Random-Access Machine

- formal model, but much closer to real hardware than the TM
- in fact, it's a family of related models, we will show the simplest of them
- RAM works with numbers (our version: the whole of $\mathbb{Z}$)
- memory: seq. of numbers, indexed by numbers (negative indices allowed)
- addressing of operands:
  - literal constant (embedded in an instruction)
  - [n] - directly addressed memory cell
  - [[n]] - indirectly — " — (read [n] to obtain another cell address)
- instructions:
  1. movement of data: $X \leftarrow Y$ — Y = any, X = any except literal
  2. arithmetics: $X \leftarrow Y \oplus Z$ — $\oplus$: +, −, *, % ; bitwise &, or, xor ; bitwise shift
  3. control:
     - halt
     - jump PLACE
     - if $X < Y$ jump PLACE — $<, >, =, \neq, \leq, \geq$
- input is stored at agreed-upon locations in memory when the program starts
- output is found — " — when the program stops

Example: sum of N numbers

In: [0] = N, [1] = x_1, ..., [N] = x_N

Out: [0] = sum

Temporary: [-1] = copy of N, [-2] = current index

```
Program:  [-1] ← [0]          copy N
          [0] ← 0             initialize sum
          [-2] ← 1            start with x1
LOOP:     if [-2] > [-1] jump END
          [0] ← [0] + [[-2]]
          [-2] ← [-2] + 1
          jump LOOP
END:      halt
```

Complexity: time = # executed instructions

space = max (cell address used) − min (...)

this varies between RAM versions, e.g. we could define cost of an instruction as

$\max \log(1+x)$

$x \in \{$operands, addresses, result$\}$

or keep cost constant, but restrict size of cells somehow...

"TM is equivalent to RAM"

- what can this mean?
  - ... they can simulate each other: for each RAM program there is an equivalent TM & vice versa
  - ... but RAM crunches numbers, while TM crunches strings

We will assume that the RAM gets a string $\in \Sigma^*$ as input:
[0] = length, [1], [2], ... = symbols of the string.
(this is WLOG since both TM and RAM can convert between all reasonable input formats)

$\boxed{\text{TM to RAM}}$ – WLOG 1-tape TM with 1-way-infinite tape

- store the contents of the written-to part of the tape in [1], [2], ... using some numbering of the work alphabet
- [0] will specify how far the — " — stretches.
- [-1] = current position of head
- position in program represents machine state
- can simulate 1 step of the TM in constant time.

$\boxed{\text{RAM to TM}}$

- representation of numbers: binary + sign symbol
- TM subroutines for arithmetics (inputs/output on special tapes)
- tape M: memory of the RAM   # cell -1 # cell 0 # cell 1 # ...
- tape A: address of memory cell in which the head on tape M is
  ... can more 1 cell left/right, possibly extending M by empty cells at both ends
- memory read: given address on tape R, copy number read to tape D (data)
  ... compare R with A, move across cells until ~~data~~ R=A, copy data from M to D
- memory write: similar, but need to expand cells if they are to small for new data
- every instruction can be composed of read/write/arithmetics
- keep position in RAM program inside state of the TM

- simulation works, but with significant slowdown (inevitable?)

Computability

We will study it only for languages (decision problems), generalization to functions is straight-forward.

Df: • Turing machine M <u>accepts</u> word $\alpha \in \Sigma^*$ ≡ computation on α ends in state q^+

rejects α ⟺ stops in q^- or runs forever (diverges)

• Language $L(M)$ <u>accepted by</u> M := $\{\alpha \in \Sigma^* \mid M \text{ accepts } \alpha\}$

• Language L is <u>decided</u> by M ≡ M always stops & $L = L(M)$.

Df: • Language L is <u>computable</u> (a.k.a. <u>decidable</u> / <u>recursive</u>) ≡ $\exists$TM M: L is decided by M.

refers to Church's formalism of recursive functions (equivalent to TM)

• Language L is <u>partially computable</u> (a.k.a. <u>partially decidable</u> / <u>recursively enumerable</u>) ≡ $\exists$TM M: L is accepted by M (i.e., $L(M) = L$).

Df: $R := \{L \mid L \text{ is computable}\}$

$RE := \{L \mid L \text{ is partially computable}\}$

since elements of Σ_1 can be arbitrary, these are proper classes. WLOG we can fix $\Sigma_1 = \{0,1\}$ to make R and RE sets.

$R \subseteq RE \subseteq 2^{\{0,1\}^*}$ ← all languages over $\{0,1\}$

are these strict? Watch out...

Enumeration (or: why "recursively enumerable"?)

Df: <u>Enumerator</u> ≡ TM with no input, potentially running forever, printing strings (formally: printer is an oracle) → language enumerated by M

L is <u>enumerable</u> ≡ $\exists$ enumerator which prints exactly the words of L

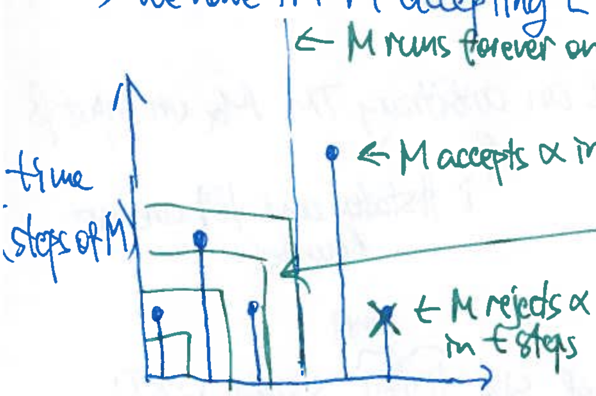
Thm: $L \in RE$ ⟺ L is enumerable

Pf: ⇐ we want to accept $\alpha \in L$ ... run enumerator, compare printed strings with α

YES ⇒ stop in q^+, NO ⇒ continue

enumerator stops ⇒ stop in q^-

⇒ we have TM M accepting L, let's build enumerator for L using M:



expand square ... for each (α, t) simulate M on α for t steps ... if it stops in q^+, in exactly t steps, print α

prints all $\alpha \in L(M)$

Strings in <u>length-lexicographic</u> order ($\alpha \le_{LL} \beta \equiv |\alpha| < |\beta| \lor |\alpha| = |\beta| \;\&\; \alpha \le_{Lex} \beta$)

[binary numbers with leading 1 removed]

<u>Homework</u>: $L \in R$ ⟺ L is enumerable in $\le_{Lex}$ order.

Universal TM (why we don't need TM program in modifyable memory)

Df: Encoding of TMs (a.k.a. Gödel numbering) ← but in our case, the codes are actually strings, not numbers

we define it for 1-tape machines with $\Sigma_1 = \{0,1\}$

alphabet: $\Gamma = \{x_0, x_1, x_2, \dots, x_m\}$

x_0, x_1, x_2 ↑ 0, 1, 2 — other symbols in arbitrary order

directions: $\{d_0, d_1, d_2\}$ ← → •

states: $Q = \{q_0, q_1, q_2, \dots, q_n\}$

q_0, q_1, q_2 ↑ initial, q^+, q^- — other states

→ start code with $1^m 0 1^n 0$ to preserve $|\Gamma|$ and $|Q|$ even if symbols/states unused

transitions: $\delta(q_i, x_j) = (q_k, x_l, d_t) \rightarrow$ encode as $1^{i+1} 0 1^{j+1} 0 1^{k+1} 0 1^{l+1} 0 1^{t+1} 0$

↳ concatenate codes of all transitions → code of machine $\langle M \rangle$

Df: • $M_\alpha :=$ machine with code α (if α not a valid code ⇒ machine which immediately halts in q^-)

👁 $\forall$ TM M $\exists \alpha : M \cong M_\alpha$

↑ isomorphism of TMs (defined in the obvious way)

↳ in fact, there are multiple such codes (we numbered Q, Γ arbitrarily &c.)

• $L_\alpha := L(M_\alpha)$

👁 $\forall L \in RE$ $\exists \alpha : L = L_\alpha$

\# codes is countable ⇒ RE is countable ••• but $2^{\{0,1\}^*}$ uncountable

⇒ $\exists L \notin RE$ (non-constructively)

Tool: Encoding of pairs $\langle \alpha, \beta \rangle$: $\langle x_1 \dots x_n, y_1 \dots y_m \rangle := x_1 0 x_2 0 \dots x_n 0 1 1 y_1 0 \dots y_m 0$

👁 encoding & decoding is computable (& well-defined)

Df: Universal language $L_u := \{ \langle \alpha, \beta \rangle \mid \alpha, \beta \in \{0,1\}^* \ \&\ \beta \in L_\alpha \}$

"contains all partially computable languages" (in a sense)

Lemma: $L_u \in RE$

Pf: Construct the universal TM, which can simulate an arbitrary TM M_α on input β

↑ WLOG multi-tape

↑ # states and $|\Gamma|$ are not bounded

tape K: copy of the code α

tape T: tape of the simulated machine: blocks of size $|\Gamma|+1$, symbol $x_i \in \Gamma$ stored as $1^i 0^{m-i}$

tape M: 1^m

↑ head on T encodes position of M_α's head

13

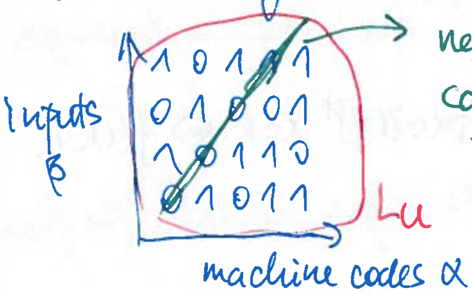
tape S: current state of M_α stored as $1^j 0^{|S|-j}$

Init: Split $\langle \alpha, \beta \rangle$, copy α to tape K, encode β on tape T, initialize tape S
↳ & set tape M

Step: Read current symbol ~~s~~ from T, find entry for state s and symbol x on K, write new symbol & state, move head on T.

Lemma: $\overline{L_u} \notin RE$

Proof: Use diagonalization



diagonal language

$L_d := \{ \alpha \in \{0,1\}^* \mid \alpha \notin L_\alpha \}$

$L_d \notin RE$ ... assume $L_d \in RE$

Then $\exists \alpha$: $L_d = L_\alpha$

but: $\alpha \in L_d \Leftrightarrow \alpha \notin L_\alpha \Leftrightarrow \alpha \notin L_d$ ↯

If $\overline{L_u}$ were partially decidable, we could modify the machine accepting $\overline{L_u}$ to a machine accepting L_d ↯

Corollaries:
- $L_u \notin R$ (R is closed under complement, so $L_u \in R$ would imply $\overline{L_u} \in R \subseteq RE$)
- $R \subsetneq RE \subsetneq 2^{\{0,1\}^*}$
  ↑ witnessed by L_u ↑ witnessed by $\overline{L_u}$
- RE is not closed under complement

Exercise: Are R and RE closed under $\cap$ or $\cup$?

Thm (Post's): $L \in R \Leftrightarrow L \in RE$ & $\overline{L} \in RE$.

Pf: $\Rightarrow$ trivial, because $R \subseteq RE$ & R closed under complement.
$\Leftarrow$ "run machines accepting L and $\overline{L}$ in parallel" (one step of each at a time) One of them certainly stops.

Operations on machine codes
- swap q^+ with q^-: given M_α ~~accepting~~ deciding L, find M_β deciding $\overline{L}$
- compose two machines: find M_γ, which runs first M_α and then M_β on its output
- substitute M_α for an oracle in M_β

all these are computable functions