② output is word w' in $R1$ when STOP is reached.

Ex. Consider the following RM on the alphabet $A = \{1, +\}$ ($1 = \sigma_1$, $+ = \sigma_2$)

1. IF FIRST, R1, GOTO 5
2. DEL R1
3. IF FIRST, R1, GOTO 8
4. GOTO 11
5. DEL R1
6. ADD, R2
7. GOTO,
8. DEL R1
9. ADD, R2
10. GOTO 3
11. R1 ← R2
12. STOP

- on an input of the form

$\underbrace{111\ldots1}_{n\ 1's} + \underbrace{11\ldots1}_{m\ 1's}$

the program:

↳ consecutively deletes the leading 1's (before the +) in R1 and writes them in R2 (in instruction loop 1 → 5 → 6 → 7)

- deletes the + once it reaches it (instructions 2 → 3)
- consecutively deletes remaining 1's in R1 and writes them (at end of word) in R2
  (instruction loop 3 → 8 → 9 → 10)
- once R1 is empty transfers word in R2 to R1 and halts
  (4 → 11 → 12)

↳ output is $\underbrace{\underbrace{111\cdots1}_{n}\underbrace{11\cdots1}_{m}}_{n+m}$

↳ i.e. the program performs "unary addition"!

Equivalence of RMs and TMs

- though RMs and TMs are different kinds of machines, can show they are equivalent as models of computation
  ... in the sense that they compute the same (partial) functions

- more precisely: spr A, B are finite alphabets

a partial function $f: A^* \to B^*$ is a function whose domain is a subset of A^* and has outputs in B^*.

Def'n a partial function $f: A^* \to B^*$ is called TM-computable if there is a finite alphabet $C \supseteq A \cup B$ and a turing machine M on C s.t. for each $w \in A^*$, M terminates on w iff w is in domain(f) (i.e. $f(w)$ is defined) and in this case the output of M is $f(w)$.

↳ TM-computable functions are just the functions you get from TMs (i.e. by viewing a TM as an input/output device -- i.e. a function)

↳ given a TM M, associated TM-computable function f_M is the one defined by:

$f_M(w)$ = output of M on input w (if M halts on input w o/w undefined)

↳ in this sense TM-computable functions are "1-1" with TMs M.

(but not literally 1-1: could be that same abstract function is computed by two different TMs)

↳ can define what it means for a function to be "RM-computable" in an analogous way.

↳ we then have the following:

Theorem (equivalence of TMs/RMs)
a function f is TM-computable iff it is RM-computable

- theorem says: for every TM M there is an RM N w/ same outputs as M

  i.e. output of M on input w
  = output of N on input w
  for all words w
  (... and the machines don't halt on the same inputs)

and vice versa

↳ won't prove theorem, but the idea of the proof is simple

- Given a TM M, can encode the instructions in the table of M as batches of instructions in some RM N, and then

simulate M by N
- conversely, given an RM N, can find a TM M whose states correspond to places in N's program
- such M simulates N (i.e. computes the same outputs)

Think: this is just like using a "virtual machine" on a Mac to run Windows or Linux software.

Church-Turing thesis

- Beyond RMs and TMs, many other "models of computation" have been studied over the years including: recursive functions, finite automata, ...actual computers!

- for all such models can show: if f is a function computed by a model of computation M (e.g. an actual computer) then there is a TM M that computes f (by simulating the M program N that computes f)

- and vice versa

- so, empirically: all (sufficiently powerful) models of computation are equivalent!
  ↳ this is known as the Church-Turing thesis

- it's a "thesis" since it's not a formal mathematical statement that can be proved
  ... no formal definition of "model of computation" is known!

Universal Turing Machines

- intuitively a TM is more akin to an individual program than a computer.

- however: it's possible to construct a universal TM U
  ↳ such a TM resembles a computer (i.e. a machine that can run any program)
  ↳ U takes as inputs strings of the form:
  (code for another TM M) followed by (input w for M)

i.e. return the same output that M returns on input w.

- Let's sketch how this coding is achieved
- First: given an alphabet $A = \{\sigma_1, \ldots, \sigma_k\}$ we can code this alphabet by strings in the alphabet $\{0,1\}$
  e.g. if $k \leq 2^n$ we can code all symbols $\sigma_1, \ldots, \sigma_k$ by binary sequences of length $\leq n$, (since 2^n of these)
  $\sigma_1 \to 0$ $\sigma_2 \to 1$ $\sigma_3 \to 00$ $\sigma_4 \to 01$, etc.
- then for each word $w \in A^*$ have a binary code $b(w) \in \{0,1\}^*$ for this word
- can check: for each M on the alphabet A can construct a TM M^* on $\{0,1\}$ s.t. for all inputs $w \in A^*$
  if M outputs v on input w
  M^* outputs $b(v)$ on $b(w)$
- i.e (encoded) input/output behaviour of M^* is same as M.

- thus any TM is simulated by one on the binary alphabet $\{0,1\}$
- can restrict our attn to machines on $\{0,1\}$

- going up a level: we can now encode TMs themselves
- that is: given a machine M, which is just a table of read/write instructions, can encode these instructions as a single string (putting them back to back, maybe w/ spaces)
- can then encode this string in a binary sequence.
- we call this sequence the binary code of M and denote it b(M)

- now we can, with a lot of effort, write a program
  (i.e. a Turing machine U on the alphabet $\{0,1\}$)
that simulates every TM M on $\{0,1\}$.

- More specifically:

Theorem There is a Turing machine U on the alphabet $\{0,1\}$ s.t.
For every TM M on $\{0,1\}$
and every input $w \in \{0,1\}^*$
if M returns w' on this input

then:

on input $b(M) * w$ (* - blank)
U returns w/ dec.

↳ we call U a universal Turing machine: