1. Formulate Church-Turing thesis. Is it a mathematical theorem? Is it a statement about the physical world?

Thesis: Anything that can be computed on any physical device can also be computed on a Turing machine.

- It is not a mathematical theorem, but a statement about the physical world.
2. Prove that the halting problem is not algorithmically solvable.

No algorithm is possible that given a problem \( p \) and data \( d \) checks if \( p \) halts on \( d \).

Prove that a halt checker is impossible.

\[
\begin{align*}
& \text{P} \quad \text{if } p \text{ halts on } d. \\
& \text{False otherwise.}
\end{align*}
\]

A program is a series of ASCII symbols which are used to turn every symbol into 0's & 1's, that put together form a long sequence of 0's & 1's. This sequence is what is stored in the PC when typing a program. Different binary strings may lead to the same number. To avoid that we append a 1 to the beginning of the string and interpret it as a binary number.

\[
001_2 = 12 \quad \text{1001}_2 = 9 \quad \text{01}_2 = 3
\]

There is an algorithm that given a number \( n \), checks if \( n \) corresponds to a program.

If \( n \) is valid, a compiler generates an executable file denoted by \( J_n \).

By contradiction:

Assume a halt checker exist.

Define function \( F(n) = \begin{cases} J_n(n) + 1 & \text{if } n \text { corresponds to a program \& halts on } n \\\n0 & \text{otherwise} \end{cases} \)

Let \( n_0 \) denote the number corresponding to the program that computes \( F(n) \).

For every \( n \), \( J_{n_0}(n_0) = F(n) \).

For \( n = n_0 \),

\[
J_{n_0}(n_0) = F(n_0)
\]

By definition, \( n_0 \) is a number corresponding to a program.

\( J_0 \) is \( F \), it always halts, so it halts on \( n_0 \) too.

Therefore, \( J_0(n_0) = J_{n_0}(n_0) + 1 \)

This is a contradiction, thus a halt checker does not exist.
3. In the proof that the halting problem is not algorithmically solvable, we use a diagonal function \( f(n) \) which was defined as follows:

- \( f(n) = j_n(n) + 1 \) if \( n \) is a valid code of a Java program and \( j_n \) halts on \( n \);
- \( f(n) = 0 \) otherwise.

Write down the first 4 values \( f(0), f(1), f(2), \) and \( f(3) \) of the diagonal function \( f(n) \) for the following case:

- \( j_0(n) = 3 \);
- \( j_1(n) = n \);
- 2 is not a valid numerical code of a program; and
- \( j_3(n) = n^2 \).

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 3 & 3 & 3 \\
1 & 0 & 1 & 2 \\
2 & - & - & - \\
3 & 0 & 1 & 4 & 9 \\
f(n) & 4 & 2 & 0 & 10
\end{array}
\]

\[
\begin{align*}
 f(0) &= j_0(0) + 1 = 3 + 1 = 4 \\
 f(1) &= j_1(1) + 1 = 1 + 1 = 2 \\
 f(3) &= j_3(3) + 1 = 9 + 1 = 10
\end{align*}
\]
4. Not all algorithms are feasible, but, unfortunately, we do not have a perfect definition of feasibility. Give two examples:

- an example of an algorithm which is feasible according to the current definition but not practically feasible, and
- an example of an algorithm which is practically feasible but not feasible according to the current definition.

1) \( L_n(x) = 10^{150} \cdot \text{len}(x) \rightarrow \text{Feasible by definition, but not in practice.} \)

2) \( t(x) = \exp(10^{10} \cdot \text{len}(x)) \rightarrow \text{Not feasible by definition, but feasible in practice.} \)
5. Briefly describe what is P, what is NP, and what is NP-hard. Give an example of an NP-hard problem. What do we gain and what do we lose when we prove that some problem is NP-hard?

**P:** Problems that can be solved in feasible time.
**NP:** Once there is a candidate for a solution, we can check, in feasible time, whether it is indeed a solution.
**NP-hard:** Every problem from the class NP can be reduced to this problem. This is, problems harder than those of NP.

*Example: SAT*

*What we gain:*
- Any method for solving a problem automatically leads to methods for solving all other problems.

*What we lose:*
- We cannot have an efficient algorithm that solves all problems feasibly.