8 Uncomputability

In this section we will study problems for which we can **prove** that there is no algorithm solving them.

8.1 What is an algorithm?

The notion of algorithm is usually defined as **Turing machines (TMs)** – model of computation developed by Allan Turing in 1930s.

Church-Turing thesis: Any process which could be **naturally** called an effective procedure (or algorithm) can be realized by a TM.

Note: This is NOT a theorem. Since it combines intuitive and mathematical terms, it cannot be proved rigorously.

Evidence in support of Church-Turing thesis:

1. Many other models of computations have been developed and proved to be equivalent to TMs.
2. The class of functions computable by TMs is remarkably insensitive to various modifications of the definition of TMs.
3. There is no known effective procedure that would not have its formal counterpart in terms of a TM.

Note: Church-Turing thesis DOES NOT state that TMs can compute things **as fast** as other computational models.

8.2 RAM model of computation

Turing machines are relatively simple computational model, however to "program" TMs requires acquiring some practice, which is out of the scope for this course. Therefore we need some other computational model which will be closer to a typical von Neumann style computer. Therefore we introduce **Random access machines**.

- **Memory:** array of registers $R_1, R_2, \ldots$; each register can store arbitrarily large natural number
- **Program:** fixed program with numbered lines

  Instruction set:

ℓ: `INC` *op*	increment operand
ℓ: `DEC` *op*	decrement operand
ℓ: `IFZERO` *op*	if operand is zero go to $\ell + 1$ otherwise go to $\ell + 2$
ℓ: `GOTO` *op*	go to line specified by operand

  Operands:

i	natural number i (constant)
R_i	value in register R_i
$@R_i$	value in register R_{R_i}

- **Input and output:** R_1 stores an input, R_2 stores output after RAM finishes

Lemma 1. *RAMs are equivalent to TMs (i.e., everything which can be computed on TMs can be computed on RAMs and vice versa).*

From Church's thesis we can now conclude that **RAMs are at least as strong as any other model of computation.**

Example: RAM to compute $f(n) = 2^n$

```
 1: INC R2          // R2:=1
 2: IFZERO R1       // while R1<>0
 3: GOTO 17
 4: IFZERO R2       // R3:=R2; R2:=0;
 5: GOTO 9
 6: INC R3
 7: DEC R2
 8: GOTO 4
 9: IFZERO R3       // R2:=2*R3; R3:=0;
10: GOTO 15
11: INC R2
12: INC R2
13: DEC R3
14: GOTO 9
15: DEC R1          // R1:=R1-1;
16: GOTO 2
```

Note: This was the last program we have written for RAMs. Instead, we will write algorithms in pseudocodes and use Church's thesis to argue that they can be rewritten as a RAM program.

8.3 Aside: Everything is a natural number

So far RAMs can only perform operations which transform one integer to another integer. What if our problem involves strings or lists of natural numbers?

List is a natural number. List $(u_1, u_2, \ldots, u_n)$ can be represented as natural number:

$$2^{u_1+1} \cdot 3^{u_2+1} \cdot \ldots \cdot p_i^{u_i+1} \cdot \ldots \cdot p_n^{u_n+1},$$

where p_i is the i-th prime number.

From the algebraic properties of prime numbers it follows that it is possible to uniquely decode such encoded list. Moreover, you can easily write functions to encode/decode such list representation.

Note: If we used $p_i^{u_i}$ instead of $p_i^{u_i+1}$, the lists $(2, 3, 0, 0)$ and $(2, 3, 0, 0, 0)$ would have the same representation which we want to avoid.

Character is a natural number. ... use ASCII code

String is a natural number. ... it is just a list of characters

RAM program is a natural number ... it is just a string

Conclusion: Every reasonable data structure can be represented as a natural number (possibly a very large one). Thus we can define any **problem as a function** $f : \mathbb{N} \to \mathbb{N}$.

(We will define $f(x) = 0$, if x does not represent a valid input of a problem.)

Definition 1. *Total function $f : \mathbb{N} \to \mathbb{N}$ is* **recursive**/*computable if there exists a RAM program that computes the function f.*

8.4 Halting problem

Problem: Given a RAM program P and input x. Does P halt on x?

$$\text{HALT}(P, x) = \begin{cases} 1 & \text{if } P \text{ halts on } x \\ 0 & \text{otherwise} \end{cases}$$

Example 1:

```
trivial_function(x):
  while x<>1 do x:=x-2
```

This program halts for odd x and loops forever for even x.

Example 2:

```
mystery_function(x):
  while x<>1 do
    if (x is even) then x:=x/2
    else x:=3*x+1;
```

For all the inputs that people tried, the algorithm finishes. However, there is no known proof that this is the case for all possible values of x. For more information, search for so called "$3x + 1$ problem".

Theorem 1. *There is no RAM program which computes function HALT.*

Proof. Proof by contradiction. Assume that there exists a RAM program that solves HALT. Then we can certainly can create a RAM program with the following pseudocode:

```
NOTHALT(P)
  if HALT(P,P)=1 then loop forever
  else return 1;
```

What happens if we try to run NOTHALT(NOTHALT)?

- **Assume NOTHALT(NOTHALT) halts.**
  By definition of HALT, HALT(NOTHALT, NOTHALT) = 1 and therefore according to our pseudocode NOTHALT(NOTHALT) should loop forever – a contradiction!
- **Assume NOTHALT(NOTHALT) does not halt.**
  Then according to definition of HALT, HALT(NOTHALT, NOTHALT) = 0. Therefore NOTHALT(NOTHALT) should finish and return 1 – a contradiction!

Therefore assuming the existence of RAM program for HALT leads to contradiction of both a claim and its negation. Therefore a RAM program for HALT cannot exist. □

8.5 Diagonalization

The proof presented above is an example of a method called *diagonalization*. Let us present the proof in a different way.

Recall: All the RAM programs are natural numbers and all the inputs are also natural numbers.

Consider a table $H(i, j)$, where $H(i, j)$ will be X, if the RAM program number i will halt on input number j:

	0	1	2	3	4	...
0	X	X		X		...
1	X	X	X	X	X	...
2						...
3		X	X	X		...
4		X	X	X		...
...	...	...	...	...	...	...
NOTHALT			X		X	...

Line for NOTHALT is in fact an "inverse" of diagonal (this follows from the pseudocode which we have given for NOTHALT).

Can NOTHALT be in the table H? NO! For any program number i, NOTHALT differs from its behaviour for input i. However, lines of the table H represent all RAM programs (because every RAM program is assigned a natural number as its representation). Therefore, there is no RAM program for NOTHALT and because HALT was the only element in NOTHALT pseudocode which was not obviously recursive, there can be no RAM program for HALT.

Other uses of diagonalization method:

- Recall **Cantor's theorem** (from calculus course):
  "There is more real numbers than natural numbers"
  "The set of real numbers is non-denumerable"
- Famous **Gödel's incompleteness theorem:**
  "Each formal mathematical system that contains arithmetic is either inconsistent or contains non-provable theorems."

8.6 Turing reductions

Definition 2. *We say that function A is* **Turing reducible to function** *B (or $A \leq^T B$) if there exists an algorithm computing A using B as a subroutine.*

Note: Differences between $A \leq^T B$ and $A \leq_P B$:

- We define $\leq^T$ for all problems, not only for decision problems.
- No restrictions on running time of the algorithm.
- No restrictions on number of calls of function B.

Lemma 2. *If A is non-recursive (uncomputable) and $A \leq^T B$, then B is non-recursive.*

Example: HALT_ALL

$$\text{HALT\_ALL}(P) = \begin{cases} 1, & \text{if } P \text{ halts on } \textbf{all inputs}, \\ 0, & \textit{otherwise}. \end{cases}$$

Lemma 3. *HALT_ALL is non-recursive.*

Proof. By reduction from HALT (i.e., we want to prove HALT $\leq^T$ HALT_ALL)

- We want to create a RAM program that solves function HALT using HALT_ALL as a subroutine.

```
HALT(P,x):
  Q:=encoding of the program
     ‘‘Q(y): return P(x);’’
  return HALT_ALL(Q);
```

i.e., given program P and an input x, we create a new program Q that ignores its input and runs P on input x. This new program Q can be created by a simple string manipulation. Then HALT_ALL is called on program Q and its output is returned as an output of the main program.

- **Claim:** Above mentioned is an implementation of function HALT.
  - **Assume P halts on x.** Then program Q halts on all inputs y and therefore HALT_ALL(Q) must return 1.
  - **Assume P does not halt on x.** Then program Q loops forever on any input y and thus HALT_ALL(Q) must return 0.
- Therefore $HALT \leq^T$ HALT_ALL, and therefore HALT_ALL is a non-recursive function (or: HALT_ALL is not computable).

□

How to prove your favourite problem Q is non-recursive?

1 Choose a function P for which we already know that it is non-recursive.

2 Write pseudocode for RAM program computing function P using Q as a subroutine.

3 Justify, that pseudocode given in 2 indeed computes P.

4 Conclude that since $P \leq^T Q$ and P is non-recursive, also Q must be non-recursive.

Example: EQUIV Given two programs (P_1, P_2), do they exhibit the same behaviour?

$$\text{EQUIV}(P_1, P_2) = \begin{cases} 0, & \text{if there exists } x \text{ such that } P_1(x) \neq P_2(x) \\ & \text{or } P_1 \text{ halts on } x \text{ and } P_2 \text{ does not} \\ & \text{or } P_2 \text{ halts on } x \text{ and } P_1 \text{ does not} \\ 1, & \text{otherwise.} \end{cases}$$

Lemma 4. *EQUIV is non-recursive.*

Proof. By reduction from HALT_ALL(i.e., we want to prove HALT_ALL $\leq^T$ EQUIV)

- We want to create a RAM program for function HALT_ALLusing EQUIV as a subroutine.

```
HALT_ALL(P):
  Q:=encoding of the program ‘‘Q(y): return 0;’’
  R:=encoding of the program ‘‘R(x): P(x); return 0;’’
  return EQUIV(Q,R);
```

- **Claim:** Above mentioned is an implementation of function HALT_ALL.
  - **Note:** Program Q always halts and returns 0
  - **Assume P halts on all inputs.** Then program R halts on all inputs and returns 0. Therefore, EQUIV$(Q, R) = 1$, and the pseudocode returns 1 as expected.
  - **Assume P does not halt on some input x.** Then program R loops forever on input x but Q halts on x. Therefore EQUIV$(Q, R) = 0$, and the pseudocode returns 0 as expected.
- Therefore HALT_ALL $\leq^T$ EQUIV, and since HALT_ALL is non-recursive, EQUIV is a non-recursive function as well.

□

8.7 Universal RAMs

Definition 3. Universal RAM *(we will denote such program by SIM) is a RAM program that takes on the input program P and number x and:*

- *if P does not halt on x, SIM(P, x) does not halt*
- *if P halts on x and returns y, SIM(P, x) halts and returns y*

Lemma 5. *Universal RAMs are recursive.*

Proof sketch.

- RAM programs are essentially very basic assembler
- Write an assembler simulator in language of your choice
- By Church's thesis – you can also do it as a RAM.

Modifications of a universal RAM.

- simulate only first t step of the program (SIM(P, x, t))
- can answer various question about status of simulated RAM after t steps or after the simulation finished

One would think that such universal simulator will be quite sophisticated and complex program.

This is not true: people are continuously "competing" who will find "smaller" universal simulators (e.g., with smaller number of lines of code, using smaller number of registers, ...)

Sketch: How to implement SIM in small number of registers?

- Store all registers of simulated RAM in a single register:

$$2^{R_1} \cdot 3^{R_2} \cdot \ldots \cdot p_i^{R_i} \cdot \ldots$$

- The rest of the simulation will need only (small) constant number of registers — **let's say** < 1000 **registers all together**

8.8 Using a universal RAM to prove non-recursiveness

The following function tries to address the following natural question: how much space programs use?

$$\text{IS\_BIG}_{10000}(P, x) = \begin{cases} 1, & \text{if } P \text{ uses} > 10000 \text{ registers on } x \\ 0, & \text{otherwise} \end{cases}$$

Lemma 6. *Function IS_BIG$_{10000}$ is uncomputable.*

Proof. By reduction from HALT (i.e., want to prove HALT $\leq^T$ IS_BIG$_{10000}$)

- We want to construct a RAM program for function HALT using IS_BIG$_{10000}$ as a subroutine.

```
HALT_ALL(P):
  Q:=encoding of the program
       ‘‘Q(x): SIM(P,x); increment R1,...,R10001;’’
  return IS_BIG_10000(Q,x);
```

- **Claim:** Above mentioned is an implementation of function HALT.
  - **Assume P halts on input x.** Then $\mathrm{SIM}(P,x)$ also halts and thus program Q halts and uses ≥ 10001 registers. Therfore $\mathrm{IS\_BIG}_{10000}(Q,x)=1$ and the pseudocode returns 1 as expected.
  - **Assume P does not halt on input x.** Then $\mathrm{SIM}(P,x)$ loops forever as well and (as stated above) it uses only small number of registers. Therefore $\mathrm{IS\_BIG}_{10000}(Q,x)=0$ and the pseudocode returns 0 as expected.
- Therefore HALT $\leq^T$ $\mathrm{IS\_BIG}_{10000}$, and since HALT is non-recursive, $\mathrm{IS\_BIG}_{10000}$ is a non-recursive function as well.

□

8.9 Using a universal RAM to prove recursiveness

The previous question was not completely indicative of whether the program runs in small amount memory. The main problem is that due to the fact that we can store arbitrarily large number in each of the registers, very complex computations can be performed in small number of registers. The following question tries to address this problem.

Definition 4. *Program P* **overflows** *on input x if it either uses >10000 registers or* **stores value** >10000 **in one of the registers**.

$$\mathrm{IS\_BIG}_{10000,10000}(P,x)=\begin{cases}1, & \text{if } P \text{ overflows on } x\\ 0, & \text{otherwise}\end{cases}$$

Good news: This is computable!

Idea: If we knew that P always halts then it would be enough to simulate it and check whether there was an overflow in the process.

Lemma 7. *If program P with k lines runs on input x without overflow for more than $\boxed{k.10001^{10000}}$ steps then:*

- *it will loop forever, and*
- *is will never overflow.*

Proof. State of the RAM is given by:

- contents of all non-zero registers
- line on which we are in the program

If we know the state of the machine S_i at time i, then state of the machine S_{i+1} at time $i+1$ is uniquely determined.

In particular, this means that if the same state of RAM occurs at two different time points $i<j$ then sequence:

$$S_i, S_{i+1}, S_{i+2}, \ldots, S_{j-1}$$

will be repeated forever and program will loop.

How many non-overflow states of RAM there can be? This is easily computed: $M=k\cdot 10001^{10000}$ (there are 10000 registers, each holding one of the numbers $0\ldots 10000$, and the program can be at one of the k lines)

So after $M+1$ steps, if RAM did not overflow or halt, by pigeon-hole principle we can conclude that **state S_{M+1} already occurred and is identical to the state S_i for some $i\leq M$.**

This means that the sequence of states $S_i, S_{i+1}, \ldots, S_M$ will repeat forever starting with state S_{M+1} and therefore no new states will be seen and program will loop forever and it will never overflow. □

Based on the lemma above we can form the following pseudocode proving that $\text{IS\_BIG}_{10000,10000}$ is recursive.

```
IS_BIG_10000,10000(P,x):
  k := number of lines of P;
  SIM(P,x,k.10000^10001+1);

  if P did overflow during simulation
     return 1;
  else
     return 0;
```

8.10 Conclusion

- Some problems cannot be solved by any algorithm.
- We can prove this fact by diagonalization method, or by a Turing reduction from known non-computable problem.
- Sometimes a small change can make a difference between computable and uncomputable problem.
- Universal RAMs are helpful in **both** proving that something is computable and that something is uncomputable.