Summary

[Definitions 2](#_Toc156895222)

[Reduction 2](#_Toc156895223)

[Recursive Set 2](#_Toc156895224)

[Recursively Enumerable Set 2](#_Toc156895225)

[Decidable Predicate 2](#_Toc156895226)

[Semi-decidable predicate 2](#_Toc156895227)

[Structure Theorem 3](#_Toc156895228)

[Projection Theorem 4](#_Toc156895229)

[Primitive Recursive Functions 4](#_Toc156895230)

[Smn-Theorem 4](#_Toc156895231)

[Saturated set 4](#_Toc156895232)

[Rice’s Theorem 5](#_Toc156895233)

[Rice-Shapiro’s Theorem 5](#_Toc156895234)

[Second Recursion Theorem 5](#_Toc156895235)

[Exercises 6](#_Toc156895236)

[URM-Machines 6](#_Toc156895237)

[Smn-theorem 8](#_Toc156895238)

[Primitive recursive functions 9](#_Toc156895239)

[Functions and computability 11](#_Toc156895240)

[Diagonalization 11](#_Toc156895241)

[Recursiveness of sets 13](#_Toc156895242)

[Rice-Shapiro 13](#_Toc156895243)

[Reduction 15](#_Toc156895244)

[Second Recursion Theorem 16](#_Toc156895245)

[Show there exist an index s.t. function is total/computable 16](#_Toc156895246)

[Show there exist an index s.t. function is not computable 16](#_Toc156895247)

[Show that a set A is not saturated 16](#_Toc156895248)

# Definitions

## Reduction

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

## Recursive Set

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

## Recursively Enumerable Set

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Specifically:

* A set is r.e. if I can check a property on a finite number of points
* A set is not r.e. if I have to check the property on an infinite number of points

## Decidable Predicate

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

## Semi-decidable predicate

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

## Structure Theorem

Immagine che contiene testo, Carattere, calligrafia, bianco

Descrizione generata automaticamenteLet a predicate. Then is semidecidable iff there is a dedicable predicate s.t.

Note: in the “notes.pdf” the predicate is written as “decidable”, but prof. says it’s semidecidable like written here. Keep this in mind.

Immagine che contiene testo, schermata, Carattere, algebra

Descrizione generata automaticamenteThis reasoning is useful inside theoretical exercises about decidability/semidecidability because it’s literally the same reasoning, reported here for the sake of completeness.

The converse does not hold, for example

Alternatively:

* Suppose holds if non-semi-decidable, otherwise would be r.e.. We know there are programs inside , e.g. the ones calculating the always undefined function, but then always holds and so it would always be inevitably undecidable

## Projection Theorem

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

This reasoning is useful inside theoretical exercises about decidability/semidecidability because it’s literally the same reasoning, reported here for the sake of completeness.

*Proof*

Let semi-decidable. Hence, by the structure theorem, there is decidable s.t. .

Now is decidable.

Hence, is the existential quantification of a decidable predicate and by the structure theorem is semi-decidable.

## Primitive Recursive Functions

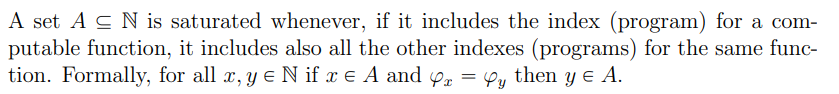
Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

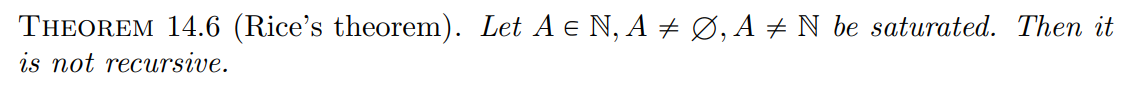
## Smn-Theorem

Given there is a total computable function s.t.

## Saturated set



## Rice’s Theorem



## Rice-Shapiro’s Theorem

Let (where is a property of functions) be a set of computable functions and let Then if is r.e. then

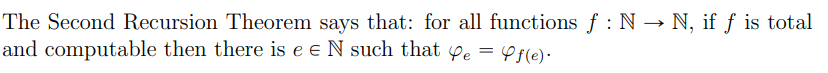


finite s.t. )

Generally, it can be used in two ways:



## Second Recursion Theorem



# Exercises

## URM-Machines

This kind of exercises was mainly present only inside partial exams.

* The exercise gives us a variant of the normal URM model which these basic instructions:
  + *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
* We have to prove the inclusion of the computable sets in both ways
  + From modified URM to normal URM
  + From normal URM to modified URM
* Define for URM-machine and (for example) the set of the model you have to show
* First step is showing
  + Not necessarily the new machine is more powerful, infact it may be even less powerful
  + Informally, we simply can code the “new” instruction/s in normal URM machine using a routine of some existing instructions (jump/transfer/successor/jump)
    - This is typically done considering say the index of an unused register by the program and a subroutine
  + Formally, we prove showing that, for each number of arguments and for each program using both sets of instructions we can obtain a URM program which computes the same function i.e. such that
  + The proof goes on by induction on the number of instructions
    - (), usually trivial, it’s already a URM program
    - (, basically I will describe the logic
      * Describe as for instance the index of instruction you want to replace and the length of computed program
      * We can build a program using a register not referenced in , for instance ( is the largest unused register)
      * Show that for the whole length of program, the jump to the subroutine can successfully replace the instruction wanted
    - The program is s.t. and it contains instructions. By inductive hypothesis, there exists a URM program s.t. , which is the desired program
* Second step is showing
  + The usual question is if inclusion holds both ways or if it is strict
  + If this second part does not hold, then it is not strict
* Usually, this is similar to the one before, but this time around, instructions of normal URM have to be encoded using only the new machine
  + This one follows, if formally, exactly the same steps as before

## Smn-theorem

* Give a function of two arguments
  + Define a case for set definition
  + Define a case for otherwise
* In this case, with smn-theorem exercises, it helps creating a function s.t.
  + the domain is where the values exist
    - so, the positive case condition is the domain or less than the domain and has to include that case inside condition
  + the codomain is the output we want to reach
    - after having written the cases, we see if the output/the computable function respects said condition
* It is computable, since it is defined by cases
* By the smn-theorem, there is
  + Write and rewrite the function defined initially again
* As observed above
  + domain given by definition
  + = codomain given by definition

In case you have and inside the function definition (just notation here, folks, the concept holds the same way, you simply have in place of ):

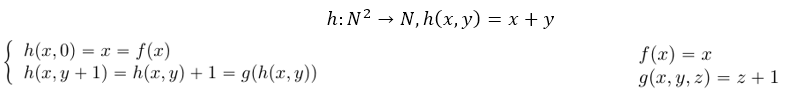
* simply use a function
* by the smn theorem, there is a total computable function
* As observed above
  + domain given by definition
  + = codomain given by definition

## Primitive recursive functions

* Write the class definition present above
* Carefully read the problem definition and write it using a combination of known functions

Consider, just for reference, these basic functions are primitive recursive functions:

* *sum*



* *product*

Immagine che contiene testo, Carattere, bianco, algebra

Descrizione generata automaticamente

* *exponential*

Immagine che contiene testo, Carattere, bianco, algebra

Descrizione generata automaticamente

* *predecessor*

Immagine che contiene testo, Carattere, bianco, calligrafia

Descrizione generata automaticamente

Immagine che contiene Carattere, testo, calligrafia, bianco

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

* *minimum*



* *maximum*



* *remainder*

Immagine che contiene Carattere, testo, bianco, linea

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, bianco, algebra

Descrizione generata automaticamente



* Immagine che contiene testo, Carattere, calligrafia, bianco

  Descrizione generata automaticamente*quotient*,

For completeness sake, always assume the sum and product are bounded, so they are primitive recursive (bounded sum and product)

* You define the function of the problem as a combination of base case and recursive case of the base functions and also some like the ones presented here

## Functions and computability

* In this case, consider the function are total
  + So, they have to define and handle all cases by definition

We have different choices to follow:

* diagonalization (subsection ahead)
* use a known non computable function, like
  + conditions are dependent on exercise, here reported just as an example
  + the general structure would be using somewhere, it can be both on positive/otherwise case
* sometimes, it happens that we use functions and subfunctions
* since the subfunction is finite, the function is too, and one can write it as a computable function

### Diagonalization

* In this case, there are notable total non-computable functions; the function is built to differ from its own values by recursion
* We then say since this holds by construction (just use the problem conditions replacing with )

Consider (conditions are dependent on exercise, here reported just as an example):

More generally, it might be something like:

The good proof (extended by this) would be:

* is total by construction
* is not computable since
  + infact, if then
  + if then
* the specified exercise property holds
* Immagine che contiene testo, schermata, Carattere, documento

  Descrizione generata automaticamenteConsider the following notable examples from the course:

Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

* Other cases, more similar to what we saw in the course, involve multiple cases, usually three, with small variations of the condition but with the same concept

The same observations about using hold.

## Recursiveness of sets

### Rice-Shapiro

* We use this one if is saturated
  + This usually happens when the exercises gives or both of them
  + A set is saturated if there is a non-trivial property (finitely characterizable)
  + and
  + You replace with and with (also with )
* This way, we show and are not r.e.
  + This may not always be the case; sometimes a set is saturated, but the set is r.e. (it means you can write a semicharacteristic function )
    - In this case, if is r.e. is not r.e (hence not recursive)
    - Conversely, if is r.e., not r.e. (hence not recursive)
* Applying the definition it means either:
  + we have a function which is in the set but a finite subfunction not in the set
  + we have a function which is not in the set but a finite subfunction which is in the set
* Usually, we use and
  + identity = defined for all natural numbers
    - if you use this one, possibly you have a function inside/not inside the set
  + always undefined function = undefined for all natural numbers
    - this one is often used as a subfunction to prove is inside the complement
    - many other times, it can simply be a function inside the normal set
* Sometimes, one can use the constant function (or constant )
* It usually works showing you have (as above, but replace with a logically correlated function to the exercise definition of specified set)
  + , but
  + , but
* This usually holds for both sets
  + If both sets are not r.e. they are not recursive either

There are the following implications:

* if is r.e. but not recursive, also is not r.e. (also not recursive, otherwise they would be both recursive)
* if is recursive, then is computable. We have is r.e. and:
  + if , then is not recursive
  + if is computable then is recursive
* If r.e., then is not – if is r.e., it means exists, but is not recursive
* If r.e. then is not – if is r.e., it means exists, but is not recursive

Side note (important):

* One can show a set is not recursive by using Rice’s theorem
  + This occurs when the set is saturated and maybe is r.e. but we ask if it is recursive
  + Then, you use and to prove hence
    - for example or

Usually, if the set is not r.e. it is also not recursive.

Practically:

* find a function which is in the set/a subfunction not in the set
  + alternatively
* find a function which is not in the set/a subfunction which is in the set

Then:

* if something is in the normal set, normally it is not in the complement
  + write the complement set so you exactly see what you are looking for

Use same reasoning for complement:

* find a function which is in the set/a subfunction not in the set
  + alternatively
* find a function which is not in the set/a subfunction which is in the set

### Reduction

To note:

* : to prove a set is not recursive
* to prove a set is not r.e.
* We use this one if is not recursive ()
  + usually something like
  + a variant with the same meaning is
  + sometimes, consider there is also:
    - this one occurs in case of both domain and codomain over index
  + it is computable and thus, by the smn theorem, we deduce that there is a total computable function such that, for each ,
* It can be shown to be the correct reduction function
  + if Therefore and . Therefore,
    - the function here is the value; if we had it would have been



* + if , . Therefore and so
* Note: if , then usually is r.e.
* We can also use the complement of the same set ()
  + usually something like
  + this starts from a computable function, like
  + it is computable since we have and thus, by the smn theorem, we deduce that there is a total computable function such that, for each ,
* It can be shown to be the correct reduction function
  + if Also, we can say is false Therefore and . Therefore,
  + if , . Also, we can say is true Therefore and so
* If this reduction from complement holds, is not r.e.
* It can also happen and so is not r.e.
* If both are valid (so and ), both sets (, ) are not r.e.

## Second Recursion Theorem

### Show there exist an index s.t. function is total/computable

* Give the theorem definition
* Give a function of two arguments for instance defined by cases
  + case for the normal condition
  + case for otherwise
* Since it is defined by cases, it's computable (since it is total, holds)
* By the smn-theorem, there exists a total computable function
* By the Second Recursion Theorem, there exists such that
* You use the function previously defined and replace with
  + inside the function, replace with
* You conclude since you fixed the point in which all the conditions you posed hold (simply use second recursion theorem definition)

### Show there exist an index s.t. function is not computable

* Give the theorem definition
* Note the function is computable but it is usually total, so you have say
* By the Second Recursion Theorem, there exists such that
* So, the original function cannot be computable

### Show that a set A is not saturated

* Give the theorem definition
* Give a function of two arguments for instance defined by cases
  + case for the normal condition
  + case for otherwise
* Since it is defined by cases, it's computable
* By the smn-theorem, there exists a total computable function
* By the Second Recursion Theorem, there exists such that
* You use the function previously defined and replace with
  + inside the function, replace with
* Now, just take such that (which exists since there are infinitely many indices for the same computable function)
* So, we have in and So, is not saturated