiene 0. In caso affermativo non c'è niente da fare. Altrimenti, con R_q che parte da 0 e R_{q+1} da 1, continua ad incrementare i due registri. Qunando R_{q+1} eguaglia R_n, avremo che R_q contiene il predecessore.

Più formalmente, si prova che $\mathcal{C}^p \subseteq \mathcal{C}$ dimostrando che, per ogni numero di argomenti k per ogni programma URM^p P si può ottenere un programma URM P' che calcola la stessa funzione ovvero tale che $f_{P'}^{(k)} = f_{P}^{(k)}$.

La prova procede per induzione sul numero h di istruzioni P nel programma P. Il caso base $h = 0$ è banale, dato che P con 0 istruzioni, è già un programma URM. Supposto vero il risultato per h dimostriamolo per $h + 1$. Il programma P contiene certamente almeno una istruzione P. Supponiamo che sia l'istruzione di indice j.

$$
\begin{array}{lll}
1 & : & I_1 \\
 & & \dots \\
j & & P(n) \\
 & & \dots \\
\ell(P) & : & I_{\ell(P)}
\end{array}
$$

Costruiamo un programma P'', utilizzando un registro non riferito da P, $q = \max\{\rho(P), k\} + 1$

$$
\begin{array}{lll}
1 & : & I_1 \\
 & & \dots \\
j & & J(1,1,SUB) \\
 & & \dots \\
\ell(P) & : & I_{\ell(P)} \\
 & & J(1,1,END) \\
SUB & : & J(n,q,RIS) \\
 & & S(q+1) \\
LOOP & : & J(n,q+1,RIS) \\
 & & S(q) \\
 & & S(q+1) \\
 & & J(1,1,LOOP) \\
RIS & : & T(q,n) \\
 & & J(1,1,j+1) \\
END & : &
\end{array}
$$

Il programma P'' è un programma URM^p tale che $f_{P''}^{(k)} = f_P^{(k)}$ e contiene h istruzioni di tipo P. Per ipotesi induttiva, esiste un programma URM P' tale che $f_{P'}^{(k)} = f_{P''}^{(k)}$ che quindi è il programma desiderato.

Vale anche l'inclusione opposta e quindi l'uguaglianza. Infatti l'istruzione $Z(n)$ può essere sostituita semplicemente da un'istruzione $T(q,n)$, dove q è un qualunque registro non usato dal programma e quindi a 0. Più precisamente, dato un programma URM P e un numero di argomenti fissato $k \in \mathbb{N}$, detto $q = \max\{\rho(P), k\} + 1$ l'indice del primo registro non usato e quindi inizialmente a 0, sostituendo in P ogni istruzione $Z(n)$ con l'istruzione $T(q,n)$, è un programma URM^p che calcola esattamente la stessa funzione.

Esercizio 1

Dare la definizione dell'insieme $\mathcal{PR}$ delle funzioni primitive ricorsive e, utilizzando esclusivamente la definizione, dimostrare che per ogni $k \geq 2$ è primitiva ricorsiva le funzione $sum_k : \mathbb{N}^k \to \mathbb{N}$ definita da $sum_k(x_1, \dots, x_k) = \Sigma_{i=1}^k x_i$.

First recall that PR is the class of primitive recursive functions containing:

- Zero function z(x) = 0
- Successor function s(x) = x + 1
- Projection functions U_i^k(x_1,...,x_k) = x_i and closed under composition and primitive recursion.

We'll proceed by induction on k:

Base case (k = 2): Define $sum_2(x_1,x_2)$ by primitive recursion on x_2:

```
sum₂(x₁,0)  = x₁  = U_1^2(x₁,0)

sum₂(x₁,y+1)  = s(sum₂(x₁,y))
```

This is primitive recursive as it uses only:

- Base case: projection function (basic)
- Recursive step: composition of successor (basic) with recursive call

Inductive step (k > 2): Assume sum_{k-1} is primitive recursive. Define:

$$\text{sum}_k(x_1, \ldots, x_k) = \text{sum}_2(\text{sum}_{k-1}(x_1, \ldots, x_{k-1}), x_k)$$

This is primitive recursive because:

- sum_{k-1} is primitive recursive (inductive hypothesis)
- sum_2 is primitive recursive (base case)
- Their composition is primitive recursive

Therefore, by induction, sum_k is primitive recursive for all $k \geq 2$.