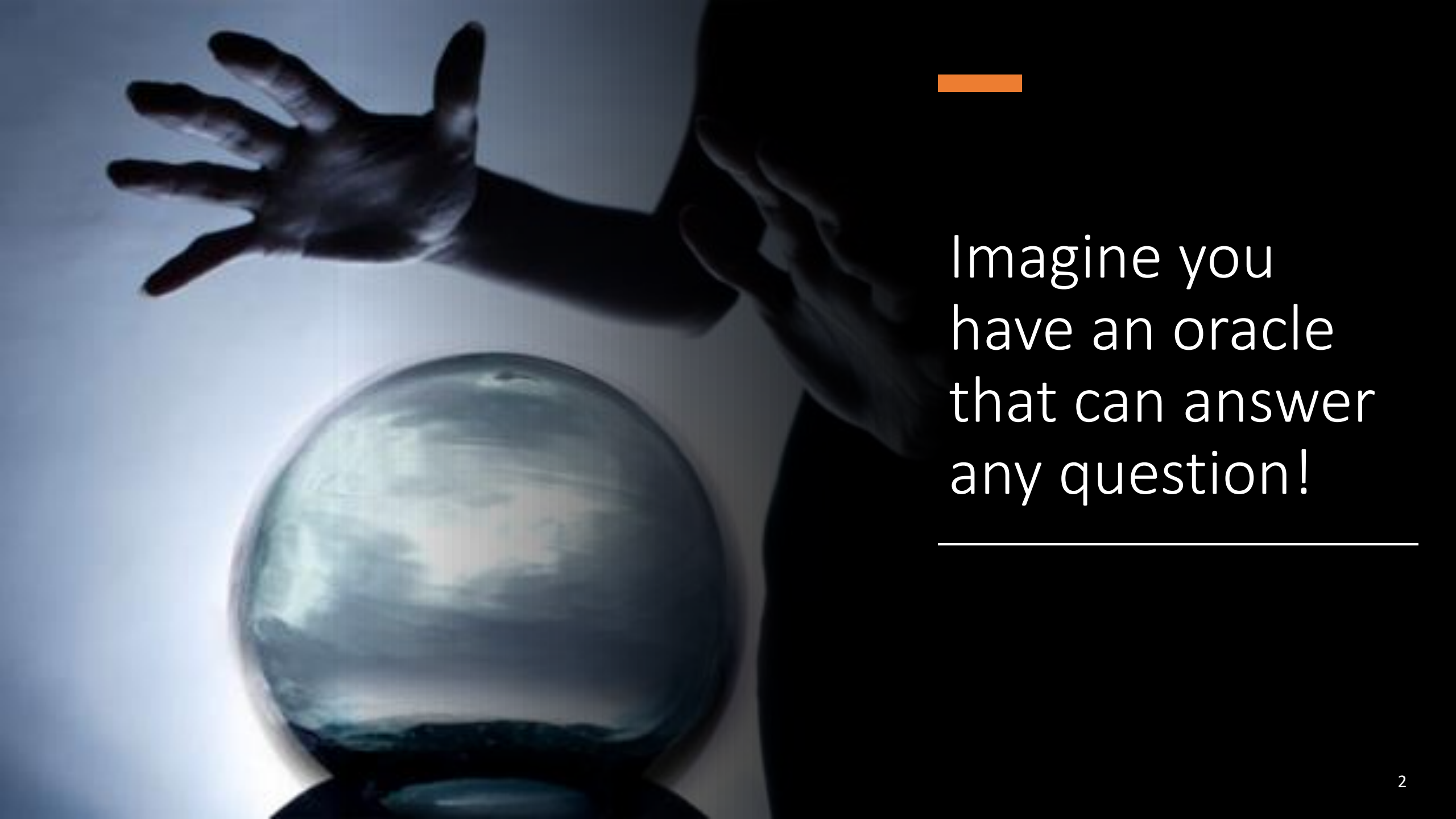


Theoretical Computer Science

Where are we now?

Where are we heading to?

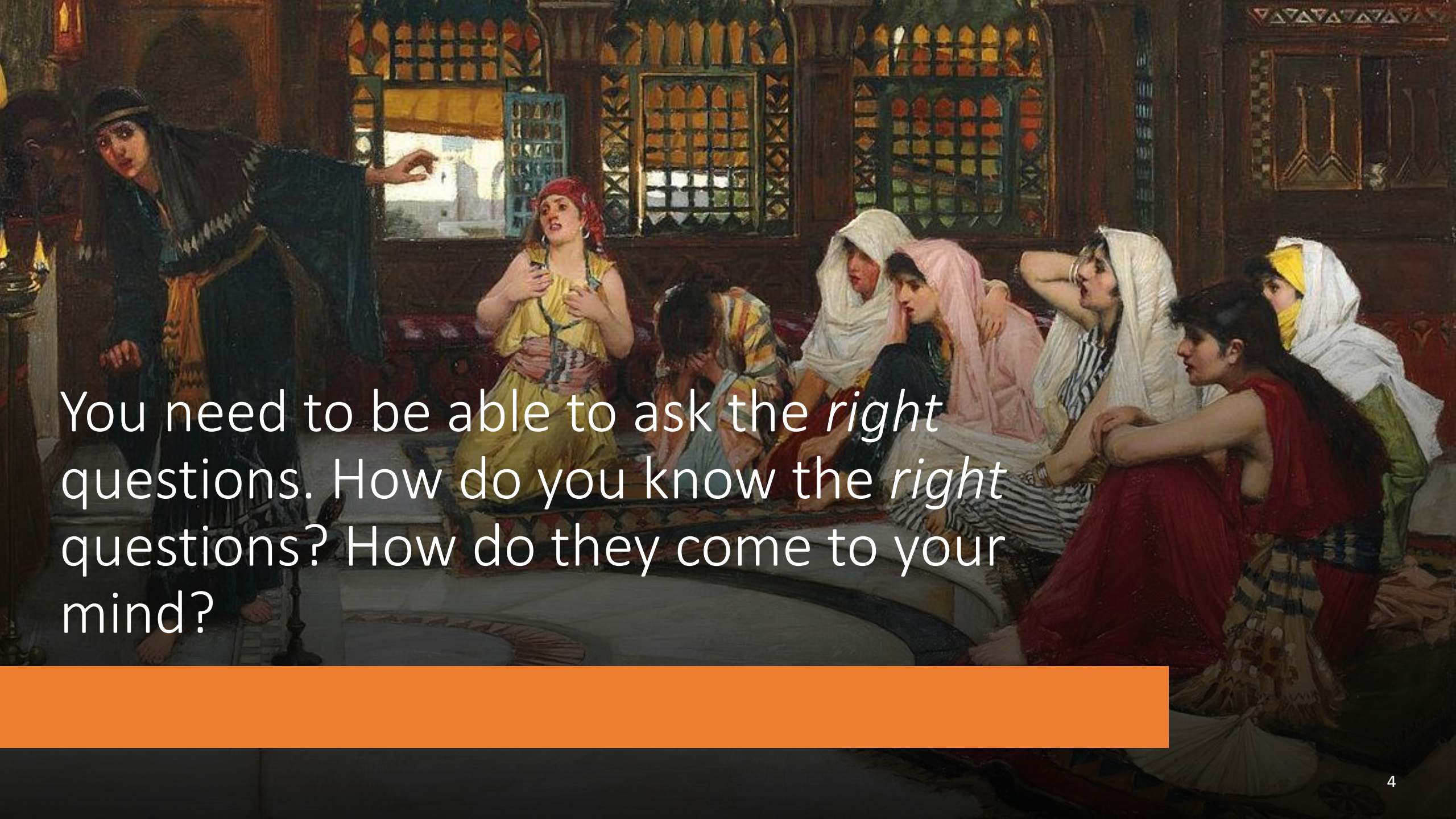
Lecture 14 - Manuel Mazzara

A person's hands are shown in silhouette, hovering over a crystal ball. The crystal ball contains a landscape with a bright sun or moon in a cloudy sky. The background is dark, and the scene is lit from below, creating a dramatic effect.

Imagine you
have an oracle
that can answer
any question!



Would you achieve
knowledge or even
more wisdom?



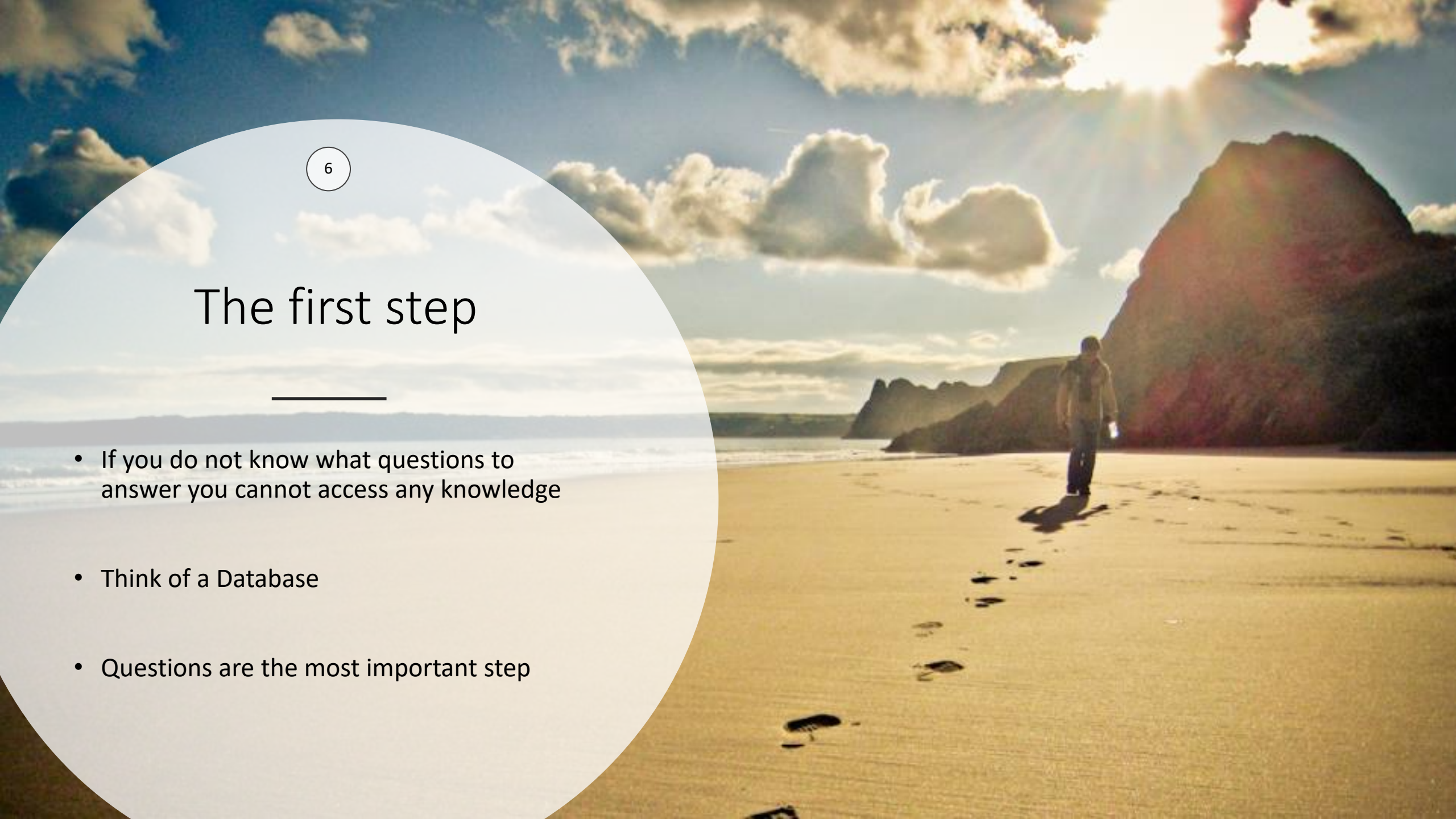
You need to be able to ask the *right* questions. How do you know the *right* questions? How do they come to your mind?

Maybe you can ask the oracle
what questions to ask? But
this is a meta-question 😊

Will the oracle answer meta questions?

The first step

- If you do not know what questions to answer you cannot access any knowledge
- Think of a Database
- Questions are the most important step



The two key questions we asked!

Do there exist computing formalisms more powerful than TMs?

- **Church-Turing thesis**

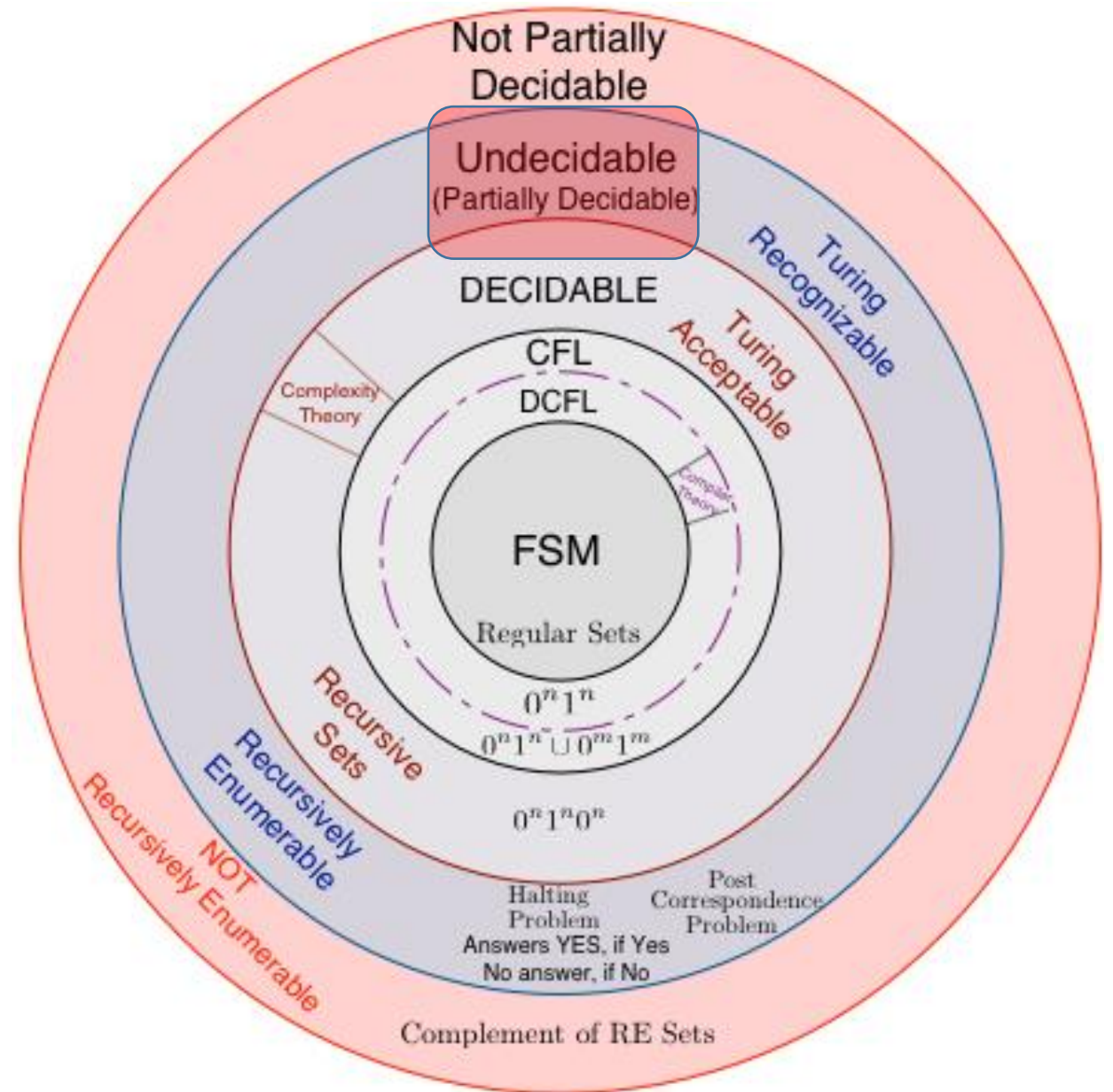
Can we always solve problems by means of some mechanical device?

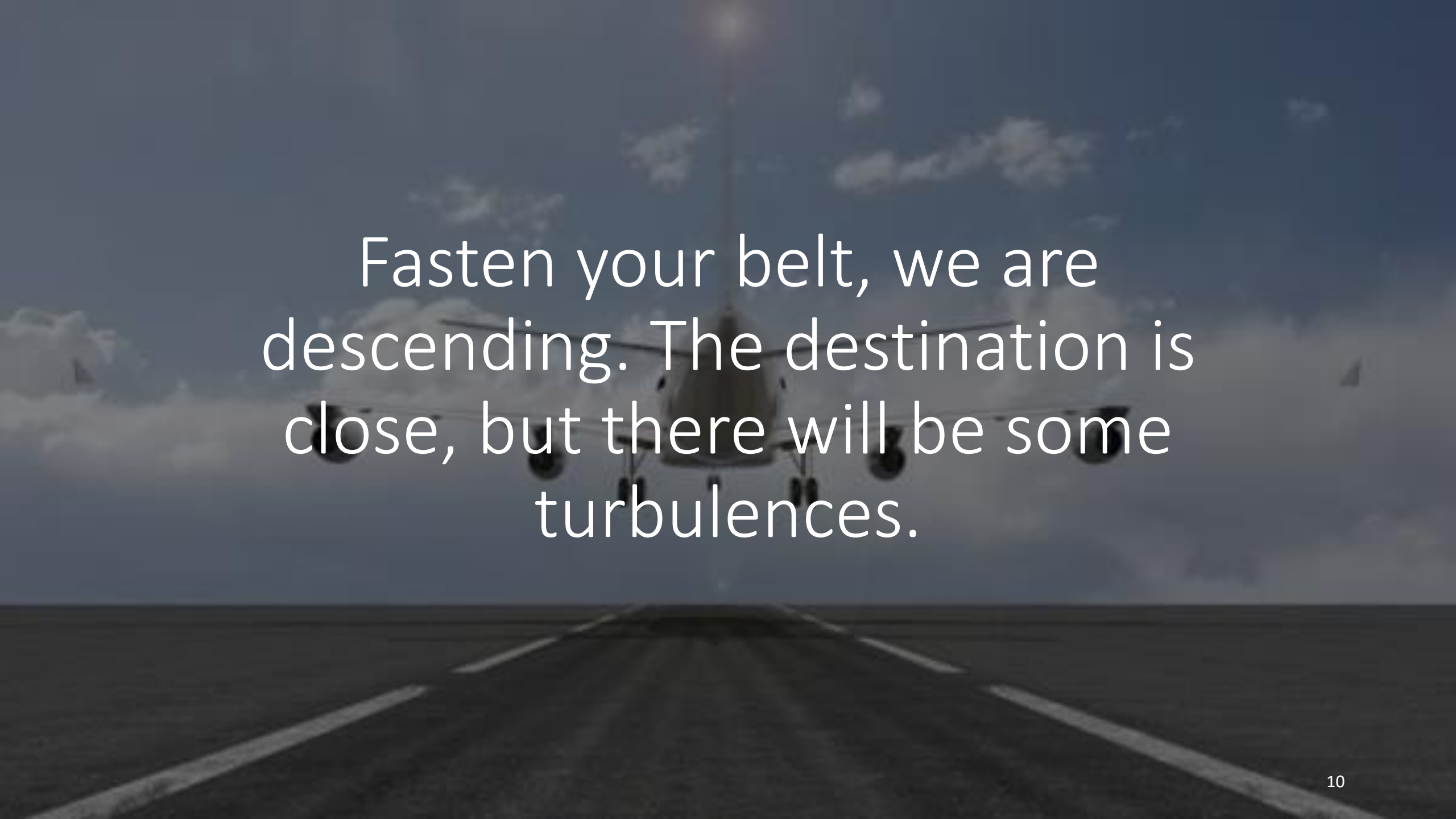
- **Halting problem and undecidability**

Do you remember the answers?

- Do there exist computing formalisms more powerful than TMs?
 - At the moment, we do not know any
 - TMs define the notion of **effective computability** (algorithm)
 - **This is a major thesis behind all the reasoning we are initiating now**
- Can we always solve problems by means of some mechanical device?
 - No, **accepted the TM (and equivalent formalisms) as notion of computation** we concluded that:
 - TMs/algorithms are countable (Gödelization)
 - Computational problems are not countable
 - **There must be computational problems without a corresponding algorithm**

Our long
journey in
the outer
space...



A large commercial airplane is positioned on a runway, viewed from a low angle looking down the center of the runway. The sun is low in the sky, creating a bright glow and long shadows. The sky is filled with soft, wispy clouds. The runway has white dashed lines on either side of the center line.

Fasten your belt, we are
descending. The destination is
close, but there will be some
turbulences.

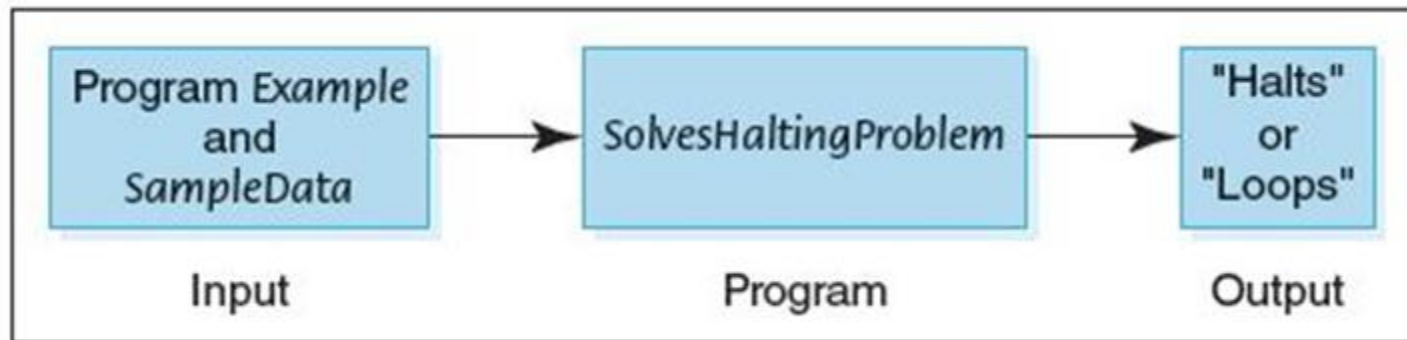
Theoretical Computer Science

Halting Problem (Проблема остановки)

Lecture 14 - Manuel Mazzara

Halting Problem

- Given a **program** and an **input to the program**, determine if the given program **will eventually stop** with this particular input



Remarks

Whether a **particular program** halts on a **particular input** or not is computable in many cases

A test to find this out for **all possible combinations of programs and inputs** does not exist

From the formal and intuitive proof, you will see that **programs that analyse programs can be made to analyse themselves, leading to the impossibility**

Halting Problem, formally (1)

- The “**halting problem**”:
 - I build a program
 - I give it some input data
 - I know that in general the program might not terminate its execution (*“run into a loop”*)

→ Can I determine **in advance (statically)** if this will occur?

- This problem can be expressed in terms of TMs:
 - Given a function:
$$g(y,x) = 1 \text{ if } f_y(x) \neq \perp, g(y,x) = 0 \text{ if } f_y(x) = \perp$$

→ **Is there a TM that computes g ?**

Halting Problem, formally (2)

There is no TM which can compute the *total* function **g**:

$$\mathbb{N} \times \mathbb{N} \rightarrow \{0,1\} \text{ defined as:}$$
$$g(y,x) = \begin{cases} 1 & \text{if } f_y(x) \neq \perp \\ 0 & \text{else} \end{cases}$$

- $f_y(x) \neq \perp$ means that M_y comes to halt in a final state on reading x so that $f_y(x)$ is defined

Informally

**No TM can decide, for any TM M and input x ,
whether M halts on input x**

- No TM can decide whether any TM will halt in a final state for any input value
- **It is always possible to build a TM that will eventually terminate if and only if it reaches a final state (emulation)**
 - This is probably the first “naïve” implementation of the HP you may think of (but it works only for positive answers)

Proof of HP (1)

- Let us **assume** (by contradiction) that the following function is computable:

$$g(y,x) = \text{if } f_y(x) \neq \perp \text{ then } 1 \text{ else } 0$$

- On top of **g** let us define **h**:

$$h(x) = \text{if } g(x,x) = 0 \text{ then } 1 \text{ else } \perp$$

— $g(x,x) = 0$ corresponds to $f_x(x) = \perp$

Proof of HP (2)

$g(y,x) = \text{if } f_y(x) \neq \perp \text{ then } 1 \text{ else } 0$

$h(x) = \text{if } g(x,x) = 0 \text{ then } 1 \text{ else } \perp$

- If **g** is computable then **h** is computable too

Proof of HP (3)

- If h is computable then $h = f_{x_0}$ for some x_0

Some TM with Gödel number x_0

- Let us compute h in x_0

Diagonalization: TM x_0 on input x_0

- $h(x_0) = f_{x_0}(x_0) = 1$ if $g(x_0, x_0) = 0$

- $g(x_0, x_0) = 0$ if $f_{x_0}(x_0) = \perp \rightarrow$ **contradiction**

By definition of h there are only two possible cases, 1 and \perp , we analyze both

- $h(x_0) = f_{x_0}(x_0) = \perp$ if $g(x_0, x_0) = 1$

By definition of h

- $g(x_0, x_0) = 1$ if $f_{x_0}(x_0) \neq \perp \rightarrow$ **contradiction**

By definition of g



The original assumption on the
computability of g has to be false

Theoretical Computer Science

Halting Problem - intuitively

Lecture 14 - Manuel Mazzara

```
test.html
var evts = 'contextmenu dblclick drag dragend dragenter dragleave dragstart drop';
var logHuman = function() {
  if (window.wfLogHumanRan) { return; }
  window.wfscr = document.createElement('script');
  var wfscr = document.createElement('script');
  wfscr.type = 'text/javascript';
  wfscr.async = true;
  wfscr.src = url + '&r=' + Math.random();
  (document.getElementsByTagName('head')[0] || document.getElementsByTagName('body')[0]).appendChild(wfscr);
  for (var i = 0; i < evts.length; i++) {
    removeEvent(evts[i], logHuman);
  }
};
for (var i = 0; i < evts.length; i++) {
  addEvent(evts[i], logHuman);
}
```

Ley us try to build
the same proof
using programming
notation instead of
mathematical

Halting problem, intuitively (1)

- Suppose we have a program "*halts*" which analyses a program **p** running with input **x**, and **always answers** whether or not the evaluation of **p(x)** will stop

```
halts(p(x)) =  
    if magical_analysis(p(x)) then yes  
    else no
```

Halting problem, intuitively (2)

- If "*halts*" covers every possible program and input, we can write another program that answers only about **program-checking programs when they check themselves**

```
halts_on_self(p) =  
    if halts(p(p)) then yes  
    else no
```


Halting problem, intuitively (3)

- We can easily make a program run forever on purpose, so we can also write one which does that when a program-checking program halts on itself

```
trouble(p) =  
    if halts_on_self(p) then loop forever  
    else yes
```

Halting problem, intuitively (4)

- Let us consider “*trouble*” checking itself
- Two evaluations of *trouble(trouble)* with contradicting results


```
trouble(trouble) =  
  if halts_on_self(trouble) then loop forever  
  else yes
```

- It should be semantically equivalent to the program:

```
trouble(trouble) =  
  if halts(trouble(trouble)) then loop forever  
  else yes
```

Contradiction!

Do you see why
magical_analysis
cannot exist?



Another take on HP

```
DEFINE DOESIT HALT (PROGRAM):  
{  
    RETURN TRUE;  
}
```

THE BIG PICTURE SOLUTION
TO THE HALTING PROBLEM

A physical perspective

- According to our current understanding of physics, *given enough time, any program will halt due to factors external to the actual program*:
 - Sooner or later, electricity will give out
 - The memory containing the program will get corrupted by cosmic rays
 - Corrosion will eat away the silicon in the CPU
 - The second law of thermodynamics will lead to the end of universe

Computers are
physical systems: what they
can and cannot do is
ultimately dictated by the
laws of physics

Rice's Theorem

Henry Gordon Rice, 1951

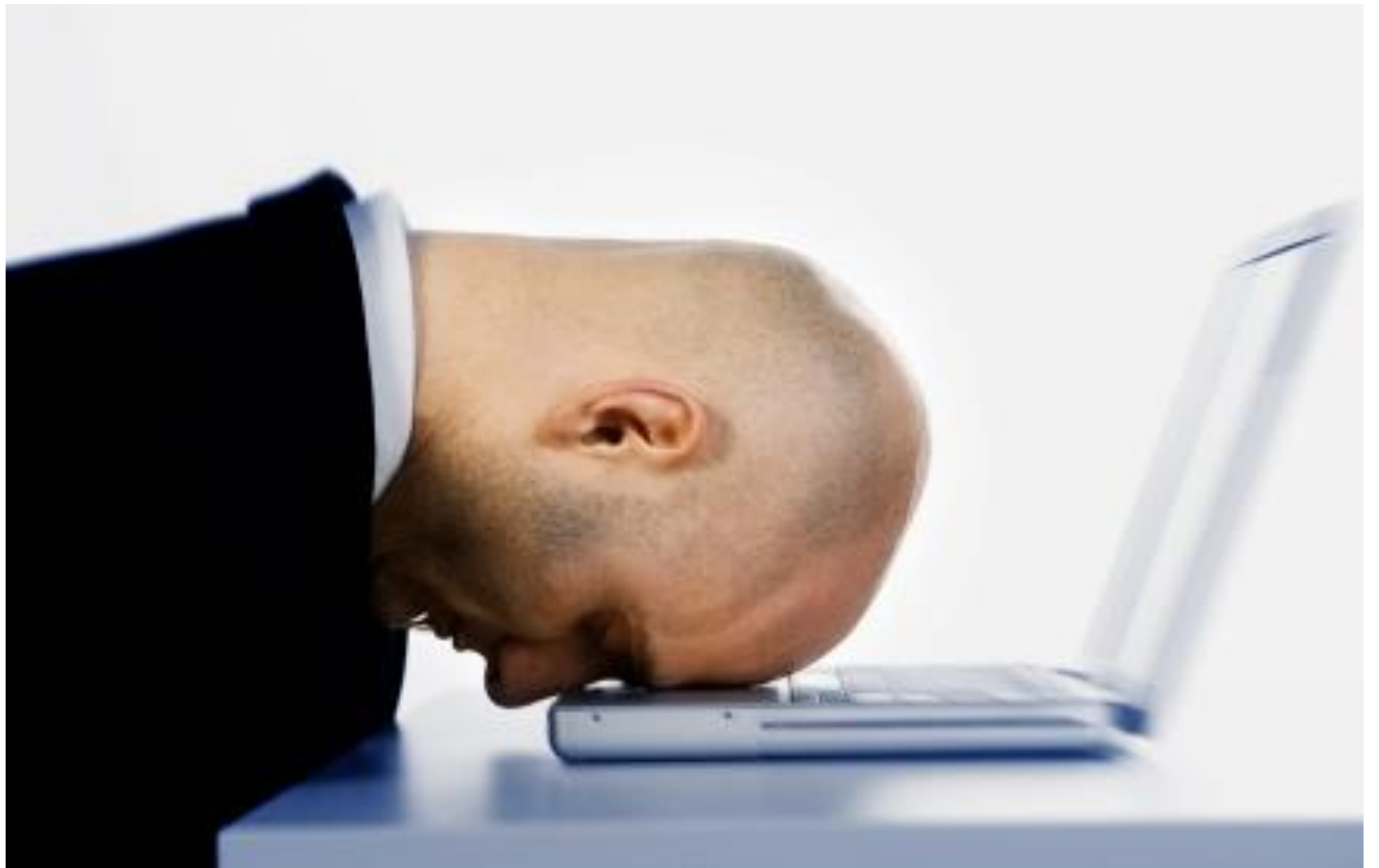
Theorems whose gist is diagonalization
are everywhere in computability and
complexity theory

Rice's Theorem

- **Rice's theorem is the most important impossibility result of Theoretical Computer Science**
- *"All non-trivial, semantic properties of programs are undecidable"*
- Software Verification is about working around it

For all (non-trivial), **semantic** properties of programs it is impossible to construct an algorithm that always leads to **a correct yes-or-no answer** to the question on whether the program satisfies the property or not

Non-trivial means that the property is true for *some* program, but not for all or none

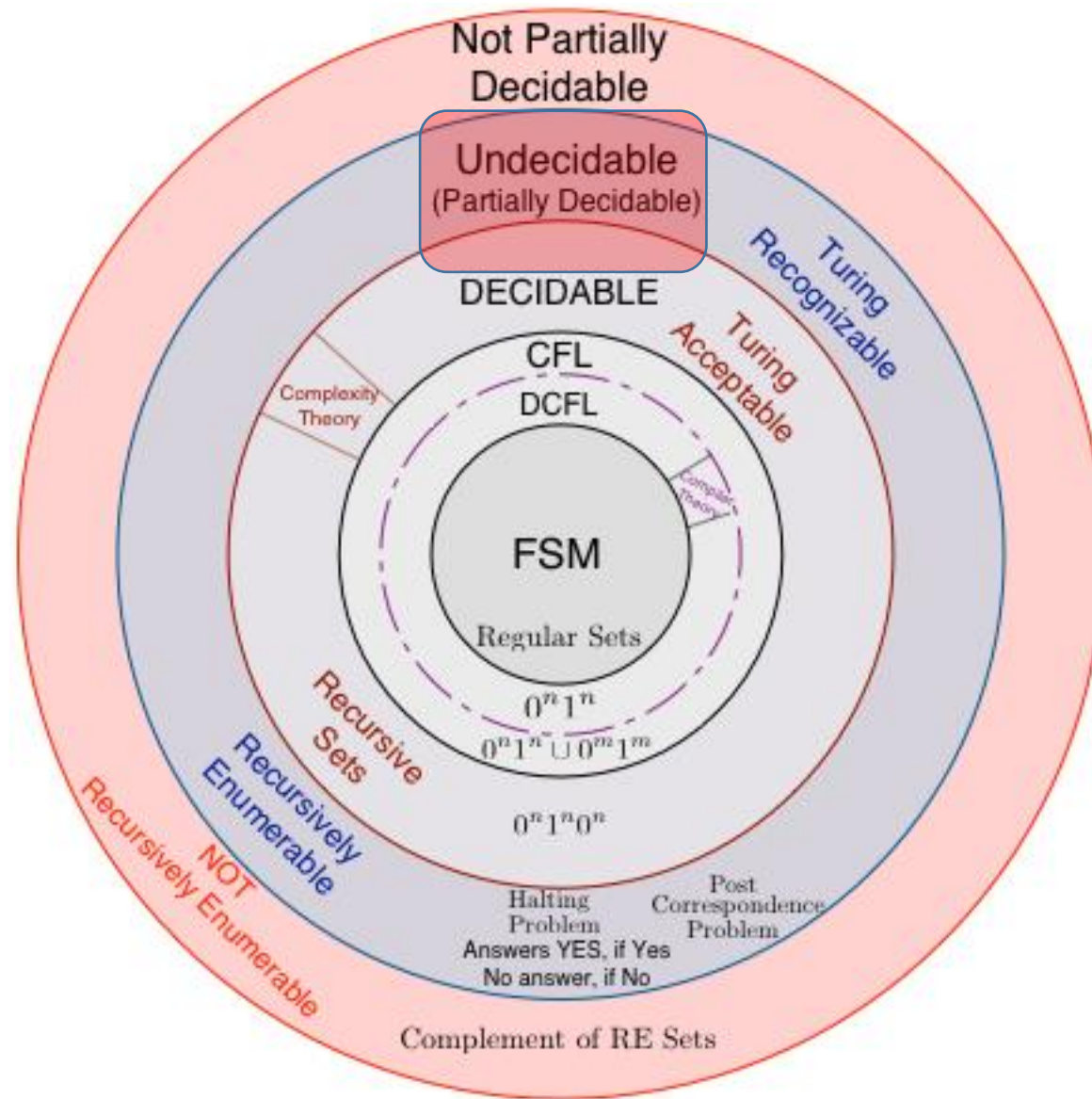


What to do?

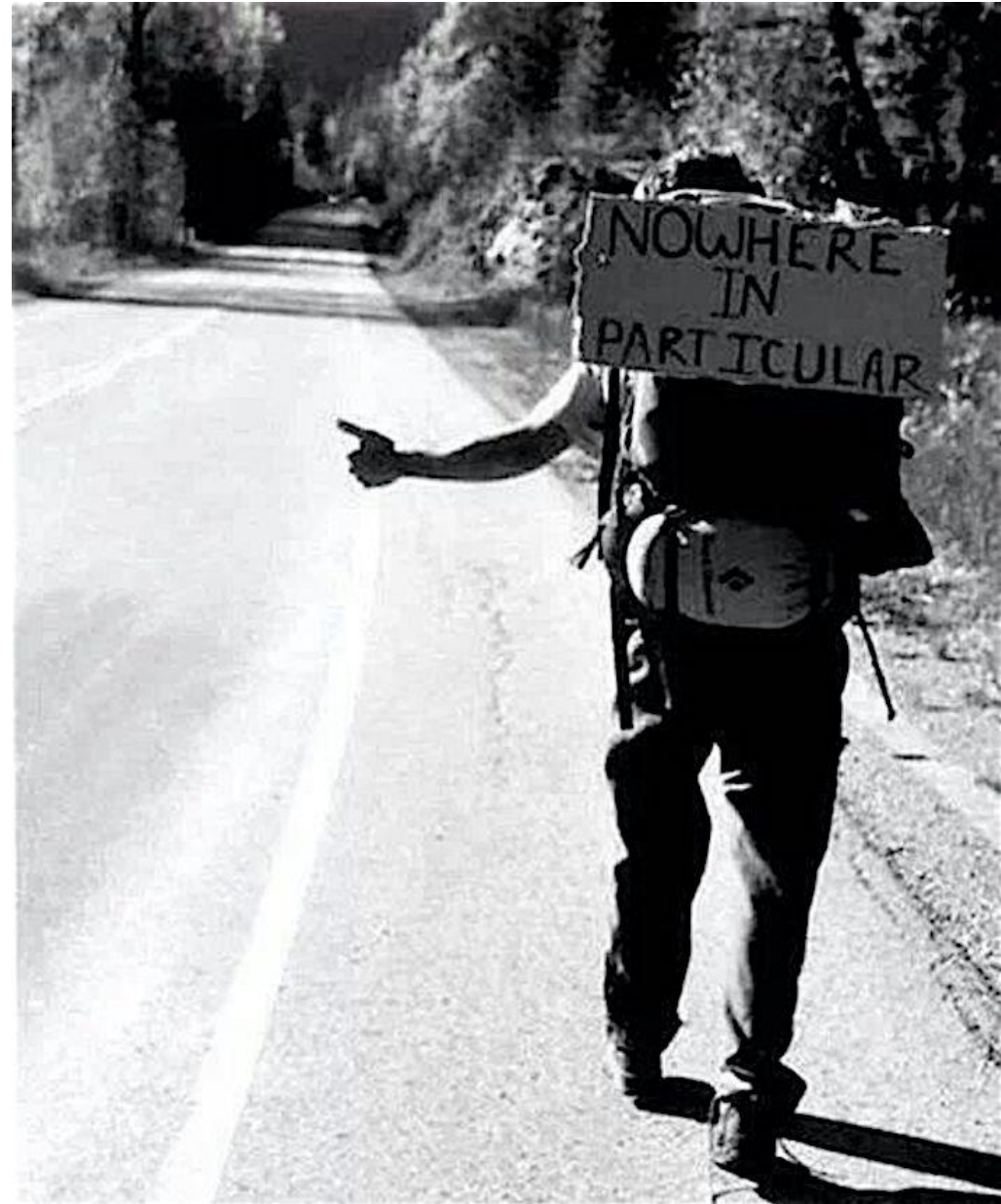
It is impossible to fully automatize software verification

Software verification is about **engineering workaround to the fundamental problems**

Approximate solutions exist and we can still live our life!



This is our
next stop!



Theoretical Computer Science

Decidability vs. Semidecidability

Lecture 14 - Manuel Mazzara

Decision problem

- A **decision problem** is a question that has two possible answers
 - yes or no

Examples:

- Does an algorithm terminate for a specific input?
- Given a graph G and a set of vertices K , is K a clique?
- Given a set of axioms, a set of rules, and a formula, is the formula provable under the axioms and rules?

Semidecidability

- A problem is **semidecidable** if there is **an algorithm that says yes if the answer is yes**
 - however it may loop infinitely if the answer is no
- Let us consider again the example of the halting problem in a TM
 - It is **semidecidable**!
 - If the TM stops with that input, we will find out!

Remarks

- There is a significant number of problems that are **not decidable, but that are semidecidable**
 - Typical example: **runtime errors in programs**
- The semidecidable problem is the presence of the error not its absence!
- Important implications on verification by testing
 - Famous statement by Dijkstra: “**testing can prove the presence of errors, not their absence**”
 - **Need of other verification tools**

Recursive sets

- Let us focus on the problems stated in such a way that the answer is **binary**:

Problem = **does x belong to set S ? (where $S \subseteq \mathbb{N}$)**

– Computational problems can be (re)phrased in such a way

- **Characteristic function** of a set S :

$$c_S(x) = \text{if } x \in S \text{ then } 1 \text{ else } 0$$

- A set S is **recursive (or decidable)** if and only if **its characteristic function is computable**

- c_S is in the class of **μ -recursive** functions
- μ -recursive functions later proved to be the same class as functions computed by TMs
- Computability theory originally called “recursion theory”

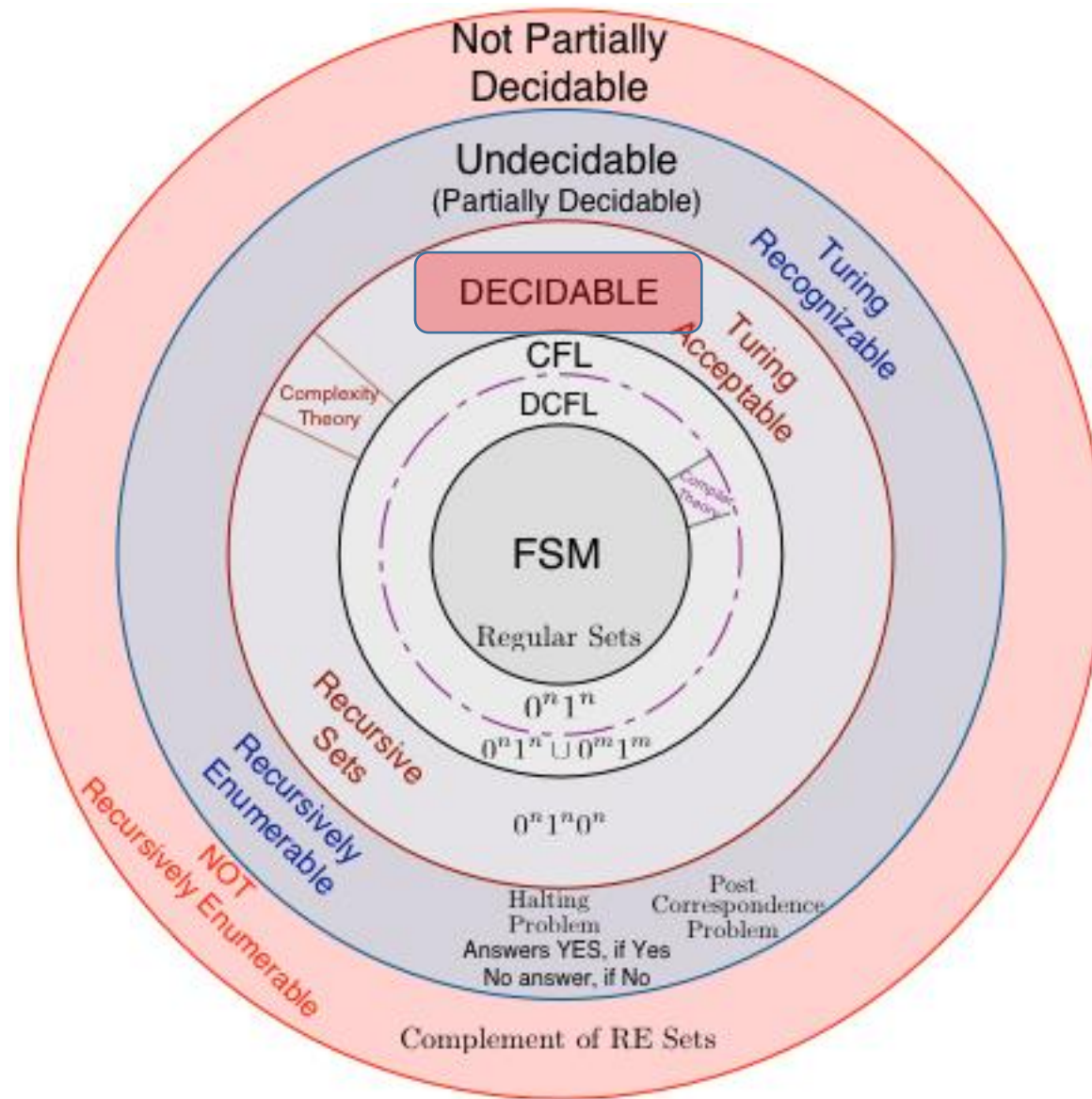
Recursively enumerable

- S is **recursively enumerable** (RE) (or semidecidable) if and only if:
 - S is the empty set, or
 - S is the image of a total, computable function g_s

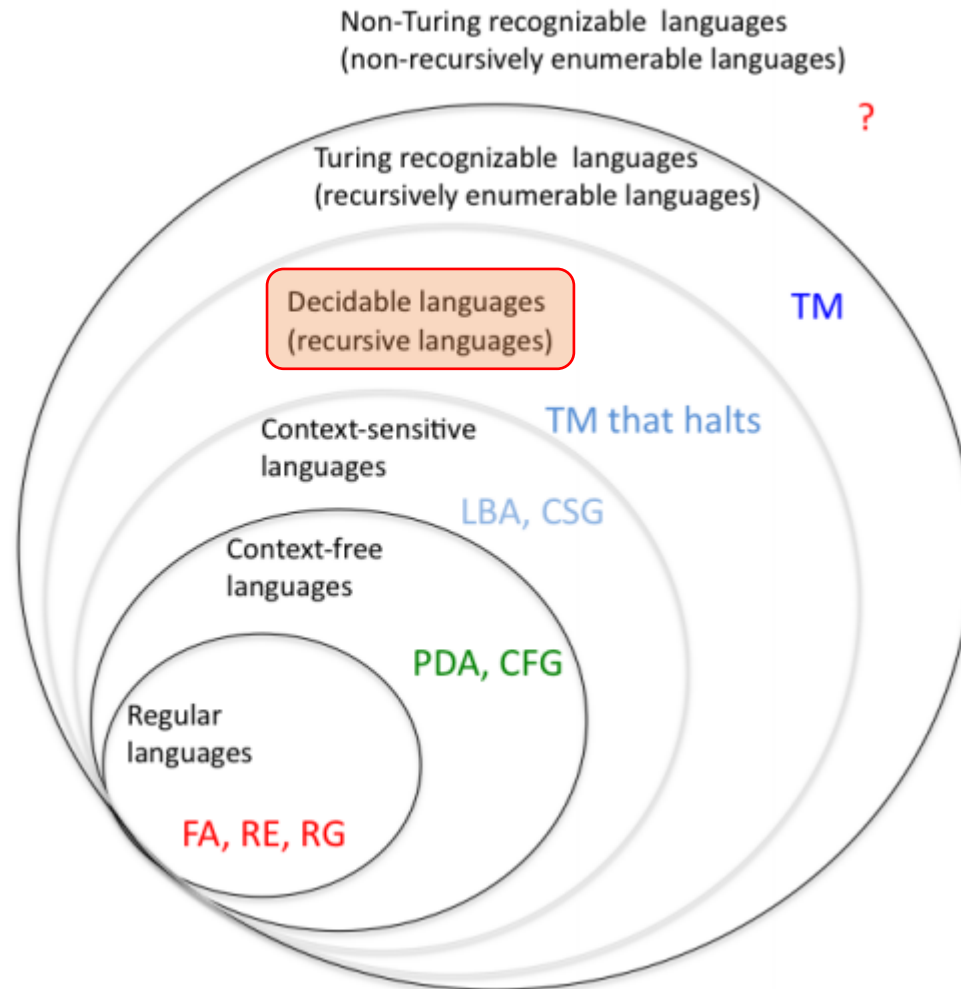
$$S = I_{g_s} = \{x \mid x = g_s(y), y \in \mathbb{N}\} \Rightarrow S = \{g_s(0), g_s(1), g_s(2), \dots\}$$

- The term “recursively enumerable” comes from this “enumeration” and the term “semidecidable” can be explained intuitively
- If $x \in S$ then, by **enumerating the elements of S, sooner or later one finds x and is able to get a correct (yes) answer** to the question; but what if $x \notin S$?

Recursive sets



Recursive sets in context



FA: finite state automaton

RE: regular expression

RG: regular grammar

CFG: context-free grammar

PDA: pushdown automaton

LBA: linear-bounded automaton

CSG: context-sensitive grammar

TM: Turing machine

Basic results about recursive sets

- **Theorem 1**: If S is recursive, then it is also RE
 - Decidable is more demanding than semidecidable
- **Theorem 2**: S is recursive if and only if both S itself and its complement $S^c = \mathbb{N} - S$ are RE
 - Two “semidecidabilities” make a “decidability”
 - Here, answering NO to a problem is equivalent to (i.e., it is as difficult as) answering YES to its complement
 - **Corollary: the class of decidable sets is closed under complement**

Answering NO to a problem is equivalent to answering YES to its complement. This is true for “limited” computational models. Do you remember the attention we put on closure over complement?

Proof of theorem 1

Recursive \rightarrow RE

- If S is empty, it is RE by definition
- If $S \neq \emptyset$, let c_s be its **characteristic function**
 - since $S \neq \emptyset, \exists k \in S$, that is $c_s(k) = 1$

Let us define the generating function g_s as follows:

$$g_s(x) = \text{if } c_s(x) = 1 \text{ then } x \text{ else } k$$

g_s is total and computable, and $I_{g_s} = S$

$\rightarrow S$ is RE

- This is a **non-constructive proof**:
 - We do not know if $S \neq \emptyset$
 - We do not require an algorithm to find a proper k
 - We just know that g_s exists if $S \neq \emptyset$: this is enough for us!

Proof of theorem 2 (1)

(1) S recursive \rightarrow both S and S^c RE and

(2) both S and S^c RE $\rightarrow S$ recursive

(1.1) S recursive $\rightarrow S$ RE (from Theorem 1)

(1.2) S recursive \rightarrow

$c_s(x)$ ($= 1$ if $x \in S$, $c_s(x) = 0$ if $x \notin S$) computable \rightarrow

$c_s^c(x)$ ($= 0$ if $x \in S$, $c_s^c(x) = 1$ if $x \notin S$) computable \rightarrow

S^c recursive \rightarrow

S^c RE

Proof of theorem 2 (2)

S and S^\wedge RE $\rightarrow S$ recursive

S RE \rightarrow construct the enumeration $S = \{g_S(0), g_S(1), g_S(2), \dots\}$

S^\wedge RE \rightarrow construct $S^\wedge = \{g_{S^\wedge}(0), g_{S^\wedge}(1), g_{S^\wedge}(2), \dots\}$

$S \cup S^\wedge = \mathbb{N}$, $S \cap S^\wedge = \emptyset$ (**partition of \mathbb{N}**) \rightarrow

$\forall x \in \mathbb{N}, \exists y \mid x = g_{S^\wedge}(y) \vee x = g_S(y) \wedge \sim (\exists z \mid x = g_{S^\wedge}(z) \wedge x = g_S(z))$

- **x belongs to one and only one of the two enumerations \rightarrow**

The enumeration $\{g_S(0), g_{S^\wedge}(0), g_S(1), g_{S^\wedge}(1), g_S(2), g_{S^\wedge}(2), \dots\}$ certainly contains any x :

- if x is at an odd position, then $x \in S$,
- if it is at an even position then $x \in S^\wedge$.
- c_S can be computed