TM: exercise

Provide a TM, M, that recognizes all strings of 0s whose length is a power of 2, i.e., it decides the language $L = \{0^{2^n} : n \geq 0\}$. Below is an implementation-level description for the TM:

M="On input string w:

1. Sweep left to right across the tape, crossing off every other 0.
2. If in stage 1 the tape contained a single 0, accept.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s is odd, reject.
4. Return the head to the left-hand of the tape.
5. Go to stage 1."

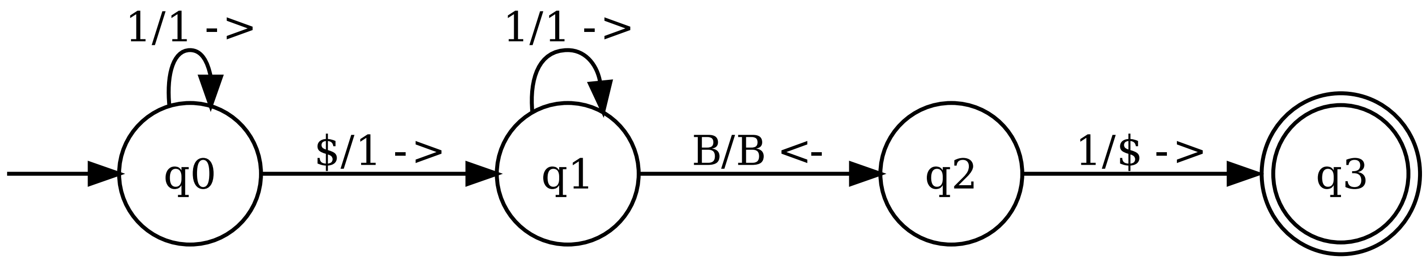
TM Example: Adder

Given two positive integers $x > 0$ and $y > 0$, design a TM that computes $x + y$.
Assumptions:

- $\widehat{x}$ denotes the unary notation for x, i.e., if x is 3, then $\widehat{x}$ is 111.
- $\widehat{x}$ and $\widehat{y}$ are placed on the tape separated by a single $\$$.
- After the computation, $\widehat{x+y}$ will be on the tape followed by a single $\$$.
- The expression below represents an accepting computation of $x + y$.

$$q_0\widehat{x}\$\widehat{y} \overset{*}{\vdash} \widehat{x+y}\$q_f B$$

TM Example: an Adder (cont.)

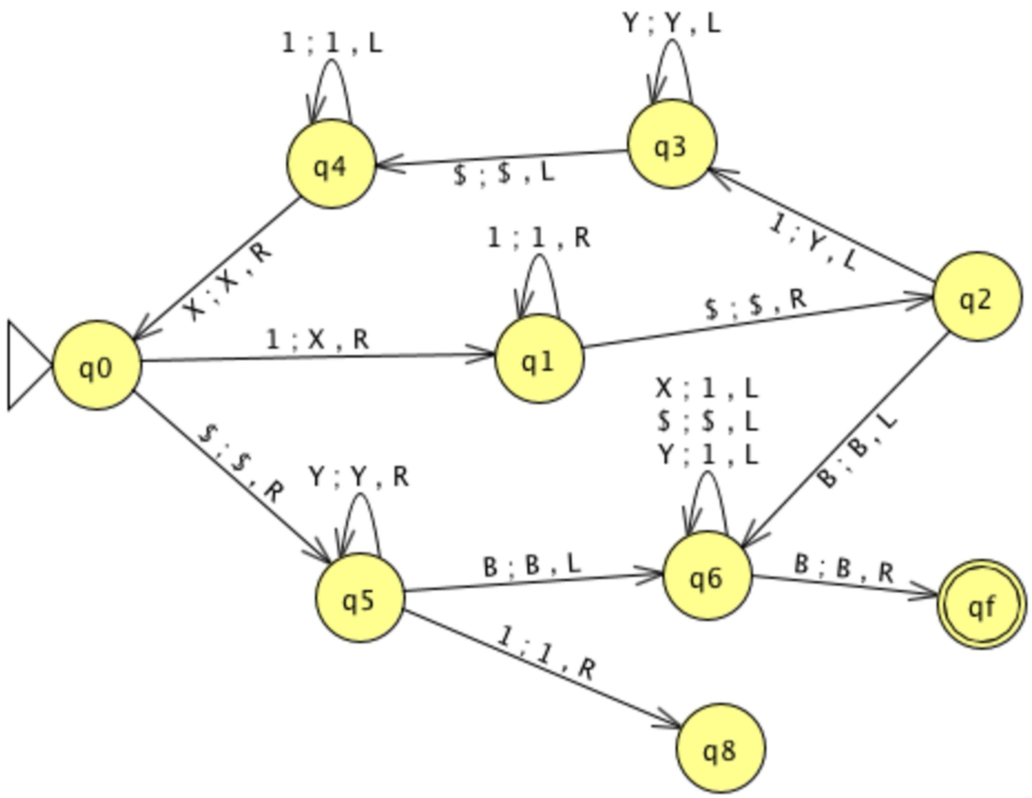


Below is the computation of $1\$11$:

$$q_0 1\$11B \vdash 1q_0\$11B \vdash 11q_1 11B \vdash 111q_1 1B \vdash 1111q_1 B \vdash$$
$$111q_2 1B \vdash 111\$q_3 B$$

TM Example: a Comparer

A TM that, given two positive integers x and y, input in the format $\widehat{x}\$\widehat{y}$ as in the previous TM, halts on a final state if $x \geq y$, or halts on a non-final state if $x < y$. In both cases (and for this example specifically) the TM should eventually position the head on the first symbol of the input string.

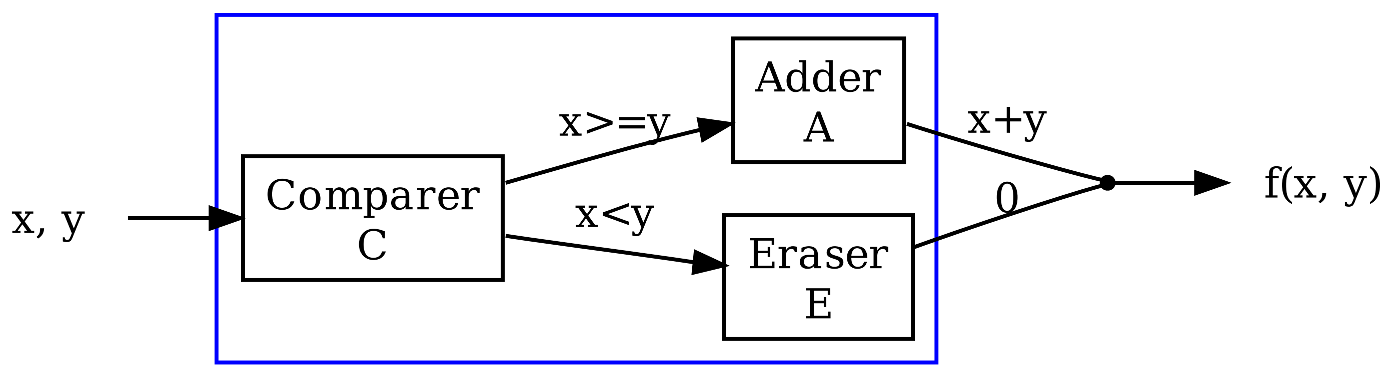


TM Example: an elaborate function

Combining TMs to perform elaborate tasks. E.g., computing the mathematical function

$$f(x,y) = \begin{cases} x + y & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases}$$

We use our previous TMs as subroutines.



TM tricks: alternative models for TMs

- A TM with the ability to stay put, i.e., keep its head on the same cell in a move. E.g:
  - $\delta(q, a) = (p, a, StayPut)$, i.e., $\alpha q a \gamma \vdash \alpha p a \gamma$
- This feature would not render the TM more powerful, as we could represent a “stay put” transition with two transitions (left, right). E.g.
  - $\delta(q, a) = (r, a, R)$ and $\delta(r, x) = (p, x, L), \forall x \in \Sigma$
  - Computation steps: $\alpha q a \gamma \vdash \alpha a r \gamma \vdash \alpha p a \gamma$
- This small example holds the key to the analysis we will do next.
  - We will introduce a repertoire of TM variants.
  - We will show their equivalence by simulating one by the other.

TM tricks: alternative models for TMs

- TM *programming techniques*: storage in the state, multiple tape tracks, subroutines, ...
- *Extensions*: multiple tapes, non-determinism, ...
- *Restrictions*: semi-infinite tape, multiple stacks, counters,...

All these tricks, however, produce models that are equivalent: they accept the recursively enumerable languages (Church- Turing thesis, 1936).

TM Multiple tracks (TM-mt): example

- Suppose symbols of Γ are pairs $[A, X]$, where X is the "real" symbol, and A is either B (blank) or $*$.
  - Input symbol a is identified with $[B, a]$.
  - The blank is $[B, B]$.
- Here's a program to find the $*$, assuming it is somewhere to the left of the present position.

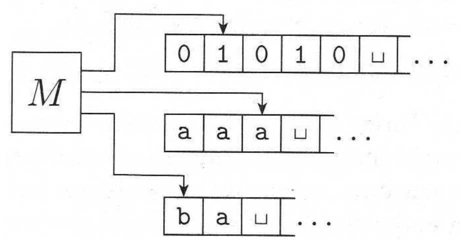
$$\delta(q, [B, X]) = (q, [B, X], L)$$
$$\delta(q, [*, X]) = (p, [B, X], R)$$

Notice that TM-mt $\equiv$ classical TM; we can encode each pair of symbols $[A, X]$ of MT-mt as an element of Γ of the classical TM.

Restricted TMs (rTMs)

- Now lets replace the tape of a TM with a semi-infinite tape (infinite only to the right). There are no cells to the left of the initial head position.
- Although apparently simpler, rTMs are as powerful as a classical TMs.
- The proof is based on the fact that we can simulate a TM using an rTM with two tracks.
  - The upper track represents the cells of the original TM that are at or to the right of the head.
  - The lower Track represents the cells to the left of the head, in reverse order.

Multitape TMs (mtTMs)



- A mtTM has some finite number of tapes k, with a head for each tape.
- The input symbols are placed in the first tape. All other cells of the other tapes are initially empty.
- A move of a mtTM depends on the state and the symbol scanned by each head. On each tape a new symbol is written, and each head makes a L,R move.

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

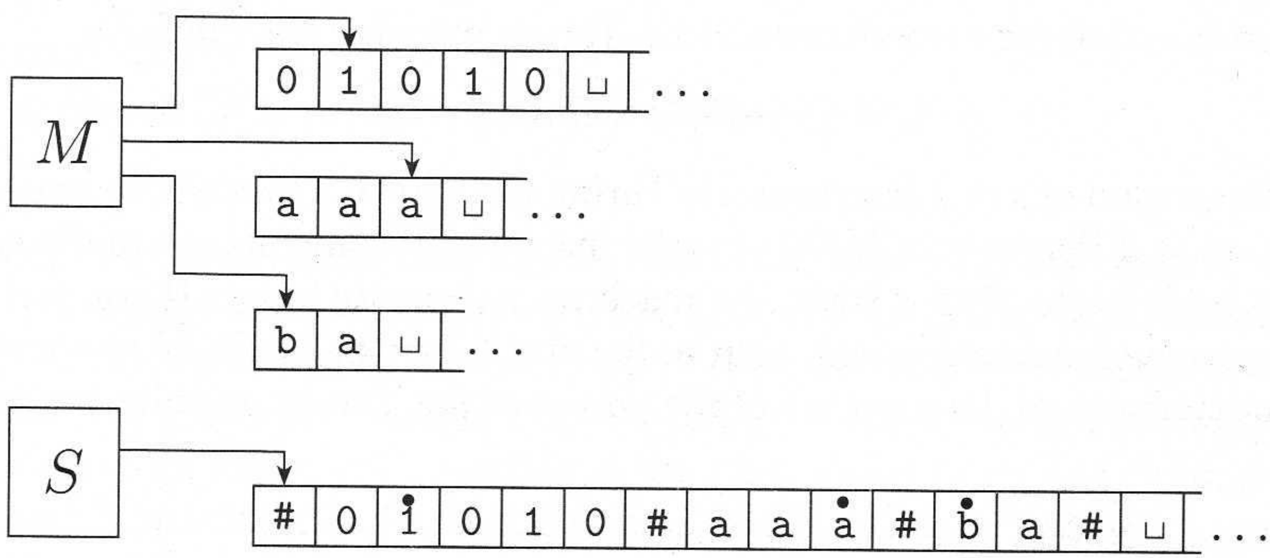
E.g.: $\delta(q_i, a_1, \cdots, a_k) = (q_j, b_1, \cdots, b_k, L, R, \cdots, L)$

Theorem: Every mtTM has an equiv. TM

Proof: Let M be a multi-tape TM. The TM S simulates M as follows:

- S uses a new symbol # as a delimiter to separate the contents of each tape.
- S uses a tape symbol (the one with a dot above) to mark the place where the head on that tape would be.

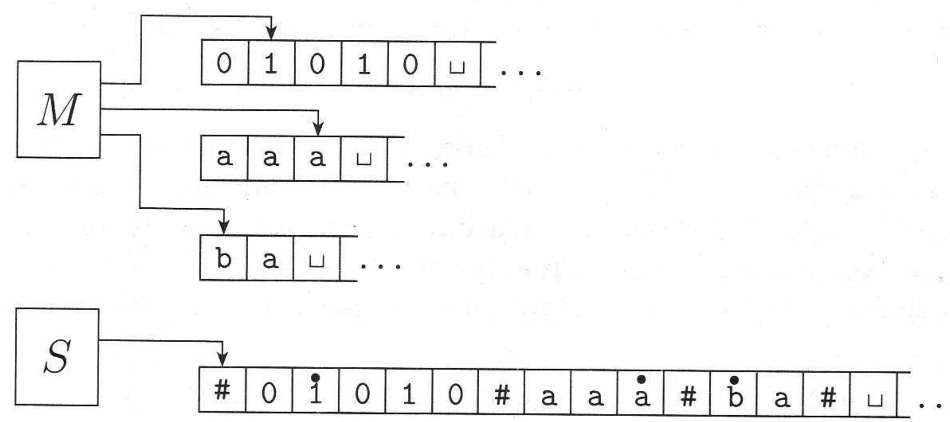
Think of these as "virtual" tapes and heads.



Simulation of a mtTM by a TM

$S =$ "On input $w = a_1 a_2 \cdots a_n$:

- First S puts its tape into the format that represents all the k tapes of a mtTM M: $\#\dot{a}_1 a_2 \cdots a_n \# \dot{B} \# \dot{B} \# \cdots \#$
- To simulate a single move, S scans its tape from the first # to the last #, which marks the right-hand end, to determine the symbols under the virtual heads.

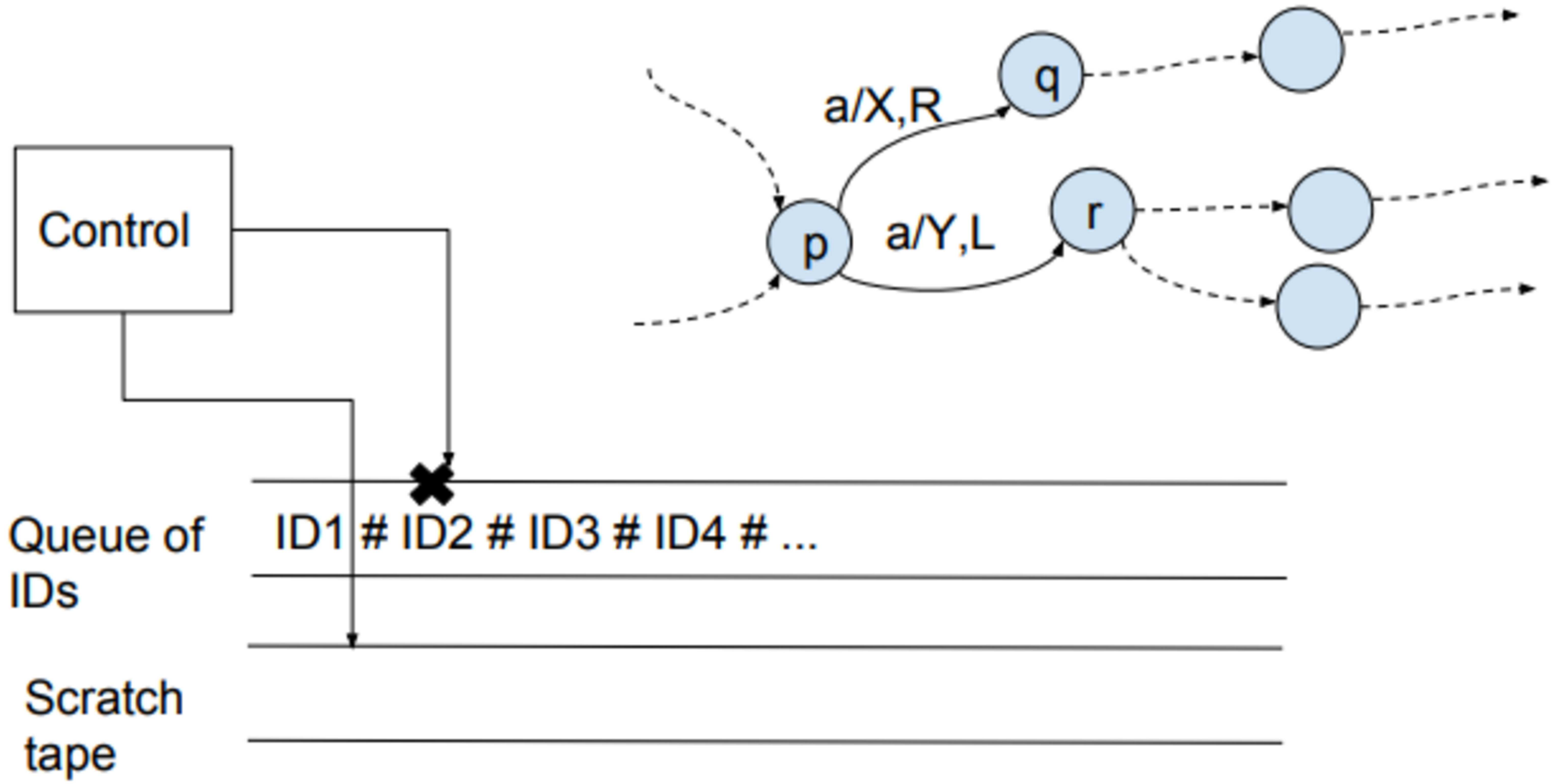


- Then S makes a second pass to update the tapes according to the way that M's transition function dictates.

Nondeterministic TMs (nTMs)

- As you may expect, at any point in a computation a nTM may proceed according to several possibilities.
- E.g., a transition in a nTM:
  $\delta(q, X) = \{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \cdots, (q_k, Y_k, D_k)\}$
- We can simulate a nTM with a deterministic TM, D, as follows:
  - Have D simulate all possible branches of N's nondeterministic computation (putting IDs on a queue, rather than a stack) .
    - View N's computation of w as a tree; each node of the tree is an ID; D traverses the tree in a breadth first search.
  - If D ever finds an accept state, it accepts; otherwise D's simulation will not terminate.

Nondeterministic TMs (nTMs)



Opening a Parenthesis

Is a push-down automaton with two stacks equivalent to a turing machine?
(proof idea on the doc camera/board)

Closing the parenthesis

TMs and Real Computers

Now let's compare TMs and the common sort of computer we use daily.

- Although these models appear different, both accept the same languages–the recursively enumerable languages.
- Next we shall informally prove that $L(TMs) = L(ModernComputers)$, and this claim will be divided in two parts:
  - A computer can simulate a Turing Machine.
  - A Turing machine can simulate a modern computer.

Simulating a TM by a computer

We could write a program that acts like a TM.

- **The input symbols**: could be implemented as character strings.
- **The finite control**: we could use a table and a couple of helper functions to implement the finite number of states and transitions rules.
- **The infinite tape....**: we can postulate an "infinite" cloud-based file storage space.
  - The program that simulates the finite control also mounts the cloud drive that holds the region of the TM tape around the tape head.
  - There are two file folders, one storing the data to the left of this region, and one storing the data to the right of it.

Simulating a computer by a TM

Simulation is at the level of *stored instructions* and *words of memory*.

- The TM has an assortment of tapes for: memory locations, and their contents; the instruction counter; memory address; computer input file; and scratch.
- Instruction cycle simulation:
  1. Find the word indicated by the instruction counter on the memory tape.
  2. Examine the instruction code, and get the contents of any memory words mentioned in the instruction, using the scratch tape.
  3. Perform the instruction, changing any words' values as needed, and adding new address-value pairs to the memory tape, if needed.

Summary

- We have seen some variants of the TM model and have shown that they are equivalent in power.
- There are many other models, all share the essential feature of TMs, viz. unrestricted access to unlimited memory, distinguishing them from FA and PDA.
- Remarkably, all models turned out to be equivalent in power.

Summary (cont.)

- Since we can simulate a deterministic TM in Java and also in Fortran, the two languages can represent exactly the same class of algorithms.
  - So do all other (Turing complete) programming languages.
- This equivalence phenomenon has an important philosophical corollary:
  - Even though there are many different computational models, the class of algorithms that they describe is unique.

Our progress so far

In our exploration of theories of computation we studied descriptors & recognizers for different classes of languages:

- RE & FA: regular languages; simple machines with single-state-based memory
- CFG & PDA: context-free languages; more advanced machines with a stack-based memory
- ? & TM: recursively enumerable languages; even more advanced machines, with a tape-based unlimited memory with unrestricted access.
  - We learned that TMs can do exactly what computers can do.
- But what are the descriptors for the languages TMs recognize?

The definition of *Algorithm*

- Informally, an *algorithm* is a collection of simple instructions for carrying out some task.
- Ancient mathematical literature contains descriptions of algorithms for a variety of tasks, e.g., greatest common divisors, aka Euclidian Algorithm (circa 300BC).
- The notion of algorithm was precisely defined only in the 1936 papers of Alonzo Church and Alan Turing.
- The Church-Turing thesis:

$$\left[\begin{array}{c}\text{Intuitive notion of}\\ \text{algorithms}\end{array}\right] \equiv \left[\begin{array}{c}\text{Turing Machine}\\ \text{algorithms}\end{array}\right]$$

- Our study now has reached a turning point:
  - We continue to speak of TM.
  - But our real focus is on algorithms.

How we'll carry out this exploration

- From now on, TMs will be used as a model for the definition of algorithms.
- We'll skip over loads of TM theory.
- Instead, we'll focus on "high-level programming" of TMs and the problems they cannot solve.

On describing TM algorithms

- What is the right level of detail to give when specifying a TM?
- There are three possibilities:
  - Formal (low-level) description: we specify $(Q, \Sigma, \Gamma, \delta, q_0, F)$; or provide a complete state diagram.
  - Implementation description: we use English prose to describe the way the TM moves its head and stores data on its tape. We don't provide details about states or transition function.
  - High-level description: we use structured English prose to describe an algorithm, ignoring the TM implementation model entirely.

Example

An implementation-level description for a TM that decides the language:

$$L = \{w : 0^n 1^n, n \geq 1\}$$

$M =$ "On input w

1. Cross off a 0 with an X and scan to the right until a 1 occurs, then cross it off with a Y. Shuttle between the 0s and 1s crossing off one of each until reading a Y.
2. Move right until reading either a blank or 1.
3. If symbol read is a blank, accept; otherwise reject."

Another example

A high-level description for a TM that decides the language:

$$L = \{w : w \text{ is not } 0^n 1^n\}$$

$M =$ "On input w

1. Run the TM of previous example on w.
2. If it rejects, accept; otherwise reject."

A new format and notation for TMs

- The input to a TM is always a string
- The encoding of an object O as a string representation will be denoted as $\langle O \rangle$
- If there are several objects $O_1, O_2, \cdots, O_k$, then we'll denote the encoded string as $\langle O_1, O_2, \cdots, O_k \rangle$.
- If $\langle A \rangle$ is the input, then we assume the TM first verifies whether the input properly encodes the object, rejecting it if it doesn't.

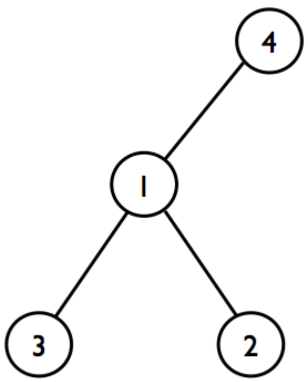
Let's see an example.

New notation for TMs: example

Let A be the language consisting of all strings representing undirected graphs that are connected (every node can be reached from every other node), formally:

$$A = \{\langle G\rangle : G \text{ is a connected undirected graph}\}$$

A possible encoding for a graph using s-expressions

$G =$ 

$\langle G\rangle =$ `"(4 3 2 1)((1 2) (3 1) (1 4))"`

Example (cnt.)

The following is a high-level description of a TM that decides $A = \{\langle G \rangle : G$ is a connected undirected graph$\}$
E.g.: $\langle G \rangle =$ `"(4 3 2 1)((1 2) (3 1) (1 4))"`
$M =$ "On input $\langle G \rangle$:

1. Select the first node of G and mark it.
2. Repeat the following step until no new nodes are marked.
   2.1 For each node in G, mark it if it is attached by an edge to a node that is already marked.
3. Scan all nodes of G to determine whether they all are marked. If they are, *accept*; otherwise *reject*."

Un)decidability: exploring the unsolvable

- The notion of algorithmic solvability is not foreign to us.
- Why study unsolvability?
  - if a problem is algorithmically unsolvable, then it has to be simplified or altered before a solution is attempted.
  - a glimpse into the unsolvable can stimulate our imagination and help us gain a better footing on computation.
- Initially we'll see some problems that are decidable by algorithms.
- Then we'll study a problem involving TMs that no TM can solve (i.e., there is no decidable algorithm for it).
- Later we'll learn how prove that a problem is unsolvable.

Decidable languages/problems

Let's start by analysing the acceptance problem for DFAs: whether a given DFA accepts a given string.

- This problem can be expressed as a language.

$$A_{DFA} = \{\langle D, w\rangle : D \text{ is a DFA that accepts } w\}$$

- Therefore, we'll formulate computational problems as tests of membership in a language.
  - the language is decidable if-and-only-if the computational problem is decidable

Theorem 1: A_{DFA} is a decidable language

Proof idea: we present a TM M that decides A_{DFA}:

$M =$ "On input $\langle D, w\rangle$, where D is a DFA, and w a string:

1. Simulate D on input w.
2. If the simulation ends in an accept state, *accept*; otherwise *reject*."

M's operational details:

- M checks if $\langle D, w\rangle$ properly encodes a DFA, and w properly encodes a string.
- M simulates D directly, e.g.:

$$\delta_M(p, a) = (q, a, \rightarrow) \text{ simulates } \delta_D(p, a) = q$$

Thrm 2: A_{NFA} is a decidable language

$$A_{NFA} = \{\langle D, w\rangle : D \text{ is an NFA that accepts } w\}$$

Proof idea: we present a TM N that decides A_{NFA}:

$N =$ "On input $\langle D, w\rangle$, where D is an NFA and w a string.

1. Convert D to a DFA C.
2. Run M (our TM from Thrm. 1) on $\langle C, w\rangle$.
3. If M accepts, *accept*; otherwise *reject*."

Thrm 3: A_{RE} is a decidable language

$A_{RE} = \{\langle R, w\rangle : R \text{ is a regular expression that generates } w\}$

Proof idea: similar to proof of Thrm 2.

Thrm 4: A_{CFG} is a decidable language

$A_{CFG} = \{\langle G, w\rangle : G$ is an CFG that generates the string $w\}$

Proof idea:

- We can't have the TM simulate derivations until it generates w; the TM may never halt! (not a decider).
- Converting G to a NPDA, then have a TM simulate it won't do, for similar reasons.
- The solution is to ensure the TM tries only finitely many derivations.
- In effect, if G is in CNF, any derivation of a string of length n will have $2n - 1$ steps.
- Therefore, checking only the derivations with at most $2n - 1$ steps would do.

Proof of Thrm 4 (cont.)

Below is the description for the TM S that decides A_{CFG}.

$S =$ "On input $\langle G, w \rangle$, where G is a CFG and w a string:

1. Convert G to CNF.
2. List all derivations with $2n - 1$ steps, where $n = |w|$.
3. If any of these derivations generate w, *accept*; otherwise *reject*."

The emptiness problem for CFGs

Given the language

$$E_{CFG} = \{\langle G\rangle : G \text{ is a CFG and } L(G) = \emptyset\ \}$$

Showing that E_{CFG} is a decidable language by determining whether the start symbol is 'generating'.
$R =$"On input $\langle G\rangle$, where G is a CFG.

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:
   2.1. Mark any variable A where G has a rule $A \to U_1U_2\cdots U_k$ and each symbol $U_1, U_2, \cdots, U_k$ has been marked
3. If the start symbol is not marked, accept; otherwise reject.

Thrm 5: CFLs are decidable by TMs

Proof idea: Let A be a context-free language (CFL). Our objective is to prove that A is decidable.

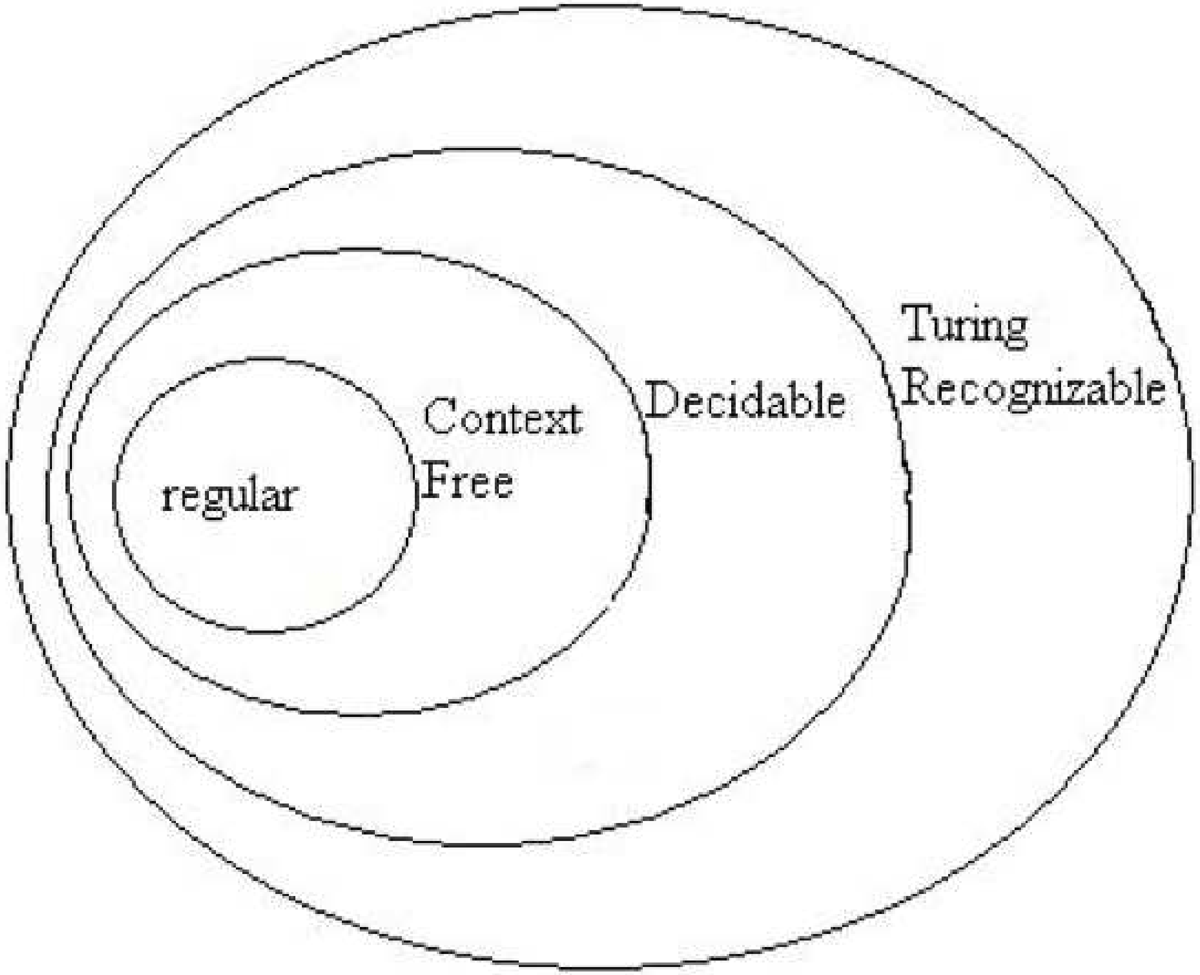
- a bad idea is to convert a PDA for A directly into a TM
- Since a PDA may be nondeterministic, branches of the execution may never end, thus the TM could not be a decider.

Let G be a grammar for A. We build G into the following TM that decides A.

$M_G =$ "On input $\langle w \rangle$:

1. Run TM S (from Thrm 4) on $\langle G, w \rangle$.
2. if S accepts, then *accept*; otherwise *reject*."

The hierarchy of languages



Beyond Turing recognizable languages

Our first goal in this exploration of undecidable problems is to prove that there is a specific problem that a TM cannot decide.

- What kind of unsolvable problems are these? Are they esoteric, dwelling only within the realm of Theoretical Comp Sci?
- In fact, they can be quite earthly, e.g.:

  Given a program P, write a program that decides whether P prints `''Hello World!''`

- We'll start by presenting our first theorem that establishes the undecidability of a specific language.