Complexity Theory

Part I

Outline for Today

- **Recap from Last Time**
  - Reviewing Verifiers
- **Nondeterministic Turing Machines**
  - What does nondeterminism mean in the context of TMs? And just how powerful are NTMs?
- **Complexity Theory**
  - A new framework for solvability.
- **The Complexity Class P**
  - What problems can be solved efficiently?

Recap from Last Time

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V is a decider (that is, V halts on all inputs.)
  - For any string $w \in \Sigma^*$, the following is true:

    $w \in L$ iff $\exists c \in \Sigma^*$. V accepts $\langle w, c \rangle$
- Some notes about V:
  - If V accepts $\langle w, c \rangle$, then we're guaranteed $w \in L$.
  - If V does not accept $\langle w, c \rangle$, then either
    - $w \in L$, but you gave the wrong c, or
    - $w \notin L$, so no possible c will work.

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V is a decider (that is, V halts on all inputs.)
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \text{iff} \quad \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

- Some notes about V:
  - If $w \in L$, a string c for which V accepts $\langle w, c \rangle$ is called a ***certificate*** for c.
  - V is required to halt, so given any potential certificate c for w, you can check whether the certificate is correct.

Verifiers and **RE**

- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof idea:*** Build a recognizer that tries every possible certificate to see if $w \in L$.
- ***Proof sketch:*** Show that this TM is a recognizer for L:

M = "On input w:
    For $i = 0$ to ∞
        For each string c of length i:
            Run V on $\langle w, c \rangle$.
            If V accepts $\langle w, c \rangle$, M accepts w."

Verifiers and **RE**

- ***Theorem:*** If $L \in$ **RE**, then there is a verifier for L.
- ***Proof sketch:*** Let L be an **RE** language and let M be a recognizer for it. Then show that this is a verifier for L:

V = "On input $\langle w, n \rangle$, where $n \in \mathbb{N}$:
Run M on w for n steps.
If M accepts w within n steps, accept.
If M did not accept w in n steps, reject."

Nondeterministic Turing Machines

Nondeterministic TMs

- A ***nondeterministic Turing machine*** (or ***NTM***) is a Turing machine in which there can be zero or multiple transitions defined at each state.
- Nondeterministic TMs do not have ε-transitions; they have to read or write something and move the tape at each step.
- As with NFAs, NTMs accept if any path accepts. In other words, an NTM for a language L is one where

  $w \in L$ iff there is some series of choices N can make that causes N to accept w.

- In particular, if $w \in L$, N only needs to accept w along one branch. The rest can loop infinitely or reject.

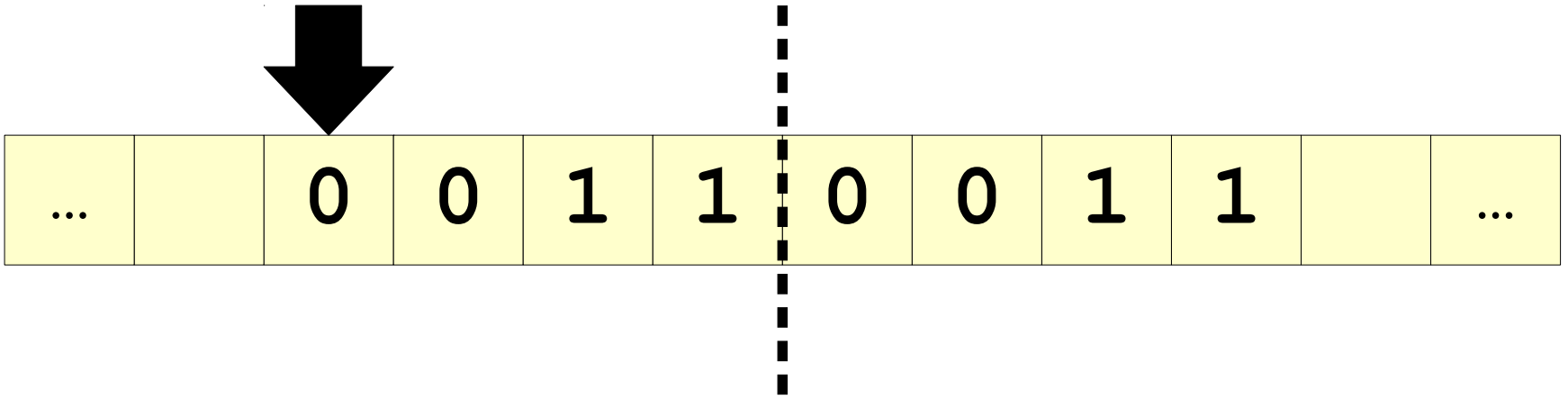
Designing an NTM

- A ***tautonym*** is a word that consists of the same string repeated twice.
- Some examples:
  - **dikdik** (an adorable petite antelope)
  - **hotshots** (people who aren't very fun to be around)
- Consider the following language ver $\Sigma = \{$ **0**, **1** $\}$:

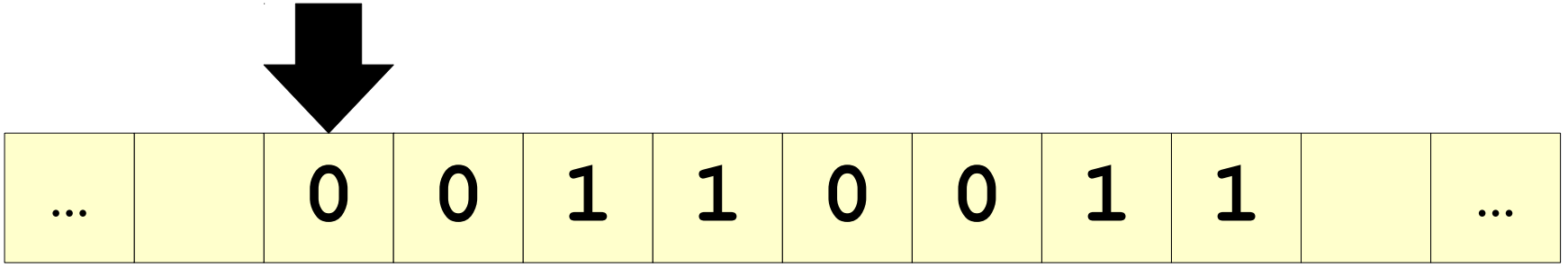
$$L = \{ ww \mid w \in \Sigma^* \text{ and } w \neq \varepsilon \}$$

- This is the set of all nonempty tautonyms.
- How might we design a TM for this language?

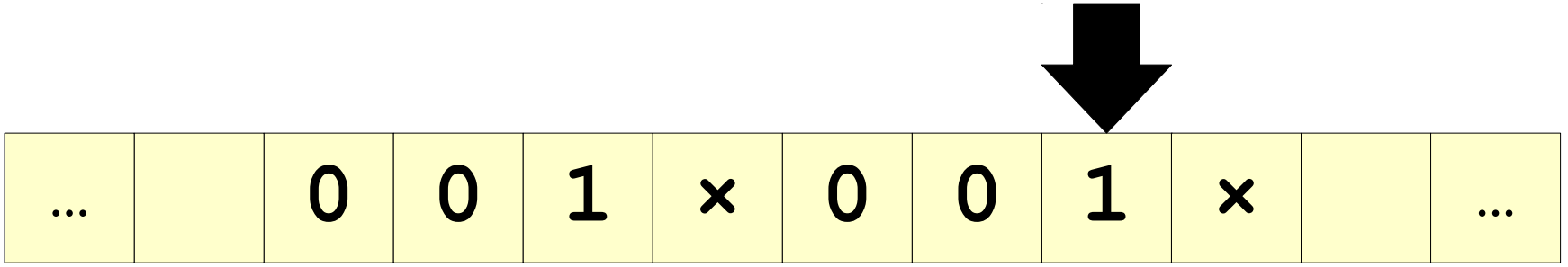
What's Tricky

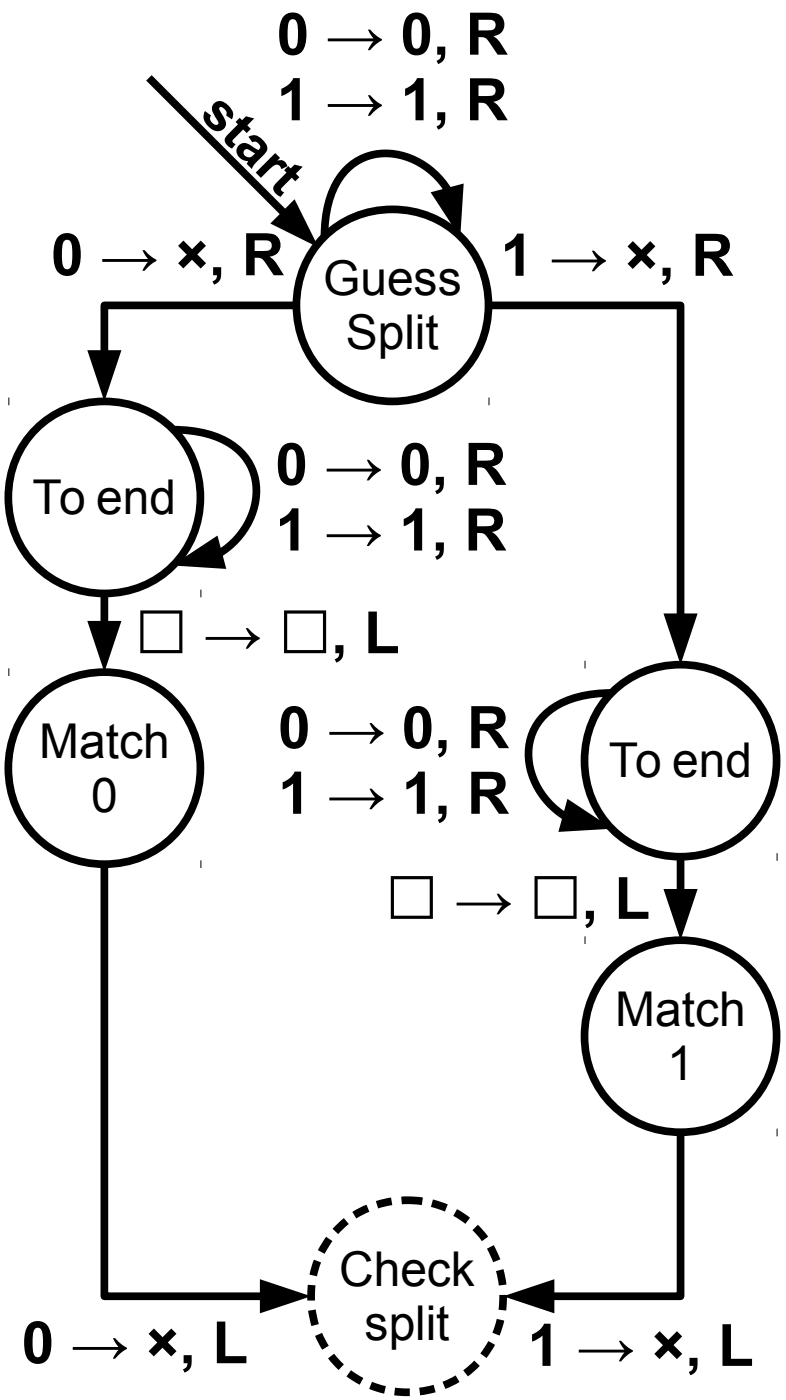


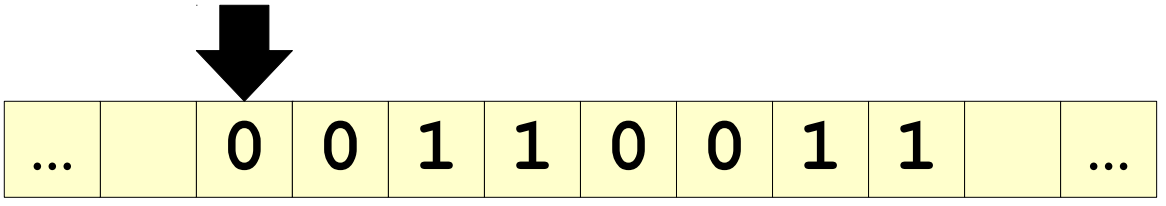
Using Nondeterminism



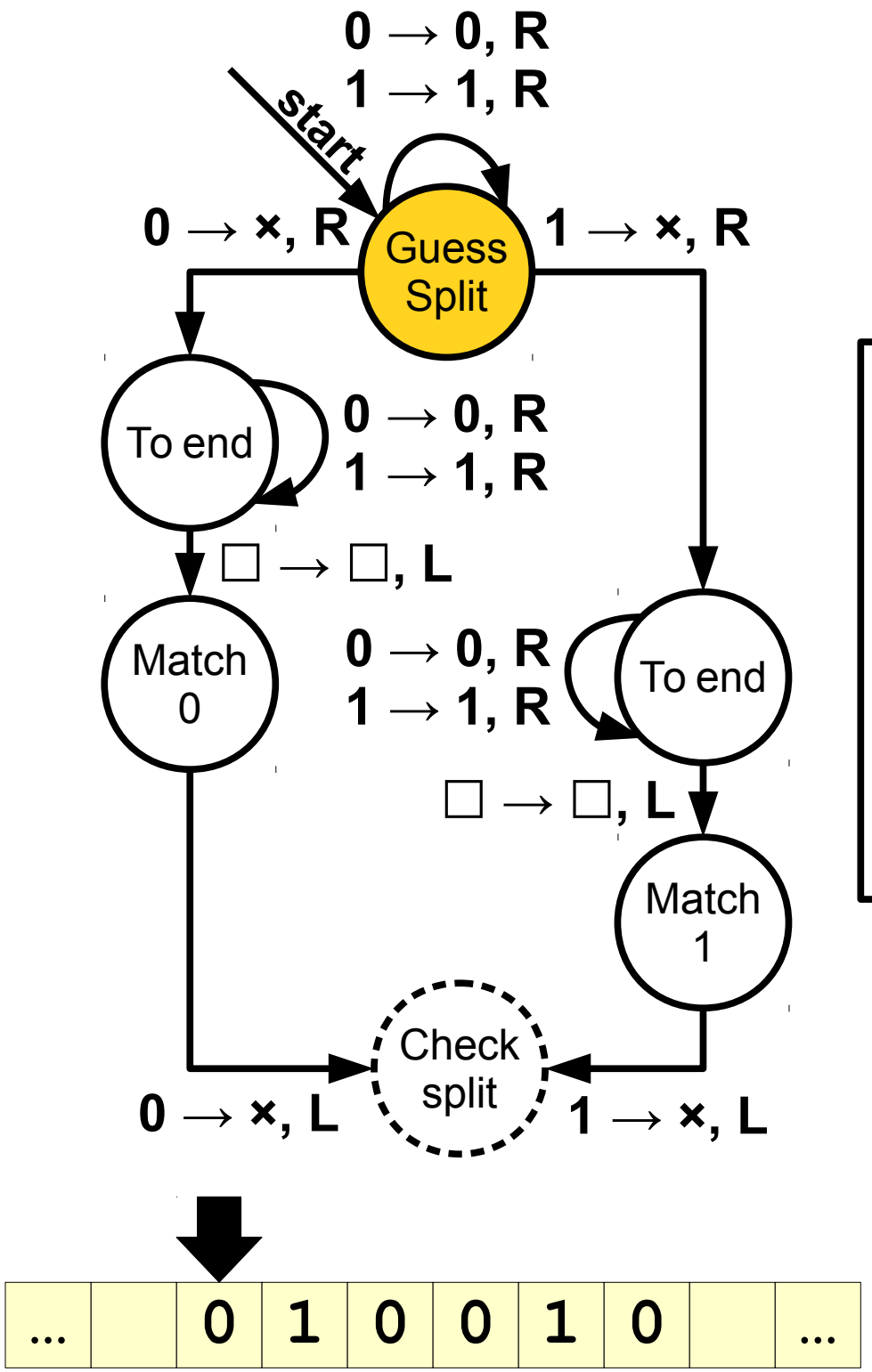
Using Nondeterminism




0 → 0, R
1 → 1, R
start
0 → ×, R
Guess
Split
1 → ×, R
To end
0 → 0, R
1 → 1, R
□ → □, L
Match
0
0 → 0, R
1 → 1, R
To end
□ → □, L
Match
1
Check
split
0 → ×, L
1 → ×, L

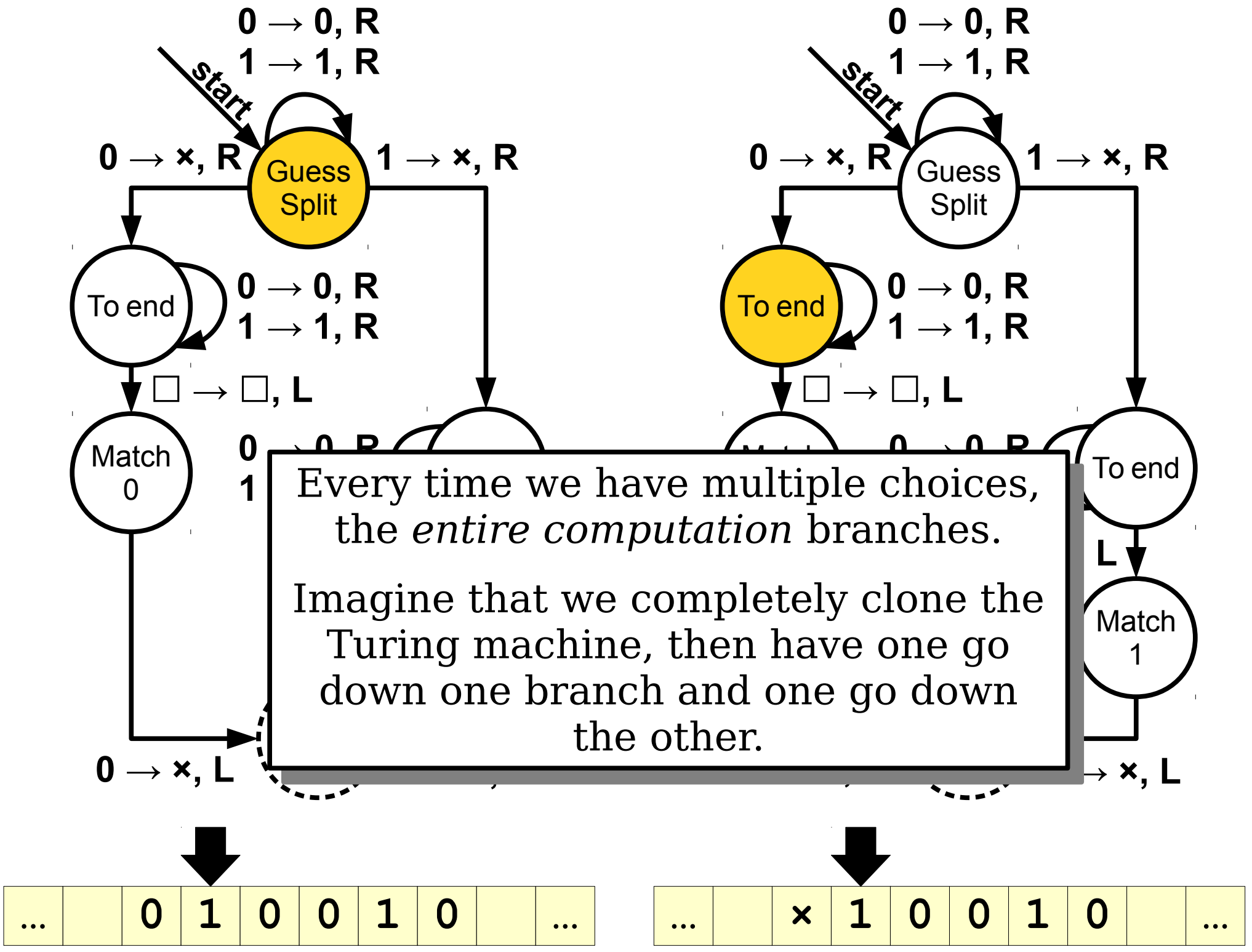

...
0
0
1
1
0
0
1
1
...

A ***huge*** difference between NTMs and NFAs.



In an NFA, we can follow multiple transitions at once by just being in many states at the same time.

That doesn't work with NTMs! The tapes will be different in each case.


0 → 0, R
1 → 1, R
start
0 → ×, R
Guess
Split
1 → ×, R
To end
0 → 0, R
1 → 1, R
□ → □, L
Match
0
0 → ×, L
0 → 0, R
1 → 1, R
start
0 → ×, R
Guess
Split
1 → ×, R
To end
0 → 0, R
1 → 1, R
□ → □, L
To end
L
Match
1
→ ×, L
Every time we have multiple choices, the *entire computation* branches.
Imagine that we completely clone the Turing machine, then have one go down one branch and one go down the other.
... 0 1 0 0 1 0 ...
... × 1 0 0 1 0 ...

Intuiting Nondeterministic TMs

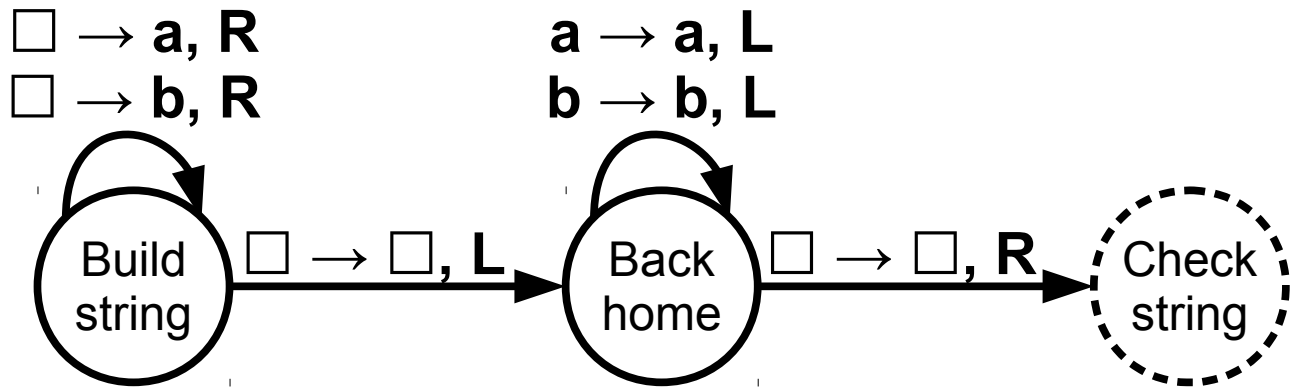
- Two of our previous NTM intuitions are useful here:
- ***Perfect guessing:*** If there is some choice of transitions that leads to an accepting state, the NTM can perfectly guess those transitions.
  - There's just one NTM, and it makes the right guess if one exists.
- ***Massive parallelism:*** The NTM tries all options. Each time it follows multiple transitions, it copies the current state of the machine once for each option, then tries each option.
  - Each step of the computation creates multiple new NTMs to try out each branch.
- The “guess-and-check” intuition from NFAs still applies here and is probably the best way to design NTMs.

Guessing Arbitrary Objects

- NTMs can use their nondeterminism to guess an arbitrary discrete, finite object.
- Idea: The NTM nondeterministically chooses a string to write on its tape, then does something with the string it just wrote.

Guessing an Arbitrary String

- As an example, here's how an NTM can guess an arbitrary string, then go do something with it:



- As a high-level description:

> N = "On input w:
> Nondeterministically guess a string $x \in \Sigma^*$.
> Deterministically check whether [...]"

Just How Powerful are NTMs?

NTMs and DTMs

- ***Theorem:*** If $L \in \mathbf{RE}$, then there is an NTM for L.
- ***Proof Sketch:*** Every deterministic TM (DTM) can be thought of as an NTM with no nondeterminism, so if L is the language of a DTM, it's also the language of an NTM. ■

NTMs and DTMs

- ***Theorem:*** If L is the language of an NTM, then $L \in \mathbf{RE}$.
- ***Faulty Proof Idea:*** Use the subset construction.
- Why doesn't this work?
  - In an NFA, the only “memory” is which states are active, so creating one state per configuration simulates the NFA with a DFA.
  - In an NTM, the memory is the current state plus the tape contents, so building one state per configuration is impossible.

NTMs and DTMs

- ***Theorem:*** If L is the language of an NTM, then $L \in \mathbf{RE}$.
- ***Proof Idea:*** Show how to construct an verifier for L using the NTM.
- We showed how to build a verifier for an arbitrary TM M by having the certificate for some $w \in L$ be the number of steps it takes for M to accept w.
- With an NTM, there might be many possible executions of length n on the string w.
- **Idea:** Our certificate will be the series of transitions that N is supposed to follow to accept w.

NTMs and DTMs

- ***Theorem:*** If L is the language of an NTM, then $L \in \mathbf{RE}$.
- ***Proof Sketch:*** Let N be an NTM for L. Then we can prove that this is a verifier for L:

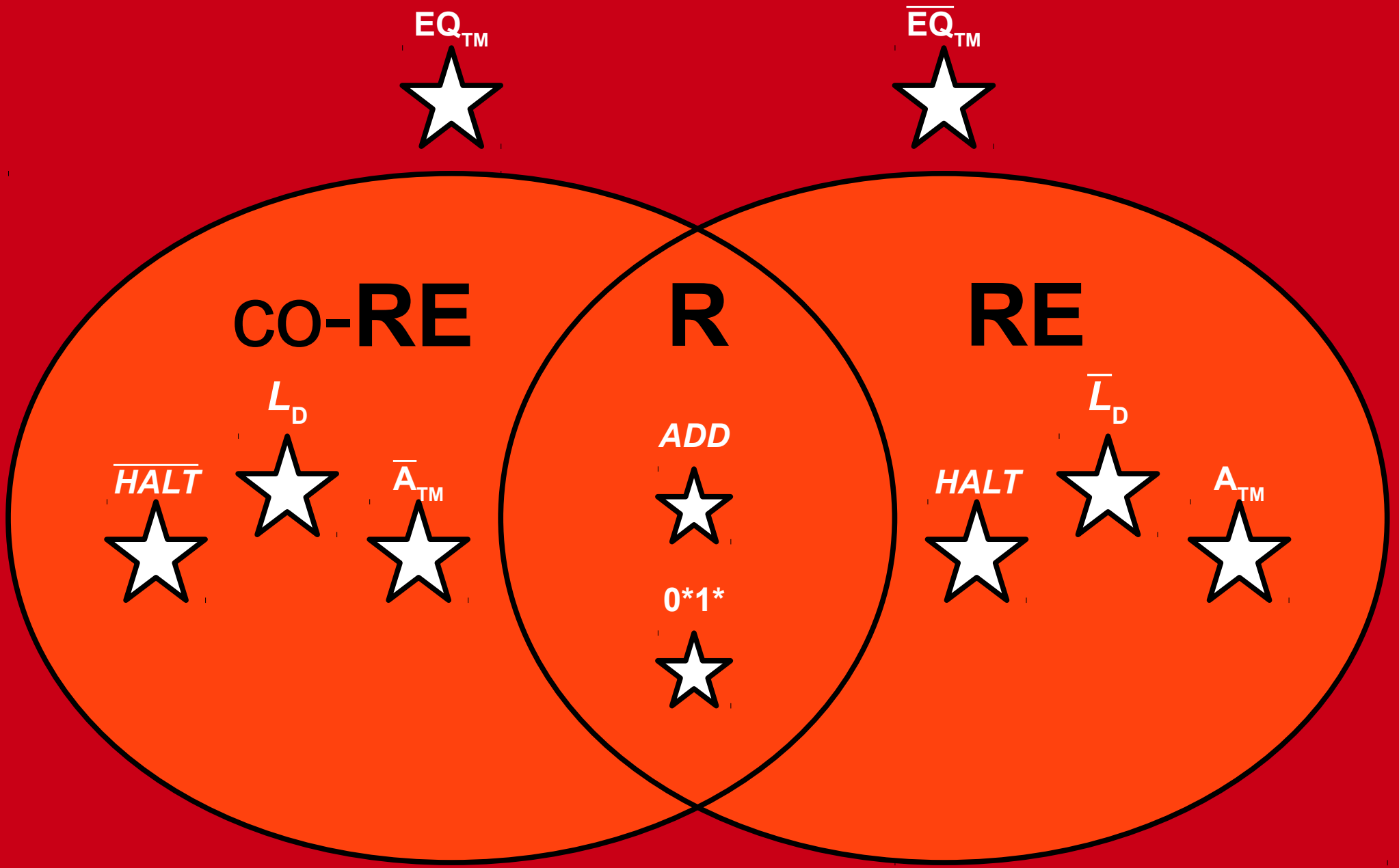
V = "On input $\langle w, T \rangle$, where T is a sequence of transitions:

- Run N on w, following transitions in the order specified in T.
- If any of the transitions in T are invalid or can't be followed, reject.
- If after following the transitions N accepts w, accept; otherwise reject.

Where We Stand

- We now know a ***lot*** about **R**, **RE**, and co-**RE**:
- The **RE** languages are the languages for which we can build a recognizer, an NTM, or which can be verified.
- The **R** languages are the decidable languages, or the languages which are both **RE** and co-**RE**.
- The co-**RE** languages are the complements of the **RE** languages and the languages for which there is a co-recognizer.
- The recursion theorem lets us find examples of languages that aren't in some of these classes.

The Limits of Computability



What problems can be solved by a computer?

What problems can be
solved ***efficiently*** by a computer?

Where We've Been

- The class **R** represents problems that can be solved by a computer.
- The class **RE** represents problems where “yes” answers can be verified by a computer.
- The class co-**RE** represents problems where “no” answers can be verified by a computer.

Where We're Going

- The class **P** represents problems that can be solved *efficiently* by a computer.
- The class **NP** represents problems where “yes” answers can be verified *efficiently* by a computer.
- The class co-**NP** represents problems where “no” answers can be verified *efficiently* by a computer.
- The *polynomial-time* mapping reduction can be used to find connections between problems.

Time-Out for Announcements!

Casual CS Dinner Tonight

- WiCS is holding their second biquarterly Casual CS Dinner tonight on the Gates fifth floor at 6PM.
- Meet fellow women CS students, chat with professors, talk with folks in industry, and have a nice dinner!

Problem Set Six Grading

- The TAs are humming along and grading PS6. It should be ready by Friday.

Your Questions

“The distribution for midterm 2 was very discouraging. I got everything correct except one detail and got put right below median. I studied extremely hard but apparently not doing it right. How can we best prepare for exams to meet such high curve?”

“I've spoken to friends in industry, and they seem to only speak of discrete math as something they did long ago – almost like a rite of passage. How will CS103 help us become better programmers? Why do we need this? I'm missing the big picture.”

“Can we have a 103 trip for the Alan Turing movie (aka the Imitation Game, staring Benedict Cumberbatch)?”

“Keith, is it possible to build a Turing Machine that checks $\{w \mid w$ is an encoding of some Turing Machine $H\}$? How about a decider?”

“Is it theoretically possible for a machine to create a machine that is more complex than itself?”

“In class, you mentioned that there is no physical analog to NFAs, and that quantum computers "do not really match" how an NFA works. I've read a bit about QFAs - could you give us a broad idea of how they can solve problems that we otherwise can't?”

“How does quantum computing relate to Turing Machines and DFAs? Do the same bounds apply?”

“I get that there are some types of problems that are impossible to compute, but so far they all seem 100% theoretical. What are the practical implications of what we're talking about? Can you give examples of real world problems impossible to solve?”

Back to CS103!

It may be that since one is customarily concerned with existence, [...] decidability, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-decidable* algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”

A Decidable Problem

- ***Presburger arithmetic*** is a logical system for reasoning about arithmetic.
  - $\forall x.\ x + 1 \neq 0$
  - $\forall x.\ \forall y.\ (x + 1 = y + 1 \rightarrow x = y)$
  - $\forall x.\ x + 0 = x$
  - $\forall x.\ \forall y.\ (x + y) + 1 = x + (y + 1)$
  - $\forall x.\ ((P(0) \wedge \forall y.\ (P(y) \rightarrow P(y + 1))) \rightarrow \forall x.\ P(x)$
- Given a statement, it is decidable whether that statement can be proven from the laws of Presburger arithmetic.
- Any Turing machine that decides whether a statement in Presburger arithmetic is true or false has to move the tape head at least $2^{2^{cn}}$ times on some inputs of length n (for some fixed constant c).

For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$
$$2^{2^1} = 4$$
$$2^{2^2} = 16$$
$$2^{2^3} = 256$$
$$2^{2^4} = 65536$$
$$2^{2^5} = 18446744073709551616$$
$$2^{2^6} = 340282366920938463463374607431768211456$$

The Limits of Decidability

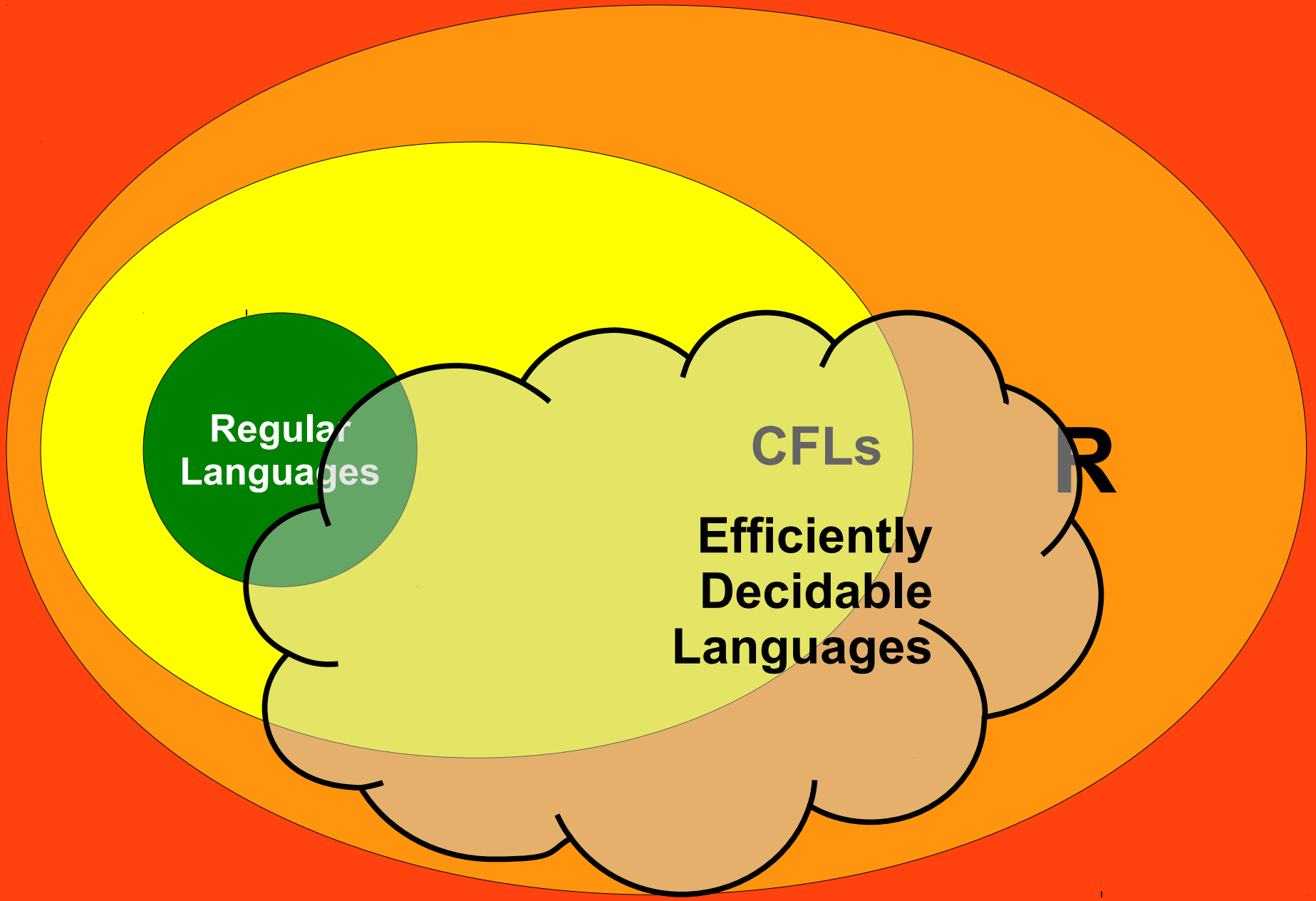
- The fact that a problem is decidable does not mean that it is *feasibly* decidable.
- In ***computability theory***, we ask the question

  Is it possible to solve problem *L* ***at all***?

- In ***complexity theory***, we ask the question

  Is it possible to solve problem *L* ***efficiently***?

- In the remainder of this course, we will explore this question in more detail.


Regular Languages
CFLs
R
Efficiently Decidable Languages
Undecidable Languages

The Setup

- In order to study computability, we needed to answer these questions:
  - What is “computation?”
  - What is a “problem?”
  - What does it mean to “solve” a problem?
- To study complexity, we need to answer these questions:
  - What does “complexity” even mean?
  - What is an “efficient” solution to a problem?

Measuring Complexity

- Suppose that we have a decider D for some language L.
- How might we measure the complexity of D?
  - Number of states.
  - Size of tape alphabet.
  - Size of input alphabet.
  - Amount of tape required.
  - Amount of time required.
    - Number of times a given state is entered.
    - Number of times a given symbol is printed.
    - Number of times a given transition is taken.
    - (Plus a whole lot more...)

What is an efficient algorithm?

Searching Finite Spaces

- Many decidable problems can be solved by searching over a large but finite space of possible options.
- Searching this space might take a staggeringly long time, but only finite time.
- From a decidability perspective, this is totally fine.
- From a complexity perspective, this is totally unacceptable.

A Sample Problem

4	3	11	9	7	13	5	6	1	12	2	8	0	10

Longest so far: 4 11

How many different subsequences are there in a sequence of n elements? $\mathbf{2^n}$

How long does it take to check each subsequence? $\mathbf{O(n)}$ time.

Runtime is around $\mathbf{O(n \cdot 2^n)}$.

A Sample Problem

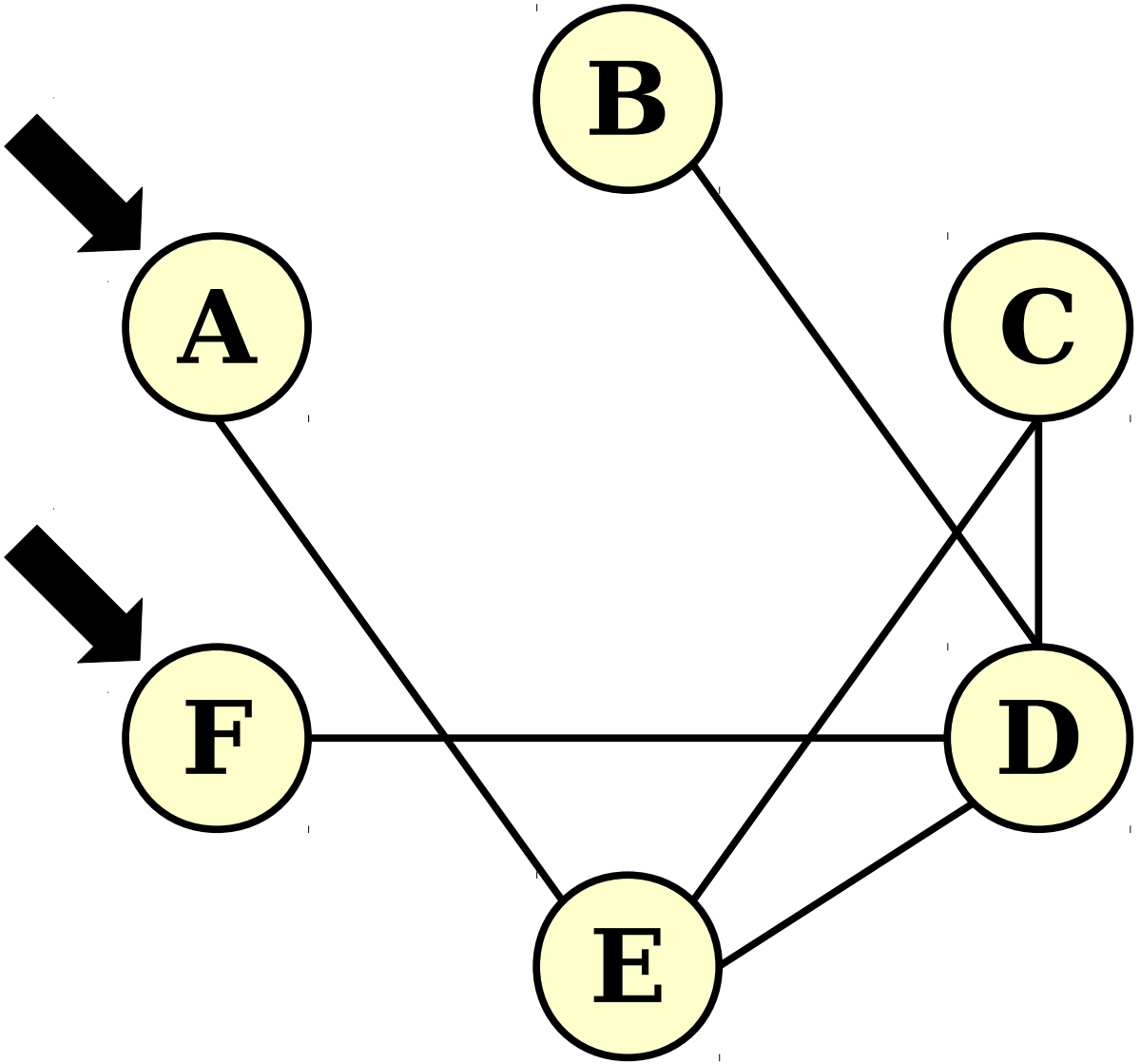
4	3	11	9	7	13	5	6	1	12	2	8	0	10

1	1	2	2	2	3	2	3	1	4	2	4	1	5

How many elements of the sequence do we have to look at when considering the kth element of the sequence? $\boldsymbol{k - 1}$

Total runtime is
$1 + 2 + \ldots + (n - 1) = \mathbf{O(n^2)}$

Another Problem

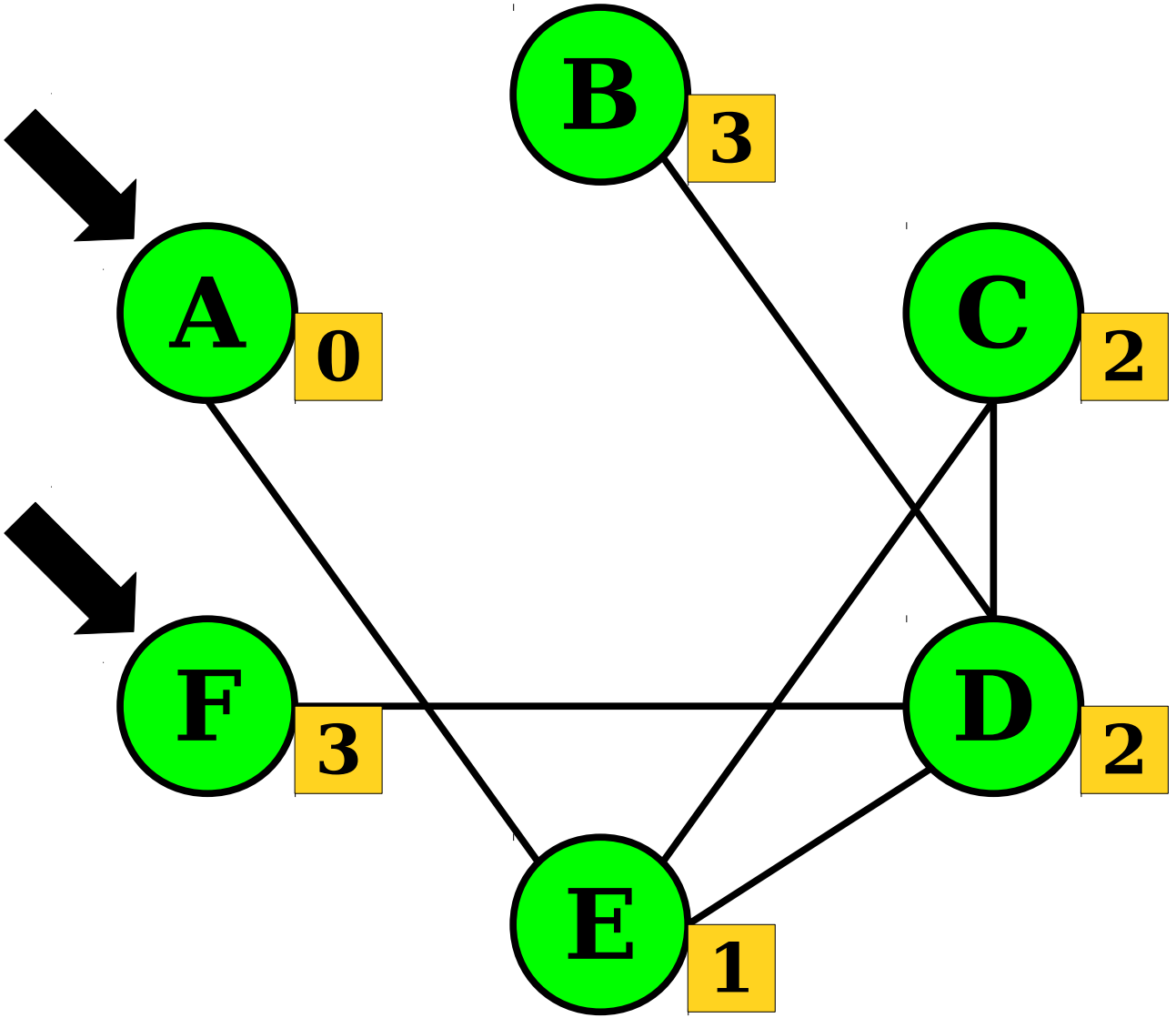


Number of possible ways to order a subset of n nodes is **$O(n \cdot n!)$**

Time to check a path is **$O(n)$**.

Runtime: **$O(n^2 \cdot n!)$**

Another Problem



With a precise analysis, runtime is **O(*n* + *m*)**, where *n* is the number of nodes and *m* is the number of edges.

For Comparison

- **Longest increasing subsequence:**
  - Naive: $O(n \cdot 2^n)$
  - Fast: $O(n^2)$
- **Shortest path problem:**
  - Naive: $O(n^2 \cdot n!)$
  - Fast: $O(n + m)$, where n is the number of nodes and m the number of edges. (Take CS161 for details!)

Defining Efficiency

- When dealing with problems that search for the “best” object of some sort, there are often at least exponentially many possible options.
- Brute-force solutions tend to take at least exponential time to complete.
- Clever algorithms often run in time $O(n)$, or $O(n^2)$, or $O(n^3)$, etc.

Polynomials and Exponentials

- An algorithm runs in ***polynomial time*** if its runtime is some polynomial in n.
  - That is, time $O(n^k)$ for some constant k.
- Polynomial functions "scale well."
  - Small changes to the size of the input do not typically induce enormous changes to the overall runtime.
- Exponential functions scale terribly.
  - Small changes to the size of the input induce huge changes in the overall runtime.

The Cobham-Edmonds Thesis

A language L can be ***decided efficiently*** if there is a TM that decides it in polynomial time.

Equivalently, L can be decided efficiently iff it can be decided in time $O(n^k)$ for some $k \in \mathbb{N}$.

Like the Church-Turing thesis, this is ***not*** a theorem!

It's an assumption about the nature of efficient computation, and it is somewhat controversial.

The Cobham-Edmonds Thesis

- Efficient runtimes:
  - $4n + 13$
  - $n^3 - 2n^2 + 4n$
  - $n \log \log n$
- "Efficient" runtimes:
  - $n^{1,000,000,000,000}$
  - 10^{500}
- Inefficient runtimes:
  - 2^n
  - $n!$
  - n^n
- "Inefficient" runtimes:
  - $n^{0.0001 \log n}$
  - 1.000000001^n

Why Polynomials?

- Polynomial time *somewhat* captures efficient computation, but has a few edge cases.
- However, polynomials have very nice mathematical properties:
  - The sum of two polynomials is a polynomial. (Running one efficient algorithm after the other gives an efficient algorithm.)
  - The product of two polynomials is a polynomial. (Running one efficient algorithm a “reasonable” number of times gives an efficient algorithm.)
  - The *composition* of two polynomials is a polynomial. (Using the output of one efficient algorithm as the input to another efficient algorithm gives an efficient algorithm.)

The Complexity Class **P**

- The ***complexity class* P** (for ***p***olynomial time) contains all problems that can be solved in polynomial time.
- Formally:

$$\mathbf{P} = \{ L \mid \text{There is a polynomial-time decider for } L \}$$

- Assuming the Cobham-Edmonds thesis, a language is in **P** if it can be decided efficiently.