Uncomputability

FOCS, Fall 2020

Turing machines

Model of computation for decision functions $f : \Sigma^* \to \{\text{true}, \text{false}\}$

- A decision function is Turing-computable if there exists a total Turing machine M such that f(u) = true exactly when M accepts string u

Claim: Turing-computability can be taken as the definition of computability

Church-Turing thesis: every feasible model of computation can be simulated by a Turing machine

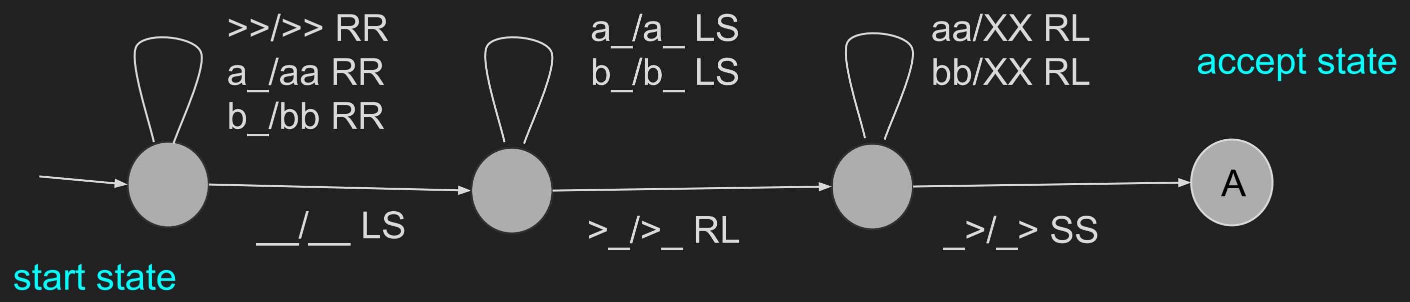
Multitape Turing machines

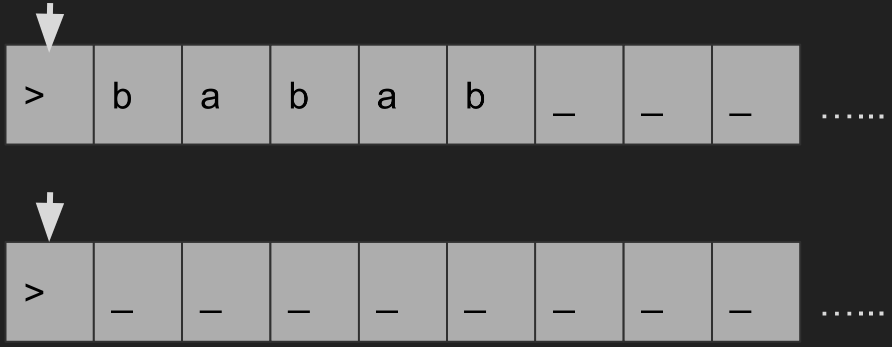
The Church-Turing thesis tells you that extensions to Turing machines don't give you additional computable languages

Consider Turing machines with k tapes ($k \geq 1$):

- defined like normal Turing machines: states, alphabets, start/accept/reject states
- during execution though, there are k tapes, each with its own tape pointer
- transitions now depend on symbols in all pointed-to cells
  - $\partial(q, a_1, \ldots, a_k) = (p, b_1, \ldots, b_k, d_1, \ldots, d_k)$ where $d_i \in \{L, S, R\}$
- first tape initially contains the input string, other tapes empty
- accept/reject on an accept/reject state as usual

Example: 2 tapes Turing machine





This machine accepts all strings that are palindromes over {a,b}

Can simulate 2 tapes with one tape

Transform a 2-tapes machine M_2 into a single tape machine M_1

- every state of M_2 is a set of states of M_1 [one state is the entry point]
- represent the two tapes on a single tape by alternating symbols from first tape and symbols from second tape
    - >abba and >xbax simulated with >axbbbaax (odd positions are on tape 1, even on tape 2)
- mark tape positions in some way (e.g., uppercase symbols)
- set of states corresponding to a state of M_2 scan tape left to right to find the two marked symbols, transform them according to the transitions of M_2, and move M_1 to the entry state of the states corresponding to the new M_2 state

Uncomputable functions

The most interesting consequence of Turing's definition of computability is that there *must* be functions that are *not* Turing-computable!

This is actually a straightforward counting argument:

There are a lot more decision functions than there are Turing machines

It takes a bit of work to make that precise, though, because there are infinitely many decision functions and infinitely many Turing machines

- what does it mean to say that there are more of one than the other?

Focus: alphabet $\{0,1\}$ — the argument generalizes to arbitrary alphabets

Comparing infinite sets

It's easy to say when a finite set A is bigger than a finite set B:

- count the elements — if A has more elements than B then A is bigger than B

The obviously doesn't work if A and B have infinitely many elements

Cantor in the 1880s showed how compare two infinite sets

- try to pair up elements of A and B

We are going to look at a very special case of Cantor's theory

Encoding Turing machines

Fix the alphabet to be {0,1}

The argument relies on encoding Turing machines into strings over {0,1}

Not surprising: on our homework 4, we encoded a Turing machine as OCaml source code:

- a string over the symbols you can use in OCaml source programs (alphanumeric characters, some punctuation)

An explicit encoding

Given $M = (Q, \Gamma, \{0,1\}, \delta, q_s, q_{acc}, q_{rej})$

We can rename Q to be $\{q_1, \ldots q_k\}$ with $q_1 = q_s$, $q_2 = q_{acc}$, $q_3 = q_{rej}$

We can rename Γ to be $\{x_1, \ldots, x_n\}$ with $x_1 = 0$, $x_2 = 1$, $x_3 = >$, $x_4 = \_$

How do we represent the transitions?

- let $d_0 = L$, $d_1 = R$
- each transition is of the form $\delta(q_{n1}, x_{n2}) = (q_{n3}, x_{n4}, d_{n5})$
- can be encoded as a string $0^{n1}10^{n2}10^{n3}10^{n4}10^{n5}$

An explicit encoding

The entire Turing machine be encoded as a string

$111\mathit{code}_1 11\mathit{code}_2 11\mathit{code}_3 11 \ldots\ 11\mathit{code}_k 111$

where each code_i encodes a transition, as above

Write <M> to represent the encoding of Turing machine M as a string over {0,1}

There *must* be uncomputable languages

This is a standard diagonalization argument

Let TM be the set of all Turing machines over input alphabet {0,1}

Let LANG be the set of all languages over {0,1}

Consider $L : TM \to LANG$ mapping every Turing machine M to the language L(M) accepted by M

I claim that L cannot be onto (surjective):

- there must be at least one element in LANG that is not mapped to by L
- that element is a language, and it is by definition uncomputable

Argument

We show that L cannot be onto by arguing by contradiction: we assume that L is onto, and show that this leads to an absurdity. Therefore L cannot be onto.

Define the following language in LANG:

$$B = \{ \langle M \rangle \mid \langle M \rangle \notin L(M) \}$$

B is the set of all strings over {0,1} that represent the encoding of a Turing machine whose language does not contain the string representing its own encoding.

Yeah, weird, I know…

Argument

Because L is onto (that's what we assumed with the hope of deriving a contradiction) and B is a language over {0,1}, there is a Turing machine M_B such that $L(M_B) = B$

M_B is a Turing machine, so it can be encoded in a string $\langle M_B \rangle$

Now ask yourself the question: does $\langle M_B \rangle \in B$ or not?

There are two possibilities: either $\langle M_B \rangle \in B$ or $\langle M_B \rangle \notin B$

Each is problematic

Argument

If $\langle M_B \rangle \in B$, then by definition of B, this means that $\langle M_B \rangle \notin L(M_B)$. But $L(M_B) = B$, so $\langle M_B \rangle \notin B$. Thus, if $\langle M_B \rangle \in B$, then we must also have $\langle M_B \rangle \notin B$ — which makes no sense

Argument

If $\langle M_B \rangle \in B$, then by definition of B, this means that $\langle M_B \rangle \notin L(M_B)$. But $L(M_B) = B$, so $\langle M_B \rangle \notin B$. Thus, if $\langle M_B \rangle \in B$, then we must also have $\langle M_B \rangle \notin B$ — which makes no sense

If $\langle M_B \rangle \notin B$, then by definition of B, this means that $\langle M_B \rangle \in L(M_B)$. But $L(M_B) = B$, so $\langle M_B \rangle \in B$. Thus, if $\langle M_B \rangle \notin B$, then we must also have $\langle M_B \rangle \in B$ — which makes no sense

Argument

If $\langle M_B \rangle \in B$, then by definition of B, this means that $\langle M_B \rangle \notin L(M_B)$. But $L(M_B) = B$, so $\langle M_B \rangle \notin B$. Thus, if $\langle M_B \rangle \in B$, then we must also have $\langle M_B \rangle \notin B$ — which makes no sense

If $\langle M_B \rangle \notin B$, then by definition of B, this means that $\langle M_B \rangle \in L(M_B)$. But $L(M_B) = B$, so $\langle M_B \rangle \in B$. Thus, if $\langle M_B \rangle \notin B$, then we must also have $\langle M_B \rangle \in B$ — which makes no sense

So neither $\langle M_B \rangle \in B$ nor $\langle M_B \rangle \notin B$ can be true. But that's absurd. So our initial assumption must have been wrong: L cannot be onto

And B itself is uncomputable

Conclusion

Language B shows that there are some languages that are not computable

By the Church-Turing thesis, it is impossible to implement a function f in *any computation model or programming language* with the property that f(u) = true exactly when $u \in B$

B feels a bit artificial though — it was built specifically for this proof

Are there any *natural* languages / decision functions that are uncomputable?

Yes!

The Halting Problem

We can define an encoding <M> for Turing machines as strings in {0,1}*

We can extend to an encoding <M, w> for Turing machines *and an input string*

- We should be able to construct <M, w> from <M> and w

Define HP = { <M, w> | M accepts or rejects w } = { <M, w> | M halts on w }

Claim: *HP is uncomputable*

Argument

Again, we proceed by contradiction. Suppose that HP were computable. I'll show that leads to an absurd situation.

Since we assumed HP was computable, then by definition there exists a total Turing machine H such that:

- H accepts <M, w> when <M, w> $\in$ HP, i.e., when M halts on w
- H rejects <M, w> when <M,w> $\notin$ HP, i.e., when M loops on w

I'm going to construct something unholy with H

Turing machine K

Construct a Turing machine K that on input <M>:

- replaces <M> by <M, <M>>
- runs H on <M, <M>> (as a subroutine)
- if H rejects, then accept
- if H accepts, then go into an infinite loop

(You can do that by having the first part of K rewrite <M> into <M, <M>>, then jump to states that do exactly what H does, except changing the reject state of H by an accept state, and replacing the accept state of H by two states that go back and forth amongst themselves)

Turing machine K

Turing machine K is interesting. It is a Turing machine, so of course it has an encoding <K>. What happens if we run K with input <K>?

Does K halt on <K>? If it does then by definition of K:

- H rejects <K, <K>>
- that is, <K, <K>> $\notin$ HP
- that is, K loops on <K>

Oops...

Turing machine K

Turing machine K is interesting. It is a Turing machine, so of course it has an encoding <K>. What happens if we run K with input <K>?

Does K halt on <K>? no

Does K loop on <K>? If it does then by definition of K:

- H accepts <K, <K>>
- that is, <K, <K>> $\in$ HP
- that is, K halts on <K>

Oops...

Turing machine K

Turing machine K is interesting. It is a Turing machine, so of course it has an encoding <K>. What happens if we run K with input <K>?

Does K halt on <K>? no

Does K loop on <K>? no

But that's impossible. K must either halt or loop on <K>. That's our contradiction. Our assumption that H exists must be false, that is, HP is not computable.

Reducibility

Once you have one uncomputable problem, you can show other problems are uncomputable by *reduction*

X is uncomputable if you can reduce solving a known uncomputable problem to the problem of solving X

Examples:

- MP = { <M, w> | M accepts w }
- NULL = { <M> | M accepts ε }

MP is uncomputable

MP = { <M, w> | M accepts w }

Show that we can solve the Halting Problem with MP (that is, reduce HP to MP).

Given a string <M, w>. Consider the Turing machine M encoded by the string. Let M' be the modification of M where every transition that goes to the reject state now goes to the accept state. It's clear that M' accepts w exactly when M accepts or rejects w. So <M', w> ∈ MP exactly when <M, w> ∈ HP

This mean MP must be uncomputable. If it were computable, you could compute HP by taking <M, w>, converting it to <M', w>, and checking if <M', w> ∈ MP

NULL is uncomputable

NULL = { <M> | M accepts ε }

Show that we can solve the Halting Problem with NULL

Consider the following family of Turing machines:

$F_{M,\ w}$ = TM that on input x replaces x by w and runs M on w, accepting if M halts.

We have $L(F_{M,w}) = \{0,1\}^*$ if M halts on w, $= \varnothing$ otherwise. So $\langle M, w\rangle \in HP$ exactly when $\varepsilon \in L(F_{M,w})$, that is when $\langle F_{M,w}\rangle \in NULL$.

NULL must be uncomputable. If it were computable, you could compute HP by taking $\langle M, w\rangle$, converting it to $\langle F_{M,w}\rangle$, and checking if $\langle F_{M,w}\rangle \in NULL$.

Rice's Theorem (restricted version)

Let P be a property of languages. If there is at least one Turing machine whose language has property P and at least one Turing machine whose language does not have property P, then $\{ \langle M \rangle \mid L(M) \text{ has } P \}$ *is uncomputable.*

Basically any interesting property of the language of Turing machines is uncomputable

⇒ Any interesting behavioral property of programs in any programming language is uncomputable

(*) For all Turing machines M, N, if L(M) = L(N), then M has P iff N has P

(A note on terminology)

We're using the terms computable, Turing-computable, uncomputable

In the literature, Turing-computability is often called decidability, and uncomputable languages/functions are called undecidable.

Stepping away from Turing machine problems

The Halting Problem and the various uncomputable languages obtained via Rice's theorem are all problems that talk about Turing machines

- e.g., the language of all (encoding of) Turing machines with a certain property

Obvious question: are there natural uncomputable languages that are not about Turing machines?

Post Correspondence Problem

Consider the following problem:

You are given an infinite supply of "dominoes", each domino one of the following form:

$$\frac{u_1}{v_1} \quad \frac{u_2}{v_2} \quad \frac{u_3}{v_3} \quad \cdots \quad \frac{u_k}{v_k}$$

Is there a way to choose a finite number of dominoes so that when you put them in a sequence, both the top row and the bottom row yield the same string?

Examples

$$\frac{a}{\varepsilon} \qquad \frac{a}{ab} \qquad \frac{b}{aa}$$

Examples

$$\frac{a}{\varepsilon} \quad \frac{a}{ab} \quad \frac{b}{aa} \longrightarrow \text{YES} \quad \frac{a}{ab}\ \frac{b}{aa}\ \frac{a}{\varepsilon}\ \frac{a}{\varepsilon}$$

Examples

$$\frac{a}{\varepsilon}\quad\frac{a}{ab}\quad\frac{b}{aa}\quad\longrightarrow\quad\text{YES}\qquad\frac{a}{ab}\quad\frac{b}{aa}\quad\frac{a}{\varepsilon}\quad\frac{a}{\varepsilon}$$

$$\frac{a}{ba}\quad\frac{b}{a}\quad\frac{b}{ab}$$

Examples

$$\frac{a}{\varepsilon} \quad \frac{a}{ab} \quad \frac{b}{aa} \quad \longrightarrow \quad \text{YES} \quad \frac{a}{ab} \quad \frac{b}{aa} \quad \frac{a}{\varepsilon} \quad \frac{a}{\varepsilon}$$

$$\frac{a}{ba} \quad \frac{b}{a} \quad \frac{b}{ab} \quad \longrightarrow \quad \text{NO} \quad \text{(bottom row longer unless you use only 2nd)}$$

Post Correspondence Problem

PCP = { $u_1\#v_1\#u_2\#v_2\#u_3\#v_3\ldots\#u_k\#v_k$ | there exists a sequence $i_1, i_2, i_3, \ldots, i_N$ with

$$u_{i1}u_{i2}u_{i3}\ldots u_{iN} = v_{i1}v_{i2}v_{i3}\ldots v_{iN}$$ }

Theorem: *PCP is uncomputable*

Argument

Same as before. We show that the Halting Problem reduces to PCP. That is, if you could solve PCP, you could solve the Halting Problem

How could you possibly do that?

- given a Turing machine M and an input w
- transform M and w into a set of dominoes with the property:
  - M halts on w exactly when you can solve PCP with those dominoes

Trick: use dominoes whose only solution describes a sequence of configurations of the Turing machine that leads to a halting configuration!

From Turing machines to dominoes...

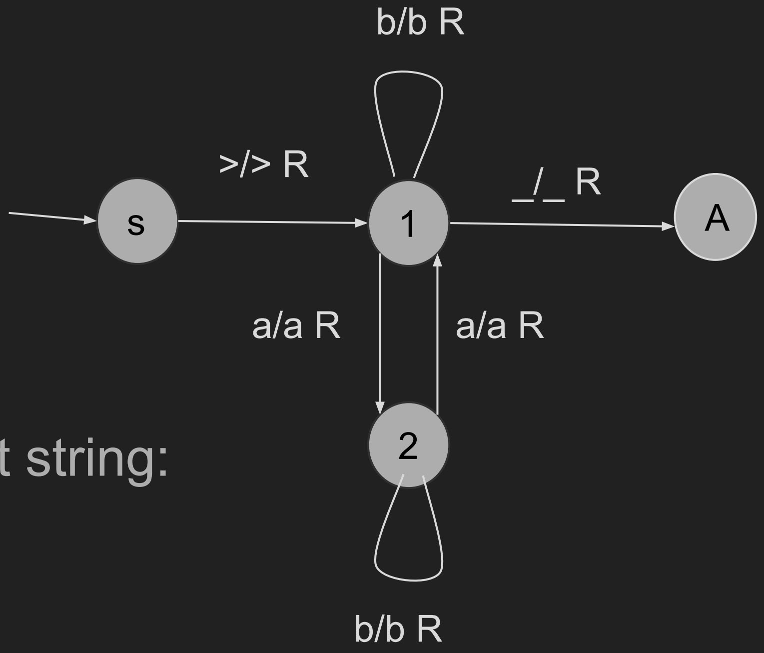
$M = (Q, \Gamma, \Sigma, \delta, q_s, q_{acc}, q_{rej})$ input string $w = a_1 \ldots a_k$

$$\frac{\#}{\#q_s{>}a_1 \ldots a_k\#} \qquad \frac{\#}{\#} \quad \frac{\#}{\_\#} \quad \frac{a}{a} \text{ (for all a)}$$

$$\frac{qa}{bp} \text{ if } \partial(q, a) = (p, b, R) \text{ (for all p, q, a, b)} \qquad \frac{q_{acc}\#\#}{\#} \quad \frac{aq_{acc}}{q_{acc}} \quad \frac{q_{acc}a}{q_{acc}} \text{ (for all a)}$$

$$\frac{cqa}{pcb} \text{ if } \partial(q, a) = (p, b, L) \text{ (for all p, q, a, b, c)} \qquad \frac{q_{rej}\#\#}{\#} \quad \frac{aq_{rej}}{q_{rej}} \quad \frac{q_{rej}a}{q_{rej}} \text{ (for all a)}$$

Example



Input string:
aba

#	s>	1b	2b
#s>aba#	>1	b1	b2

1a	2a	1_	>	a	b	_
a2	a1	_A	>	a	b	_

A##	>A	A>	aA	Aa
#	A	A	A	A

bA	Ab	_A	A_	#	#
A	A	A	A	#	_#

(missing dominoes with reject state)

$$\frac{\#}{\#s>aba\#}$$

$$\frac{\#}{\#s{>}aba\#} \quad \frac{s{>}}{{>}1} \quad \frac{a}{a} \quad \frac{b}{b} \quad \frac{a}{a} \quad \frac{\#}{\#}$$

#	s>	a	b	a	#	>	1a	b	a	#
#s>aba#	>1	a	b	a	#	>	a2	b	a	#

$\frac{\#}{\#s>aba\#}$ $\frac{s>}{>1}$ $\frac{a}{a}$ $\frac{b}{b}$ $\frac{a}{a}$ $\frac{\#}{\#}$ $\frac{>}{>}$ $\frac{1a}{a2}$ $\frac{b}{b}$ $\frac{a}{a}$ $\frac{\#}{\#}$

$\frac{>}{>}$ $\frac{a}{a}$ $\frac{2b}{b2}$ $\frac{a}{a}$ $\frac{\#}{\#}$

#	s>	a	b	a	#	>	1a	b	a	#
#s>aba#	>1	a	b	a	#	>	a2	b	a	#

>	a	2b	a	#	>	a	b	2a	#
>	a	b2	a	#	>	a	b	a1	_#

#	s>	a	b	a	#	>	1a	b	a	#
#s>aba#	>1	a	b	a	#	>	a2	b	a	#

>	a	2b	a	#	>	a	b	2a	#	>	a	b	a	1_	#
>	a	b2	a	#	>	a	b	a1	_#	>	a	b	a	_A	#

$$\frac{\#}{\#s>aba\#}\ \frac{s>}{>1}\ \frac{a}{a}\ \frac{b}{b}\ \frac{a}{a}\ \frac{\#}{\#}\ \frac{>}{>}\ \frac{1a}{a2}\ \frac{b}{b}\ \frac{a}{a}\ \frac{\#}{\#}$$

$$\frac{>}{>}\ \frac{a}{a}\ \frac{2b}{b2}\ \frac{a}{a}\ \frac{\#}{\#}\ \frac{>}{>}\ \frac{a}{a}\ \frac{b}{b}\ \frac{2a}{a1}\ \frac{\#}{\_\#}\ \frac{>}{>}\ \frac{a}{a}\ \frac{b}{b}\ \frac{a}{a}\ \frac{1\_}{\_A}\ \frac{\#}{\#}$$

$$\frac{>}{>}\ \frac{a}{a}\ \frac{b}{b}\ \frac{a}{a}\ \frac{\_A}{A}\ \frac{\#}{\#}$$

$$\frac{\#}{\#s{>}aba\#}\;\frac{s{>}}{{>}1}\;\frac{a}{a}\;\frac{b}{b}\;\frac{a}{a}\;\frac{\#}{\#}\;\frac{>}{>}\;\frac{1a}{a2}\;\frac{b}{b}\;\frac{a}{a}\;\frac{\#}{\#}$$

$$\frac{>}{>}\;\frac{a}{a}\;\frac{2b}{b2}\;\frac{a}{a}\;\frac{\#}{\#}\;\frac{>}{>}\;\frac{a}{a}\;\frac{b}{b}\;\frac{2a}{a1}\;\frac{\#}{\_\#}\;\frac{>}{>}\;\frac{a}{a}\;\frac{b}{b}\;\frac{a}{a}\;\frac{1\_}{\_A}\;\frac{\#}{\#}$$

$$\frac{>}{>}\;\frac{a}{a}\;\frac{b}{b}\;\frac{a}{a}\;\frac{\_A}{A}\;\frac{\#}{\#}\;\frac{>}{>}\;\frac{a}{a}\;\frac{b}{b}\;\frac{aA}{A}\;\frac{\#}{\#}$$

#	s>	a	b	a	#	>	1a	b	a	#
#s>aba#	>1	a	b	a	#	>	a2	b	a	#

>	a	2b	a	#	>	a	b	2a	#	>	a	b	a	1_	#
>	a	b2	a	#	>	a	b	a1	_#	>	a	b	a	_A	#

>	a	b	a	_A	#	>	a	b	aA	#	>	a	bA	#
>	a	b	a	A	#	>	a	b	A	#	>	a	A	#

#	s>	a	b	a	#	>	1a	b	a	#
#s>aba#	>1	a	b	a	#	>	a2	b	a	#

>	a	2b	a	#	>	a	b	2a	#	>	a	b	a	1_	#
>	a	b2	a	#	>	a	b	a1	_#	>	a	b	a	_A	#

>	a	b	a	_A	#	>	a	b	aA	#	>	a	bA	#
>	a	b	a	A	#	>	a	b	A	#	>	a	A	#

>	aA	#
>	A	#

#	s>	a	b	a	#	>	1a	b	a	#
#s>aba#	>1	a	b	a	#	>	a2	b	a	#

>	a	2b	a	#	>	a	b	2a	#	>	a	b	a	1_	#
>	a	b2	a	#	>	a	b	a1	_#	>	a	b	a	_A	#

>	a	b	a	_A	#	>	a	b	aA	#	>	a	bA	#
>	a	b	a	A	#	>	a	b	A	#	>	a	A	#

>	aA	#	>A	#
>	A	#	A	#

$$\frac{\#}{\#s>aba\#}\quad\frac{s>}{>1}\quad\frac{a}{a}\quad\frac{b}{b}\quad\frac{a}{a}\quad\frac{\#}{\#}\quad\frac{>}{>}\quad\frac{1a}{a2}\quad\frac{b}{b}\quad\frac{a}{a}\quad\frac{\#}{\#}$$

$$\frac{>}{>}\quad\frac{a}{a}\quad\frac{2b}{b2}\quad\frac{a}{a}\quad\frac{\#}{\#}\quad\frac{>}{>}\quad\frac{a}{a}\quad\frac{b}{b}\quad\frac{2a}{a1}\quad\frac{\#}{\_\#}\quad\frac{>}{>}\quad\frac{a}{a}\quad\frac{b}{b}\quad\frac{a}{a}\quad\frac{1\_}{\_A}\quad\frac{\#}{\#}$$

$$\frac{>}{>}\quad\frac{a}{a}\quad\frac{b}{b}\quad\frac{a}{a}\quad\frac{\_A}{A}\quad\frac{\#}{\#}\quad\frac{>}{>}\quad\frac{a}{a}\quad\frac{b}{b}\quad\frac{aA}{A}\quad\frac{\#}{\#}\quad\frac{>}{>}\quad\frac{a}{a}\quad\frac{bA}{A}\quad\frac{\#}{\#}$$

$$\frac{>}{>}\quad\frac{aA}{A}\quad\frac{\#}{\#}\quad\frac{>A}{A}\quad\frac{\#}{\#}\quad\frac{A\#\#}{\#}$$

#	s>	a	b	a	#	>	1a	b	a	#
#s>aba#	>1	a	b	a	#	>	a2	b	a	#

>	a	2b	a	#	>	a	b	2a	#	>	a	b	a	1_	#
>	a	b2	a	#	>	a	b	a1	_#	>	a	b	a	_A	#

>	a	b	a	_A	#	>	a	b	aA	#	>	a	bA	#
>	a	b	a	A	#	>	a	b	A	#	>	a	A	#

>	aA	#	>A	#	A##
>	A	#	A	#	#

Tricky part of proof is showing that this is the only way to solve PCP with these dominoes