Rovesti Gabriel

Computability simple (for real)

Summary

[Introduction 2](#_Toc148772627)

[Algorithms, effective procedures, non-computable functions 4](#_Toc148772628)

[URM Computability 10](#_Toc148772629)

[Decidable predicates and computability on other domains 22](#_Toc148772630)

[Generation of computable functions 25](#_Toc148772631)

# Introduction

Course reference page: <http://www.math.unipd.it/~baldan/Computability>

(This lesson is based on the only set of slides of the course, available as “Intro-en.pdf”)

We start by a simple reflection; can we give the enumeration of all numbers and store them efficiently? A suggestion might be, “rather than the phone number itself, you might store a program that generates the number”. So, instead of We can write

It isn’t convenient; there are *numbers*  such that, for all *program P* generating *n*, . These are defined as *random numbers*; we observe there are an infinite number of them. There is no program capable of determining whether a number is random or not, because such a program does not exist.

Exercise (coming from the 8th slide):

1. Prove that there are infinitely many random numbers
2. Prove there is no program able whether a number is random or not

Notes on the previous:

1. We see this as compression, as a function that takes a set of inputs and with a special property we take back the previous file, because we inject it; for this reason, we can’t compress every single file
2. We take a programming language of preference to try to prove this, but we miss the full theory course to prove that entirely; still, we can try

Solutions

My proof for the first question (not proof-corrected by any teacher, so to take with a grain of salt)

1. We define a set of functions , where there is a natural number (phone number) mapped to another phone number. This represents a set of functions.
   * We assume there can be only finitely random numbers, such as . We call this set .
   * We let as a natural number inside of
   * Let be the program generated by for
   * Let the program generated by for , where
   * Let’s extend the previous concept of here
     + For each , there exists a program such that such that generates with . This follows the definition of the compression function .
     + Because the compression should not lose data in computation, we consider a number random in such a way the compression function should be injective for
2. For each , we have:
   1. generating
3. Consider a new number not inside the set
4. If is random, there should be a program generated by for such that:
   1. generates
5. However, since is not in the set , must have a different representation from . This is because if were the same as any of the , it would generate a number from the set , which is a contradiction.
6. By this, we prove there is a new program that generates a new program that generates a number not inside the original set and contradicts the assumption there are only finitely many random numbers

Proof for the second question (again, not proof-corrected by any teacher, so to take with a grain of salt):

* To prove this problem, for the sake of contradiction, we might argument there exists a program that can determine, given an input , if this number is random or not.
* We construct a list of numbers, each corresponding to any programming language of preference, structured low level as binary strings; consider for instance:
  + Program
  + Program
  + Program
* We create a program as follows:
  + For an input , does the opposite of what does (if structured as a Turing machine problem, it would be for example the complement of the TM that takes as input the previous strings)
    - If says that is random, outputs “not random”
    - If say that is not random, outputs “random”
* Now, consider what happens when we apply B to its own description. That is, we ask whether is a random program or not: . This leads to a paradox:
  + If is "random," then by the definition of , should be "not random." But this contradicts the assumption that can correctly determine randomness.
  + If is "not random," then by the definition of , should be "random." Again, this contradicts the assumption that can correctly determine randomness.

What we do know is that not all problems are not solvable by a computer, because of power constraints and limitations of machines, e.g. the halting problem and the program correctness (it’s impossible for even simple specifications). A natural question we naturally ask: “Which problems can we solve by a computer / by an effective procedure?”. Some problems are intrinsically theoretical, so they are completely independent from the underlying computation model.

Other specific questions:

* What is an *effective procedure*?
  + Maybe the simple program can do the job, but we must prove it formally
* What does it mean that *a problem is solved by an effective procedure*?
* Characterize the problems that can and those that cannot be solved
  + Problems that are not always binary
* Relating *unsolvable* problem (degree of unsolvability)

We tend to classify *solvable/unsolvable problems without limitations on the use of resources* (memory and time). For example, the complexity theory, considering the resources and classifying solvable problems in an hierarchy according to their “difficulty”.

*Computability theory* is a branch of computer science and mathematics that explores the theoretical limits of computation, this well before its proper birth. It revolves around the concept of decidability and undecidability, focusing on what can and cannot be computed algorithmically. So, *computer science* may be described as “the ability of building and using tools, according to some (codified) procedure, is a distinctive feature of human beings”. It depends on “how we use the tools and what we find out when we do”, according to *Djikstra.*

We don’t tend to think meaningfully always, but to think *according to patterns*, because there is a general combinatoric procedure to find all truths, reasoning and deriving consequences from a set of premises. Thing is, it doesn’t depend on the language, but we can try to represent things abstractly as a set of customized symbols (creating laws or languages), compute them logically (arithmetically) without contradiction and evaluating problems with procedures, to avoid controversies of decidability and solvability as criteria (*Leibniz, Boole, Lullus and others*) using *logic* as the main foundation.

Others posed the need of an artificial language, formally with syntactic and manipulation rules that can be programmed via *variables* and *statements*. Using cases like Russell’s paradox, we can use the same tools we already have to contradict ourselves and pushing further, even finding new meanings, possibly having a *consistent* system, where it proves itself as correct solidly (*Hilbert*). Many times, this observation led to creation of special-purpose machines, able to compute a specific class of problems.

We might try to take problems considering a small set of rules, which may not be always complete or prove the consistency of the theory (*Godel*). There may be a machine which computes a problem given a computable function and the same language, given a specific input and an output (*Turing*). We may express a universal machine to make *any kind of calculation*, storing the result of operations (memory) and solving problem discretely (*Von Neumann*).

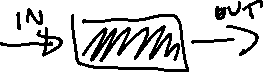
Other things:

* On Moodle there are unofficial notes
* There are the exercises with solutions (suggested the ones with no solutions)
* There will be tutoring activities for this course

# Algorithms, effective procedures, non-computable functions

An effective procedure it’s just a sequence of *elementary steps* which are describing a procedure intended to solve a problem (reaching some objective mechanically), transforming some *input* into some *output*.

We can see an algorithm as a black box of sort:



If this is deterministic, we can mathematically describe a function , where each possible input will uniquely determine the corresponding output (we will see later this happens on *partial functions*, so maps between two sets and that may not be defined on the entire set ; an example might be the square root, where not all real numbers have real square roots so *we can compute it but not always solve it*).

A function is computable if *there exist*s an algorithm such that the induced function is (so is the function computed if is *effectively computable*). It’s important to note the algorithm that computer must exist.

We informally expect some functions to be computable, given the definition above, such as:

* (eventually an n-th prime will be found)
  + this is a series that converge to and we work with techniques to allow rounding the error, such as truncating the series or rounding the computation

Let’s give an interesting example:

* + More generally, it can be written, for example as
    - (where iff means “if and only if”)

The naïve idea of this last one is:

* compute all the digits of
* check if there are digits of in a row

This, however, is not an algorithm, because we can’t exclude entirely the generation on at some point. Since ’s decimal expansion is non-repeating and doesn't follow a simple pattern, we cannot guarantee that the algorithm won't eventually find the desired sequence of 's (given is an irrational number), so we may run it indefinitely and will eventually become infeasible, because we have no way of returning .

Is this function computable? In the case of this one, we don't have an effective procedure known to us to determine whether it's computable or not (hence, it’s *not an effective procedure*). The fact that we can't exclude the existence of an effective procedure *doesn't mean* the function is computable, but it also *doesn't definitively prove* that it is computable.

Let’s consider now a slightly different example, for a function :

* + We deduce that, somehow, we will reach as constant substituting the values
  + More generally:

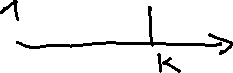
Consider

We then have two possibilities (with plot of the functions reported here, given its quite simple shape):

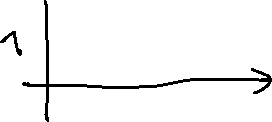












This implies the function is computable, because it behaves regularly (step function, so either or , or just a constant function, so they can be computed by simple programs). Even though we won’t know the exact shape of the function, this way we proved it’s computable (the function shape is irrelevant in knowing which program will compute the function, but if finite, they can be a simple tool to see it).

Can we use the same argument for ?

Let

and take:



Problem is, is not computable in the slightest, because the set is possibly infinite and there is no such a thing as a finite representation for it (in the notes, it’s also present an example of a function which is , otherwise; since the condition does not depend on the variable, it can have either way or as value, so the function remains computable, but if posed inside the set would be equally incomputable).

This poses the question for the existence of non-computable functions, because it suggests is computable, because the set is possibly infinite, so we can’t provide a finite representation.

A good algorithm should satisfy the following characteristics which can be ideally implemented in a theoretical machine we call *computational model*, this way being considered *effective*:

* it has a *finite length*
* there exists a *computing agent* able to execute the algorithm instructions
  + this agent has a *memory* to store the input, results and steps and it is *unbounded*
    - even if the algorithm will be finite, we assume it is unbounded for the sake of analyzing if it’s computable or not (large, but never using the full space)
    - this way, we will be able to define algorithms working on any possible input and there is no limit on the memory that can be used
  + the computation consists in *discrete steps*, not probabilistic or not-deterministic
  + finite limit to number of instructions and the power of their complexity
    - this way representing a finite machine
* the computation can
  + terminate in a finite yet unbounded number of steps output
  + diverge (never terminate) no output

Let’s recall the *math notation* needed to understand the subsequent inference of non-computable functions for every “effective” computational model.

* set of *natural numbers* (so finite and always with a successor)
* as *Cartesian product* (combine two sets to create an ordered one)
  + We will write, having set,
* *binary relation* or *predicate* as







* , the *partial* function, special relation such that
  + We write
* In words, we essentially say it’s a mathematical relationship that associates elements from a set to elements in a set , but it may not be defined for all elements in (for example, not all pairs)
* When you apply the partial function to an element in its domain, you write to indicate that the function is defined and yields a result. Conversely, if you try to apply the function to an element outside its domain, you write to signify that the function is undefined for that input.

Given a set , we indicate with the cardinality (number of elements), then we define, for sets and :

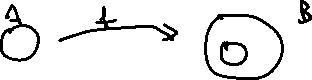


* (unique and complete mapping)

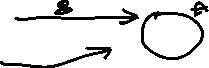


* (no two different inputs map to the same output)



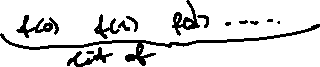


* (covering the entire codomain – all possible outputs)



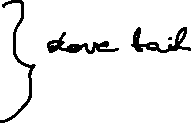
Observe also that if , having injectivity in between.

* (listing all the elements one after the other)



* A countable union of countable sets is countable:

Idea (just to visualize the whole thing, place the elements in a matrix and enumerate them in diagonals):



This so called “dove tail enumeration” means systematically listing all functions from to :

* Begin by listing the element at position in the matrix, which is the function that maps to .
* Then, move along the diagonals of the matrix, listing the elements in order

Let’s come back to the existence of non-computable functions: we focus on unary function over the natural numbers (function that takes a single argument or input variable and produces a single output):

We then fix a model of computation, which then induces a set of algorithms, for example a set of all algorithms inside of it. Given an algorithm we compute a function , which is said to be *computable* in our model if there exists an algorithm that computes it.

Hence, we define *functions computable in*  like:



Clearly we have . Is this inclusion strict? (so, , which means (is there a non-computable function?)



The answer is yes, because the algorithms are too few to compute all the functions, so they must be countable in some way, hence by logical closure computable.

By assumption, an algorithm is a finite sequence of instructions from an instruction set , which we assume finite. We can interpret all of this as a big union of finite algorithms.

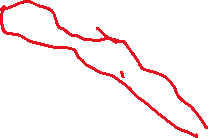
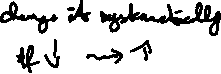
(countable union of countable sets 🡪 countable)

Given and since we have (which means it is surjective be definition), we have:

What we say in words is this: the set of all algorithms in our fixed computational model and , the set of computable functions are as many as the natural numbers.

On the other hand, the set of all functions is not countable. Why? Assume for the sake of contradiction that it is so:

We can list the elements of like we did before (taking, with diagonalization, the main diagonal, then systematically changing diagonal values):



then build a function on that, like this one:



is a function which is *total* (so, defined for every natural number) in so there is

* (meaning is not defined at , since and it means we are not enumerating the current function inside the natural numbers, which we assume we can always do since is countable; so there is the contradiction)
* (again, not defined in and we do not enumerate the function assuming we can, hence another contradiction)

Since is distinct from all the functions in the enumeration, it demonstrates that the set of all functions is uncountable, because it cannot be put in one-to-one correspondence with the set of natural numbers .

Summing up in math notation (there are more function than natural numbers, even though finite algorithms are as many as natural numbers):

Note that we can’t count non-computable functions, so:

We conclude that:

* no computational model can compute all functions
* there are more non-computable than computable functions

# URM Computability

To give a good notion of computability, we must choose a good model of computation, inducing a class of algorithms and computable functions. There can be:

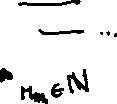
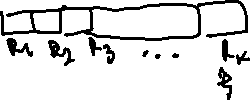
* Turing Machines (finite-state control, reading, writing, initial/final configuration)
* -calculus (a design of programming where one designs/applies functions based on primitives)
* Partial recursive functions (functions calculated with specific function that build partially)
* Canonical deduction systems (system used to create proofs logically via connectives and trees)
* URM (Unlimited Register Machines)

Whatever the model, we may concern if a specific theory may be valid for the specific model.

According to the Church-Turing thesis, a function is computable by an *effective procedure* if and only if it’s *computable by a Turing machine* (we resort to this to shorten the proof that a certain function is computable and it’s used informally as notion of effectiveness, then must be supported by evidence). This says that a function if computationally robust and we can choose whatever model one likes.

The notion of computable function will be formalized by using the URM-machine, abstraction based on the Von Neumann’s model. It has many characteristics:

* *memory is unbounded*, using an infinite number of *registers* storing each a natural number (where a sequence of registers is called *configuration*);



* it executes a program, based on a finite list of instructions (and a *computing agent* able to execute it);

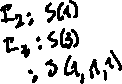
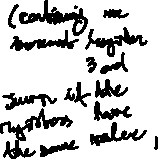


* it has *arithmetic instructions*, characterized by the fact that the instruction to be executed in the next step is the one following the current instruction in the program. They are:
  + *zero* , which sets the content of register to zero:
  + *successor* , which increments by 1 the content of register :
  + *transfer* , which transfers the content of register into , which staying untouched:
  + *conditional jump*: , which compares the content of register and , so:
    - if then jumps to (jumps to -th instruction)
    - otherwise, it will continue with the next instruction

The computation:

* starts from an initial configuration of registers and executes
* terminates if
  + the instruction to be executed does not exist
  + it’s the last instruction
  + you jump out of the program yourself

An example might be the following one:



In LaTeX form, to not kill your eyes that much, coming from the notes:

Immagine che contiene testo, Carattere, numero, linea

Descrizione generata automaticamente

As we’re using the Church-Turing thesis, we’re defining a machine, so we must describe which *states* it has: there is a *register configuration* , taking the register content and index the next instruction via a program *counter* . Also, *operational semantics* can be defined via .

A computation can possibly diverge (not terminate); consider for instance this program:

Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente

Immagine che contiene numero, Carattere, linea, testo

Descrizione generata automaticamenteLet be an program. Given a sequence of natural numbers , indicates the computation of from :

* if the computation eventually terminates (*halts*)
* if the computation diverges (*never halts*)

We work on computations that start from an initial configuration where only a *finite number of registers contain a non-zero value*. So, given denotes for .

The notation then extends to the previous ones, stating that at the end of a program we will have a valid value 🡪 for and in final configuration .

For URM-computable functions, given a function (possibly partial), we say is URM-computable if there is a program such that, , and .

In words, for any input tuple of natural numbers, if you run the URM program on this input it will eventually have a result equal to the output of the function for input. This way, computes .

We then define, as the classes of computable functions. Therefore is the union of all of them.

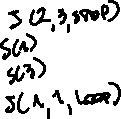
We next list some examples of URM-computable functions, providing the corresponding programs:

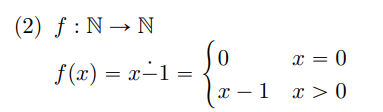
Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente



Idea: Incrementing and until contain the same value, resulting in adding to the content of . Specifically:



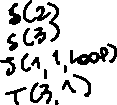




Now, let's analyze how this program works:

* If , it will jump to instruction , which presumably indicates the end of the program.
* If , it will go through instructions and , effectively setting to .
* If , it will go through instructions , and , effectively setting to .





The core concept behind this program is to continually subtract from the input value, considering the partial nature of the function. This means that the program might not always terminate, even if the function is computable, or it might terminate when the function is not computable.

In this specific example, the program checks if two values are equal; if they are, it jumps to a different instruction. If they are not equal, it subtracts one from the value. This subtraction continues until there is memory available for further operations.

Courtesy of notes (slightly different, but same example):

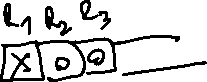
Immagine che contiene testo, schermata, Carattere, algebra

Descrizione generata automaticamente

Let’s consider a different function:

Immagine che contiene testo, Carattere, bianco, numero

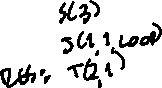
Descrizione generata automaticamente



The function behaves as follows:

* If the input number is even, the function returns half of (store an increasing even number in then storing its’ half in )
* If the input number is odd, the function does not terminate (indicated by the symbol ).

The program continues executing these instructions in a loop. If the initial input is even, it will eventually reach a point where equals the even number in , and it will jump to instruction , halving the input . If is initially odd, the program keeps increasing the even number in , and it never reaches the halting condition, resulting in a non-termination, as indicated by .



Given a program , for some fixed number of parameters , there exists a unique function computed by that we denote as follows as . More precisely:

Immagine che contiene testo, Carattere, linea, calligrafia

Descrizione generata automaticamente

In words: given a fixed number of parameters, the program halts if there is a final character of computation, otherwise the function will terminate when the program terminates. Remember:

* a program terminates or not, a function is defined or not. A function is not computing, only the program does (they are correlated, of course).
* the same function can be computed via different algorithms, which means different problems

*Question*: given how many computing are there computing ?

*Answer*: We can have infinitely many if and the only if the function is computable; so, or infinitely many

Exercise

Consider , class of URM machines without transfer instructions (so, no ). We indicate the class of URM computable functions. How does compare to ? (in math notation, )

(Thoughts)

We can use as we can zero in and increment the register until it reaches .

The idea is:

(if the registers are equal, it exits the loop)

(back the program to the loop beginning)

In plain terms, this program aims to achieve a similar effect as the transfer instruction by repeatedly incrementing the value in register until it matches the value in register . Once they are equal, it exits the loop.

Proof

We show that . Let computable in i.e. there is in . Just observe that is also a program . As said in thoughts, ideally we replace the transfer instruction as the step with the previous subroutine:

Immagine che contiene testo, Carattere, bianco, design

Descrizione generata automaticamente

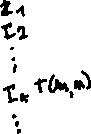
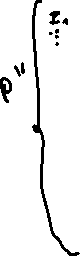
Let then . Hence there is a URM program such that .

We show there exists machine such that .

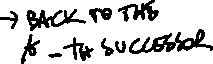
Remember: We’re assuming is well formed: if it terminates, it will at instruction

We proceed by induction on , which is the number of transfer instructions in . We can assume, without loss of generality, that when a program halts it does so at the index of the last instruction plus one (induction logic at its core, in words).

* : trivial, as with no transfer instructions is already a program, hence
* : let be the URM program with transfer instructions. Hence, we replace with a jump to the subroutine:











We call this program which has transfer instructions and . By inductive hypothesis there is a program such that . Putting things together:



*Note*: for any URM-program there is a well-formed program computing the same function. In fact:

In words:

For any URM program , you can create an equivalent well-formed program that computes the same function. To do this, you replace any conditional jump instruction where is greater than the total number of instructions () with . This ensures that the program always jumps to a valid instruction and remains well-formed while achieving the same computational result as the original program.

Exercise

Variant machine where there are no traditional transfer instructions such that , but (swap instructions) like to exchange context of registers. How does compare to ? (is ? ) [To note: The teacher says the proof is simple and very similar to the one we did before]

My take on the proof

Let us prove . We can replace the swap instructions with a few transfer instructions, formalizing how can be encoded in means of the routine. We can explain this in terms of how a swap instruction works in programming: we allocate a new register/new variable, we assign the variable to save to the new variable, then the new variable will get assigned to the second variable.

So, we create something like:

Formally, having a function we have a program as a program such that . Proceeding by induction:

* , the program is already a URM program and (such as before)
* , where the program, by injection, must have at least some transfer instructions to realize how a swap works. So, if only if this program uses both and instructions (the swap can’t be explained otherwise, and we need this statement to make this work correctly).
  + This way we can prove that with the swap, will have at least swap instructions, given the swap will be given via a jump instruction reaching the transfer instructions, hence creating the swap.
  + Inductively, there exists a program for and , concluding for having swaps recursively

Exercise

Consider without jump (where the apex indicates “minus minus”). How does compare with ? (is ?). [To note: it’s difficult, but one can start characterizing the shape of the functions in ]

My take on the proof

Let us prove . As said from the hint, we can characterize the shape of functions inside of it. We first observe that is strictly contained in C, since there are total computable functions in C that cannot be computed by a machine due to the lack of jumps.

If we try to think logically, we have zero, transfer, successor as the available functions. This means this function is strictly linear and can only perform execution as a fixed sequence of inputs, potentially up until a constant number of operations. This says they always terminate, so we can have:

Or also (having as the constant we were discussing above, which will be inside ):

This can be proven by induction, but operating with something that makes the computation possible. In this case, it should be something with a number, just to prove can come out of it. So, recursively it must recreate the shape of . We will use a register describing the execution of a given number of steps, say , so is equal to .

* , we have , fine because with the base case is trivial, having already or which will turn it as alone
* , so in this case the only thing this can do is the other three functions:
  + , concluding trivially because the next step, having fr and we conclude we’re inside and this is hence respected. When , infact, the operation resets to and the function will keep its form
  + , so will allow us to get the sum of the instruction, given , again concluding by inductive hypothesis. This way the function will have its expanded form , because we continuously sum
  + ; this is uncertain because the function depends on two values this time around, ; when they are different from each other, the result way be unknown (one can be 1, the other we can’t know for sure, making the underlying function assume shapes unknown.)
    - When (or equal) we will know will do exactly the transfer of steps; otherwise, if , we won’t know what happen for sure, it can jump many instructions
    - The proof goes well if we assume we have exactly steps, so the function for or even can go exactly linearly assuming we will execute exactly *only that* number of steps. This happens because we will keep inside the function

Let’s give the official solution to the previous exercises:

* where we replace the transfer instruction () with the swap one ().

Proof

We want to prove the two sets are equal.

* (Case )

Given .

If then there is a program program . We know that there is program without transfer instructions . But is also a -machine program.

In this case, so .

* (Case )

Take and let a program . We want to “transform” into a program .

So, the instruction can be encoded in a new subroutine, using which is something new and not used by the program. So:



is replaced with:

1. : This moves the value at location to a new, unused location .
2. : This swaps the value at location with the value at location , performing the swap.
3. : Finally, it restores the original value at location m to the new location .

A program can be transformed into a -program . We proceed by induction on , which is the number of instructions in .

* 🡪 is already a program, take
* 🡪 Let has instructions. The program can be seen as:



To complete the proof, we need:

* always terminates (if it does) at time
* is used in
  + This equation calculates the maximum of two sets: the set of registers used in program and the set . The purpose of this is to ensure that the value of is chosen to be greater than any register used in the program .

Then, and has instructions (hence, they compute the same instruction). Hence, by inductive hypothesis, there is a program . Thus,

The proof is wrong: we’re using the inductive hypothesis on which is not a -program (it contains both and ). You can make it work by proving a stronger assertion, specifically:

“Every program which uses all instructions, including and can be transformed in a -program .” This way, using all we already know up until now, we can conclude the proof solidly (so, if it works for all values in induction, we can safely conclude).

* Consider without jump instructions.

Proof

A program has this structure, and we know it terminates after steps:



All functions in are total (defined for all possible input values from its domain), so , e.g. (meaning it diverges for all values, because “program without jump always terminate”)

(not sufficient to say “it uses jump” computes ; we’re basically saying this does not hold inside because not in all cases can terminate if there’s a jump it doesn’t terminate and diverges).

Let’s restrict the program executing to unary functions (which take one argument or input); since there is no jump and it was the only way to alter the control independently from the input, we will always do the same thing



The shape of the functions will be either or , for a suitable constant.



Denote = content of after of computation starting from



We prove by induction on that or

* 🡪 🡪 OK
* ( 🡪 By inductive hypothesis or

In this case we will have only three possibilities (given the fact that we can’t jump):



As said, three cases, so:

1) , then we have two subcases (ind. hypoth.):

* , (base case)
* 🡪 OK, by inductive hypothesis, we have zeroed correctly

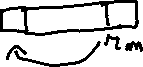
2) , again two subcases:

* , 🡪 OK, by inductive hypothesis (successor zero and all good)
* 🡪 OK, by inductive hypothesis (proceeding inductively works)

3) , again two subcases:

* 🡪 OK by inductive hypothesis

In this subcase we’re lost, because transfer instructions can cause issues when trying to maintain a specific structure for *unary functions*, particularly when the transfer instructions lead to values that cannot be effectively controlled within the defined structure of unary functions (in other case, as seen inductively, we know which instruction comes next, here we don’t know it for sure).



So, how do we proceed?

Idea 1: is “useless”. Ok, but this observation requires the jump to make it work.

The key observation is that the same property holds for all registers:

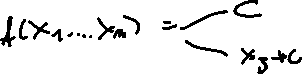


content of after steps of computation, starting from

Show by induction on that for all



The proof goes smoothly in this case (exercise here, so we get one input plus the constant for the specific function). For functions:



Solution (made by me, to take with a grain of salt):

Given all these values, let’s try to solve this inductively in cases as seen until now. We have the function here, which will be used to compute all values and will store the intermediate computation value. We will express such function, for with a new function that computes steps as:

where this function operates on its arguments. Let’s show this inductively:

* 🡪
* , assuming that for step k, or , we will now show how this assumption extends to step k+1. , we will assume
  + We introduce the function as to represent the inductive computation. We will consider all the subcases as before, given the instruction for and for a suitable constant :
    - * , here it will hold for each instruction before given the property can be seen as transfer of data so is given by given it’s defined linearly for all the function before

After examining the inductive step, we have shown that the properties we assumed for , namely , extend to step . This extension has been demonstrated through the function , which operates on the intermediate values to represent the inductive computation for .

In summary, we have successfully established that for all steps k, the properties for hold, and by extension, the computed function is of the desired form:

This completes the inductive proof, confirming that the functions adhere to the specified structure.

# Decidable predicates and computability on other domains

Consider as mathematical property the *divisor*:

As computer scientists, we can also see the divisor as a function:

Immagine che contiene testo, Carattere, bianco, calligrafia

Descrizione generata automaticamente

In the context of computability and formal logic, we introduce the concept of a predicate, which is a statement or function that takes one or more inputs and evaluates to either true or false, typically based on some condition or relationship.

The predicate on indicates the property can be true or false, formally describing:

* a function (note that we represent as values)
* a set

We write to denote or . This means will be computable if there exists a -tuple returning if , otherwise.

Then, given . We say it’s decidable if the *characteristic function* (also called *indicator function*, used to represent a specific property or set membership in a binary way) is like:

Remember also is a total function (again, defined for all possible input values from its domain and dealing with decidability of predicates, involves only total functions).

Let’s give some examples of decidable predicates:

1)



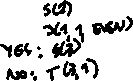
This function essentially encodes the result of applying the predicate to a pair of natural numbers. It returns if the numbers are equal (satisfying the predicate ) and if they are not.

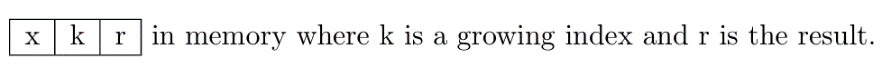
Now, let's see how this program works to compute :

* If , the program executes the jump in instruction , which sends it to instruction . It then increments register , making it , and transfers this value to register ().
* If , the program keeps looping at instruction (the self-jump) without changing the value of register . Therefore, register () remains .

So, after the execution of this program, register will contain either (if ) or (if ), which corresponds to the value of.

2)



The program essentially starts with at and checks whether is equal to . If is equal to , it means that is even, so it increments the result . If is not equal to , the program increments and repeats the process. This continues until is equal to , at which point is set to , indicating that x is even. In memory:

The program employs a simple iterative approach to determine if a given number is even, and it does so by incrementing until it matches . If the program exits with equal to , it means that is even. This program effectively computes the characteristic function for the predicate "," making it a decidable problem.

Let’s make a digression, using computability not only confined to a specific model, but resorting to the notion of effective encoding (used to map elements from one set [the domain] to elements in another set, typically inside natural numbers) in a way that is algorithmically or effectively computable. This allow us to extend the concept to other domains, defining then the computability on other domains.

Consider we’re interested in computability of a domain of objects, which is countable (so one-to-one correspondence with natural numbers), and:

where:

* *bijective* means “establishing a one-to-one correspondence (bijective mapping) between elements in the domain and the set of natural numbers”.
* *effective* means “the process of encoding an element from the domain to a natural number should be algorithmically computable”
* there exists an *inverse function*, which should map natural numbers back to elements in the domain effectively (and so are effective)
* once an effective encoding is established , it can be employed to define computability on the domain . This means that functions and predicates over can be represented using natural numbers through the encoding.

Consider for example the strings domain , where its domain size is smaller than real numbers set and it’s countable or other sets like (infinite sequences of elements from a given set, also called *streams*), .

Immagine che contiene diagramma, Carattere, origami, tipografia

Descrizione generata automaticamenteLet’s define a *computable function on a generic domain*; given function we say is computable if:

is URM-computable (the symbol means the composition of functions)

In words: if is defined and and its inverse are effective, is computable (you can see the mapping).

Let’s see this more concretely, shall we? Suppose we want to pose *computability on the integer numbers* (over ). We the need an encoding , given this encoding across the many which can be made:

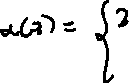


Immagine che contiene testo, Carattere, bianco, diagramma

Descrizione generata automaticamentewhich is an effective function with inverse:

Consider then the absolute value function:

Is this one computable? In this encoding, it is.

Immagine che contiene testo, Carattere, diagramma, schermata

Descrizione generata automaticamente

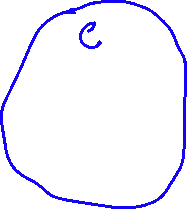
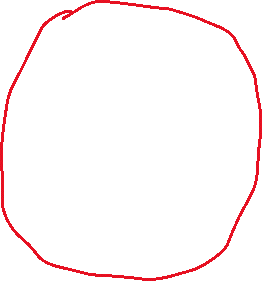
In the final part where is expressed for even and odd cases, it shows how the composition of functions and the encoding function α results in a computable function. Here's what the expressions mean:

1. If is even: In this case, is computed as , which simplifies to . This means that when is even, the absolute value of is the same as itself, and the composition function is equal to .
2. If is odd: In this case, is computed as , which simplifies to ``. When is odd, the absolute value of is because the negative of an odd integer is one more than its absolute value. Therefore, the composition function is equal to when is odd.

The expressions show how behaves for even and odd values of in terms of the encoding function α. The goal of these expressions is to demonstrate that is URM-computable for all cases, making the function computable on the integers by encoding and decoding integers using and its inverse

# Generation of computable functions

A function will be *computable* if it can be obtained from a set of basic operations that are known to be computable. Essentially, we show that having two functions we produce an operation inside in a way that composing them (for example, via is still in ).



The class will be closed under:

* (generalized) composition
* primitive recursion
* unbounded minimalization

To prove a function is computable, we can write a URM program of use the closure theorems of choosing the operations carefully (the ones listed above).

The basic functions following are URM-computable:

1) Constant zero

2) Successor

3) Projection

They are in as they are computed respectively by:

1) computed by

2) computed by

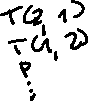
3) computed by

To prove the closure properties we will need to “combine” programs so we need some notation that we will give now. Given a program , we define:

* [aka largest register index]
* [aka number of instructions in P]
* if in standard form if, whenever it terminates, it does so at an instruction
  + for each instruction, (stopping at instruction as just said)
* concatenation, given programs (from now on we assume they are standard), where often we will have to concatenate programs. Given programs , their concatenation is obtained by considering followed by the instructions of , updating instructions properly



* Given a program we write program taking the input from and outputs in without assuming registers different from the input are set to . We do this by using transfer and reset operations, executing up until , so last instruction given the whole register space.
  + More precisely, it we express as follows:



Exercise: Write (\*) properly in this case

Solution: We write the program as , so we just transfer a value from the output register, do a reset operation and transfer the value back again inside the original register, considering the problem structure.

We define the composition, given you define

Immagine che contiene testo, Carattere, bianco, algebra

Descrizione generata automaticamente

(In words: the composition function will be made on all the sub functions if they all halt)

For example, consider:

(In words: we use the empty set function, so we define the emptying on all values)

Another example:

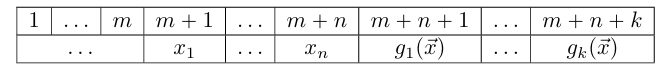
(In words: composition holds for both values inside functions and if value is not zero, it will output the first, otherwise it just diverges)

Now, we argue is closed under *generalized composition*.

Proof

Given and consider is in .

Let be programs (in standard form) for . The program for can be:



It is important to note that the registers from onwards can be used freely without the risk of interferences. Let us consider here the largest possible register, so . The program for composition then is:

Immagine che contiene testo, Carattere, ricevuta, bianco

Descrizione generata automaticamente

In words, the composition program provides inputs using additional registers and auxiliary input, then applying transfer operation on each following register and finally computing the result.

Because the following registers can compute the result, if it was defined before, the property continues to hold.

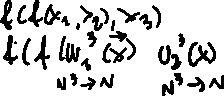
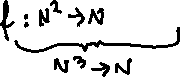
We then conclude that .

Let’s give another example: known to be in . We define with . Are we really doing generalized composition? Yes, but we can use projection.

We define such projection on as (we use projection on and then as (again, projection on ). This way, we will have a function of three arguments correctly using generalized composition, having finally:



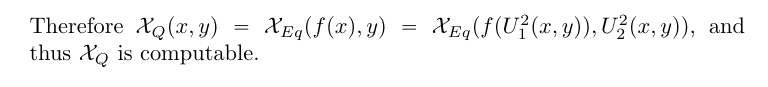
Basically, in drawing form:



Another example: let computable, decidable?

computable?

We know that is computable. We then obtain the computable result via composition (means equality and we know it’s computable).



There is a big problem: we’re not considering the case of undefined (we assumed was total, but that seems not to be the case, because the function is partial, so we map *some* values). To have it correct, change the definition of the predicate putting: “let computable and total”, thus it will work.

Recursion is a familiar concept to us computer scientists: it allows to define a function specifying its values in terms of other values of the same function (while other functions are possibly already defined).

Two classic examples of those:

(1) the factorial (the product of all positive integers less than or equal to a given positive integer)

Immagine che contiene Carattere, testo, bianco, calligrafia

Descrizione generata automaticamente

(2) Fibonacci (a sequence in which each number is the sum of the two preceding ones)

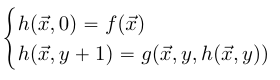
Immagine che contiene Carattere, bianco, testo, tipografia

Descrizione generata automaticamente

In our case we define a very basic and “controlled” version of recursion (also from domains, you can see they are recursively defined). Let’s give a proper definition then.

Given and functions, define by primitive recursion as follows:

So, the definition of combines and to compute its value for different inputs. It starts with a base case where equals for input 0 and then uses the recursive case to compute for other values by using both and the previously computed values of .



The function is defined in an equational manner, but it is an implicit definition, and it is not obvious that such exists or that it is unique, but it does for both.

We define a set of functions over the natural numbers, an operator for computing the recursive formula, a function of fixed points where there is always an upper bound which allows us to do operation in the continuous (so, always inductively defined). This can be drawn as:



We just say here: there might be problems given the recursive nature of computation if it doesn’t match our requirements. Let’s give other examples:

Consider the sum function:

Immagine che contiene testo, Carattere, calligrafia, tipografia

Descrizione generata automaticamenteImmagine che contiene Carattere, testo, bianco, calligrafia

Descrizione generata automaticamente

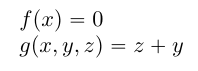
Let’s go ahead and define the product function:

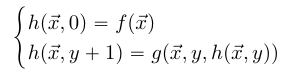
Immagine che contiene Carattere, bianco, testo, tipografia

Descrizione generata automaticamente

As proposition, we say is closed by primitive recursion, so functions obtained from total functions by generalized composition and primitive recursion are total.

Proof

Let: be in and let programs in standard form for . We want to prove that defined through primitive recursion:



is computable.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente



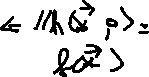
Essentially, we compute the following instruction until we get to .

Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente

Immagine che contiene testo, ricevuta, Carattere, bianco

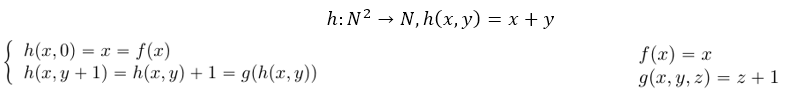
Descrizione generata automaticamente



In words: we’re just making a for loop with using closure under primitive recursion (a function that can be computed by a computer program whose loops are all "for" loops) and under composition (so, each previous function can be used to compute the following one), we will go ahead until we reach ; to avoid conflicts, we determine the maximum register ensuring there’s enough space.

We define a list of computable functions, implanting recursion through recursion.

* *sum*



* *product*

Immagine che contiene testo, Carattere, bianco, algebra

Descrizione generata automaticamente

* Immagine che contiene testo, Carattere, schermata

  Descrizione generata automaticamente*exponential*

Immagine che contiene testo, Carattere, bianco, algebra

Descrizione generata automaticamente

* Immagine che contiene testo, Carattere, schermata, design

  Descrizione generata automaticamente*predecessor*

Immagine che contiene testo, Carattere, bianco, calligrafia

Descrizione generata automaticamente

Immagine che contiene Carattere, testo, calligrafia, bianco

Descrizione generata automaticamente

* Immagine che contiene testo, schermata, Carattere, algebra

  Descrizione generata automaticamente*difference*

Immagine che contiene testo, Carattere, calligrafia, bianco

Descrizione generata automaticamente

Immagine che contiene Carattere, diagramma, bianco, design

Descrizione generata automaticamente

* *sign*

Immagine che contiene Carattere, testo, bianco, tipografia

Descrizione generata automaticamente

* *negative sign* (or *complement sign*)

Immagine che contiene Carattere, bianco, calligrafia, diagramma

Descrizione generata automaticamente

Immagine che contiene Carattere, bianco, testo, tipografia

Descrizione generata automaticamente

* Immagine che contiene testo, Carattere, schermata, design

  Descrizione generata automaticamente*Immagine che contiene testo, Carattere, schermata, design

  Descrizione generata automaticamenteminimum*



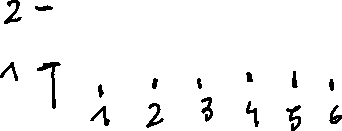
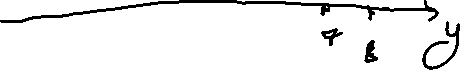
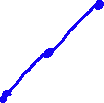
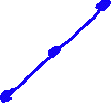
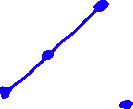
* *maximum*



* *remainder*, specifically the remainder of the integer division of by

Immagine che contiene Carattere, testo, bianco, linea

Descrizione generata automaticamente



By far, we use the definition by primitive recursion:

Immagine che contiene testo, Carattere, bianco, algebra

Descrizione generata automaticamente





* *quotient*,

Immagine che contiene testo, Carattere, calligrafia, bianco

Descrizione generata automaticamente

Immagine che contiene Carattere, bianco, diagramma

Descrizione generata automaticamenteLet’s give a *definition* *by cases*; let total and computable and decidable (mutually exclusive between each other) predicates (so, for each, exactly one of holds) and let total computable where:

then is computable and total.



Proof



So essentially, the right function will be selected and effectively computed, given all the marked functions are computable (sum and product) and composition is itself computable. We then conclude is computable.

Still, there is a mistake one can do: not assuming the functions are total and the proof will never be correct. Let’s show a counterexample:



The mistake in the given statement is the assumption that is computable for all , which is not the case in this counterexample (because it’s explicitly told it diverges). Therefore, the statement is not valid.

Important: We have a proof only if the component functions are total, otherwise it will be always undefined. *The proof is wrong if we don’t assume totality of functions*; keep in mind that *for now*.

Let’s define the algebra of decidability. Let be decidable predicates. Then:

can be considered decidable.

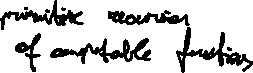
Proof

1. (evaluates to 1 if true, 0 if false)
2. (evaluates to 1 if both are true, to 0 if one of them is false)
3. (evaluates to 1 if either of them is true, to 0 if both are false)

Bounded sum/product

1) Sum

Let be a total computable function and let’s define . Then:



In simpler terms, it’s like adding up the values of the function for all starting from up to . It starts with 0 for and then, for each increment of , adds the value of to the previous total.

1. Product

The product is defined by:



It's like taking the product of the results for all starting from up to .

By closure under composition, the bound can be a total computable function. Another consequence concerns the decidability of the bounded quantification of predicates.

Let decidable:



Solution (made by me, so to take with a grain of salt)

We start from the base case, so if brings as true, the whole expression is true, otherwise it’s false. We then check systematically if all values bring that composition (true for all values = true/false for all values=false).

One can consider using a bounded operation there, for example the bounded sum. Given it will be defined on all values, we might consider defining a function which can be described as the computation of the bounded sum, so:

Given they are defined by primitive recursion, we know this holds and this is computable. More formally:

**Base Case:**

* For , we check if is true. If it's true, the whole expression is true for . If it's false, then the expression is false for .

**Inductive Step:**

* We now consider the remaining values of z, from 1 to y. For each z, we check if is true. If it's true for a specific , it contributes to the truth of the expression
* We repeat this process for all values of from to . If, for any value of in this range, is false, then the entire expression is false.

**Bounded Sum:**

* To streamline this process and handle all values of z systematically, we use a bounded function
* The function calculates the bounded sum of for z ranging from 0 to y (exclusive).
* Since the bounded sum is a well-defined mathematical operation, we know that is also well-defined and computable.
* Now, if equals , it means that for all values of from to , is true. Thus, the entire expression , is true.
* Conversely, if is not equal to , it means that there exists at least one value of within the range from to for which is false. In this case, the entire expression is false.

Similarly to this one, for the other one we use the bounded product

**Base Case:**

* For , we check if is true. If it's true, the whole expression is true for . If it's false, then the expression is false for .

**Inductive Step:**

* We now consider the remaining values of z, from 1 to y.
* For each , we check if is true. If it's true for a specific z, it contributes to the truth of the expression , . We repeat this process for all values of z from 1 to y.
* If, for any value of z in this range, is true, then the entire expression , is true.

**Bounded Product:**

* To streamline this process and handle all values of z systematically, we use the bounded product
* The function calculates the bounded product of for z ranging from to (exclusive).
* Since the bounded product is a well-defined mathematical operation, we know that is also well-defined and computable.
* Now, if is true (equals ), it means that for at least one value of from to , is true. Thus, the entire expression , is true.
* Conversely, if is not true (not equal to 1), it means that for none of the values of in the range from to , is true. In this case, the entire expression is false.

Essentially, we used this lemma from notes:

Immagine che contiene testo, Carattere, bianco, ricevuta

Descrizione generata automaticamente

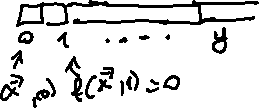
Otherwise, it can be shown as:

Immagine che contiene testo, Carattere, calligrafia, bianco

Descrizione generata automaticamenteImmagine che contiene testo, Carattere, calligrafia, bianco

Descrizione generata automaticamente

Let’s also define the bounded minimalization. Given a total function , define a function as follows:



In simpler terms, searches for the smallest integer (less than ) at which the function becomes equal to . If it finds such a , it returns that value. If there is no such , it returns .

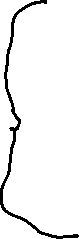
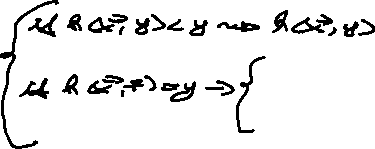
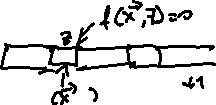
Observation: If is computable then computable



The observation correctly emphasizes that when the original function f is computable, the process of finding this minimum value , as described, is also computable (via composition of computable sum/product/sign)

Proof We have a definition by primitive recursion.





We observe the following functions are computable:

1)

(so, the division can be also written as the negation of remainder, so it will be 1 if there is no rest, 0 otherwise, as you see here for dividing)

2)



We can put the non-strict bound there, like . Formally, we can say that is defined via composition over the previous function, posing (given its recursive nature, we compute the current one if the last one was computed already).

3)

(So, it calculates the absolute difference between the number of divisors of and the given value . This difference measures how far the number of divisors of is from , in other terms, we can compute this absolute difference as ).

4)

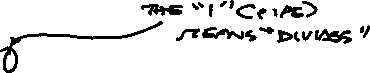
We use primitive recursion to do this.



The goal is to find a number that is prime and greater than the prime (); there, we use quotes there because this is not a formal definition. So, let’s define the search for the prime number properly, bounded to the recursive product of all of the other factors.



Let be a prime divisor of then ,



In summary, this argument demonstrates that if is a prime divisor of the product of the first prime numbers plus , it cannot be equal to any of the first primes. Instead, it must be greater than or equal to the prime, which is the next prime in the sequence. This establishes the relationship between prime divisors and the order of primes in the sequence, recursively because it’s bounded.

5)

When we calculate , we're essentially finding the exponent to which the prime number is raised in the prime factorization of . This implies that we're looking for the minimum such that if we consider raised to , it's no longer a divisor of . Via bounded minimalization, represents the highest exponent to which the prime number can be raised in the prime factorization of .

Hence, we conclude this is computable by bounded minimalization.

Exercise: Show that all functions obtained from the basic functions using composition and primitive recursion are total.

Solution (made by me, so to take with a grain of salt)

We begin by establishing the totality of all the basic functions (say, they include the successor and the zero function, for instance). For each initial function, we formally prove that it is defined for all possible inputs, ensuring that it produces a result for every valid input (because they are total). This is our base case, so define as successor and as the zero function. For any natural number , both and are well-defined and produce a result, ensuring totality.

* Induction on Composition
  + To demonstrate that composition preserves totality, we utilize mathematical induction. We assume that when composing total functions, the result remains total.
    - Base case
      * Let’s define, given two total functions and a composition such as . Let’s show that that is total
    - Inductive step
      * For to be total, we must demonstrate for any valid input it produces a result
      * We can then show is defined and produces a result, establishing the totality of it
      * Say for example we define
      * Given and are total, for any input, the composition is well-define, so which is is total
* Induction on Primitive recursion
  + To prove that primitive recursion preserves totality, we again use mathematical induction, this time on the number of applications of composition and primitive recursion needed to derive a function from the initial functions.
    - Base case
      * In the base case, we consider functions derived from the initial functions using no composition or primitive recursion. These are easily shown to be total. Say, is total, so is already that.
    - Inductive step
      * We assume that functions derived by applying composition and primitive recursion to functions that are already established as total remain total
      * We then consider a new function obtained by applying composition of primitive recursion; given the inductive hypothesis, the composition will be based over recursion on total function from the base case, hence they will keep being total
      * This new function may be
      * To show that is total, we need to demonstrate that for any valid input , produces a result, denoted as
      * Using the inductive hypothesis on , we can conclude that is also well-defined and produces a result, denoted as
      * This ensures the totality of the function

This seems strictly rigid, but let’s reason on another example, the Fibonacci function.

The Fibonacci function, as conventionally defined with two base cases and a recursive relationship involving the previous two values, is not a strictly primitive recursive function in the traditional sense of primitive recursion within computability theory.

The reason for this lies in the binary nature of the recursive relationship (adding the previous two values), which goes beyond the simple predecessor relationship found in typical primitive recursive functions. Given that is defined in terms of and , it does not completely adhere to the primitive recursion schema.



We can show that is computable by resorting to a new function :

Let’s see an encoding in of pairs (and n-tuples) of natural numbers that will be useful for some considerations on recursion. Define a pair encoding as, given

Immagine che contiene Carattere, bianco, testo, tipografia

Descrizione generata automaticamente

A pair encoding refers to a method of representing ordered pairs (and -tuples) of natural numbers using a single natural number. The goal is to encode the information in a way that preserves the relationship between the elements of the pair or tuple while allowing for effective (computable) operations on these encodings

Note is bijective (uniquely decode for original pair to encoding) and effective (encoding/decoding are computable), so computable. For example, if you have a pair , you can use to encode it as a single natural number . Later, you can use the inverse operation to decode the original pair from the encoded value.

Immagine che contiene testo, Carattere, bianco, algebra

Descrizione generata automaticamenteWe also have effective and can be characterized in terms of two computable functions and that give the first and second component of a natural number seen as pair:

It can be easily generalized in case for encoding -tuples, defining the recursive nature over functions previously computed and with projection always obtain a natural number given as factor.

On this encoding, let’s consider the Fibonacci function, which is defined as:



Given the function is not totally defined as the primitive recursion definition, we can show this is computable using the encoding in pairs. We then define:

therefore, by primitive recursion, we can write:



and so is computable by primitive recursion and is defined computable by composition.

More in detail:

* using the primitive recursion principle, and are defined based on the pair encoding and the previously computed values
* since encodes the pair ( can be derived as , where is a projection function that extracts the first component of an encoded pair. This means that is defined and computable by composition