Limits of computability

Does this program exist?

Is it possible to write a Java method of the form

```java
public boolean test(String program, String input) {
    // some code here
}
```

such that `test` returns
- `true` if the string `program` is the code of a Java method that prints “Hello world!” when called with `input`, and
- `false` otherwise?

Does this program exist?

For example, if the string `program` is

```java
public void abc(String s) {
    if (s == "abc") {
        print "Hello world!";
    } else {
        print "Whatever";
    }
}
```

then
- `test` called with `program` and `input``abc`` yields true;
- `test` called with `program` and any other input yields false.

A helper function

To see if `test` exists, we shall consider a funny program on the next slide. First, we need a helper function: let

```java
int[] enumerate(int i) { ... }
```

(Remember Cantor’s Zig-Zag!)
maybeHelloWorld

```java
String maybeHelloWorld(String s) {
    int x, y, z, n, i;

    i = 1;

    while( true ) {
        (x,y,z,n) = enumerate( i );
        if( exp(x,n) + exp(y,n) == exp(z,n) ) {
            print("Hello world!");break;
        }
        i = i + 1;
    }
}
```

The program `maybeHelloWorld` does the following:

- If there are positive integers \(x, y, z, n\) such that \(n \geq 3\) and
  \[x^n + y^n = z^n,\]
  the it prints “Hello World!”.
- Otherwise, it goes into an infinite loop.
- What happens if we call `test` on `maybeHelloWorld` (and any input)?

Turing machines

```java
public void test2(String program) {
    if(test(program,program)==false) {
        print("Hello world!");
    } else {
        print("Whatever");
    }
}

What happens if we call `test2` with its own code as argument?
Motivation

- Real-life programming languages are too big and messy to make precise claims about computability.
- To address this issue, we shall use idealized notions of computation.
- The first such notion we shall study is that of Turing machine.
- Introduced by Alan Turing in the 1930’s before programming languages even existed.

Computable functions

A function is defined to be
- effectively computable if there is a finite list of instructions by following which one could in principle compute its value for any given argument.
- Turing computable if it is computable by a Turing machine.

Turing’s thesis

- It will be obvious that “Turing-computable” implies “effectively computable”.
- The converse is called “Turing’s thesis”.
- Turing’s thesis cannot be proved, but we shall accumulate evidence during this course.

Turing machine

movable "tape head" (can read and write)

infinite tape consisting of "squares"
Configurations of a Turing Machine

- At any time, the tape head is in one of a finite number of states, and it is scanning one particular square.
- Each tape square is blank (written as 0) or contains a stroke (written as 1).
- We require that at any given time there are only finitely many 1’s.

Actions of the tape head

- \(W_0\): write 0 into current square
- \(W_1\): write 1 into current square
- \(L\): move one square to the left
- \(R\): move one square to the right

Representation of configurations

- Tapes are represented by binary strings, e.g. 1101101.
- There are supposed to be infinitely many 0’s on the left and right.
- Configurations are represented e.g. like 11_q01101. The state (here: \(q\)) is written as a subscript on the scanned symbol.

Formal definition of a Turing machine

**Definition.** A Turing machine (TM) is given by:

- A set \(Q\) of states.
- A **tape alphabet** \(\Sigma\) (we shall use \(\Sigma = \{0, 1\}\)).
- A **transition function** \(\delta: Q \times \Sigma \rightarrow Q \times \{W_0, W_1, L, R\}\). The value of \(\delta(q, a)\), if defined, is a pair \((q', X)\) consisting of a next state \(q'\) and an action \(X\).
- An initial state \(q_0\).

Note the similarity with DFA.
How a TM operates

Let \( q \) be the state of the head, and \( a \) the scanned symbol (i.e. the symbol underneath). Then

- If \( \delta(q, a) \) is undefined, the machine halts.
- Otherwise, let \( (q', X) \) be \( \delta(q, a) \). The head executes action \( X \) and goes to state \( q' \).
  - If \( X \) is \( W_0 \), the head writes \( 0 \).
  - If \( X \) is \( W_1 \), the head writes \( 1 \).
  - If \( X \) is \( L \), the head moves left
  - If \( X \) is \( R \), the head moves right.

Transition table

A Turing machine can be presented as a transition table, for example:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( W_1q_0 )</td>
<td>( Lq_1 )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( W_1q_1 )</td>
<td>( Lq_2 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( W_1q_2 )</td>
<td></td>
</tr>
</tbody>
</table>

Example: doubling the number of 1s

Some functions computable by TM’s

- The machine for doubling strokes can be seen as an implementation of the function that sends the positive integer \( x \) to \( 2 \times x \).
- There is also trivial machine for addition: it needs to do nothing, because the number of strokes after the computation is deemed to be the result.
- There is also a Turing machine for multiplication \( y \times x \) (see Boolos/Burgess/Jeffrey, somewhat messy).
Functions computable by TM’s

- We shall gather strong evidence that every effectively computable function is computable by a TM (i.e. that Turing’s Thesis holds).
- We shall focus on functions that take $k$ natural numbers to a natural number, for some $k \geq 1$:

$$f : N^k \rightarrow N.$$  

- To state what it means for such an $f$ to be Turing-computable, we need some definitions.

Representing natural numbers

- The argument given to the machine is only a representation of a number: string of digits.
- Decimal system, e.g. 1969.
- Binary system: e.g. 11110110001.
- Roman numerals: e.g. MCMLXIX.
- Monadic or tally notation: e.g. the number five is represented by five strokes: 11111.
- The representation turns out to be inessential; for Turing machines, we choose monadic notation.

Standard initial configurations

Definition. A standard initial configuration for arguments $x_1, \ldots, x_k$ is a configuration such that
- there are $k$ blocks of strokes, separated by single blanks;
- the $i$-th block consists of $x_i$ strokes;
- the head is in the start state (state 1) and scans the leftmost stroke. (E.g. $1_{q_0}1101111011$ is the standard initial configuration for arguments 3, 4, 2.)

Halting configurations

Definition. A halting configuration is a configuration that allows no further transition (i.e. if $q$ is the current state and $a$ is the scanned symbol, then $\delta(q, a)$ is undefined).
Standard final configurations

**Definition.** A standard final configuration for result $y$ is a configuration that consists of one block of $y$ strokes, such that the head scans the leftmost stroke and is in a halting configuration.

Definition of Turing computability

**Definition.** A function $f : N^k \rightarrow N$ is computed by a Turing machine $M$ if

- whenever $f(x_1, \ldots, x_k) = y$, then $M$ takes the standard initial configuration for $x_1, \ldots, x_k$ to a standard final configuration with $y$ strokes on the tape;
- whenever $f(y_1, \ldots, x_k)$ is undefined, $M$ never halts, or halts with a non-standard final conf.

A function $f : N^k \rightarrow N$ is called Turing-computable if there is a Turing machine that computes it.

Exercise

Design a TM that will do the following. Given a tape containing a block of 1’s and otherwise blank, if the machine is started the leftmost 1 on the tape, it will eventually halt scanning the rightmost stroke on the tape, having neither printed nor erased anything.
Exercise

Design a TM that, started scanning the leftmost 1 of an unbroken block of 1’s on an otherwise blank tape, adds another 1 to the block and halts, scanning the leftmost 1.

Exercise

Design a TM that will do the following. Given a tape containing a block of \( n \) strokes, followed by a blank, followed by a block of \( m \) strokes, and otherwise blank, if the machine is started scanning the leftmost 1 on the tape, it will eventually halt leaving a block of \( m - n \) strokes if \( m \geq n \) or \( n - m \) strokes if \( n \geq m \), scanning the leftmost stroke on the tape if any stroke is left.

Exercise

Design a TM that, started scanning the leftmost 1 of an unbroken block of 1’s on an otherwise blank tape, eventually halts, scanning a square on an otherwise blank tape, where the square contains a blank or a 1 depending on whether there were an even or an odd number of strokes in the original block.

Hint: recall the parity-checker DFA.

Generalized TM’s

One could allow
- more than two tape symbols,
- replace the tape by a rectangular grid,
- use several heads, several tapes, etc...

Turing’s thesis implies that no generalization will enlarge the class of functions computable. This bold claim turns out to be true for the cases above.
Uncomputability

As we shall see, every Turing machine can be encoded into a list of natural numbers.

- So the set of Turing machines is enumerable.
- As we have seen, the set of functions \( N \rightarrow N \) is not enumerable.
- So there must be functions \( N \rightarrow N \) which are not Turing-computable.
- We shall find concrete examples of non-computable functions.

Enumerating TM’s

Recall that we can present a Turing machine as a transition table.

<table>
<thead>
<tr>
<th>( q )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_1 )</td>
<td>( W_{1q_1} )</td>
<td>( Lq_2 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( W_{1q_2} )</td>
<td>( Lq_3 )</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( W_{1q_3} )</td>
<td></td>
</tr>
</tbody>
</table>

The table can be presented as a list of quadruples:

\( q_10W_{1q_1}, q_11Lq_2, q_20W_{1q_2}, q_21Lq_3, q_30W_{1q_3} \).
Enumerating TM’s

- The sets $Q, \Sigma = \{0, 1\}$, and $\{W_0, W_1, L, R\}$ are finite.
- So the set $Q \times \Sigma \times \{W_0, W_1, L, R\} \times Q$ of quadruples is finite, and in particular enumerable.
- So the set $(Q \times \Sigma \times \{W_0, W_1, L, R\} \times Q)^*$ of finite lists of quadruples is enumerable.
- Because every TM can be represented by such a list, the set of TM’s is enumerable.

Functions not Turing-computable

- So there must be functions $N \rightarrow N$ that are not Turing-computable, for otherwise $f_1, f_2, f_3, \ldots$ would be an enumeration of $N \rightarrow N$, which is impossible because of the diagonalization argument.

Enumerating Turing-computable functions

- We have an enumeration of TM’s:
  
  $M_1, M_2, M_3, \ldots$

- Letting $f_i : N \rightarrow N$ be the function computed by $M_i$, we have an enumeration of the set of Turing-computable functions:
  
  $f_1, f_2, f_3, \ldots$

The diagonal function

- Let $f_1, f_2, f_3, \ldots$ be an enumeration of Turing machines. The diagonal function $d$ is defined as follows:

  $$d(n) = \begin{cases} 
  \bot & \text{if } f_n(n) \text{ is defined}, \\
  1 & \text{otherwise}
  \end{cases}$$

(Recall that we write $\bot$ for “undefined”.)
Link with the Java example

- Recall the Java program `test2` that takes a string program, “runs program on itself”, prints “Hello World!” if program does not print “Hello World!”, and “Whatever” otherwise.
- The diagonal function $d$ is similar: it takes a natural number $n$ that represents a Turing machine, “runs $n$ on itself”, and “does the opposite” of the result.

Uncomputability of $d$

- As we have seen, the Java program `test2` cannot be Java-computable.
- Similarly, the diagonal function $d$ cannot be Turing-computable (see lecture for proof).

The halting function

The **halting function** is defined as follows:

$$h(n, k) = \begin{cases} 2 & \text{if } M_n \text{ halts on input } k \\ 1 & \text{otherwise} \end{cases}$$

Naïve attempt at computing $h(n, k)$

- Run the $n$-th TM, $M_n$, on input $k$.
  - If the computation halts, then go into an infinite loop.
  - If the computation does not halt, return 1.
  - But how long do we wait for the computation to halt?
Self-halting

The **self-halting function** is defined by 
\[ s(n) = h(n, n). \]

**Proposition.** The self-halting function \( s \) is not Turing-computable.

**Proof.** See lecture.

Uncomputability of the halting function

**Corollary.** The halting function \( h \) is not Turing-computable.

**Proof.** See lecture.

Summary

- The main result is that the halting function is not Turing-computable.
- Informally, this means that there is no TM that takes as inputs a code \( n \) for a TM and a number \( m \) and decides whether \( M_n \) halts on \( m \) or not.

Some proofs written up

In case you had difficulties taking notes during the last lecture, I have written up all proofs I did on the OHP.
The diagonal function

Let $M_1, M_2, M_3, \ldots$ be an enumeration of Turing machines, and let $f_1, f_2, f_3, \ldots$ be the resulting enumeration of Turing-computable functions. The diagonal function $d$ is defined as follows:

$$d(n) = \begin{cases} \bot & \text{if } f_n(n) \text{ is defined,} \\ 1 & \text{otherwise} \end{cases}$$

(Recall that we write $\bot$ for “undefined”.)

Uncomputability of $d$

**Proposition.** The diagonal function is not Turing-computable.

**Proof.** By contradiction. So suppose that $d$ is Turing-computable. Then $d$ is the $n$-th Turing-computable function for some $n$, i.e., $d = f_n$. We have

$$d(n) = 1 \iff f_n(n) \text{ is undefined} \iff d(n) \text{ is undefined}$$

(by definition of $d$) \quad (because $d = f_n$)

This is a contradiction, so $d$ cannot be Turing-computable.

The halting function

The halting function is defined as follows:

$$h(n, k) = \begin{cases} 2 & \text{if } M_n \text{ halts on input } k \\ 1 & \text{otherwise} \end{cases}$$

Self-halting

The self-halting function is defined by

$$s(n) = h(n, n).$$

**Proposition.** The self-halting function $s$ is not Turing-computable.
Proof (part 1 of 2)

By contradiction. Suppose that \( s \) is computable by some TM \( M \). From \( M \), we build new TM \( M' \) with the following property:

\[
\begin{align*}
M' & \text{ halts on input } n \quad \text{iff } s(n) = 1 \\
M' & \text{ does not halt on input } n \quad \text{iff } s(n) = 2
\end{align*}
\]

Suppose we have \( M' \). Then we get a contradiction as follows: we know that \( M' \) is the \( k \)-th TM for some \( k \), i.e. \( M' = M_k \). Now

\[
\begin{align*}
M' & \text{ halts on input } k \quad \text{iff } M_k \text{ halts on input } k \quad (\text{because } M' = M_k) \\
M' & \text{ does not halt on input } k \quad \text{iff } h(k, k) = 2 \quad (\text{by definition of } h) \\
& \text{iff } s(k) = 2 \quad (\text{by definition of } s) \\
& \text{iff } M' \text{ does not halt on input } k \quad (\text{because } M' \text{ has the above property})
\end{align*}
\]

This is a contradiction. So \( s \) cannot be Turing-computable.

On the next slide, we convince ourselves that \( M' \) can be built from the (hypothetical) TM \( M \).

Proof (part 2 of 2)

The machine \( M' \), on input \( n \), first proceeds like \( M \). Because \( M \) computes \( s \), we know that \( M \) halts with configuration \( 1_q \) or \( 1_q1 \) for some state \( q \) (depending on whether \( s(n) \) is 1 or 2.) Now \( M' \) checks whether there are one or two strokes on the tape. First, \( M' \) moves right, into some configuration \( 10_r \) or \( 11_r \). In the case \( 10_r \), \( M' \) halts. In the case \( 11_r \), \( M' \) goes into an infinite loop \( 11_r \to 11_r \to 11_r \to \ldots \). The details of building \( M' \) (which I showed in the lecture last time) are straightforward.