Phil 320
Chapter 4: Uncomputable functions and the halting problem

There are some functions that cannot be computed using a Turing machine. The chapter provides a **general argument** that uncomputable functions exist, and **three examples** of uncomputable functions.

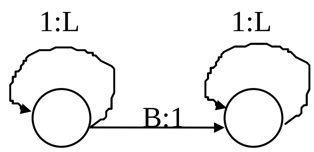
<u>**General argument.**</u> The set of Turing machines is enumerable. So the set of Turing-computable functions $f:P \to P$ is enumerable. But (chapter 2) the set of all functions $f:P \to P$ is not enumerable.

<u>I. Example 1: diagonal function</u>

1. Effective list (or coding) of all Turing-computable 1-place functions.

<u>Step 1</u>: A canonical representation for Turing machines.

a) Start with the quadruple representation.



$q_1S_0S_1q_2$ $q_1S_1Lq_1$
$q_2S_1Lq_2$

b) Add a halting state with no instructions, if needed, as the highest-numbered state.
c) Ensure that a quadruple is present for each state (except the halted state) and each possible scanned symbol (both B and 1). (Add quadruples that do nothing, if needed.)

$q_1S_0S_1q_2$ $q_1S_1Lq_1$
$q_2S_0S_0q_3$ $q_2S_1Lq_2$

d) Abbreviate by dropping the first two symbols in the quadruple, which are obvious from position, and write as a single list:

S_1q_2, Lq_1, S_0q_3, Lq_2

<u>Step 2</u>: Code each Turing machine with a positive integer.

a) Convert the canonical representation to a finite sequence of numbers using 1-4 for the overt action [S_0 = 1, S_1 = 2, L = 3, R = 4] and *i* for state *i*.

2,2,3,1,1,3,3,2

Important: no two machines yield the same list.

b) Convert the finite sequence to a single number, using powers of the prime numbers taken in increasing order:

$$2^2 \cdot 3^2 \cdot 5^3 \cdot 7 \cdot 11 \cdot 13^3 \cdot 17^3 \cdot 19^2$$

c) Place all Turing Machines in order by their code number. This gives an effective listing:

M: $M_1, M_2, \ldots$

and also an effective list of all 1-place Turing-computable functions

L: $f_1, f_2, \ldots$

2. Diagonalization.

Given the above list L, define the *diagonal function d* as follows:

$$d(n) = \begin{cases} 2 \text{ if } f_n(n) \text{ is defined and } = 1 \\ 1 \text{ otherwise} \end{cases}$$

Then d is a total function from P → P, but d is distinct from each entry on the list and hence is not Turing-computable.

1) If Turing's thesis is correct, d is not effectively computable.

2) Puzzle: why isn't d computable? (Halting problem)

4) Second puzzle: take the list of functions f_1, f_2, f_3, ... that you get from the above enumeration of Turing machines, but omitting all the partial functions. Define d as before, but using this new list. Then d is total but distinct from each f_n and hence not Turing-computable. Why not?

II. Example 2: Halting function

With reference to the above list M of Turing machines, define the *halting function* h(m, n) by:

$h(m, n) =$ {1 if machine M_m halts when started with input n
{2 if machine M_m does not halt when started with input n.

- *Halting problem*: find an effective procedure that, given any Turing machine M and any input n, will enable us to determine whether M will halts given input n. [This amounts to showing that h is an effectively computable function.]
- If h were effectively computable, then d would be effectively computable. So if we use Turing's thesis, we know that h is not Turing-computable.

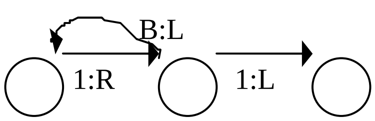
Here, we give a different direct argument that h is not Turing-computable.

Theorem 4.2: The halting function *h* is not Turing computable.

Proof:

Step 1: The *copying machine*, C. Given input n [n 1's], outputs two blocks of n 1's separated by a blank, halting at leftmost 1. *Exercise*: C can be implemented as a Turing machine.

Step 2: The *dithering machine*, D. Given input n, this machine halts (at leftmost 1) if $n > 1$, but otherwise it never halts.



Step 3: Assume, for a contradiction, that there is a machine H that computes h. "Glue together" the machines for H and C. The combined machine (call it G) computes the *self-halting function*

$g(n) = h(n, n) =$ {1 if M_n halts on input n
{2 if M_n does not halt on input n.

Now glue on D to produce the machine M. If M is started on input n:

M halts ↔ $g(n) = 2$ ↔ M_n does not halt on input n
M does not halt ↔ $g(n) = 1$ ↔ M_n halts on input n

We have a contradiction. M is M_m for some m. But if M_m halts on input m, then M (= M_m) does not halt on m (a contradiction); and if M_m does not halt on m, M (= M_m) does halt on m (also a contradiction).

III. Example 3: The Scoring and Productivity Functions

a) Definition of the *scoring function s*.

- A k-state Turing machine is a Turing machine with k-states (not counting the halted state).

- If M is a k-state Turing machine, and we run M with input k (i.e., k 1's), define the *score* of M as follows:

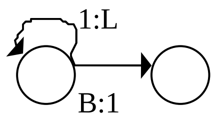
  Score of M = {0 if M does not halt or halts out of standard final position
  {n if M halts in standard position with output n

- Define $s(k)$ = highest score achieved by any k-state Turing machine.

b) The scoring function *s* is not Turing computable.

Proof:

i) If s is Turing-computable, then so is t given by $t(k) = s(k) + 1$. Just glue the following on to the flow diagram for s (letting the leftmost node be the former halting state for s):



ii) t is not Turing-computable. For suppose the k-state machine M computes t. Then if we run M with input k, we halt with output $t(k)$. So the score of M is $t(k)$, which is impossible because $t(k) = s(k) + 1$, but $s(k)$ is the highest score achievable by a k-state Turing machine.

Analysis: Why isn't s effectively computable by surveying the finitely many k-state machines? At any given time, we may not know whether or not all the k-state machines that are going to halt have halted. Again, we come back to the halting problem.

c) The productivity (busy-beaver) function p is not Turing-computable.

- Start a Turing machine M on a blank tape. Define the productivity of M as follows:

  productivity of M = {0, if M does not halt or halts out of s.f.p.
  {n, if M halts in s.f.p. with output n

- Define the *productivity function p* as follows:

  $p(n)$ = productivity of the most productive n-state machine (not counting halted state)

Claim: p is not Turing-computable.

<u>Stage 1: Facts about productivity and *p*</u>

Fact 1: p(1) = 1.

Proof: There are finitely many 1-state machines that use just B and 1. We take a survey (using the flow-graphs) and show that the maximum productivity is 1. There are always two arrows coming out of state 1, but the machines differ in whether 0, 1 or 2 arrows come back to state 1, and in what is printed on the arrows.

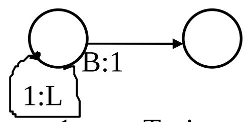
- 0 arrows: productivity is 0 (most of the time) or 1 (if it writes a single 1).
- 2 arrows back to q_1. These machines never halt (because there is always a next instruction to follow), so all have productivity 0.
- 1 arrow back to q_1.

Could be labeled "1:…", where "…" is R or L or B or 1. Never executed, since the arrow pointing to the halted state q_2 is labeled "B: …". The machine has productivity 0 or 1.

The arrow could be labeled "B:…", where "…" is R or L or B or 1. Either the machine never halts and productivity is 0, or (if B:1) it prints a 1 and then follows the other arrow to the halted state; if the instruction is 1:1, the productivity is 1 and otherwise (1:B or 1:L or 1:R) the productivity is 0.

Fact 2: $p(n+1) > p(n)$

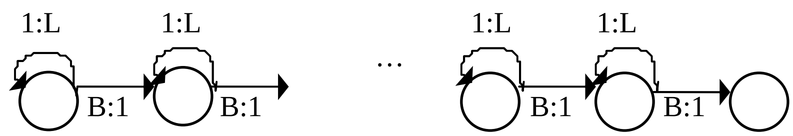
Proof: Take an n-state machine of maximum productivity. Modify its last (halted) node:



This is an n+1-state Turing machine with productivity $p(n) + 1$. So $p(n + 1) > p(n)$.

Fact 3: There is an n-state machine (not counting halted state) that begins on a blank tape, writes n 1's, and halts at the leftmost 1.

Proof:



Fact 4: $p(n + 11) \geq 2n$ for all n.

Proof: Take an n-state machine M_n that writes n 1's (by Fact 3). On the right, glue the 11-state doubler of chapter 3, so that the halted state of M_n is modified to look like state 1 of the doubler. The combined machine takes a blank input, produces a block of n 1's which it feeds to the doubler routine, and finally halts with a block of $2n$ 1's. So its productivity is $2n$. Since this machine has n+11 states, this shows that $p(n+11) \geq 2n$.

<u>Step 2: Proof that the busy beaver problem is unsolvable</u>

Suppose there is a machine BB that has k states (not counting the halted state) and computes $p(n)$: starting on leftmost on n 1's, it halts in the halted state scanning leftmost of $p(n)$ 1's.

First: $p(n + 2k) \geq p(p(n))$ for any n. (1)

As in Fig. 4-3: string "Write n 1's", "BB" and "BB" together (plus a final halted state added at the end). This is an n+2k-state machine that writes $p(p(n))$ 1's. This machine has n + 2k states. So we've shown that $p(n + 2k) \geq p(p(n))$.

Second: since p is strictly increasing (from Fact 2), it must be that

$n+2k \geq p(n)$, for all n; (2)

otherwise, the inequality just proven would be wrong.

Third: to see that (2) is wrong, substitute "$n + 11$" for n in (2) to get:

$n + 11 + 2k \geq p(n + 11)$

But $p(n + 11) \geq 2n$, so for all n we have $n + 11 + 2k \geq 2n$,

which implies that for all n,

$11 + 2k \geq n$. (3)

But (3) is absurd! If $n = 12 + 2k$, for example, we get $11 \geq 12$, which is plainly false. ////

So p is not computable by a Turing machine in standard form. (Again, the halting problem is to blame.)