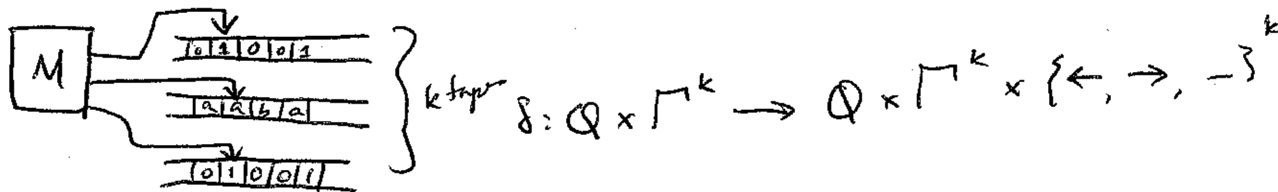
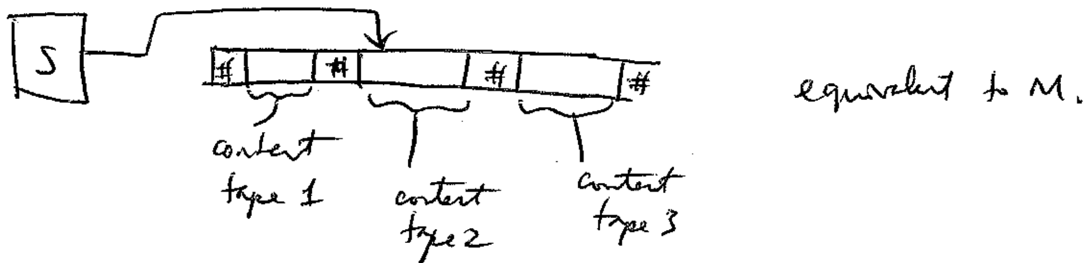
Variants of TM

Multitape TM



$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\leftarrow, \rightarrow, -\}^k$

This is equivalent to a single-tape machine



equivalent to M.

Nondeterministic TM

$\delta : Q \to \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\leftarrow, \rightarrow, -\})$ ← non-determinism

Is a Non-deterministic TM more powerful than a DTM? No!

- Show this by simulating a NTM with a DTM.

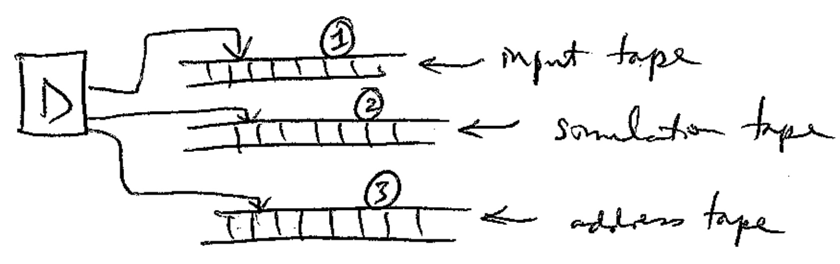
The NTM computation is a tree, where each node is a configuration of the NTM.

← every node in the tree can have at most b children, where b is the size of the largest possible choices given by the transition function.

Our DTM will traverse this tree searching for an accepting configuration.

We will do a breadth first search! Depth first search might not finish!

To do this, design a DTM with 3 tapes:



Input tape - input to our NTM, never altered.

Simulation tape - Copy of NTM's tape on the current configuration of the NTM

address tape - keeps track of location in NTM's computation tree.

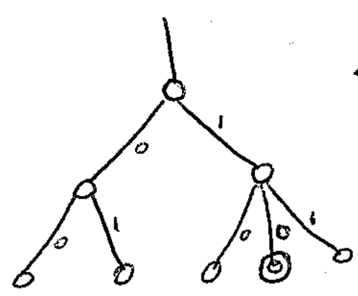
↑ Assign an address string over alphabet $\Sigma_b = \{1, 2, \ldots b\}$ to each node. 231 is root → 2nd child → 3rd child → 1st child

D works as follows.

1. Start with input w on tape 1, tapes 2, 3 blank.
2. Copy tape 1 to tape 2.
3. Use tape 2 to simulate NTM with input w on one branch of computation. Look at symbols on tape 3 to determine if choice is allowed by NTM's transition function. If not, then go to 4. If reject, go to 4. If accepting config., accept.
4. Replace string with next string lexicographically. Go to 2.

Ex:

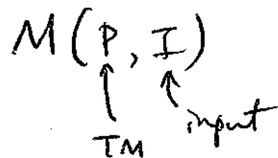
$w = 10$
$\Sigma_b = \{1, 2, 3\}$



ε
~~1~~
2
~~3~~
~~11~~
~~12~~
~~13~~
21
[22]
23

Universal Turing Machine

a TM that can simulate any other TM.



M basically simulates P on input I

Halting problem

Halt: M(P,I): returns yes if P will halt given input I; no if it won't } the Halt program should always halt.

Turns out the Halting problem is <u>undecidable</u>!

Before we show this, let's daydream. What if it were decidable?
⇒ What if we could tell if a program P would halt on input I?

- We could answer many mathematical mysteries!

Eg. <u>Goldbach conjecture</u>: Every even number is the sum of two primes.

```
void Goldbach(void) {
    for (i=0; ; i+=2) {
      if i not sum of primes return;
    }
}
```

} no need to run this program. Just give it as input to halt and halt will tell us whether it stops or not!

- Automated software verification.

Proof Halting Problem is undecidable

Suppose M exists.

M'(P) : if M(P,P) = yes, then ↗
if M(P,P) = no, then halt.

Now ask: $M'(M') \Rightarrow M(M', M')$

if this halts then M = yes, then $M'(M') = \nearrow$

if this doesn't halt, M = no, then $M'(M')$ = halts!

<u>Contradiction!</u>

Hence neither M' nor M can exist.

But the halting problem is Turing-recognizable (accepted by some turing machine).

|| recursively enumerable

Theorem

A language L is decidable iff it is Turing recognizable and co-Turing recognizable

co-Turing recognizable means that it's complement is Turing recognizable.

<u>Proof</u>:

L decidable $\rightarrow$ Turing recognizable (← recursive) & co-Turing recognizable (← recursively enumerable)

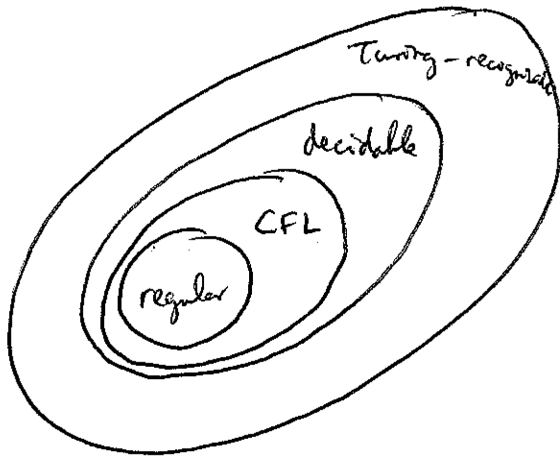
Any decidable language is Turing recognizable.

If it is decidable, then by switching the accept, reject we show that it is co-Turing recognizable.

M_1 = Turing recognizable
&
M_2 = Co-Turing recognizable
$\rightarrow$ decidable

1. Run M_1 & M_2 in parallel
2. if M_1 accepts, accept.
   if M_2 accepts, reject.

So the complement of the halting problem is not Turing recognizable. if it were, then halting would be <u>decidable</u>!



decidable = recursive

Turing recognizable = recursively enumerable
= partial recursive

Diagonalization Georg Cantor

Ex:

	0	1	2	3	4	...
∅	0	0	0	0	0	...
even	1	0	1	0	1	...
odd	0	1	0	1	0	...
prime	0	1	1	1	0	...
all	1	1	1	1	1	...

diagonal

Can we enumerate the set of all sets?

No!

← flip the ones and zeros along the diagonal. Guaranteed not to exist anywhere because it will differ by at least one element.

Halting problem

inputs

program						
	1	1	1	1	1	1
	0	1	0	1	0	1
	1	1	1	0	1	1
	0	0	0	1	0	1
	...				0	
→	0	0	0	0	1	

0 → halt
1 → halt

← result of feeding a program to itself.

Program that does the opposite not on the list.