Rovesti Gabriel

Computability simple (for real)

Summary

[Introduction 2](#_Toc147757179)

[Algorithms, effective procedures, non-computable functions 4](#_Toc147757180)

# Introduction

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

We start by a simple reflection; can we give the enumeration of all numbers? We can’t (the slide refers to a compressing algorithm, for example *gzip*).

Example:

* 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 from the Introduction set):

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 take 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 (see first question, same considerations):

* 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, for example 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 means 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. 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”.

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. 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), 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* general purpose. 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*).

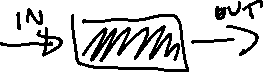
We might try to take problems considering a small set of rules, which may not be always complete or prove the consistency of the theory (*Godel*). There may be a machine which computes a problem given a computable function and the same language, yet not in all cases it might work (*Turing*). We may express 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* data 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.

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 n 5's (given is an irrational number), so we may run it indefinitely and will eventually become computationally infeasible.

Is this function computable? In the case of this function, 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:

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

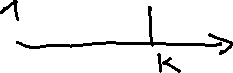
Consider

We then have two possibilities (with graph of the functions reported here):

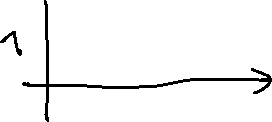










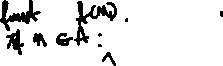


This implies the function is computable, because it behaves regularly (step function, so either 1 or 0, or just a constant function, as the plots suggested). Even though we won’t know the exact shape of the function, this way we proved it’s computable.

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 unofficial Moodle 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 1 or 0 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, cause the set is possibly infinite and 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 evert “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 A to elements in a set B, but it may not be defined for all elements in A (for example, not all pairs)
* When you apply the partial function to an element in its domain, you write f(a)↓ 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 f(a)↑ 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 :

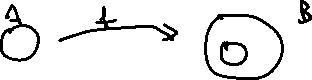


* (unique and complete mapping)

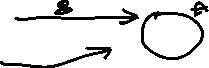


* (no two different inputs map to the same output)



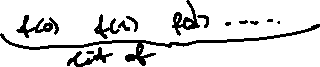


* (covering the entire codomain – all possible outputs – with

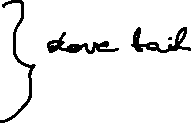


Observe also that if , having injectivity in between.

* (listing all the elements one after the other)



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



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

* Begin by listing the element at position (0, 0) in the matrix, which is the function that maps a0 to b0.
* Then, move along the diagonals of the matrix, listing the elements in the order pa0, b0q, pa0, b1q, pa1, b0q, pa0, b2q, pa1, b1q, pa2, b0q, ... and so on.
* A countable union of countable sets is countable:

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.

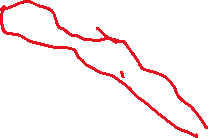
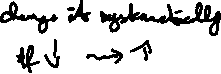
(countable union of countable sets 🡪 countable)

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:

i.e. 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:



is a function which is total (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