Unsolvable Problems

Part Two

Recap from Last Time

What does this program do?

```
bool willAccept(string program, string input) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willAccept(me, input)) {
      reject();
   } else {
      accept();
   }
}
```

What happens if...

... this program accepts its input?
It rejects the input!

... this program doesn't accept its input?
It accepts the input!

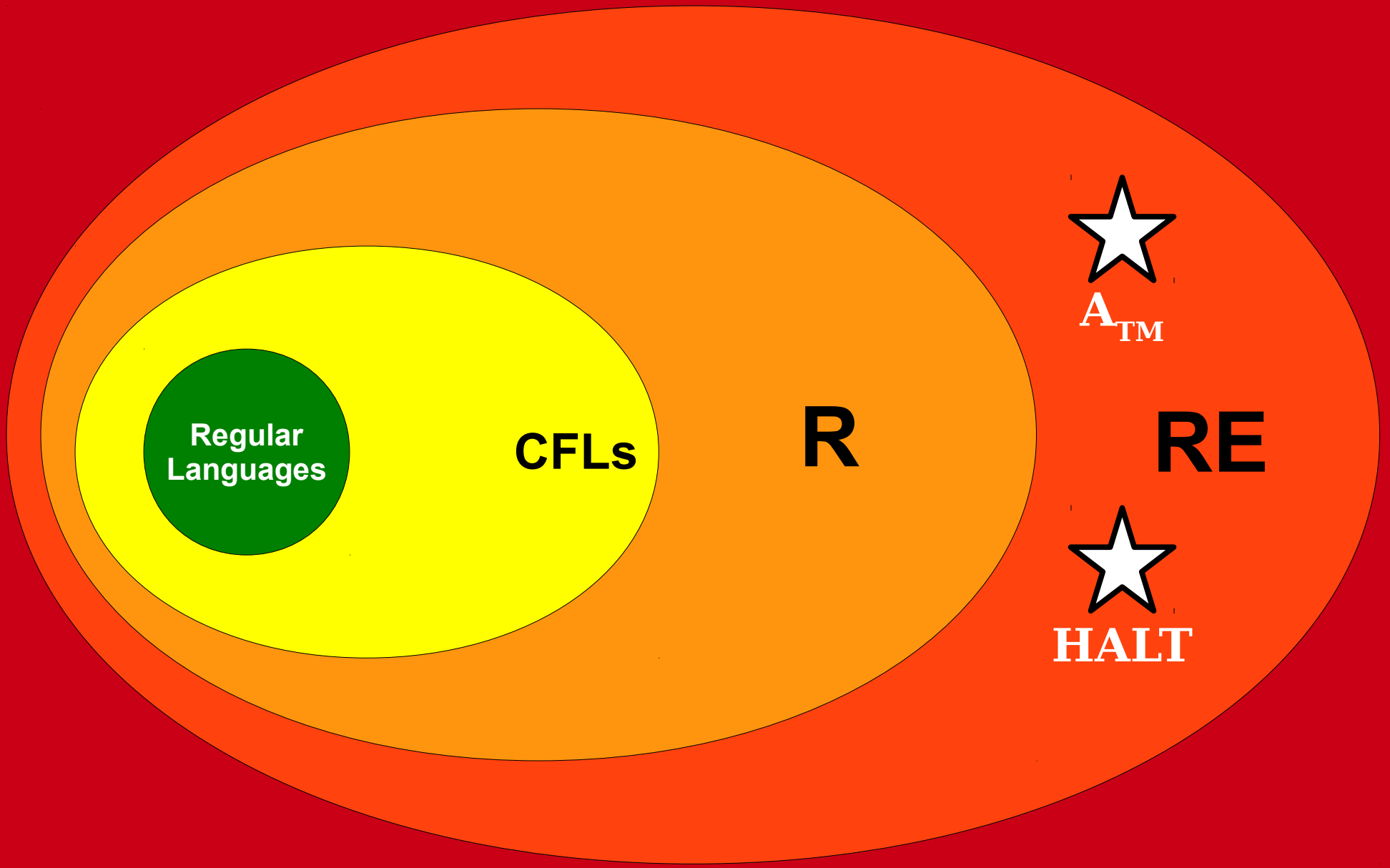
What does this program do?

```
bool willHalt(string program, string input) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willHalt(me, input)) {
      while (true) {
         // loop infinitely
      }
   } else {
      accept();
   }
}
```

What happens if...

… this program halts on this input?
It loops on the input!

… this program loops on this input?
It halts on the input!


Regular Languages
CFLs
R
RE
A_{TM}
HALT
All Languages

New Stuff – Lots of It!

So What?

- These problems might not seem all that exciting, so who cares if we can't solve them?
- Turns out, this same line of reasoning can be used to show that some very important problems are impossible to solve.

Secure Voting

- Suppose that you want to make a voting machine for use in an election between two parties.
- Let $\Sigma = \{$`r`, `d`$\}$. A string in w corresponds to a series of votes for the candidates.
- Example: `rrdddrd` means “two people voted for `r`, then three people voted for `d`, then one more person voted for `r`, then one more person voted for `d`.”

Secure Voting

- A voting machine is a program that accepts a string of `r`'s and `d`'s, then reports whether person `r` won the election.
- Formally: a TM M is a secure voting machine if $\mathscr{L}(M) = \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more `r`'s than `d`'s $\}$
- ***Question:*** Given a TM that someone claims is a secure voting machine, could we automatically check whether it actually is a secure voting machine?
  - That is, is there an *algorithm* we can follow to determine this?

Secure Voting

- The ***secure voting problem*** is the following:

  Given a TM *M*, is the language of *M*
  { *w* ∈ {r, d}* | *w* has more r's than d's }?

- ***Claim:*** This problem is not decidable – there is no algorithm that can check an arbitrary TM to verify that it's a secure voting machine!

Secure Voting

- Suppose that the secure voting problem is decidable. Then we could write a function

```
bool isSecureVotingMachine(string program)
```

  that would accept as input a program and return whether or not it's a secure voting machine.
- As you might expect, this lets us do Cruel and Unusual Things...

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   bool answer = countRs(input) > countDs(input);
   if (isSecureVotingMachine(me)) answer = !answer;

   if (answer) accept();
   else reject();
}
```

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

What happens if...

… this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This program is a secure voting machine.

What happens if...

… this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This program is a secure voting machine.

It should accept precisely the strings in { $w \in$ {`r`, `d`}* | w has more `r`'s than `d`'s }.

What happens if...

… this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This program is a secure voting machine.

It should accept precisely the strings in { $w \in$ {`r`, `d`}* | w has more `r`'s than `d`'s }.

What happens if we run it on `ddd`?

What happens if...

… this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This program is a secure voting machine.

It should accept precisely the strings in { $w \in$ {`r`, `d`}* | w has more `r`'s than `d`'s }.

What happens if we run it on `ddd`?

What happens if...

… this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This program is a secure voting machine.

It should accept precisely the strings in { $w \in$ {`r`, `d`}* | w has more `r`'s than `d`'s }.

What happens if we run it on `ddd`?

What happens if...

… this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();
    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This program is a secure voting machine.

It should accept precisely the strings in { $w \in$ {`r`, `d`}* | w has more `r`'s than `d`'s }.

What happens if we run it on `ddd`?

What happens if...

... this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();
    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **false**

This program is a secure voting machine.

It should accept precisely the strings in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on **ddd**?

What happens if...

... this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **false**

This program is a secure voting machine.

It should accept precisely the strings in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on **ddd**?

What happens if...

... this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **true**

This program is a secure voting machine.

It should accept precisely the strings in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on **ddd**?

What happens if...

... this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **true**

This program is a secure voting machine.

It should accept precisely the strings in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on **ddd**?

What happens if...

... this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **true**

This program is a secure voting machine.

It should accept precisely the strings in
{ $w \in \{$**r**, **d**$\}^*$ | w has more **r**'s than **d**'s }.

What happens if we run it on **ddd**?

What happens if...

… this program is a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **true**

This program is a secure voting machine.

It should accept precisely the strings in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on **ddd**?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if...

... this program is a secure voting machine?
then it's not a secure voting machine!

... this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on a string in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

... this program is a secure voting machine?
then it's not a secure voting machine!

... this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on a string in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on a string in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();
    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on a string in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();
    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **true**

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on a string in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **true**

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on a string in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **true**

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on a string in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What happens if we run it on a string in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

... this program is a secure voting machine?
then it's not a secure voting machine!

... this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What if we run it on a string ***not*** in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What if we run it on a string ***not*** in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();
    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What if we run it on a string ***not*** in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();
    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What if we run it on a string ***not*** in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();
    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **false**

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What if we run it on a string ***not*** in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **false**

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What if we run it on a string ***not*** in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **false**

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What if we run it on a string ***not*** in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **false**

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What if we run it on a string ***not*** in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **false**

This isn't a secure voting machine, so its language is not
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What if we run it on a string ***not*** in
{ $w \in$ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if...

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?
then it is a secure voting machine!

Theorem: The secure voting problem is undecidable.

Proof: By contradiction; assume that the secure voting problem is decidable. Then there is some decider D for the secure voting problem. If D is given any TM, it will determine whether the TM is a secure voting machine and report back the answer.

Given this, we could then construct the following TM:

> M = "On input w:
> Have M obtain its own description, $\langle M \rangle$.
> Have M determine whether w has more **r**'s than **d**'s.
> Run D on $\langle M \rangle$ and see what it says.
> If D says that M is a secure voting machine, flip the result from the previous step.
> If the net result is "yes," accept; else reject."

We consider two cases. First, suppose M is a secure voting machine. Trace through the execution of M on the string **ddd**. M will determine that **ddd** does not have more **r**'s than **d**'s. When it runs D on itself, it will determine it is a secure voting machine and then flip the previous result. It then accepts **ddd**. This means that $\mathscr{L}(M) \neq \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more **r**'s than d's $\}$, so M is not a secure voting machine, contradicting what we said earlier.

Next, suppose M is not a secure voting machine. Then $\mathscr{L}(M) \neq \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more **r**'s than **d**'s $\}$. Now, consider any string $w \in \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more **r**'s than **d**'s $\}$. Tracing through the execution of M on w, we see that M will determine that w has more **r**'s than **d**'s, not flip the result, and therefore accept w. Similarly, if we run M on any string not in $\{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more **r**'s than **d**'s $\}$, then M will determine that w has at least as many **r**'s as **d**'s, not flip the result, and therefore reject w. Overall, this means that M accepts precisely the strings in $\{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more **r**'s than **d**'s $\}$, so $\mathscr{L}(M) = \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more **r**'s than **d**'s $\}$, contradicting our initial assumption.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, the secure voting problem is undecidable. ■

Interpreting this Result

- The previous argument tells us that *there is no general algorithm* that we can follow to determine whether a program is a secure voting machine. In other words, any general algorithm to check voting machines will always be wrong on at least one input.
- So what can we do?
  - Design algorithms that work in *some,* but not *all* cases. (This is often done in practice.)
  - Fall back on human verification of voting machines. (We do that too.)
  - Carry a healthy degree of skepticism about electronic voting machines. (Then again, did we even need the theoretical result for this?)

Beyond **R** and **RE**

Beyond **R** and **RE**

- We've now seen how to use self-reference as a tool for showing undecidability (finding languages not in **R**).
- We still have not broken out of **RE** yet, though.
- To do so, we will need to build up a better intuition for the class **RE**.

What exactly is the class **RE**?

RE, Formally

- Recall that the class **RE** is the class of all recognizable languages:

  RE = { L | there is a TM M where $\mathscr{L}(M) = L$ }

- Since **R** ≠ **RE**, there is no general way to "solve" problems in the class **RE**.
- When designing TMs for **RE** languages, the goal is just to be able to confirm "yes" answers.

Key Intuition:

A language L is in **RE** if, for any string w, if you are *convinced* that $w \in L$, there is some way you could prove that to someone else.

Verification

- ***Recall:*** When focusing on the **RE** languages, we need to abandon the idea that we can “solve” the problems we're looking at.
- Rather than *solving problems*, we can think about *checking answers*.

Verification

		7		6		1		
					3		5	2
3			1		5	9		7
6		5		3		8		9
	1						2	
8		2		1		5		4
1		3	2		7			8
5	7		4					
		4		8		7		

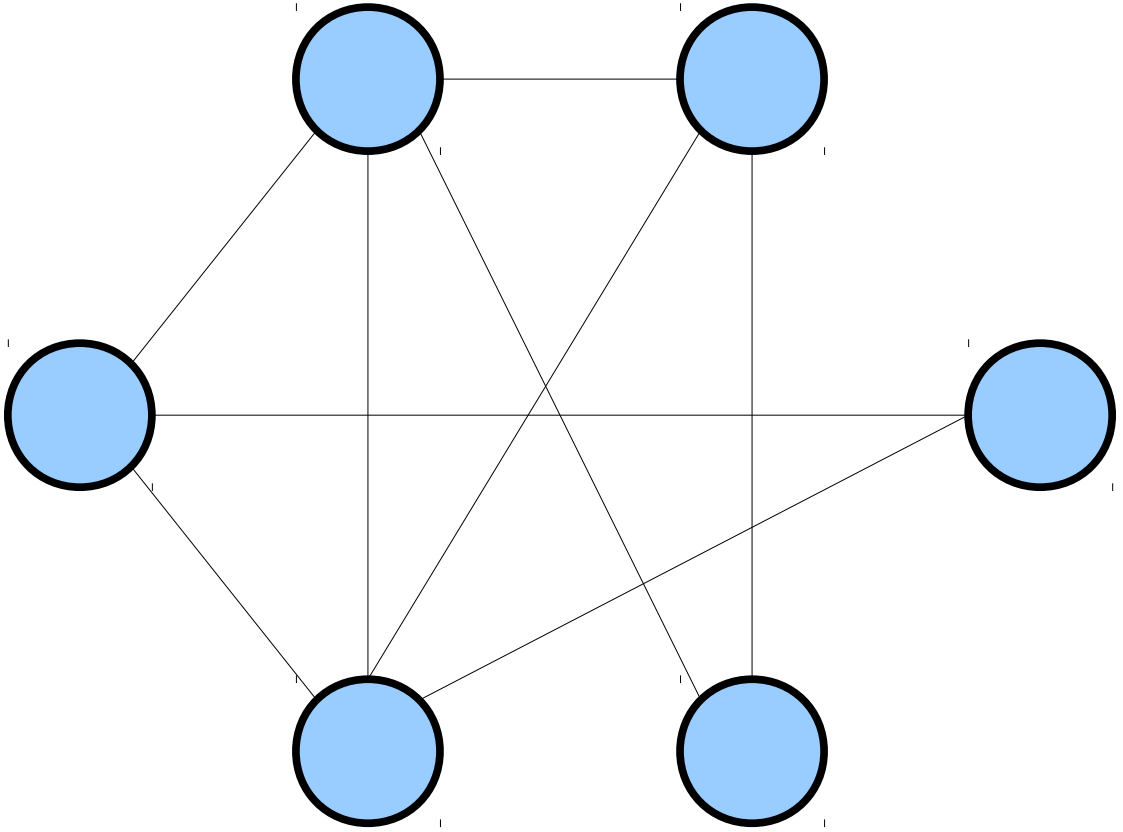
Does this Sudoku puzzle have a solution?

Verification

2	5	7	9	6	4	1	8	3
4	9	1	8	7	3	6	5	2
3	8	6	1	2	5	9	4	7
6	4	5	7	3	2	8	1	9
7	1	9	5	4	8	3	2	6
8	3	2	6	1	9	5	7	4
1	6	3	2	5	7	4	9	8
5	7	8	4	9	6	2	3	1
9	2	4	3	8	1	7	6	5

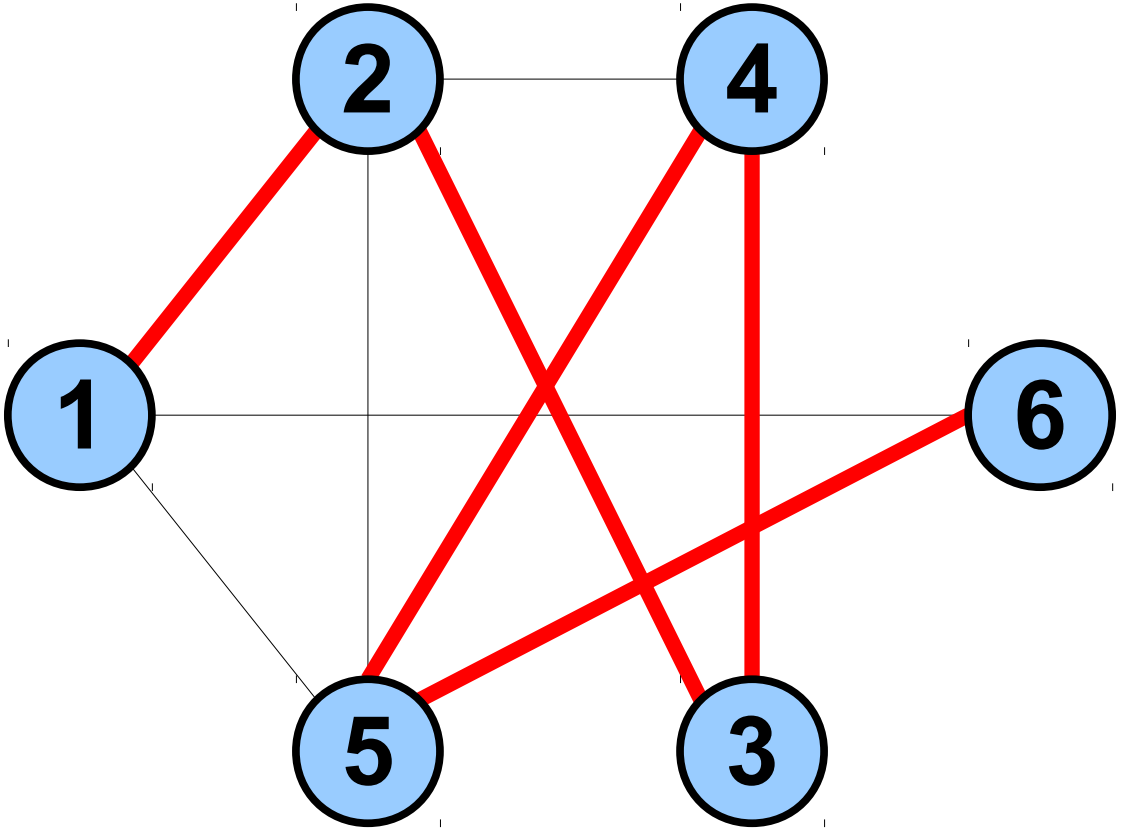
Does this Sudoku puzzle have a solution?

Verification



Is there a simple path that goes through every node exactly once?

Verification



Is there a simple path that goes through every node exactly once?

Verification

11

Does the hailstone sequence
terminate for this number?

Verification

11

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

34

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

17

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

52

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

26

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

13

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

40

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

20

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

10

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

5

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

16

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

8

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

4

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

2

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

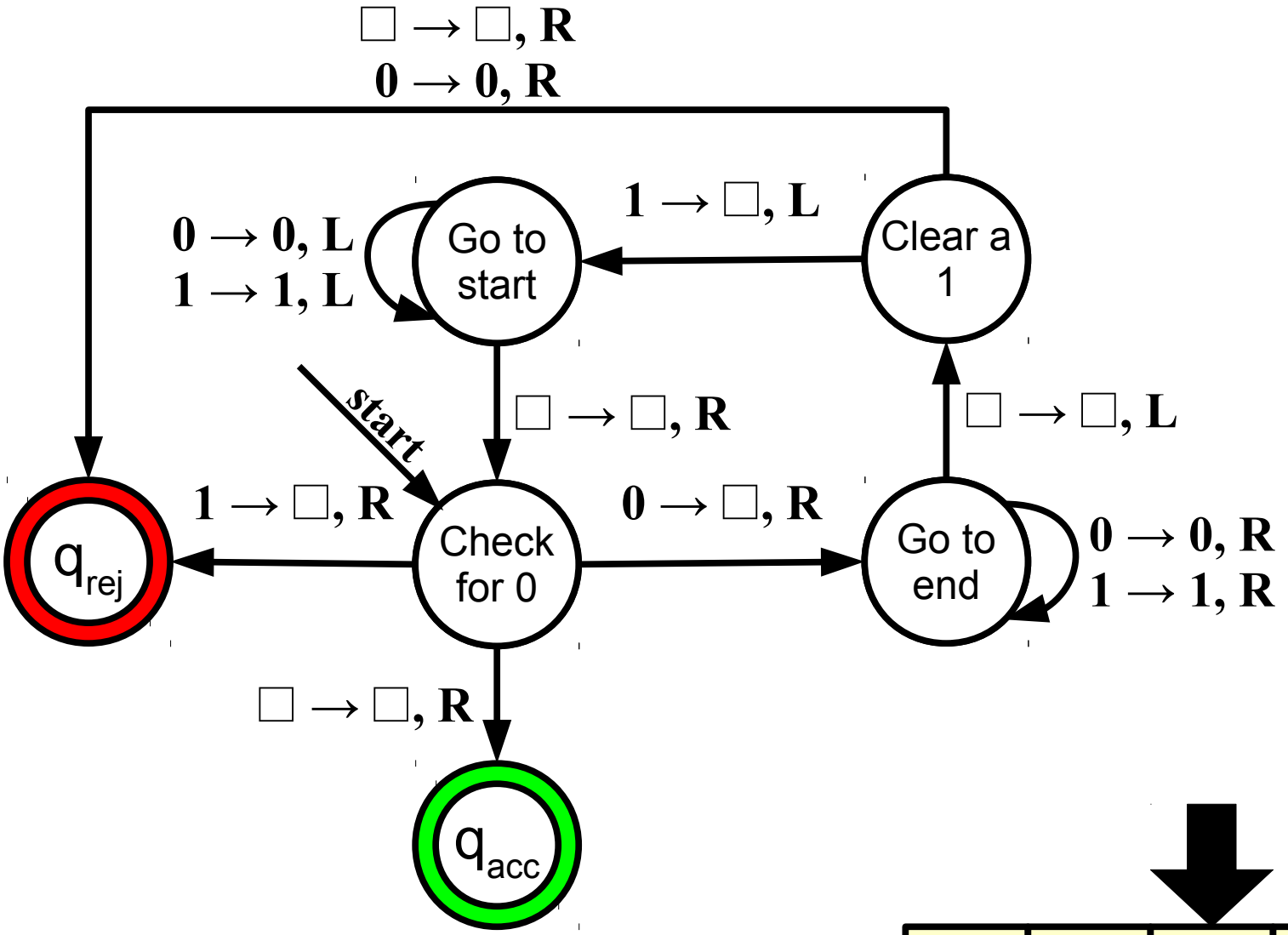
Verification

1

Try running fourteen steps of the Hailstone sequence.

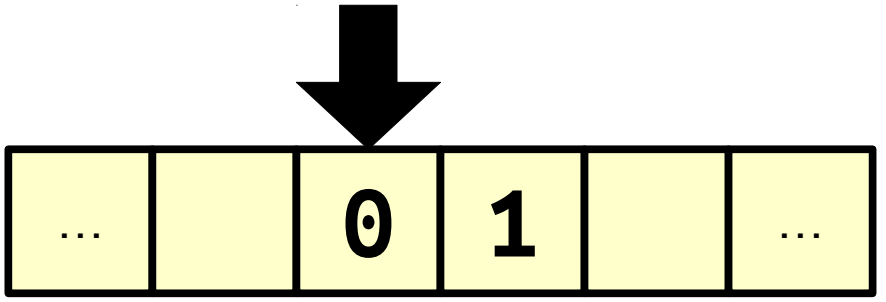
Does the hailstone sequence
terminate for this number?

Verification

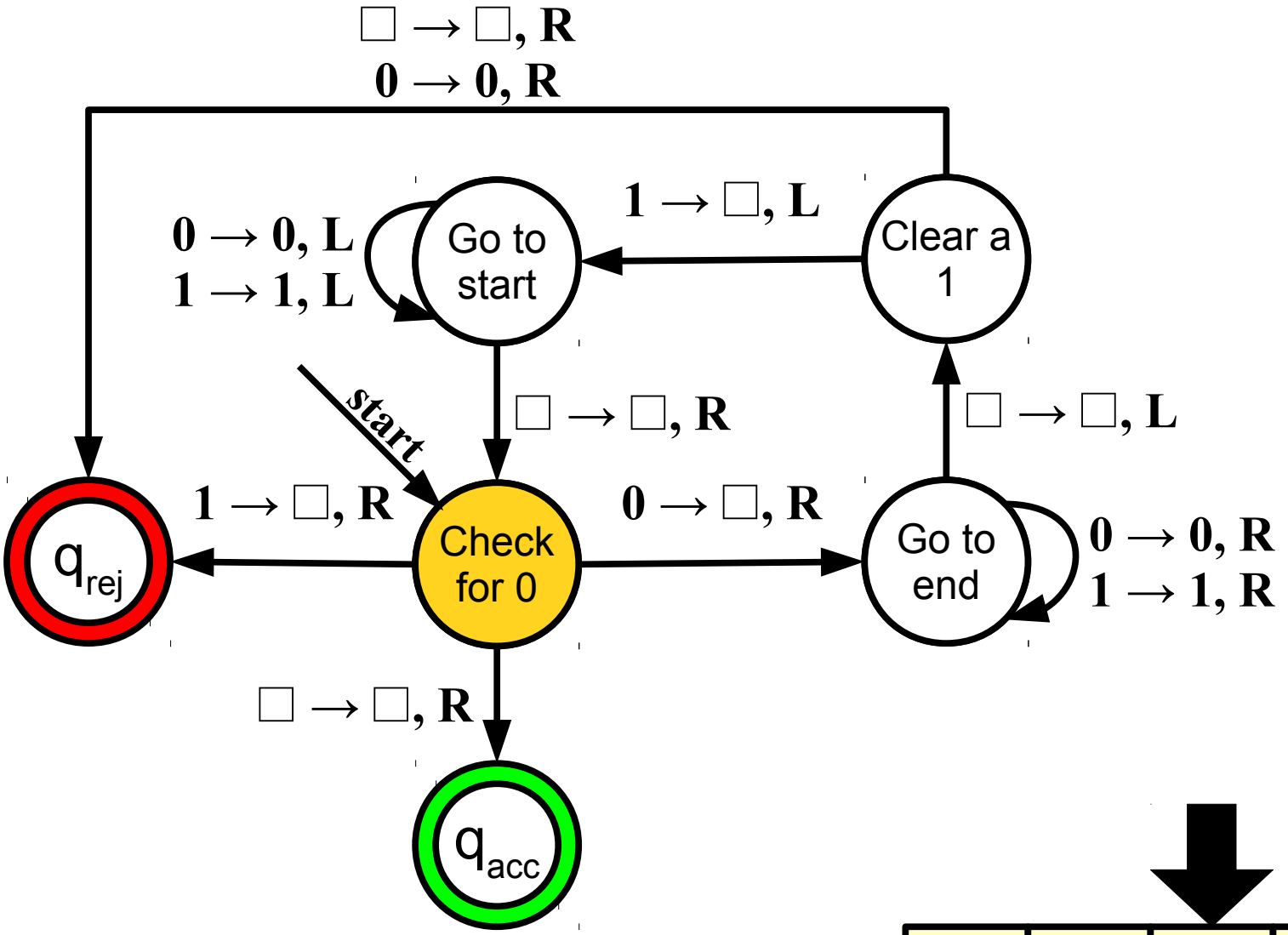


Try running it for six steps.

Does this TM halt on this input?

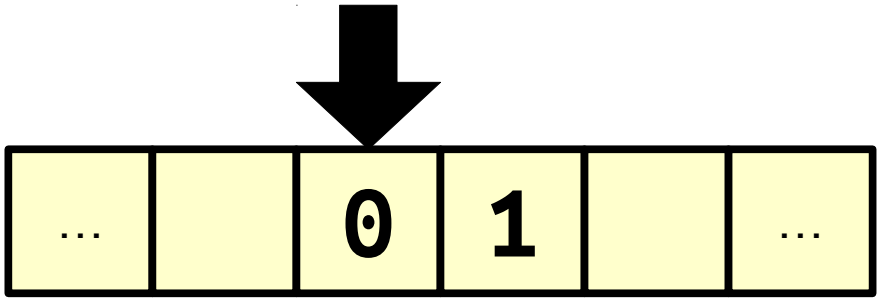


Verification

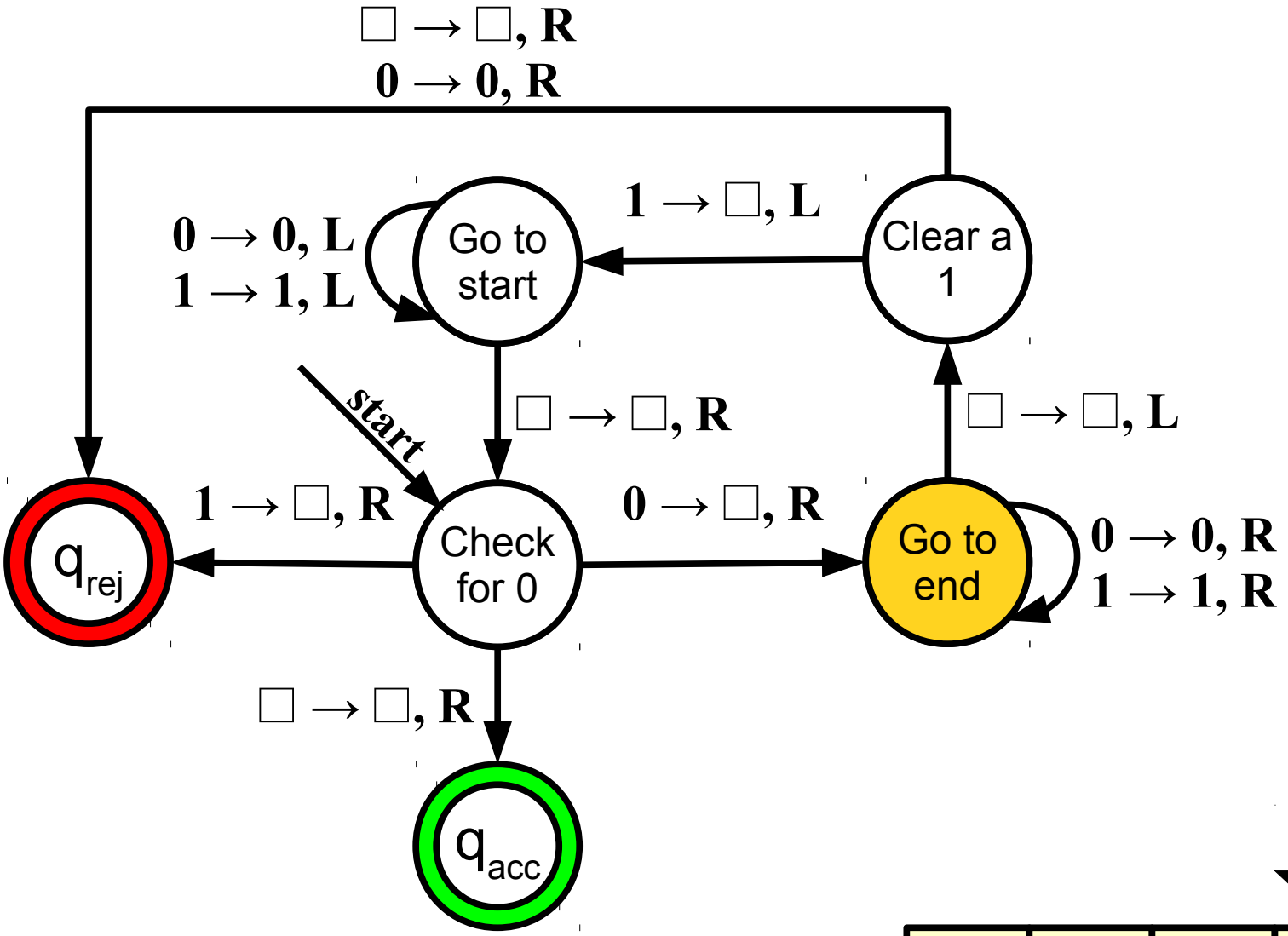


Try running it for six steps.

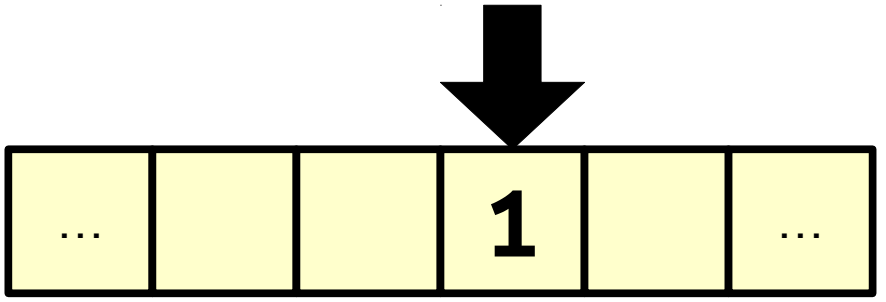
Does this TM halt on this input?



Verification

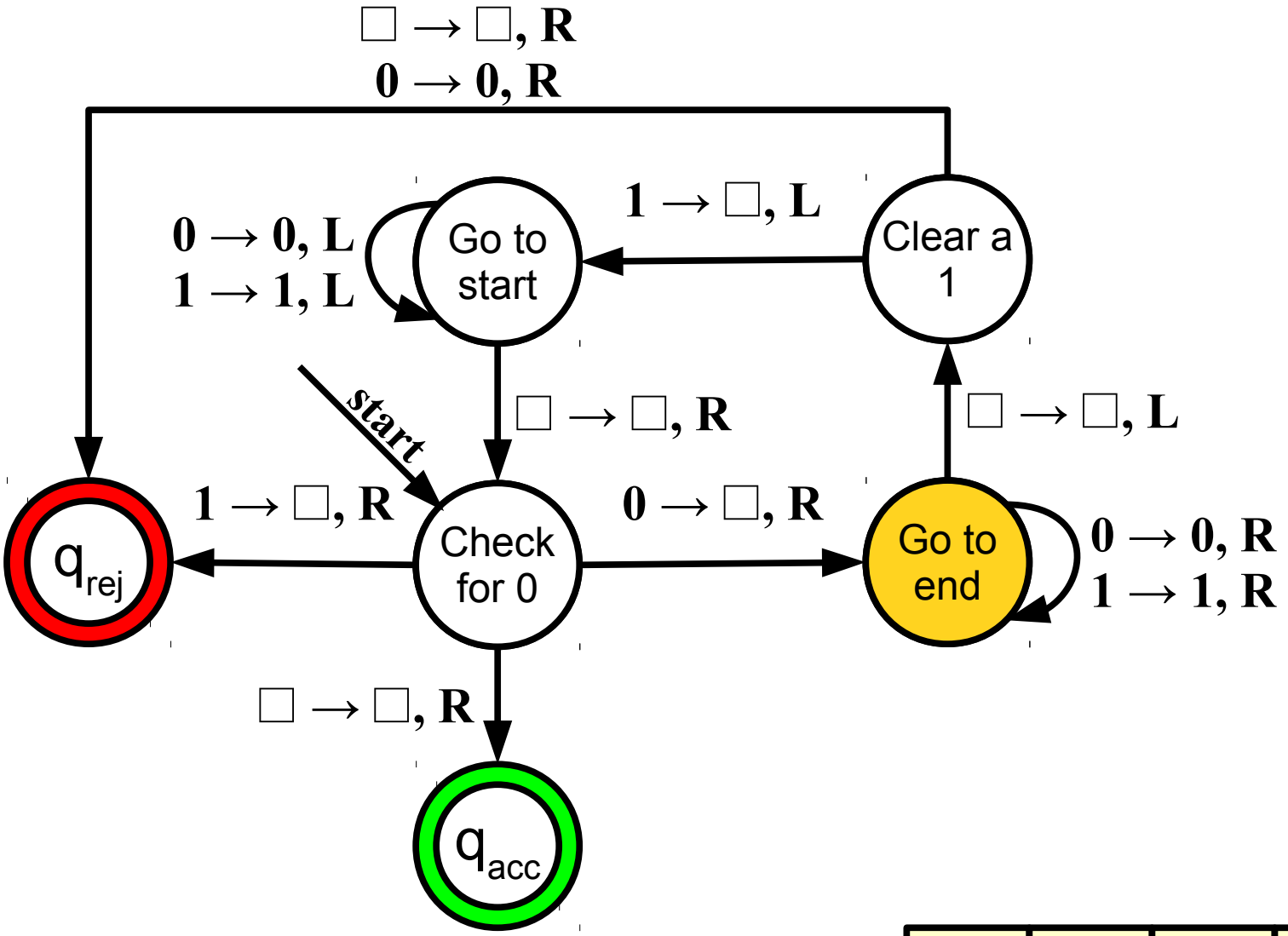


Try running it for six steps.

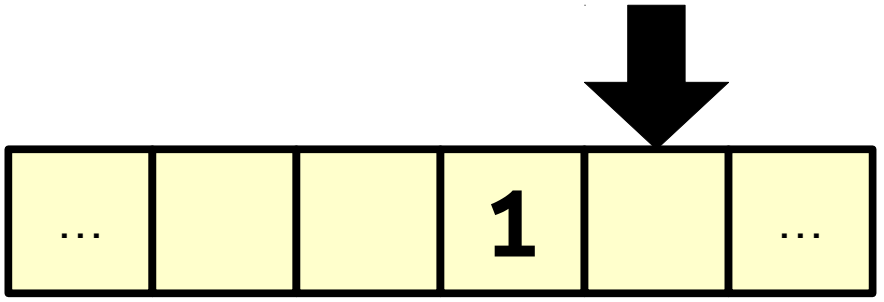


Does this TM halt on this input?

Verification

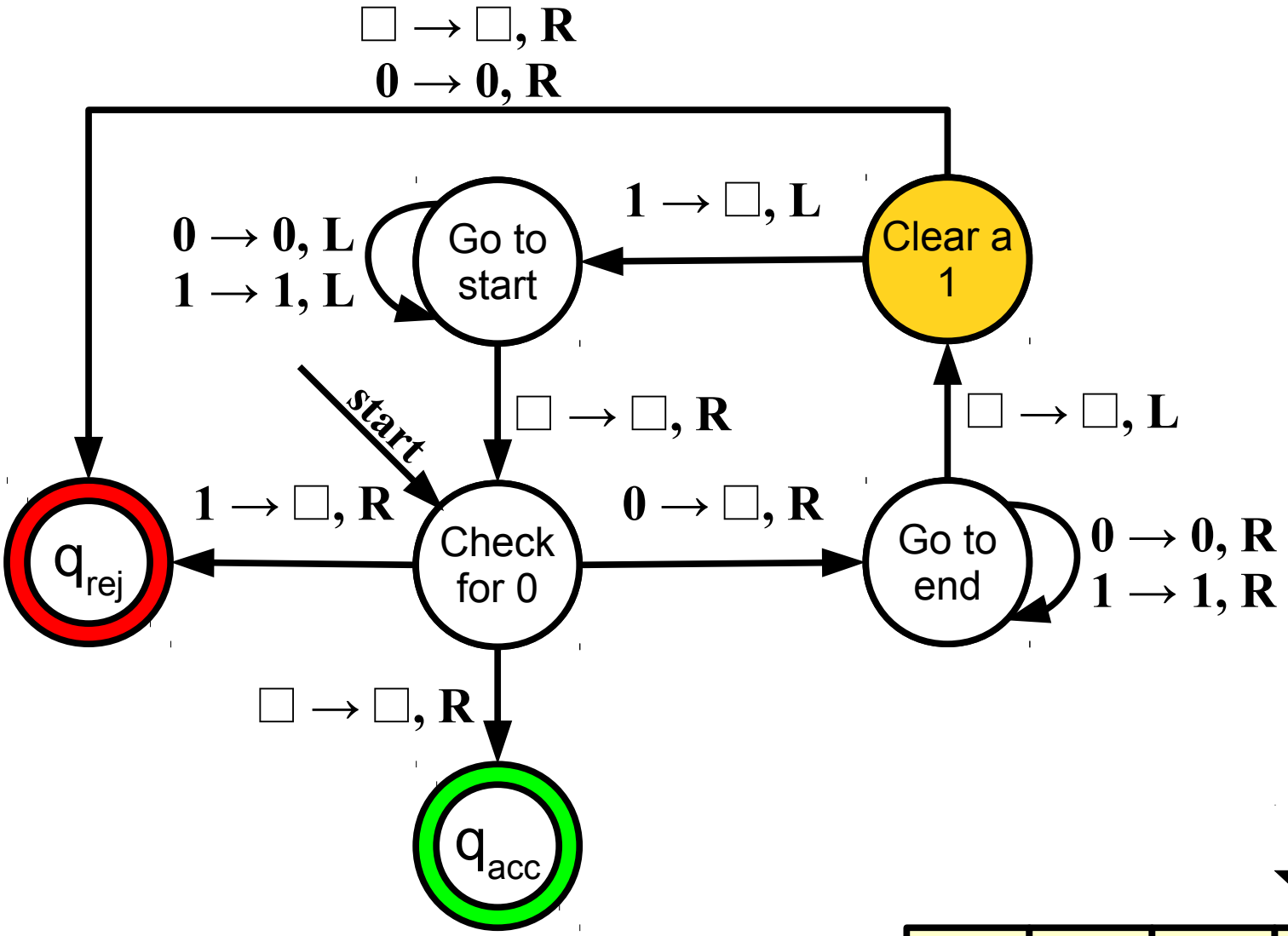


Try running it for six steps.

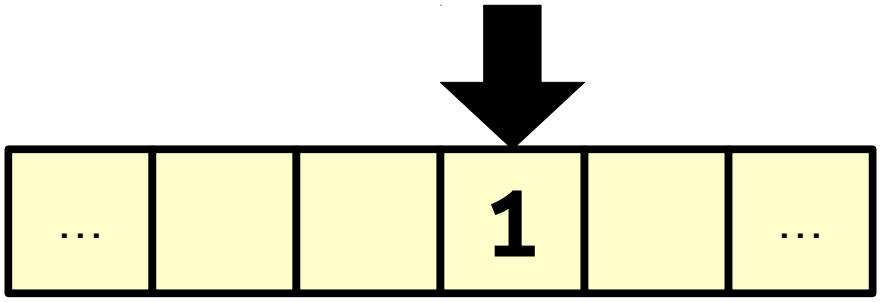


Does this TM halt on this input?

Verification

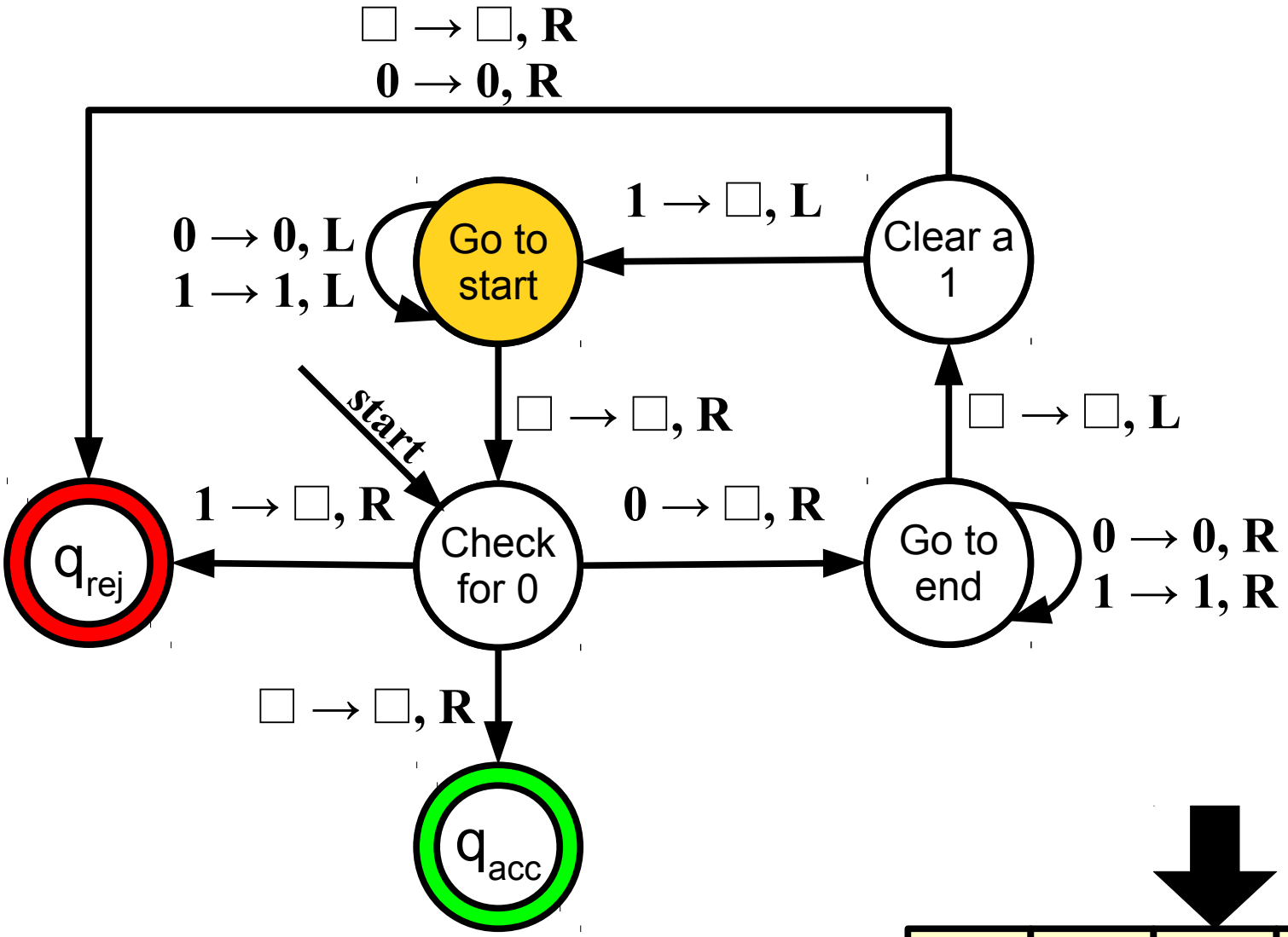


Try running it for six steps.

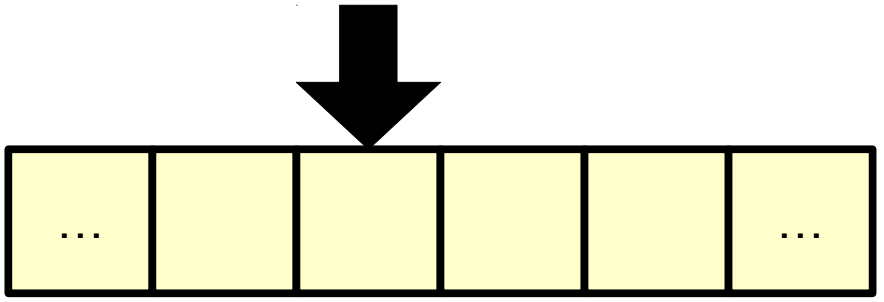


Does this TM halt on this input?

Verification

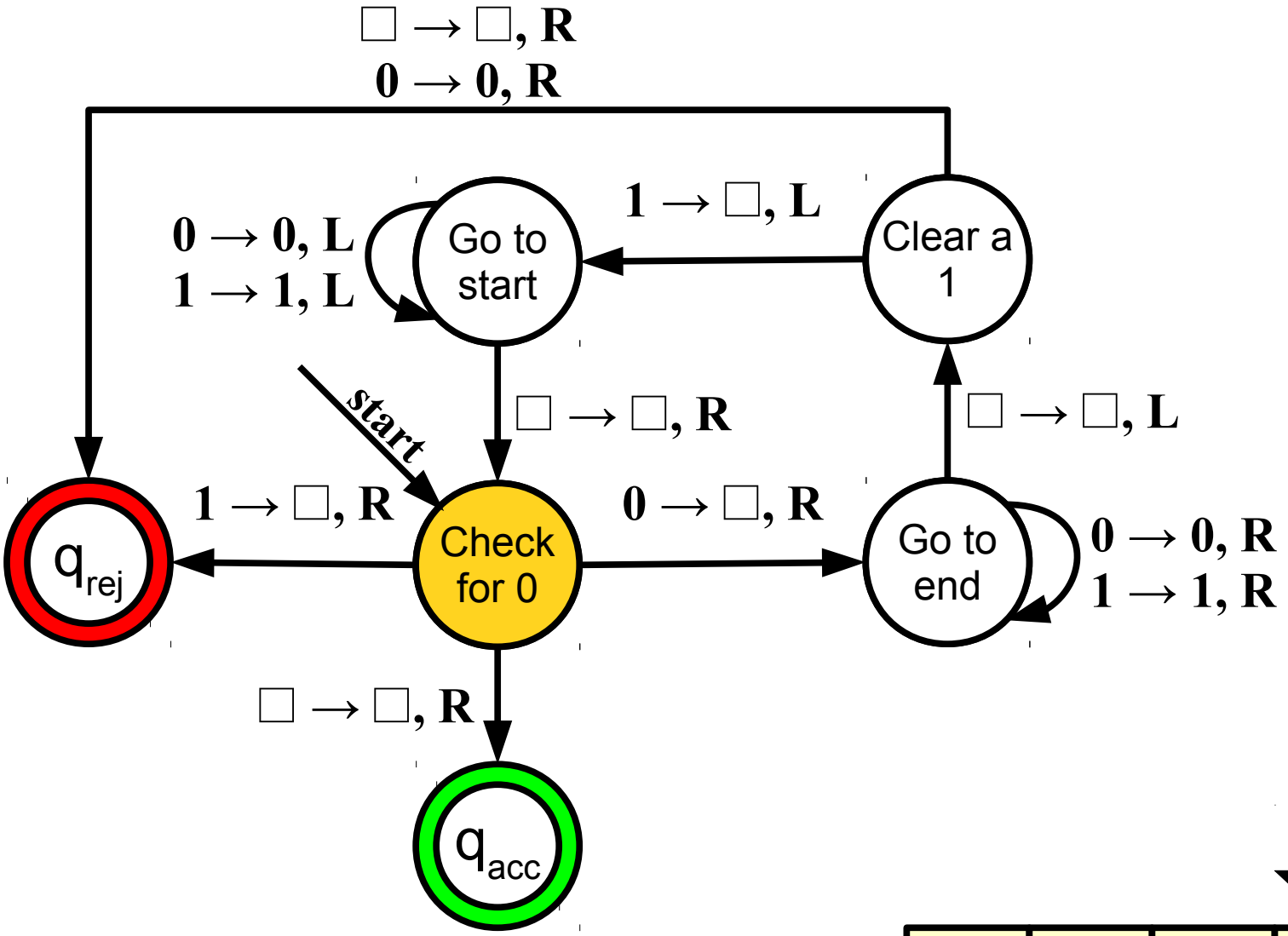


Try running it for six steps.

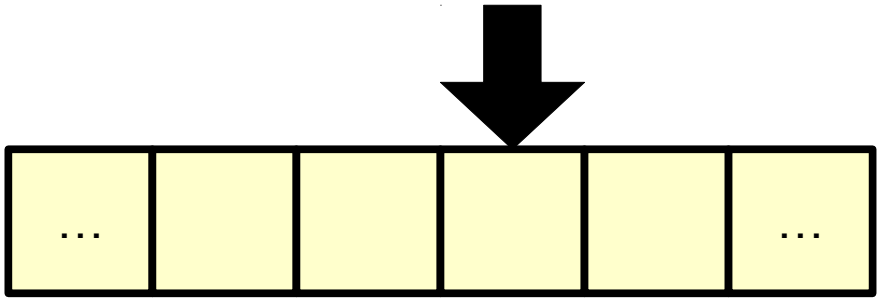


Does this TM halt on this input?

Verification

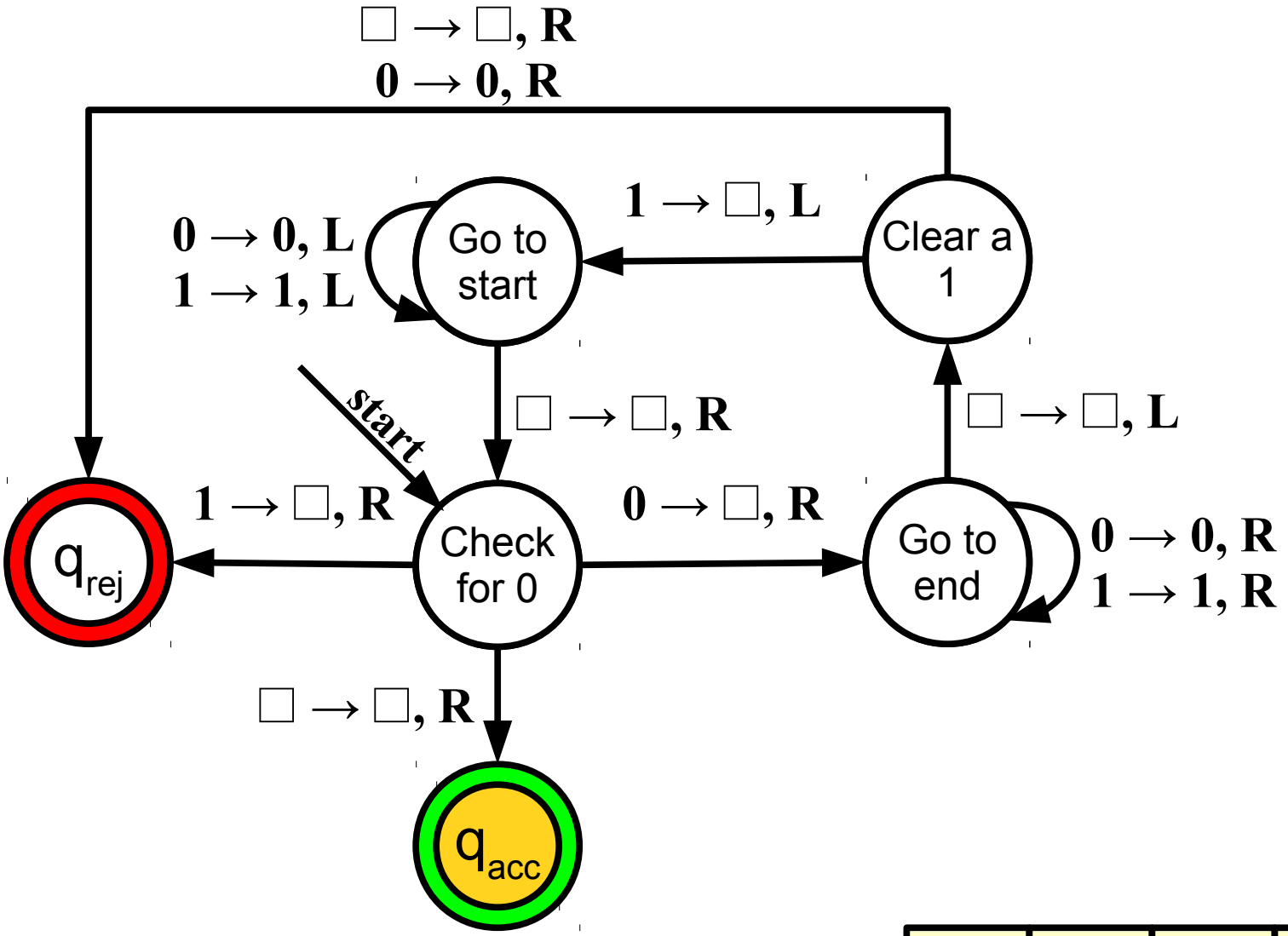


Try running it for six steps.

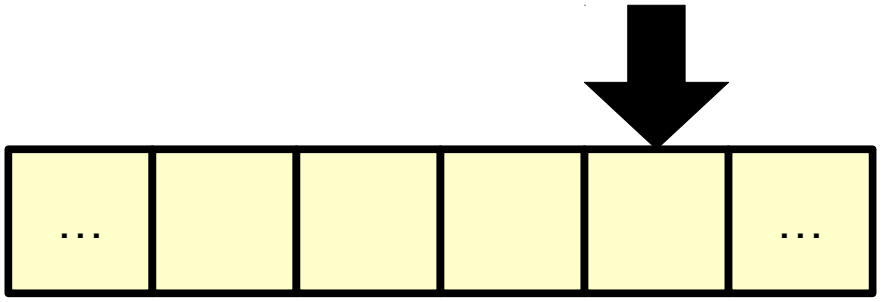


Does this TM halt on this input?

Verification



Try running it for six steps.



Does this TM halt on this input?

Verification

		7		6		1		
					3		5	2
3			1		5	9		7
6		5		3		8		9
	1						2	
8		2		1		5		4
1		3	2		7			8
5	7		4					
		4		8		7		

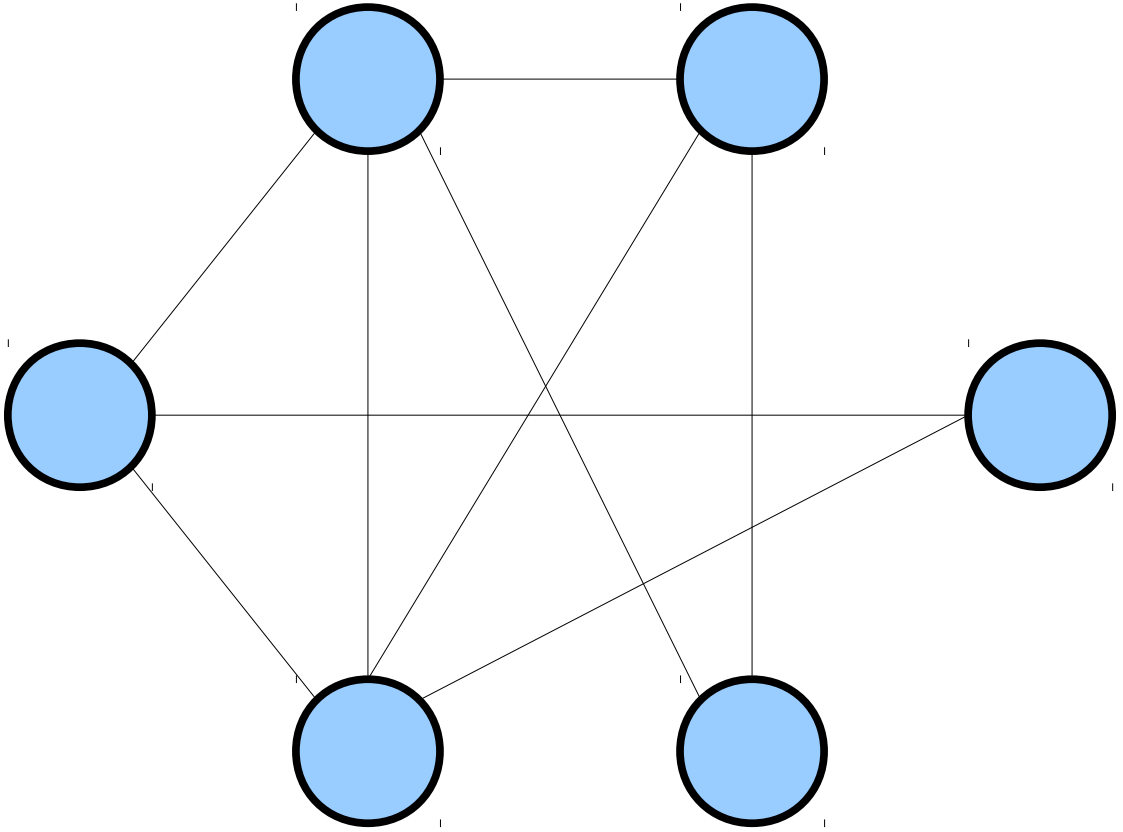
Does this Sudoku puzzle have a solution?

Verification

1	1	7	1	6	1	1	1	1
1	1	1	1	1	3	1	5	2
3	1	1	1	1	5	9	1	7
6	1	5	1	3	1	8	1	9
1	1	1	1	4	1	1	2	1
8	1	2	1	1	1	5	1	4
1	1	3	2	1	7	1	1	8
5	7	1	4	1	1	1	1	1
1	1	4	1	8	1	7	1	1

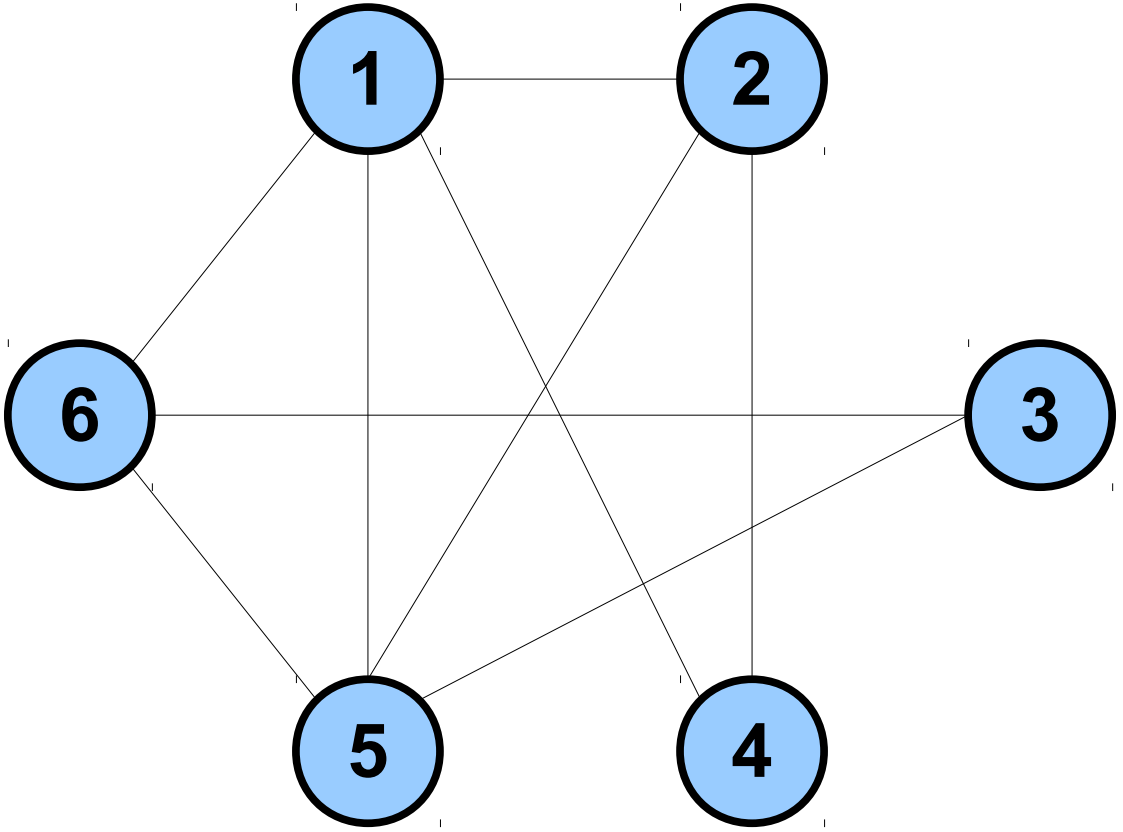
Does this Sudoku puzzle have a solution?

Verification



Is there a simple path that goes through every node exactly once?

Verification



Is there a simple path that goes through every node exactly once?

Verification

11

Does the hailstone sequence
terminate for this number?

Verification

11

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

34

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

17

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

52

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

26

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

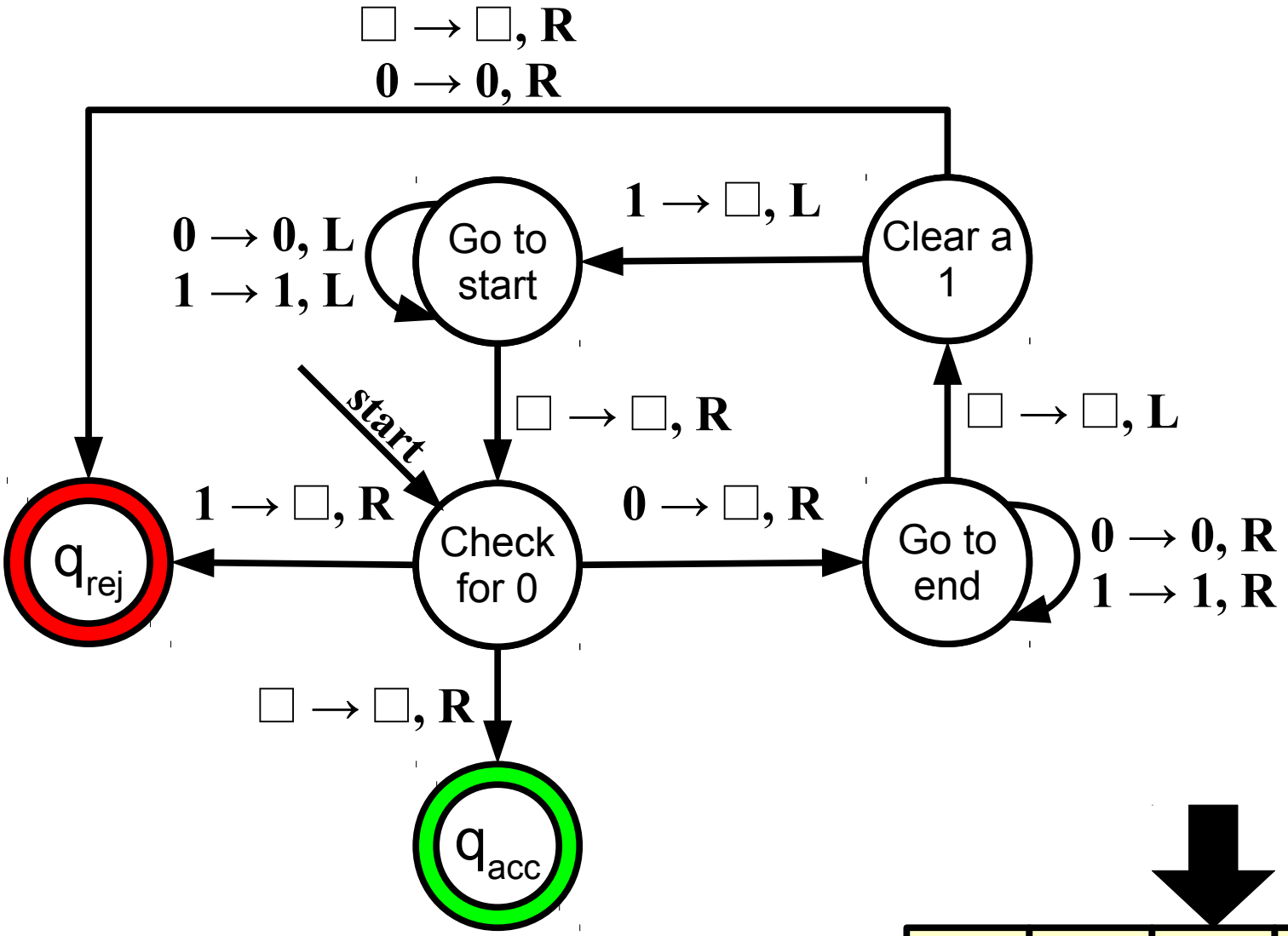
Verification

13

Try running five steps of the Hailstone sequence.

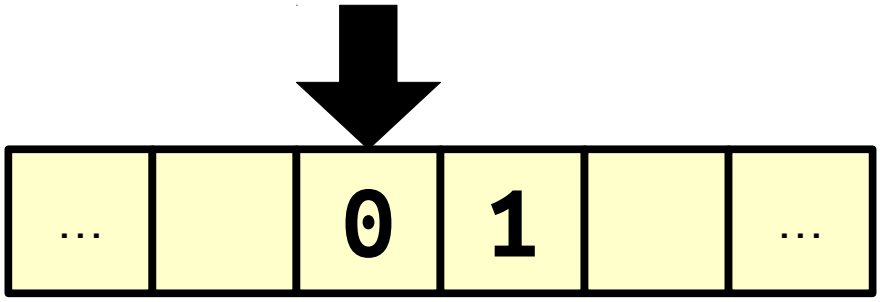
Does the hailstone sequence
terminate for this number?

Verification

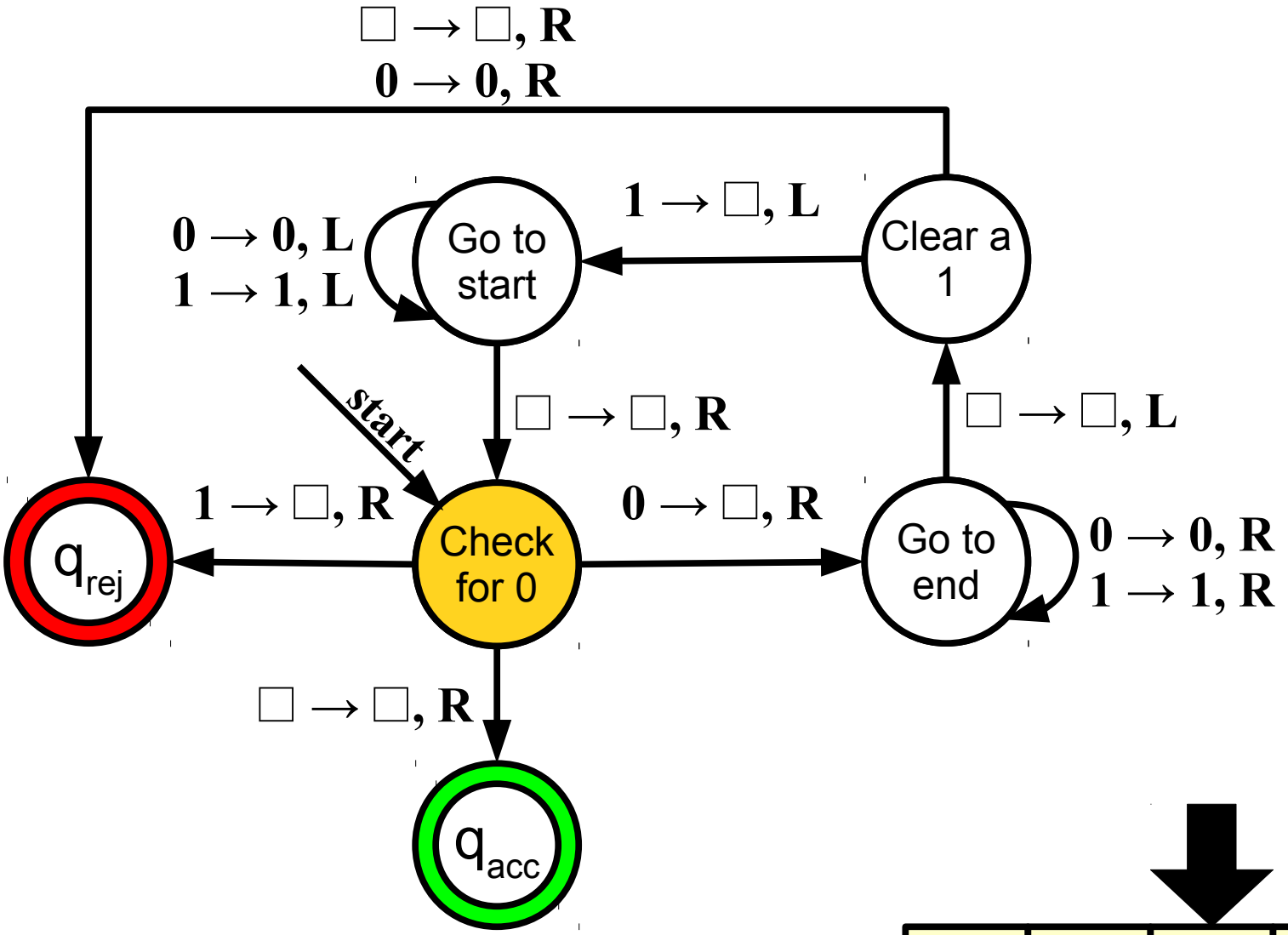


Try running it for four steps.

Does this TM halt on this input?

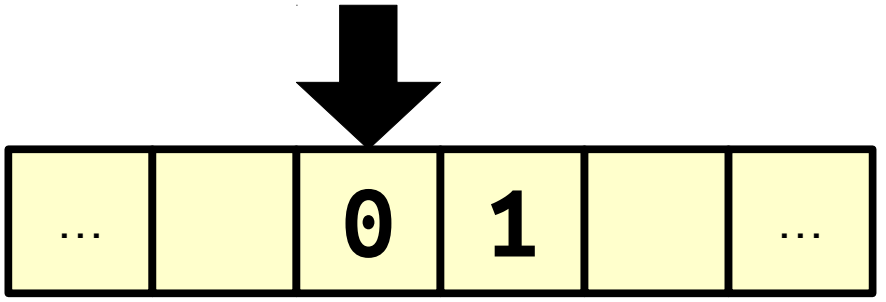


Verification

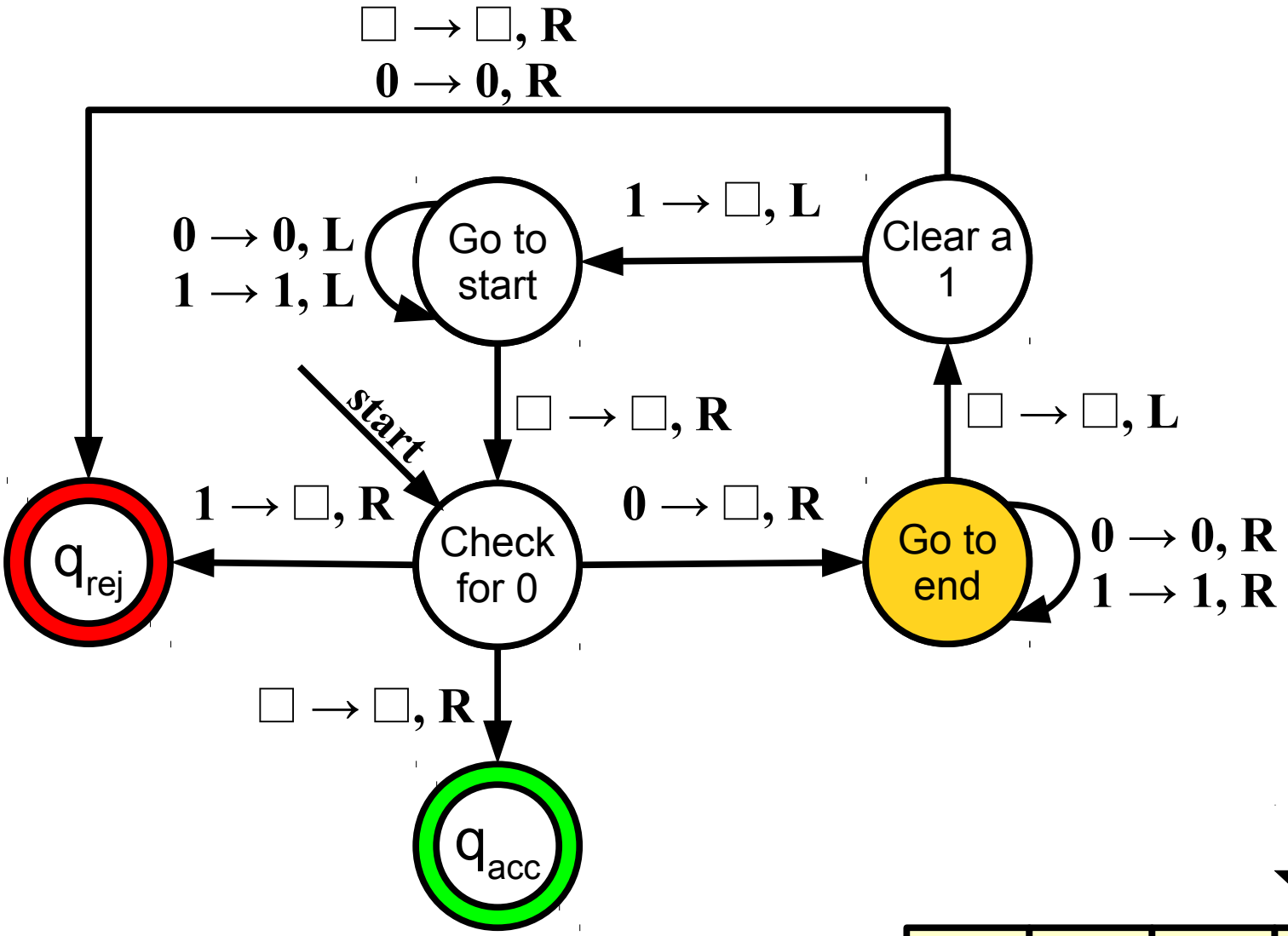


Try running it for four steps.

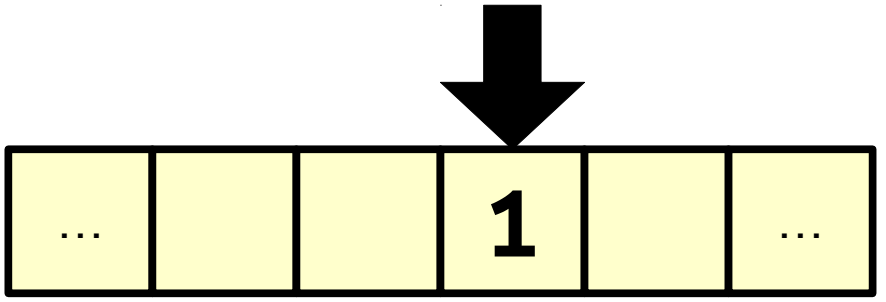
Does this TM halt on this input?



Verification

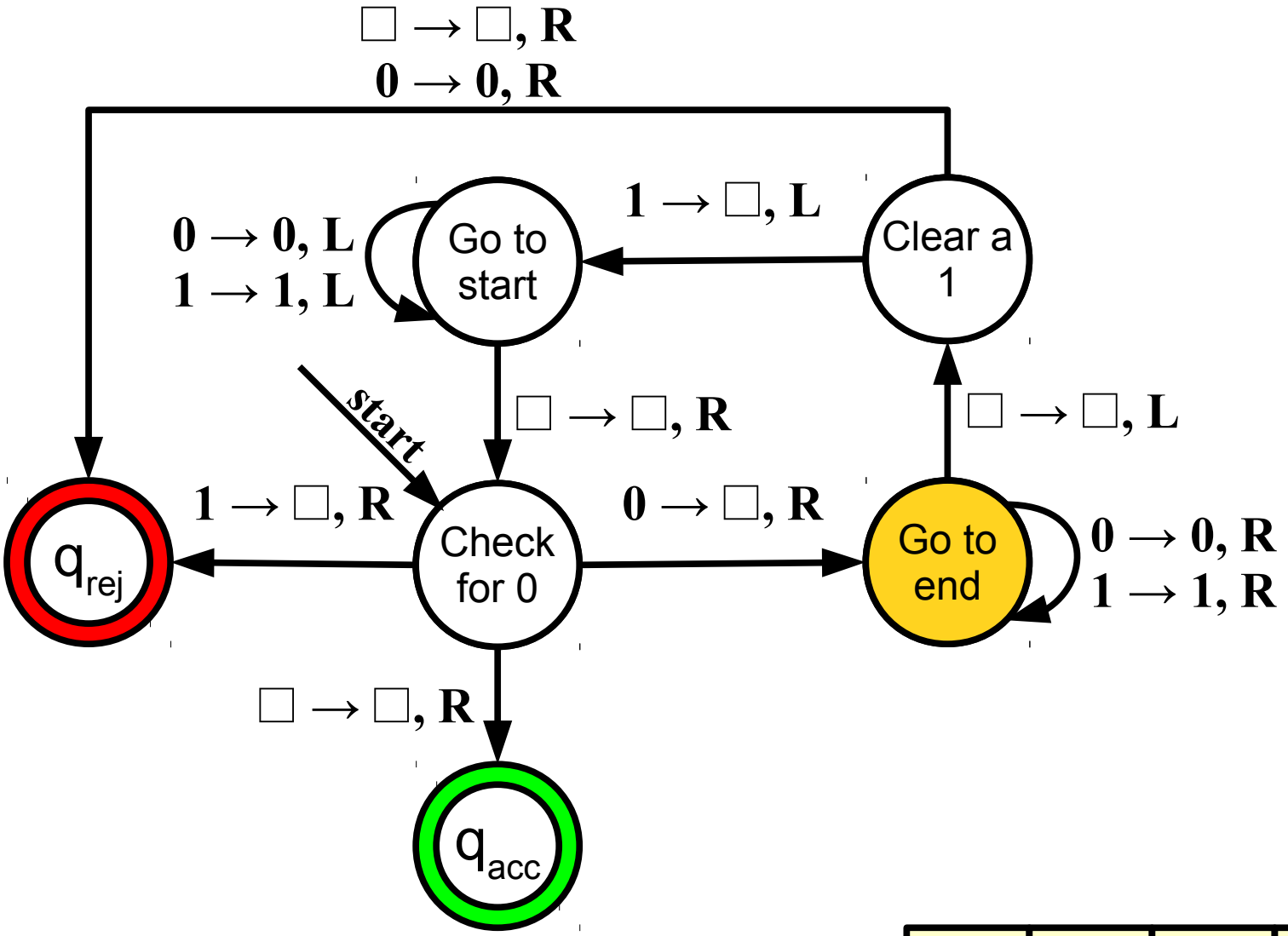


Try running it for four steps.

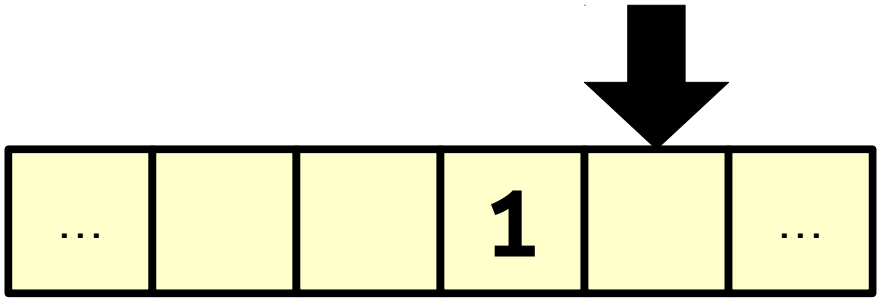


Does this TM halt on this input?

Verification

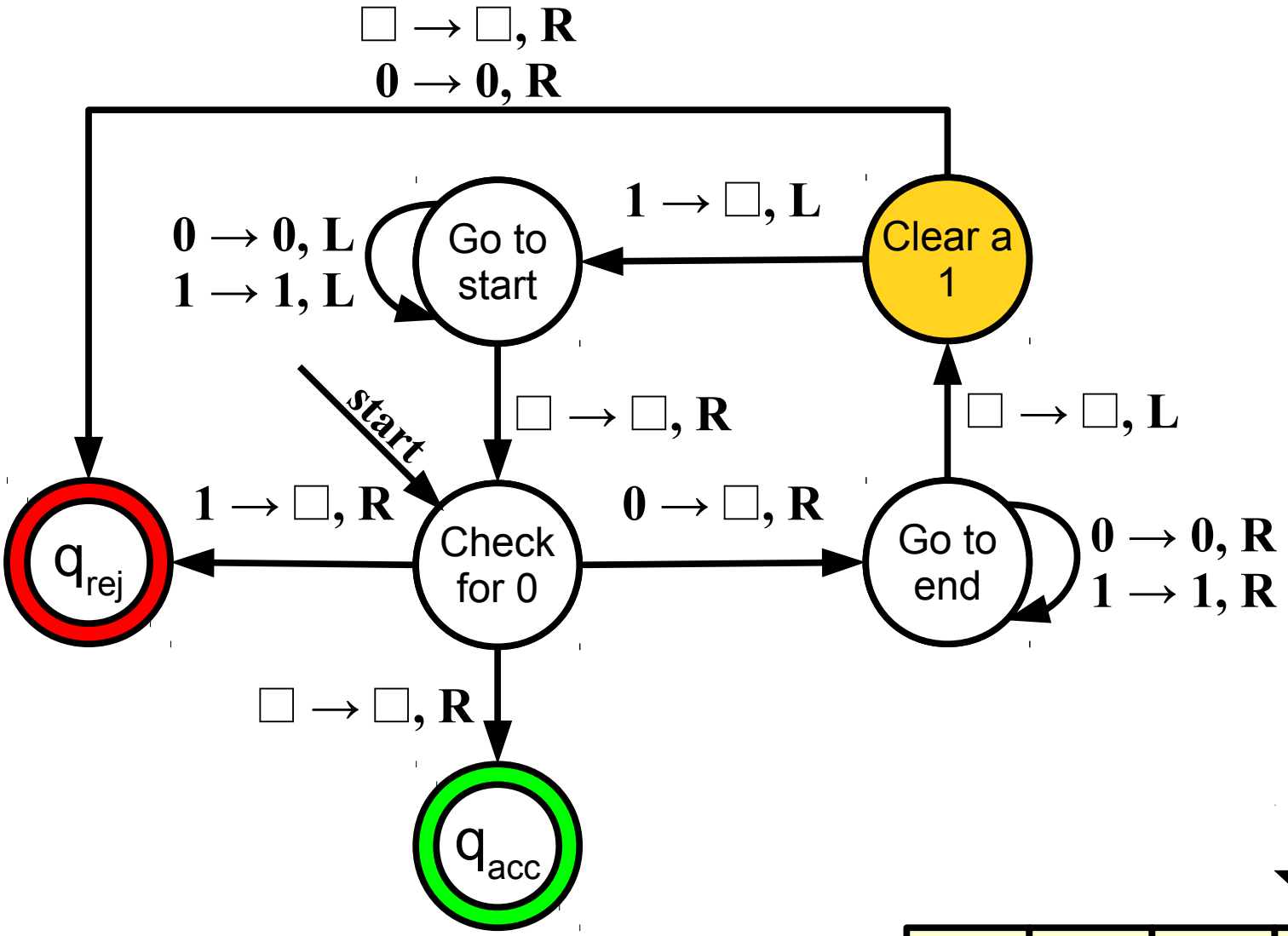


Try running it for four steps.

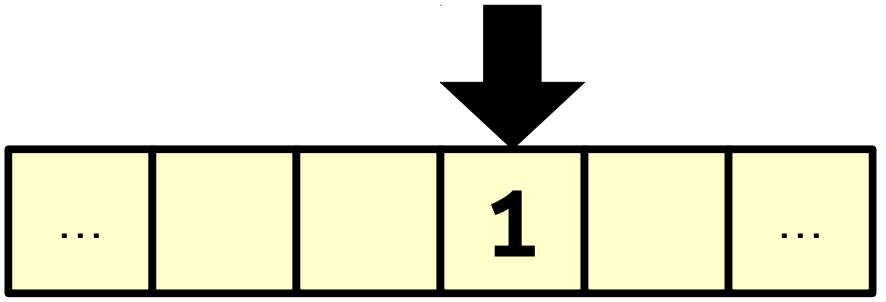


Does this TM halt on this input?

Verification

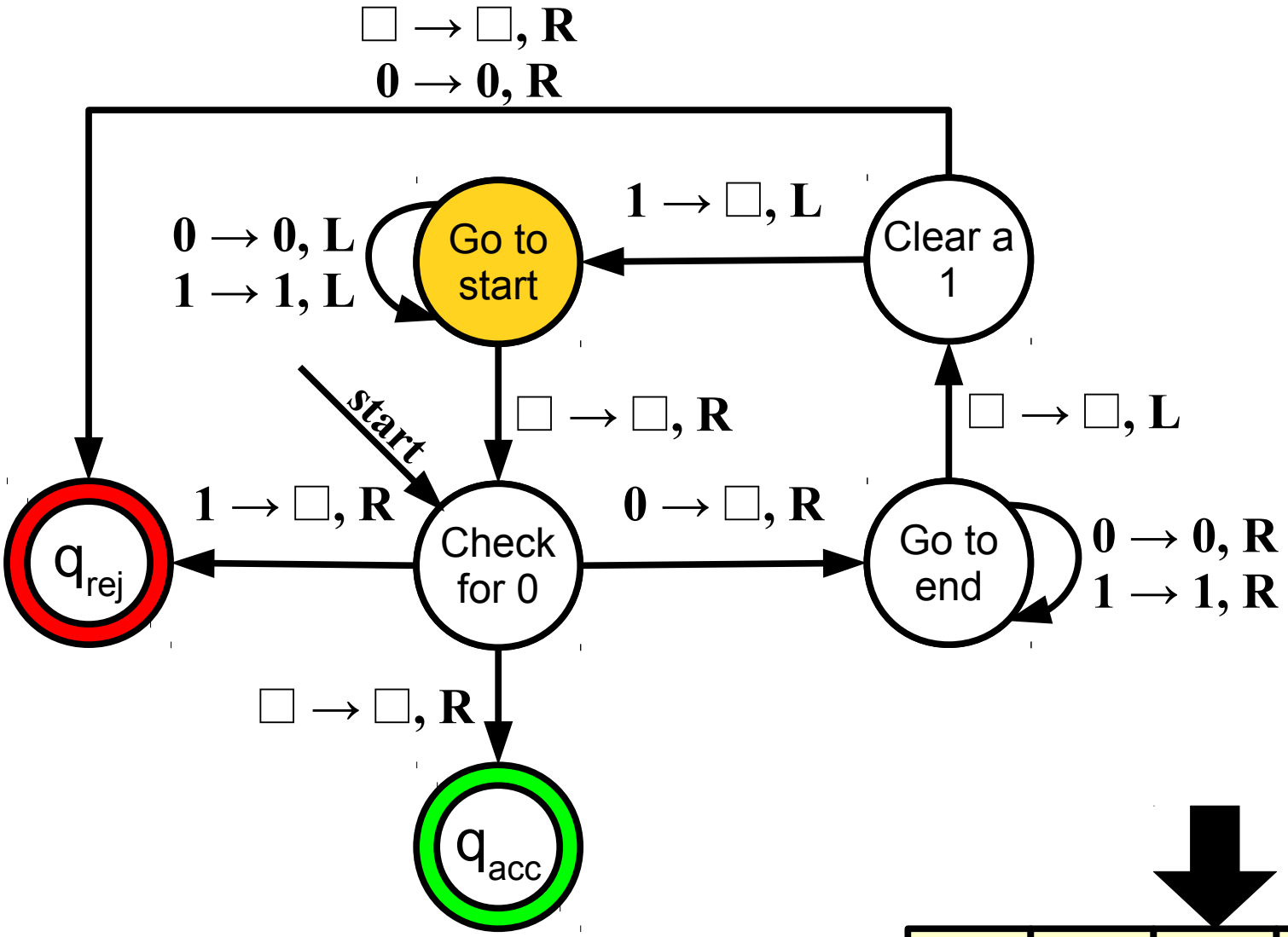


Try running it for four steps.

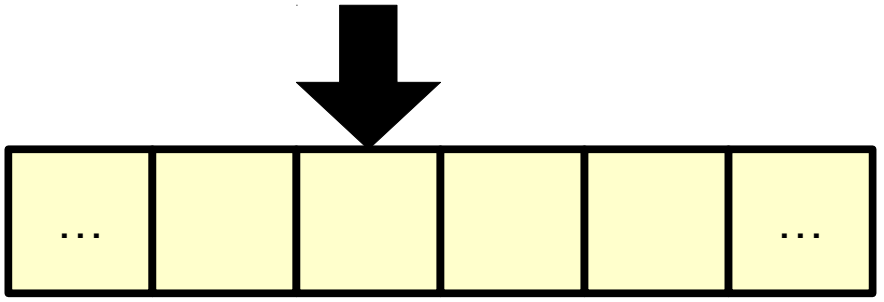


Does this TM halt on this input?

Verification

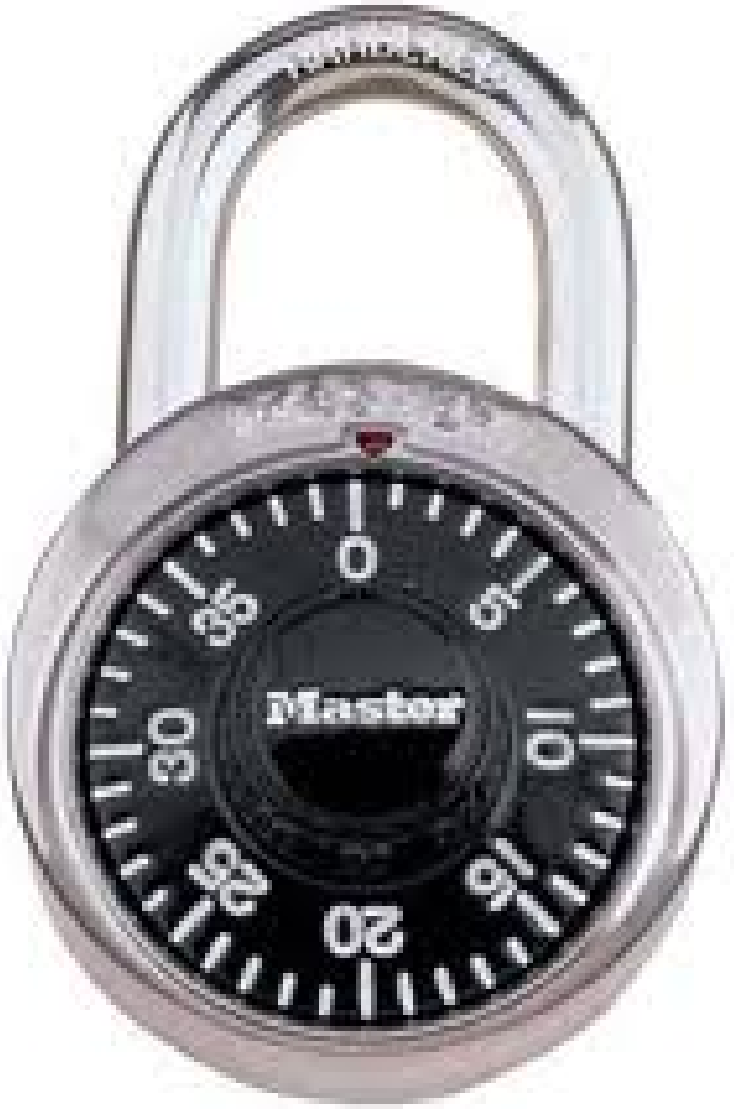


Try running it for four steps.



Does this TM halt on this input?

Intuiting Verifiers



Question:
Can this lock be opened?

What Just Happened?

- In each of the preceding cases, we were given some problem and some evidence supporting the claim that the answer is “yes.”
- Given the correct evidence, we can be certain that the answer is indeed “yes.”
- Given incorrect evidence, we aren't sure whether the answer is “yes.”
  - Maybe there's *no* evidence saying that the answer is “yes,” or maybe there is some evidence, but just not the evidence we were given.
- Let's formalize this idea.

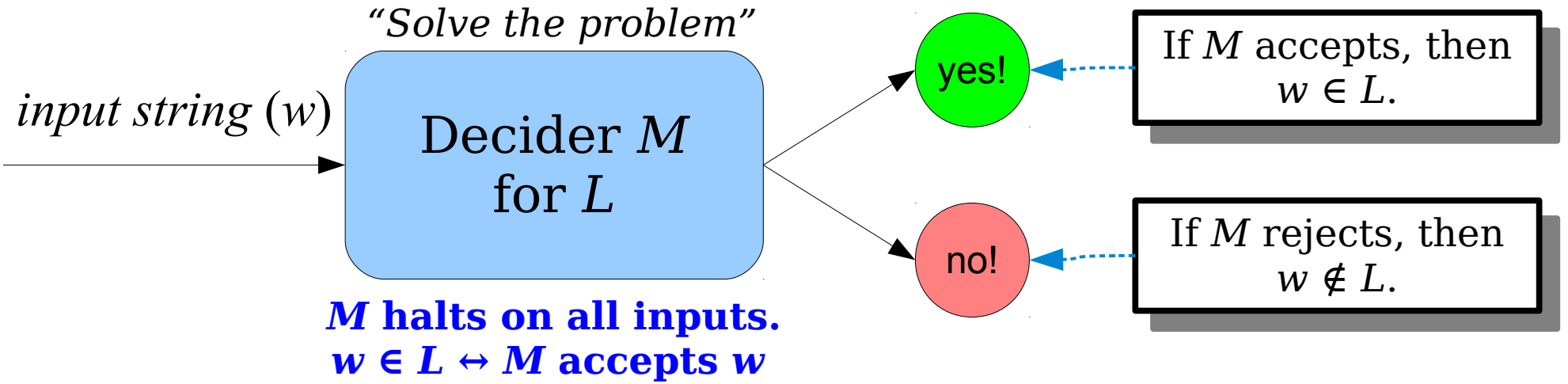
Verifiers

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

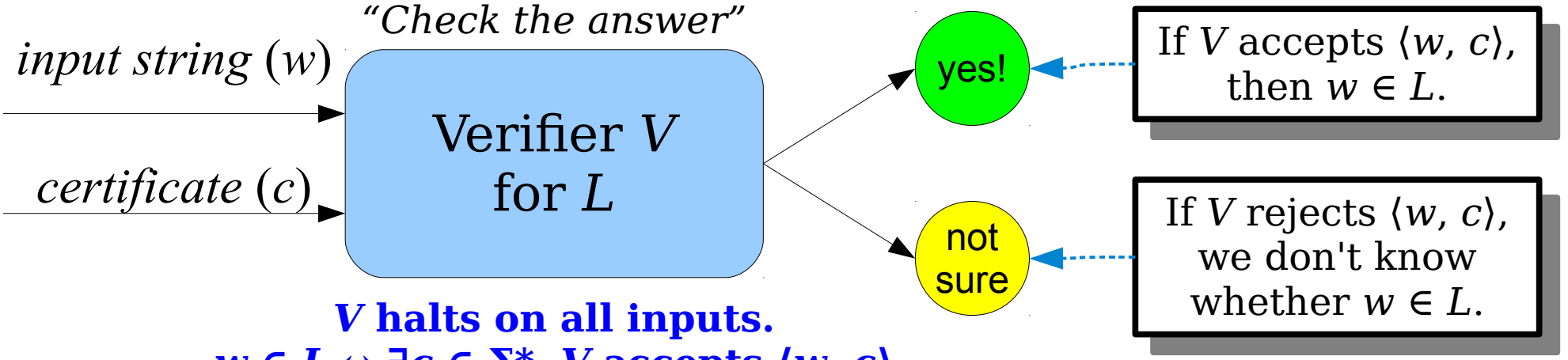
$$\boldsymbol{w \in L \leftrightarrow \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

- Intuitively, what does this mean?

Deciders and Verifiers



***M* halts on all inputs.**
w* ∈ *L* ↔ *M* accepts *w



***V* halts on all inputs.**
***w* ∈ *L* ↔ ∃*c* ∈ Σ*. *V* accepts ⟨*w*, *c*⟩**

Verifiers

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

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

- 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 halts on all inputs.
  - For any string $w \in \Sigma^*$, the following is true:

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

- Some notes about V:
  - Notice that c is existentially quantified. Any string $w \in L$ must have at least one c that causes V to accept, and possibly more.
  - V is required to halt, so given any potential certificate c for w, you can check whether the certificate is correct.

Verifiers

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

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

- Some notes about V:
  - Notice that $\mathscr{L}(V) \neq L$. *(Good question: what <u>is</u> $\mathscr{L}(V)$?)*
  - The job of V is just to check certificates, not to decide membership in L.

Some Verifiers

- Let L be the following language:

$$L = \{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}$$

- Let's see how to build a verifier for L.

Verification

11

Does the hailstone sequence
terminate for this number?

Verification

11

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

34

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

17

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

52

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

26

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

13

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

40

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

20

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

10

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

5

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

16

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

8

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

4

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

2

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

1

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Some Verifiers

- Let L be the following language:

$$L = \{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}$$

V = "On input $\langle n, k \rangle$, where $n, k \in \mathbb{N}$.
Check that $n \neq 0$.
Run the hailstone sequence, starting at n, for at most k steps.
If after k steps we reach 1, accept.
Otherwise, reject."

- Do you see why $\langle n \rangle \in L$ iff there is some k such that V accepts $\langle n, k \rangle$?
- Do you see why V always halts?

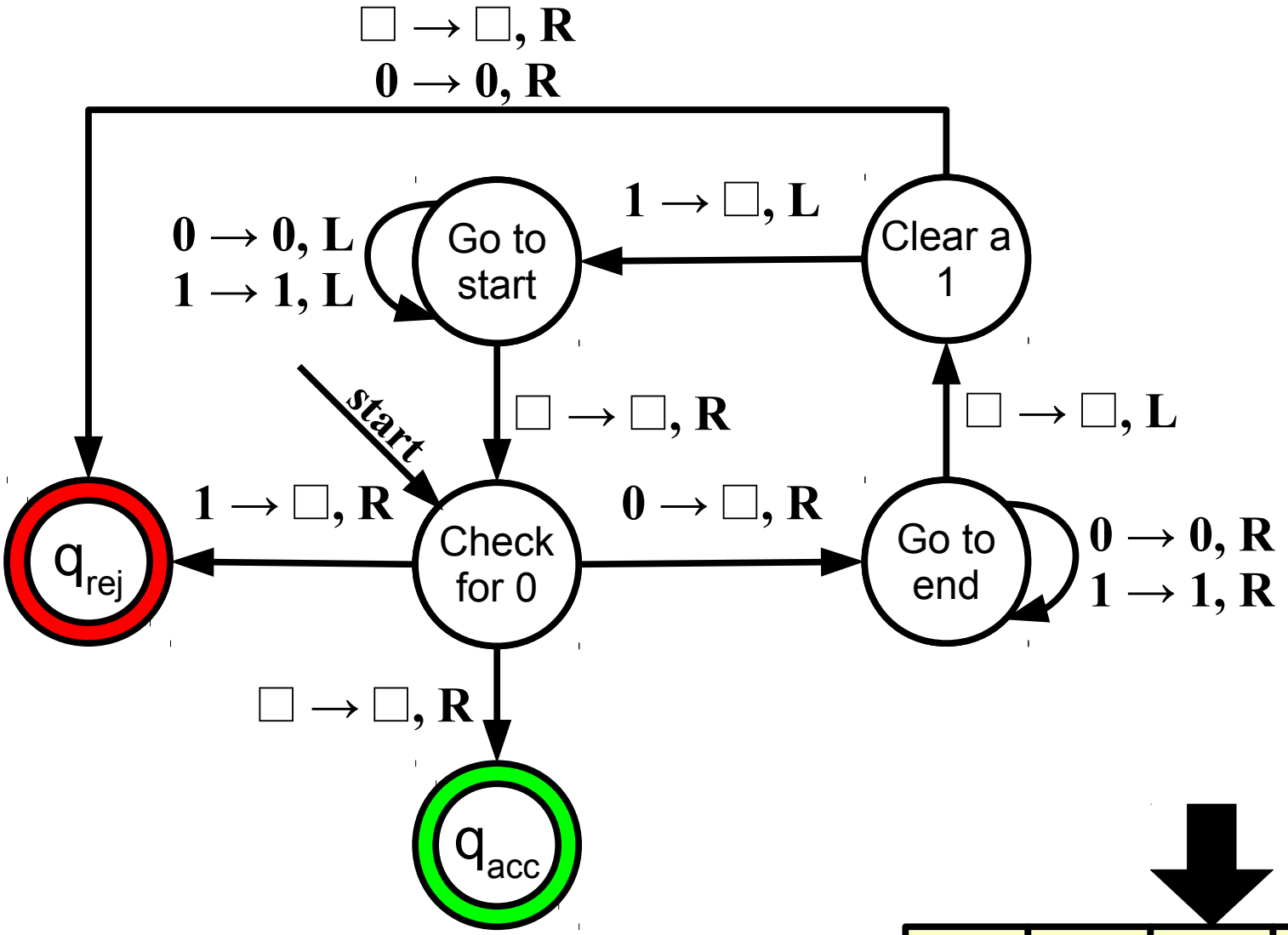
Some Verifiers

- Consider *HALT*:

  $HALT = \{ \langle M, w \rangle \mid M \text{ is a TM that halts on } w \}$

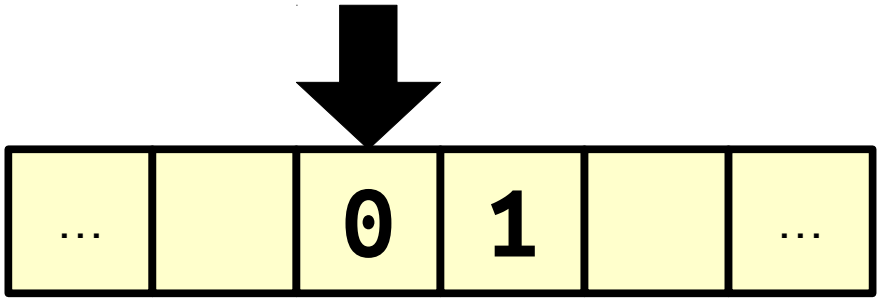
- Let's see how to build a verifier for *HALT*.

Verification

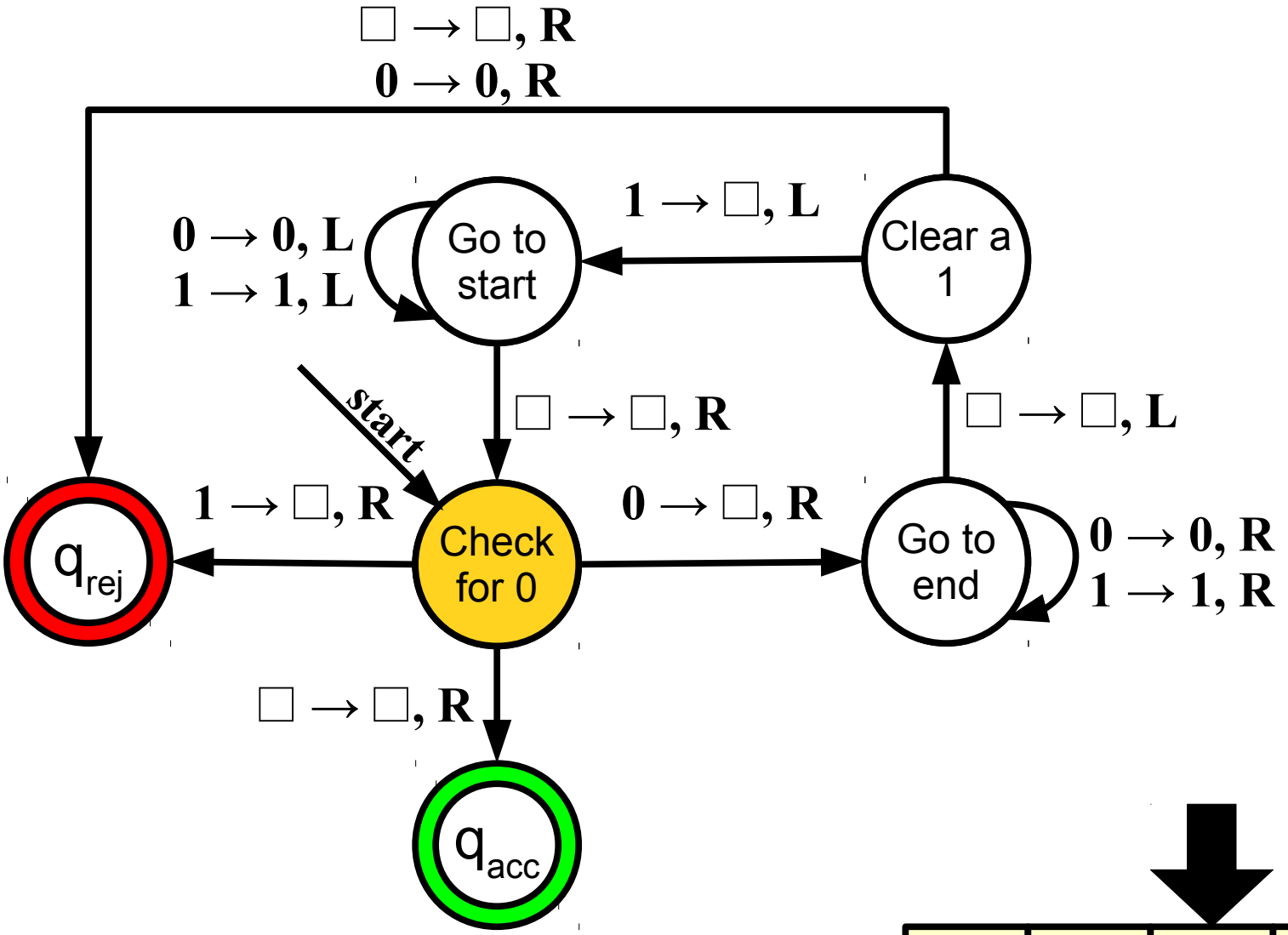


Try running it for six steps.

Does this TM halt on this input?

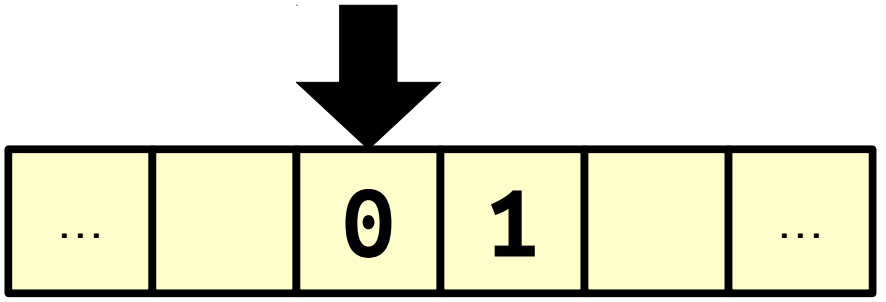


Verification

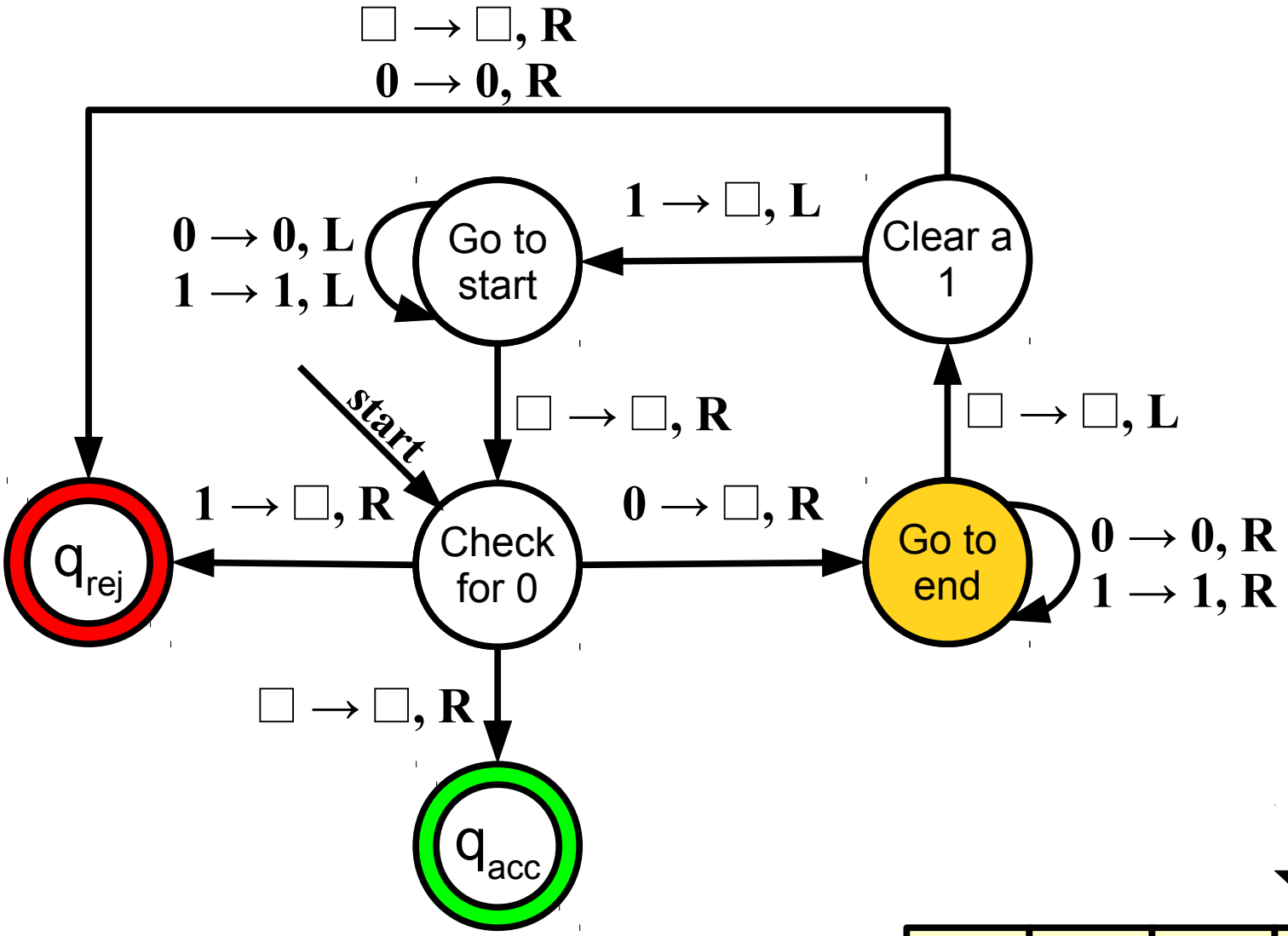


Try running it for six steps.

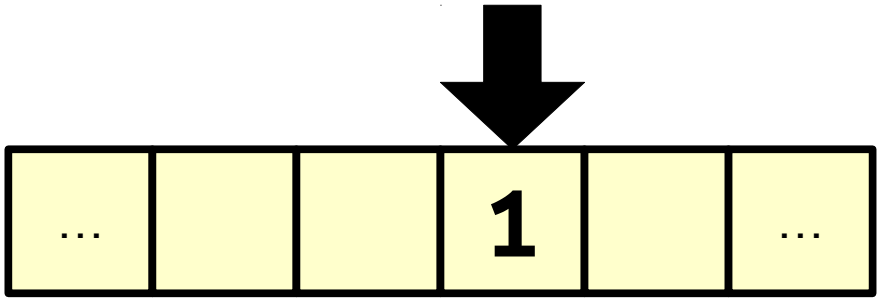
Does this TM halt on this input?



Verification

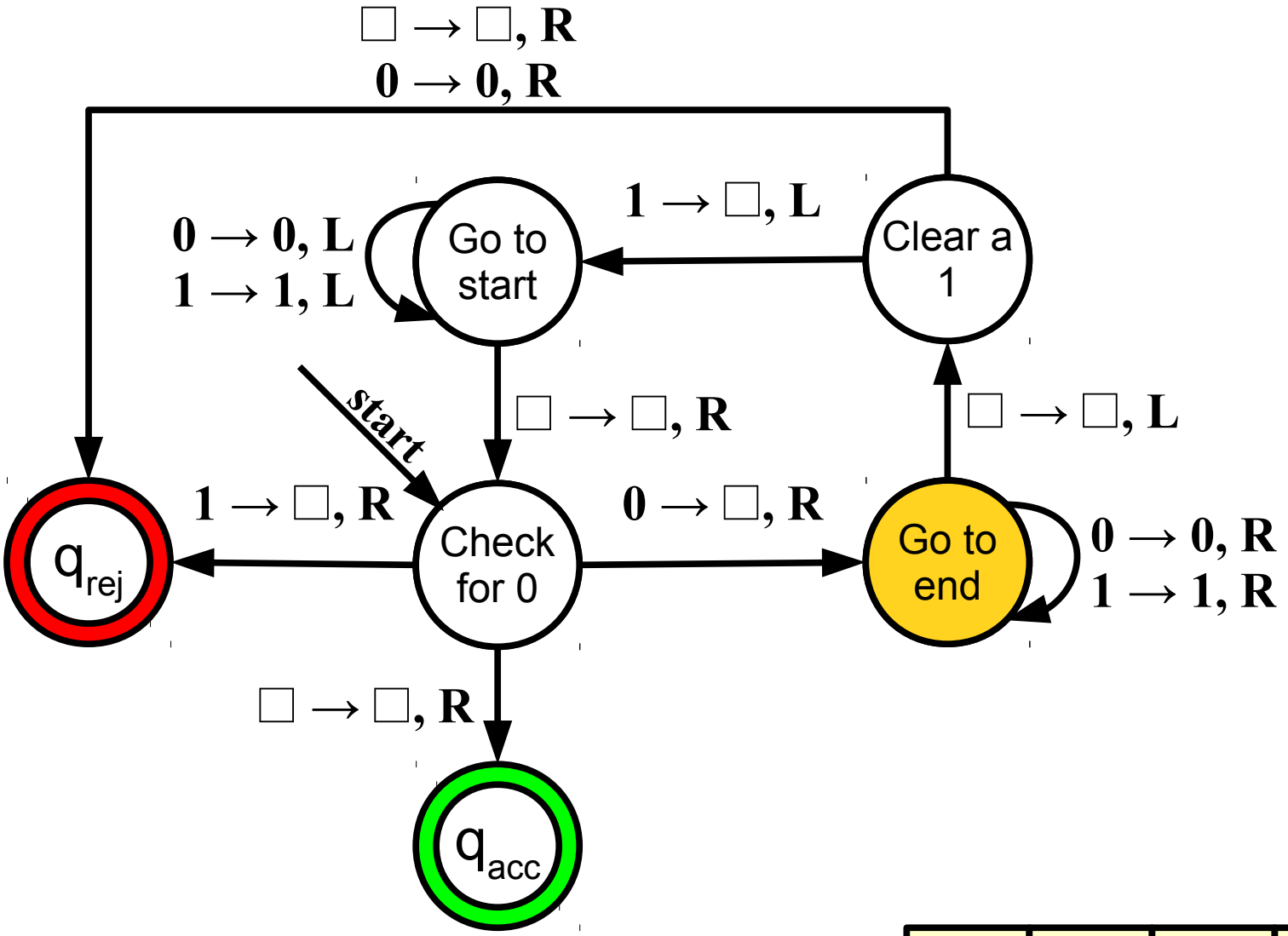


Try running it for six steps.

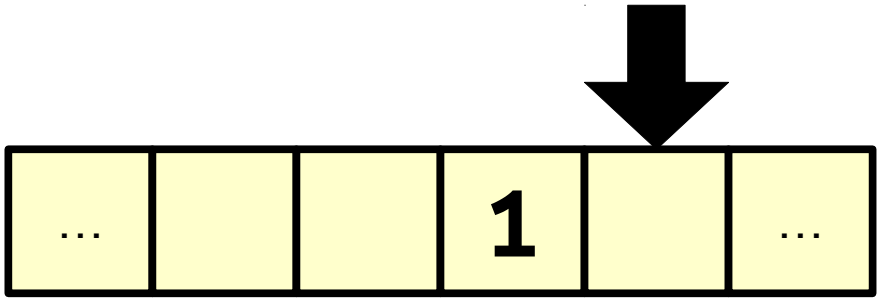


Does this TM halt on this input?

Verification

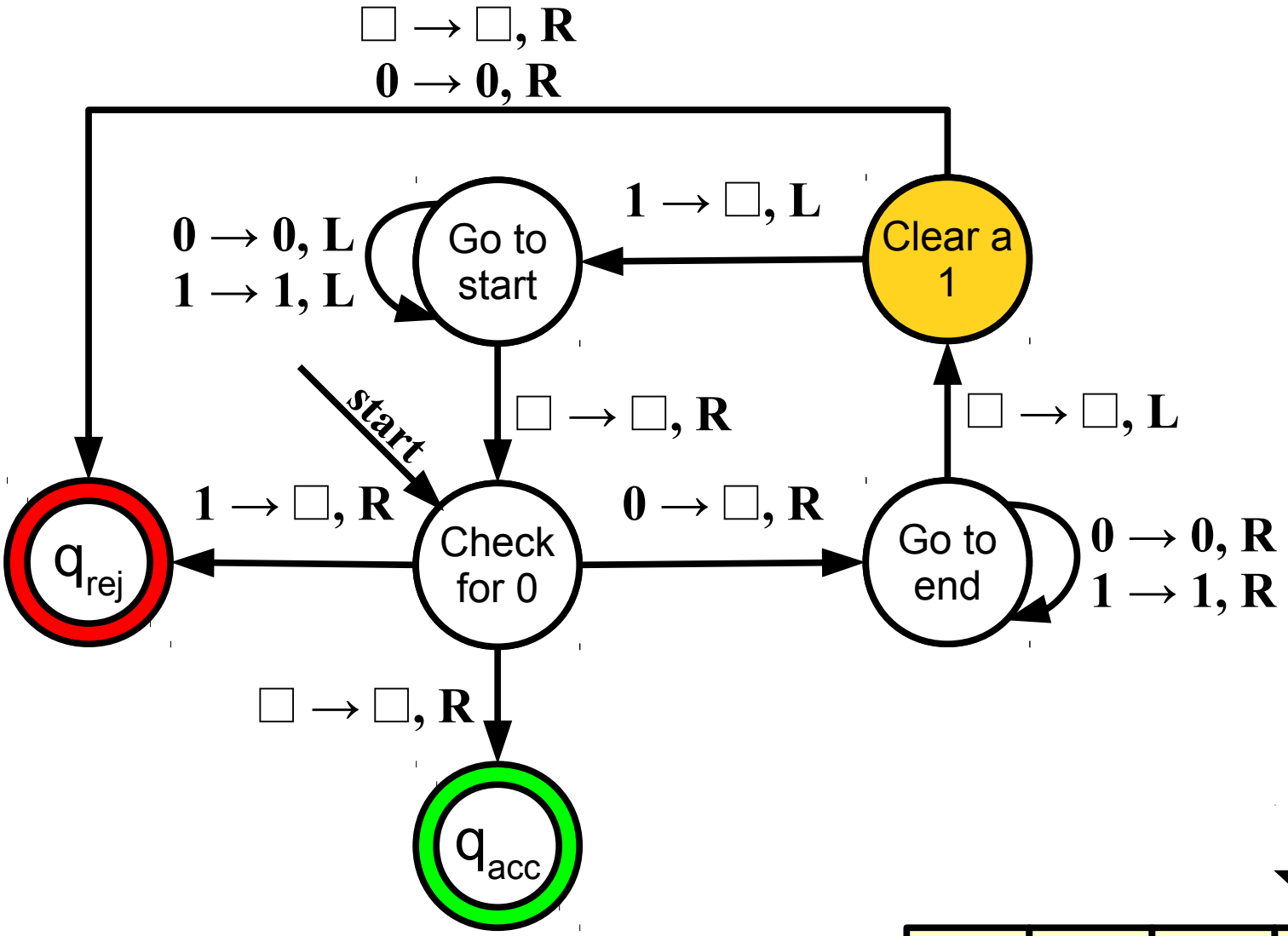


Try running it for six steps.

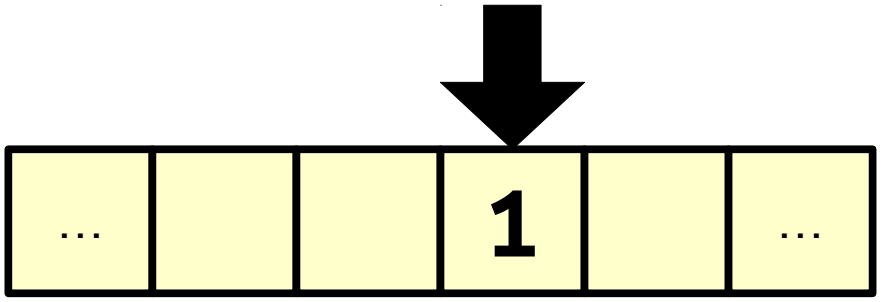


Does this TM halt on this input?

Verification

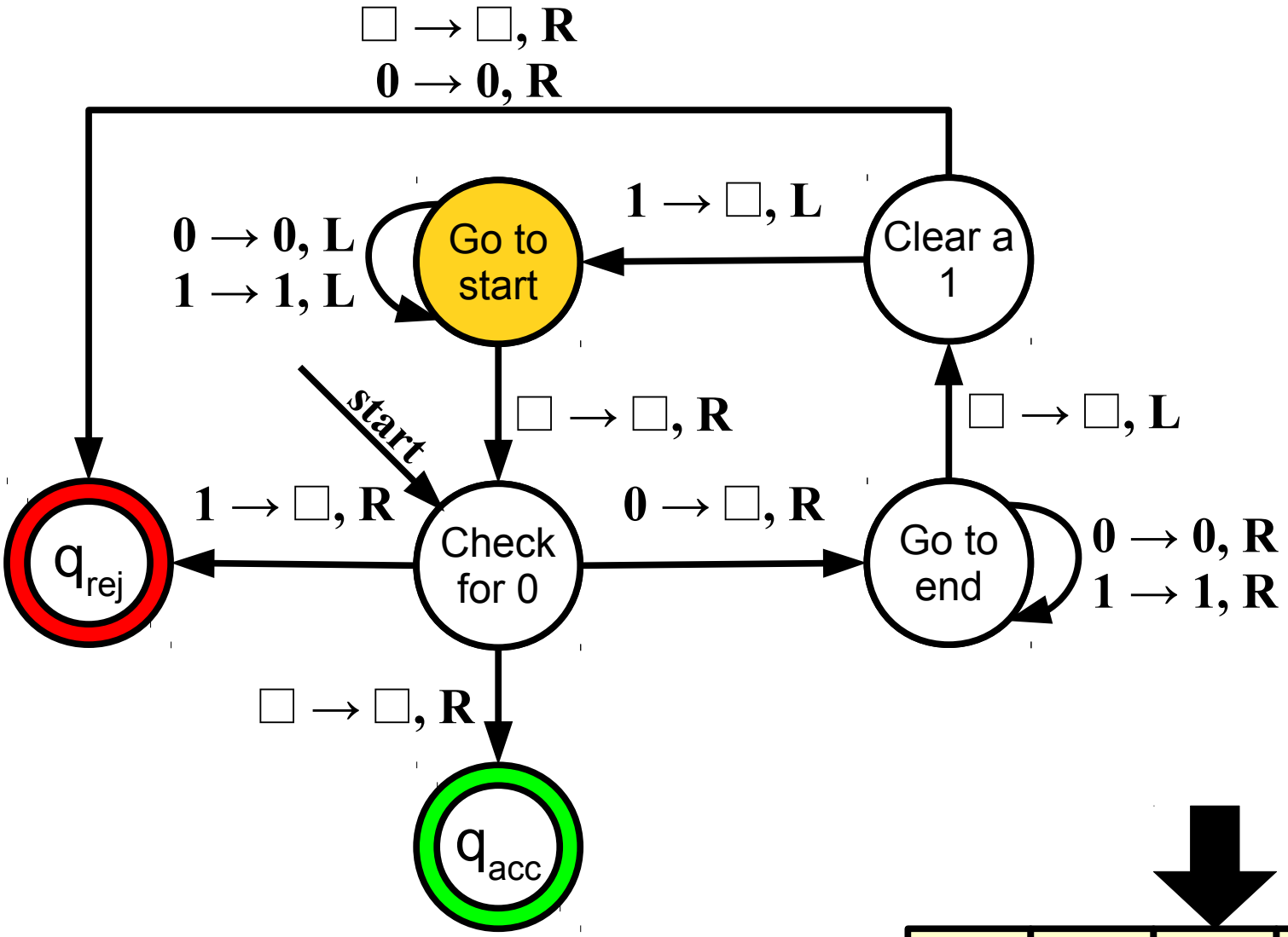


Try running it for six steps.

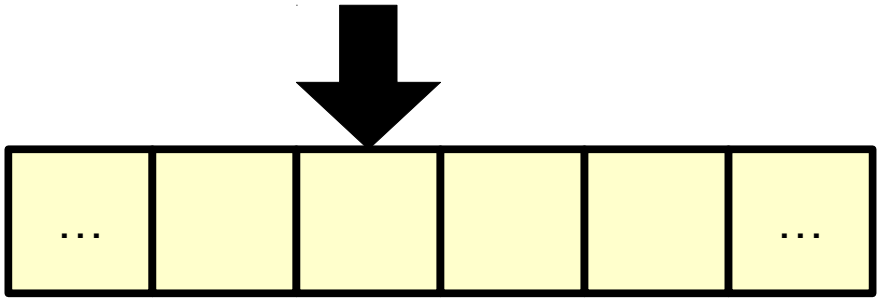


Does this TM halt on this input?

Verification

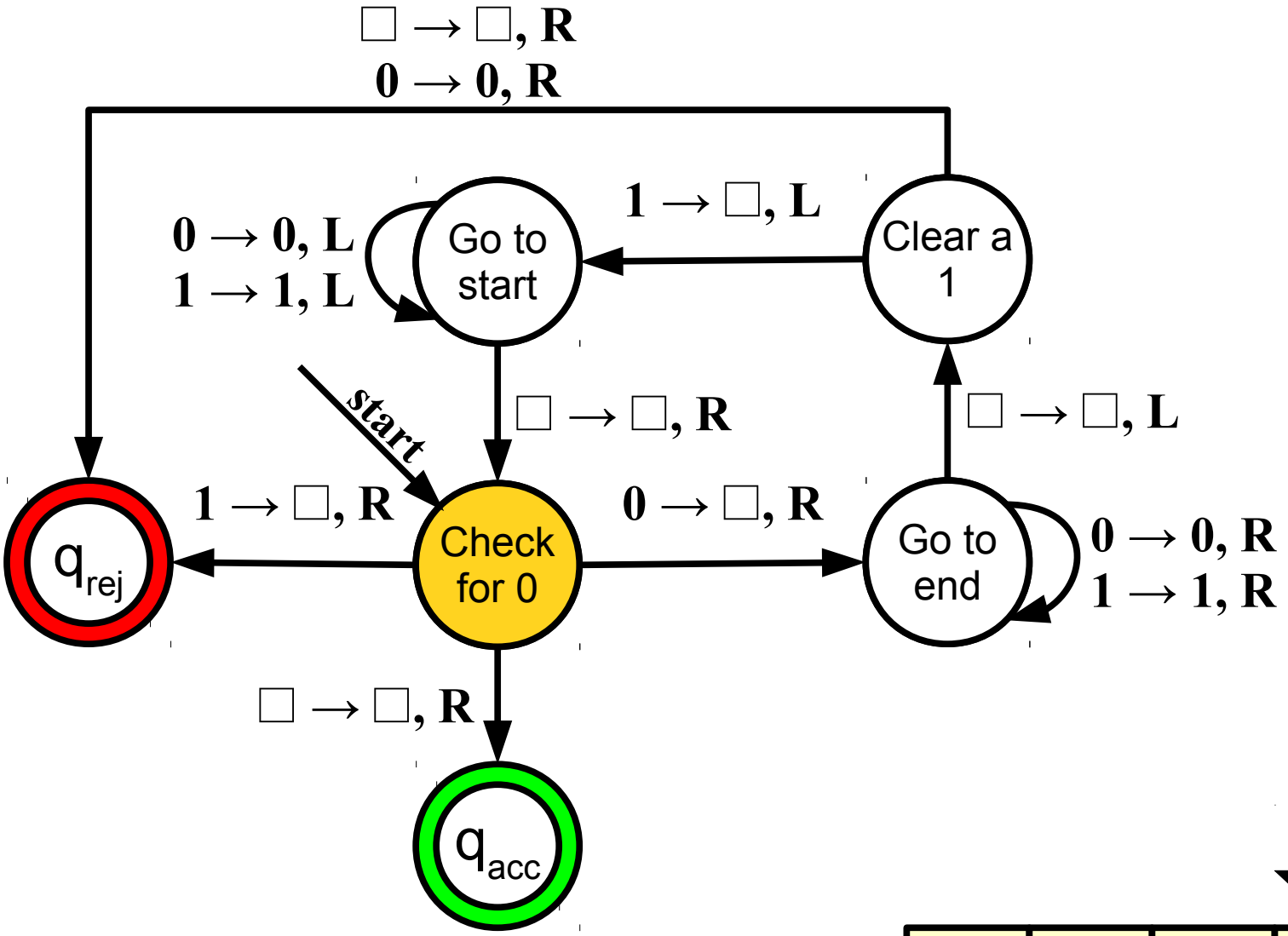


Try running it for six steps.

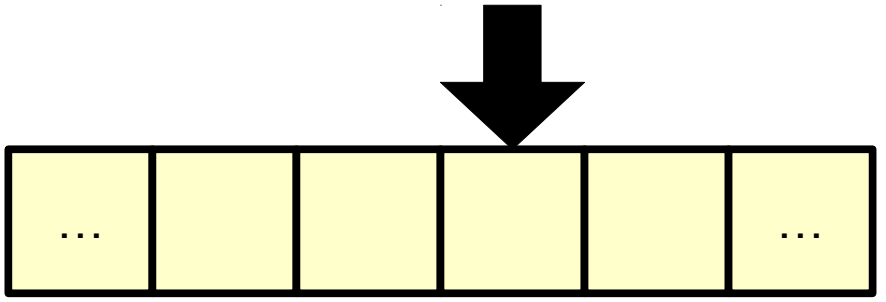


Does this TM halt on this input?

Verification

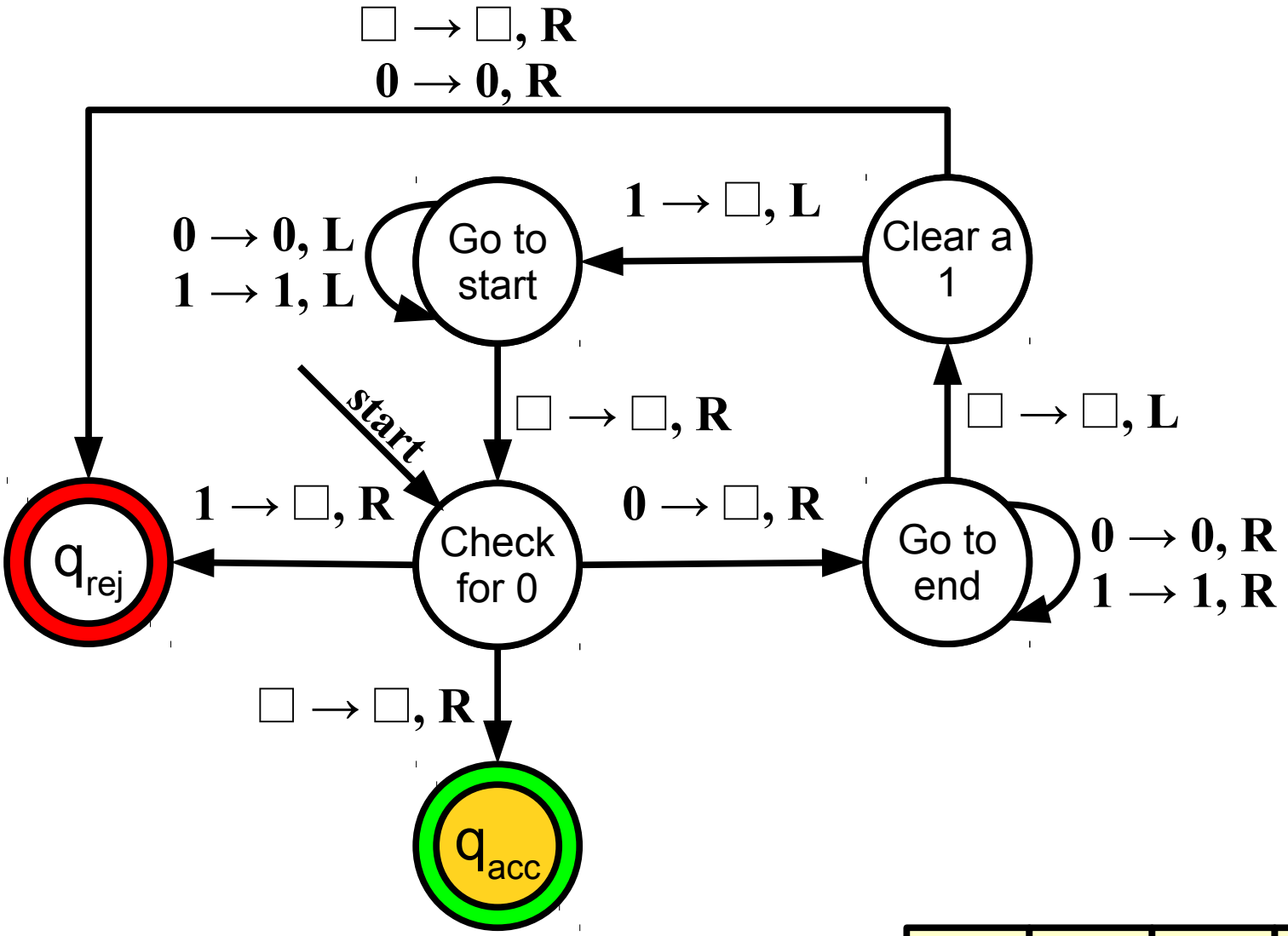


Try running it for six steps.

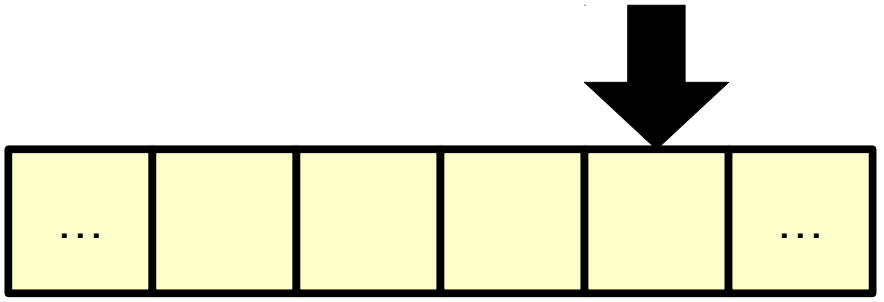


Does this TM halt on this input?

Verification



Try running it for six steps.



Does this TM halt on this input?

Some Verifiers

- Consider *HALT*:

  $HALT = \{ \langle M, w \rangle \mid M \text{ is a TM that halts on } w \}$

> V = "On input $\langle M, w, k \rangle$, where M is a TM, w is a string, and $k \in \mathbb{N}$:
> Run M on w for k steps.
> If M halts in this time, accept.
> Otherwise, reject.

- Do you see why M halts on w iff there is some k such that V accepts $\langle M, w, k \rangle$?
- Do you see why V always halts?

What languages are verifiable?

Theorem: If L is a language, then there is a verifier for L if and only if $L \in \mathbf{RE}$.

Verifiers and **RE**

- ***Theorem:*** If there is a verifier V for a language L, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.
- Imagine you have a string w and you have the verifier V. You know that if $w \in L$, then there is some $c \in \Sigma^*$ where V will accept $\langle w, c \rangle$. You also know that if $w \notin L$, then there is *no* choice of c such that V accepts $\langle w, c \rangle$.
- You need to build a recognizer M such that if $w \in L$, then M accepts w, and if $w \notin L$, then M does not accept w.
- We know nothing about L except that V exists and how V behaves. Therefore, to see whether $w \in L$, we need to see whether any certificate c exists that causes V to accept $\langle w, c \rangle$.
- ***Idea:*** What if we try running V on every possible certificate?

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 goal:*** Beginning with a recognizer M for the language L, show how to construct a verifier V for L.
- The machine M will accept w if $w \in L$. If M doesn't accept w, then $w \notin L$.
- A certificate is supposed to be some sort of "proof" that the string is in the language. Since the only thing we know about L is that M is a recognizer for it, our certificate would have to tell us something about what M does.
- We need to choose a certificate with the following properties:
  - We can decide in finite time whether a certificate is right or wrong.
  - A "good" certificate proves that $w \in L$ (meaning M accepts w)
  - A "bad" certificate never proves $w \in L$.
- ***Idea:*** If M accepts w, it will do so in finitely many steps. What if our certificate is the number of steps?

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."

RE and Proofs

- Verifiers and recognizers give two different perspectives on the "proof" intuition for **RE**.
- Verifiers are explicitly built to check proofs that strings are in the language.
  - If you know that some string w belongs to the language and you have the proof of it, you can convince someone else that $w \in L$.
- You can think of a recognizer as a device that "searches" for a proof that $w \in L$.
  - If it finds it, great!
  - If not, it might loop forever.

RE and Proofs

- If the **RE** languages represent languages where membership can be proven, what does a non-**RE** language look like?
- Intuitively, a language is *not* in **RE** if there is no general way to prove that a given string $w \in L$ actually belongs to L.
- In other words, even if you knew that a string was in the language, you may never be able to convince anyone of it!

Time-Out for Announcements!

Midterm Grading

- You're done with the midterm! Woohoo!
- We're going to grade the exam over the weekend and try to get things returned on Monday.
- SCPD students – we messed up the exam return last time. We're going to make a point to get it right this time. We're genuinely sorry about that.

A Reminder: Honor Code

- We're coming up to the point in the quarter where we start to see a marked uptick in the number of assignments submitted that are pretty obviously copied from old solution sets.
- I know that a lot of you are stressed, tired, and worn out at this point. However, please, please, *please* don't do this. The costs are really high and it's super easy to spot.
- ***To the campus community at large:*** if someone you know (a friend, a dormmate, a problem set partner, etc.) is really suffering, please reach out to them and make sure they're okay. It's easy for people to get isolated and overwhelmed at this point in the quarter, and (from experience) that can be an awful, awful feeling.

Your Questions

“Do you believe that there exists anything that the human mind can do that a Turing machine cannot?”

I actually *don't*, but I don't lose much sleep over it. I figure that the human mind can be way less powerful than TMs but still be able to discover and experience so much that the point is purely theoretical.

I'm willing to be proven wrong, though!

“Staring endlessly at an exam problem in vain, and then thinking of the solution on the walk home, is a pretty awful feeling. Any words of comfort / encouragement / advice?”

I'm sorry to hear that this happened.

The idea of writing proofs under time pressure on an exam is entirely artificial. You never do things like this in the real world. It's great that you eventually figured it out, since that's ultimately the skill we're hoping for you to develop. It's too bad that it won't count toward your exam grade, but that matters less in the long term.

Back to CS103!

Finding Non-**RE** Languages

Finding Non-**RE** Languages

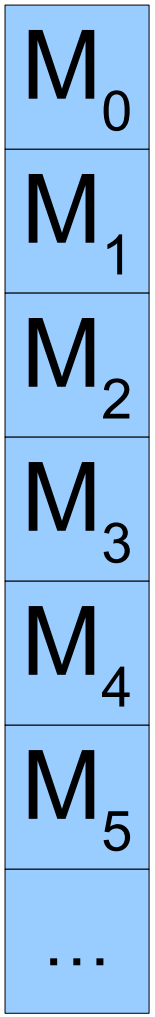
- Right now, we know that non-**RE** languages exist, but we have no idea what they look like.
- The first non-**RE** language we're going to see is one that is, essentially, constructed specifically to not be an **RE** language.
- Having seen that language, we'll then try to see if we can develop a more nuanced understanding of how to find non-**RE** languages.

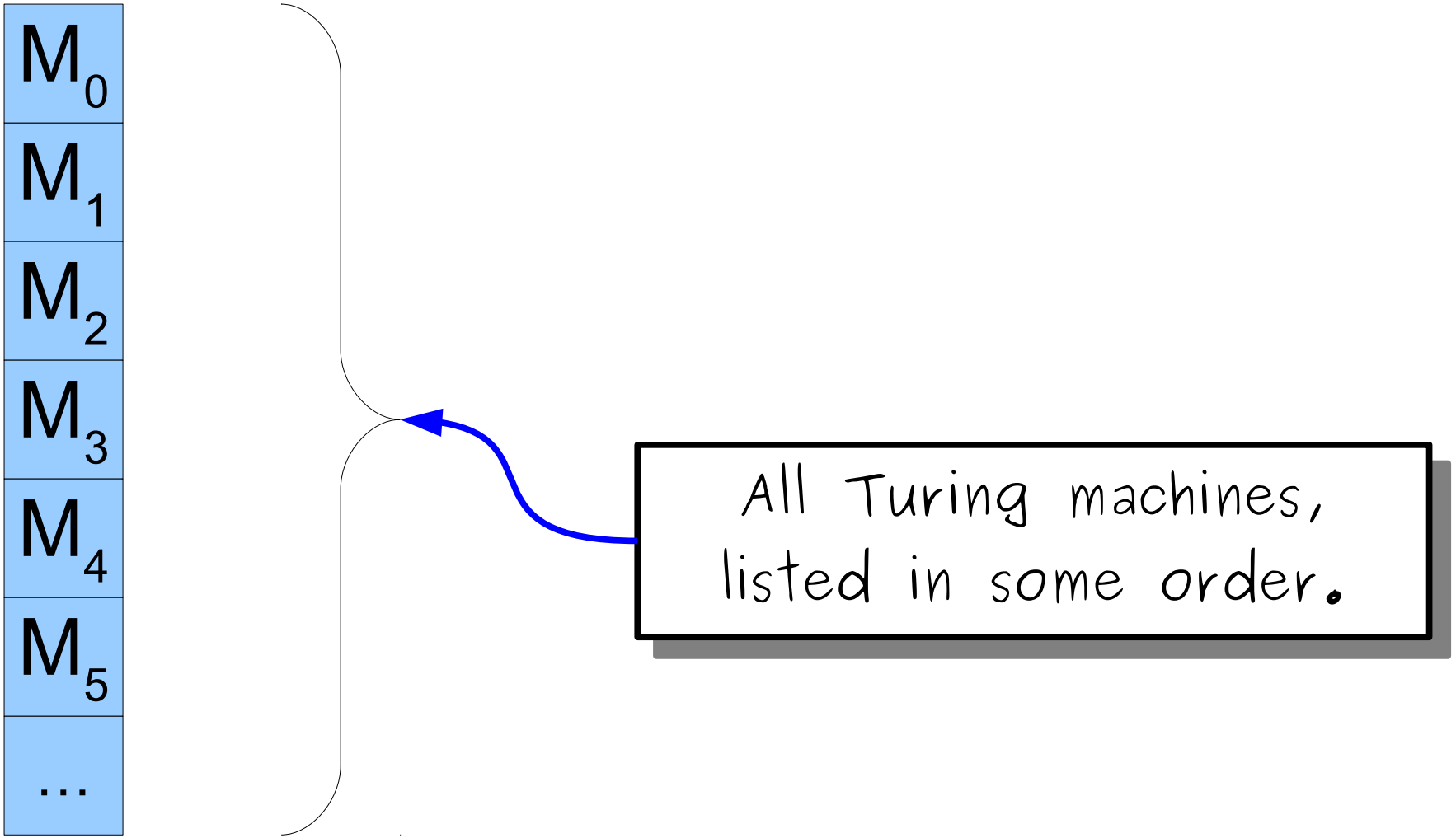
Languages, TMs, and TM Encodings

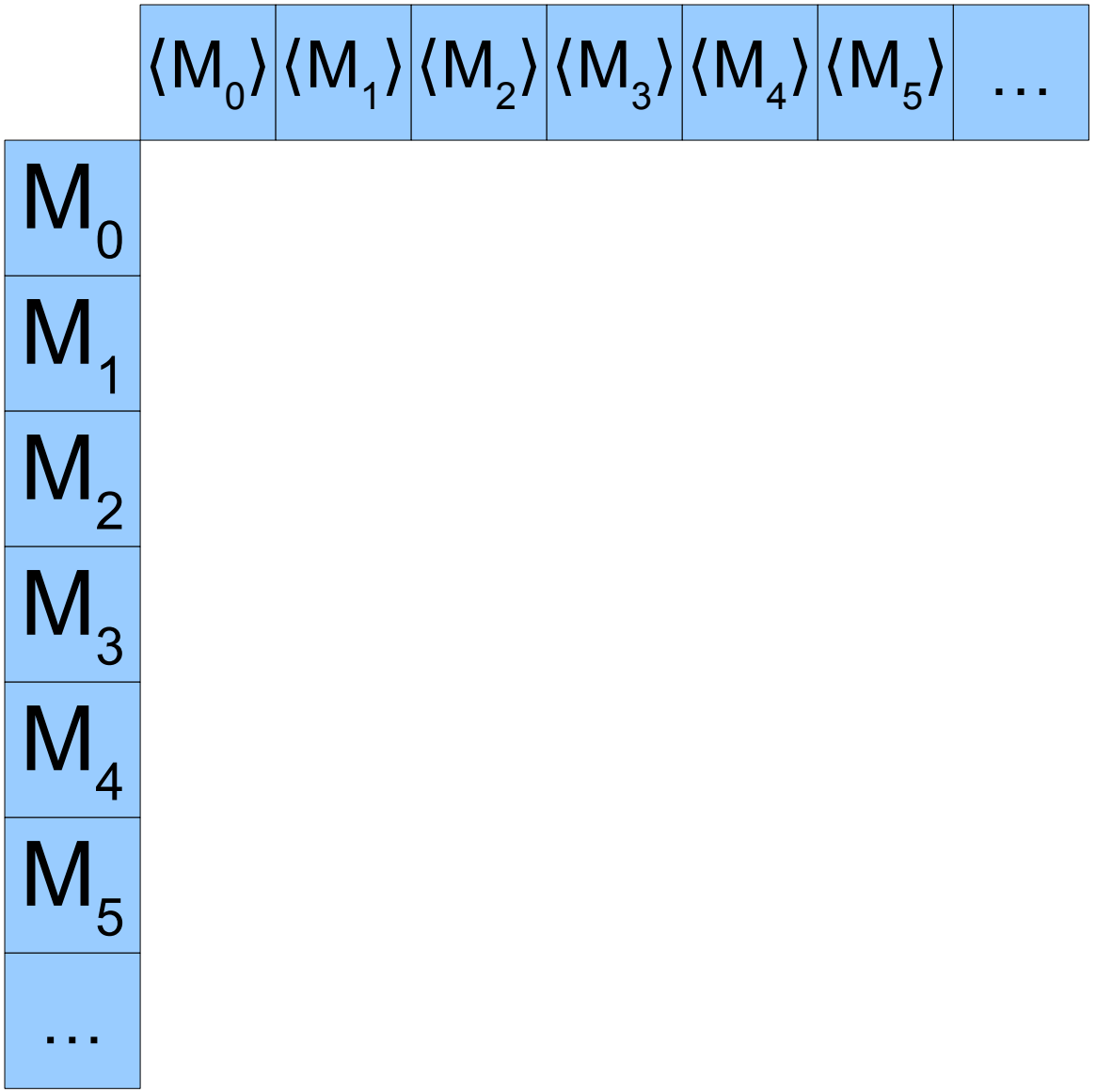
- Recall: The language of a TM M is the set

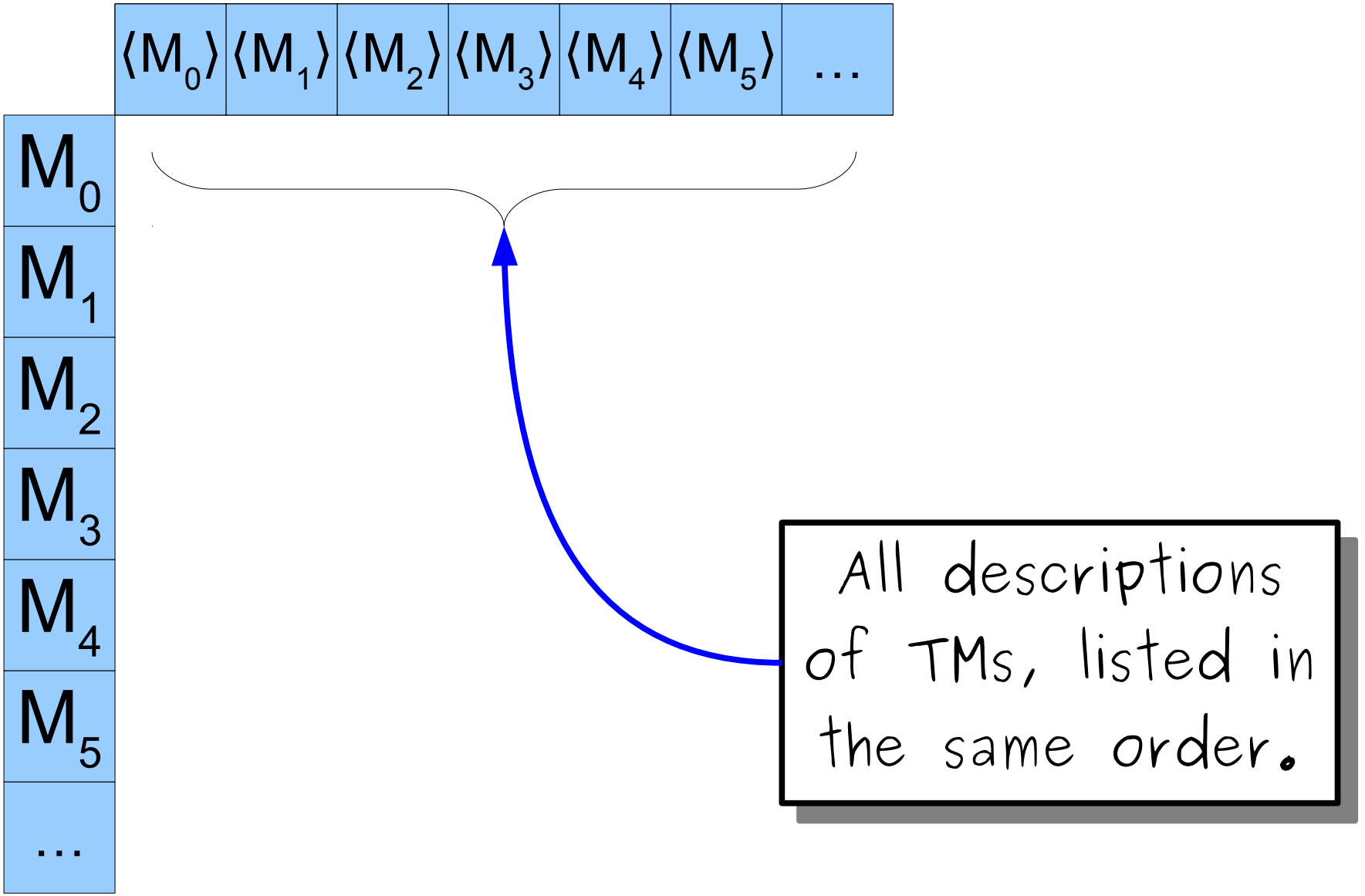
$$\mathscr{L}(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

- Some of the strings in this set might be descriptions of TMs.
- What happens if we just focus on the set of strings that are legal TM descriptions?


M_0
M_1
M_2
M_3
M_4
M_5
...


M_0
M_1
M_2
M_3
M_4
M_5
...
All Turing machines, listed in some order.


⟨M0⟩
⟨M1⟩
⟨M2⟩
⟨M3⟩
⟨M4⟩
⟨M5⟩
…
M0
M1
M2
M3
M4
M5
…


⟨M0⟩ ⟨M1⟩ ⟨M2⟩ ⟨M3⟩ ⟨M4⟩ ⟨M5⟩ …
M0
M1
M2
M3
M4
M5
…
All descriptions of TMs, listed in the same order.

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1							
M_2							
M_3							
M_4							
M_5							
…							

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2							
M_3							
M_4							
M_5							
…							

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3							
M_4							
M_5							
…							

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4							
M_5							
…							

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5							
…							

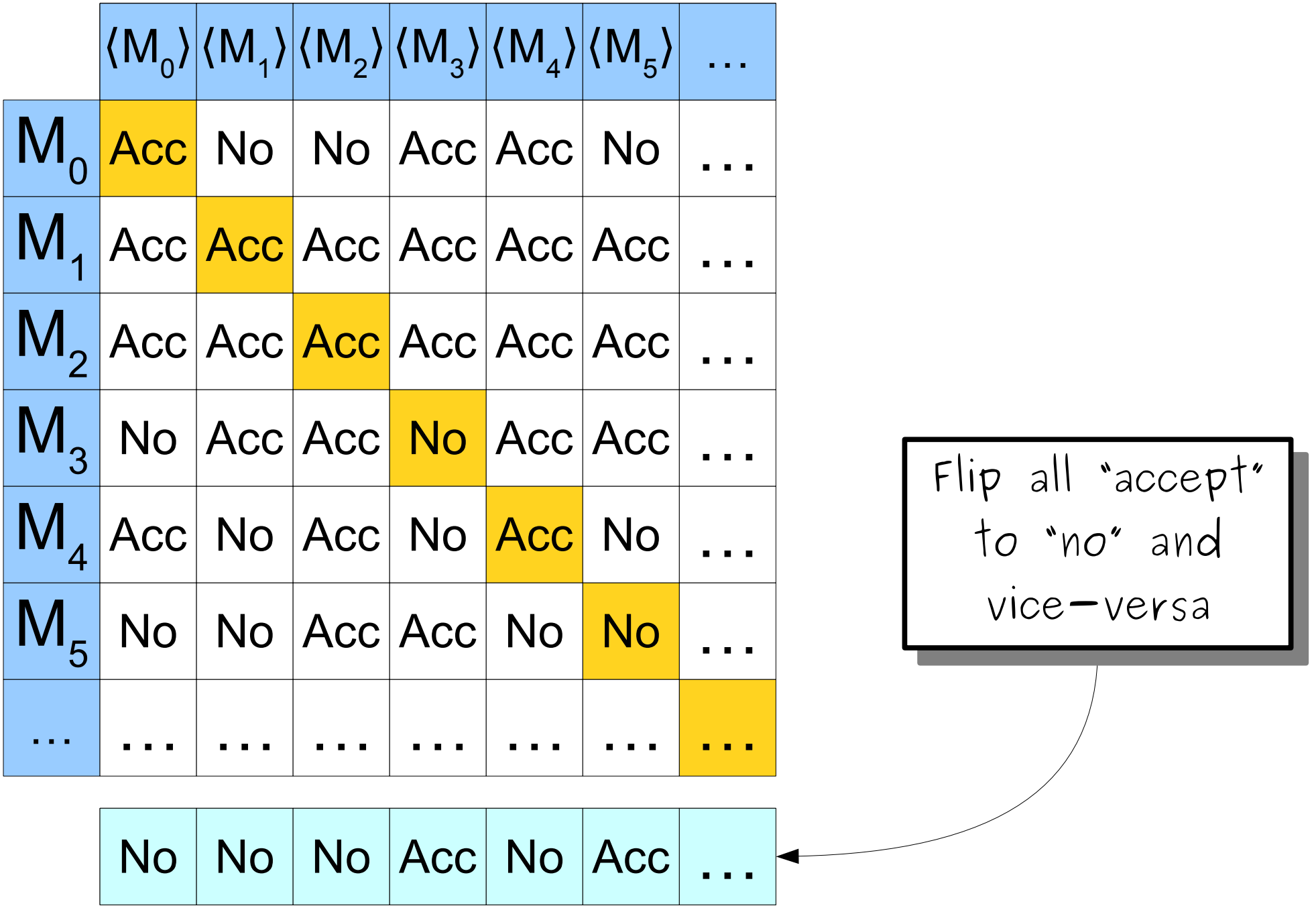
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…							

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

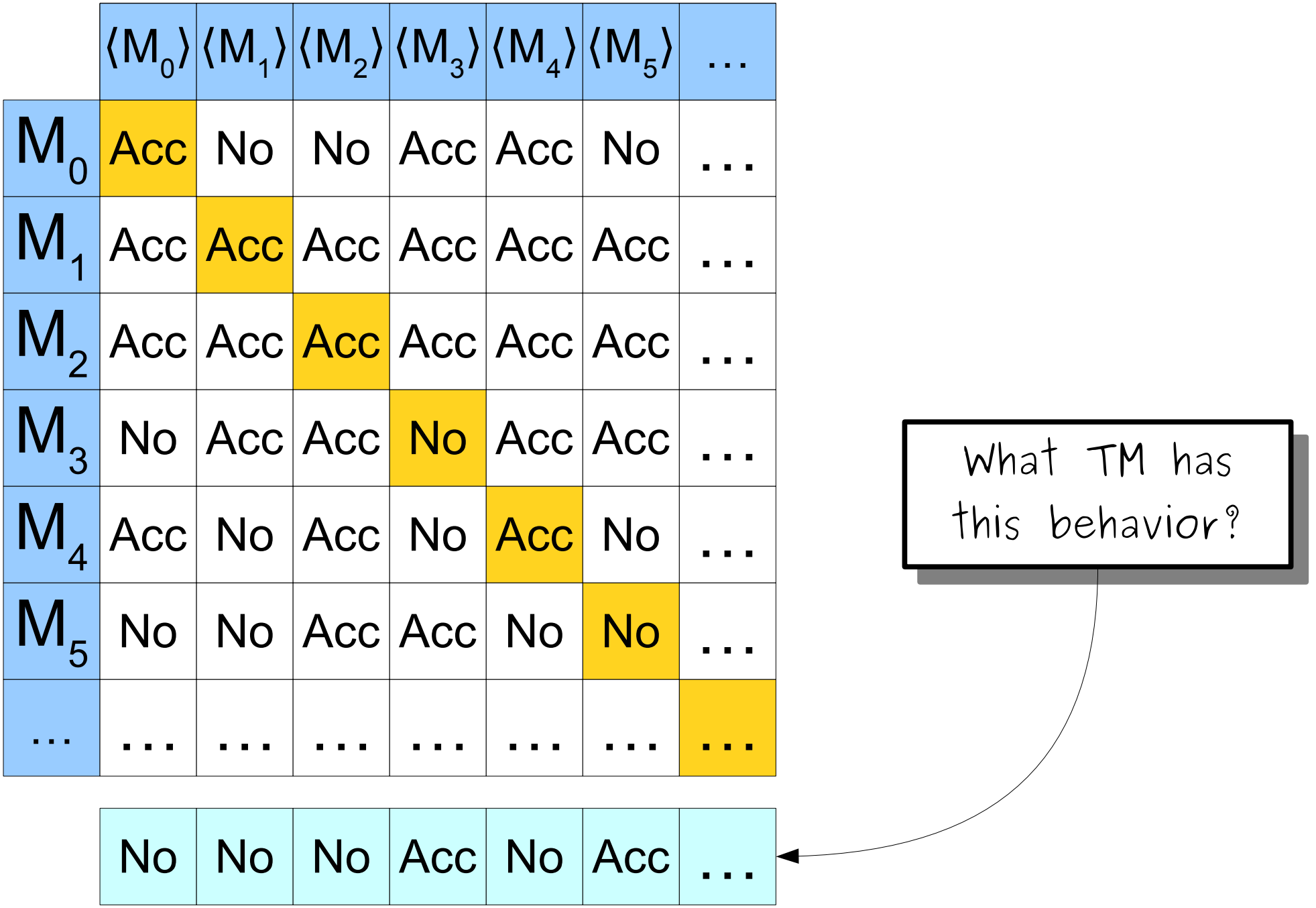
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

Acc	Acc	Acc	No	Acc	No	…


⟨M0⟩ ⟨M1⟩ ⟨M2⟩ ⟨M3⟩ ⟨M4⟩ ⟨M5⟩ …
M0 Acc No No Acc Acc No …
M1 Acc Acc Acc Acc Acc Acc …
M2 Acc Acc Acc Acc Acc Acc …
M3 No Acc Acc No Acc Acc …
M4 Acc No Acc No Acc No …
M5 No No Acc Acc No No …
… … … … … … … …
No No No Acc No Acc …
Flip all "accept" to "no" and vice-versa

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

	No	No	No	Acc	No	Acc	…


⟨M0⟩ ⟨M1⟩ ⟨M2⟩ ⟨M3⟩ ⟨M4⟩ ⟨M5⟩ …
M0 Acc No No Acc Acc No …
M1 Acc Acc Acc Acc Acc Acc …
M2 Acc Acc Acc Acc Acc Acc …
M3 No Acc Acc No Acc Acc …
M4 Acc No Acc No Acc No …
M5 No No Acc Acc No No …
… … … … … … … …
No No No Acc No Acc …
What TM has this behavior?

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

	No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

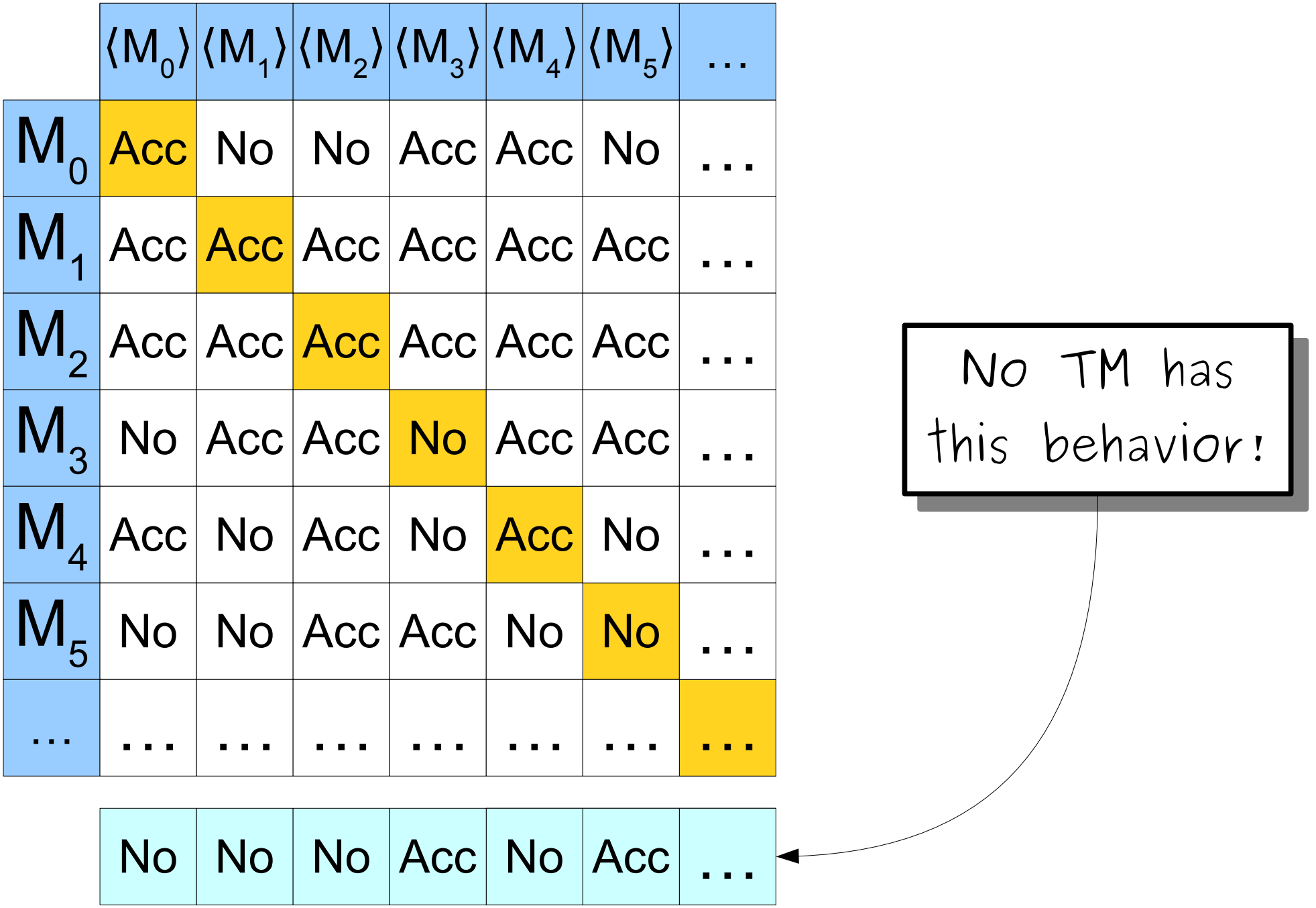
No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

No	No	No	Acc	No	Acc	…


⟨M0⟩ ⟨M1⟩ ⟨M2⟩ ⟨M3⟩ ⟨M4⟩ ⟨M5⟩ …
M0 Acc No No Acc Acc No …
M1 Acc Acc Acc Acc Acc Acc …
M2 Acc Acc Acc Acc Acc Acc …
M3 No Acc Acc No Acc Acc …
M4 Acc No Acc No Acc No …
M5 No No Acc Acc No No …
… … … … … … … …
No No No Acc No Acc …
No TM has this behavior!

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

	No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…

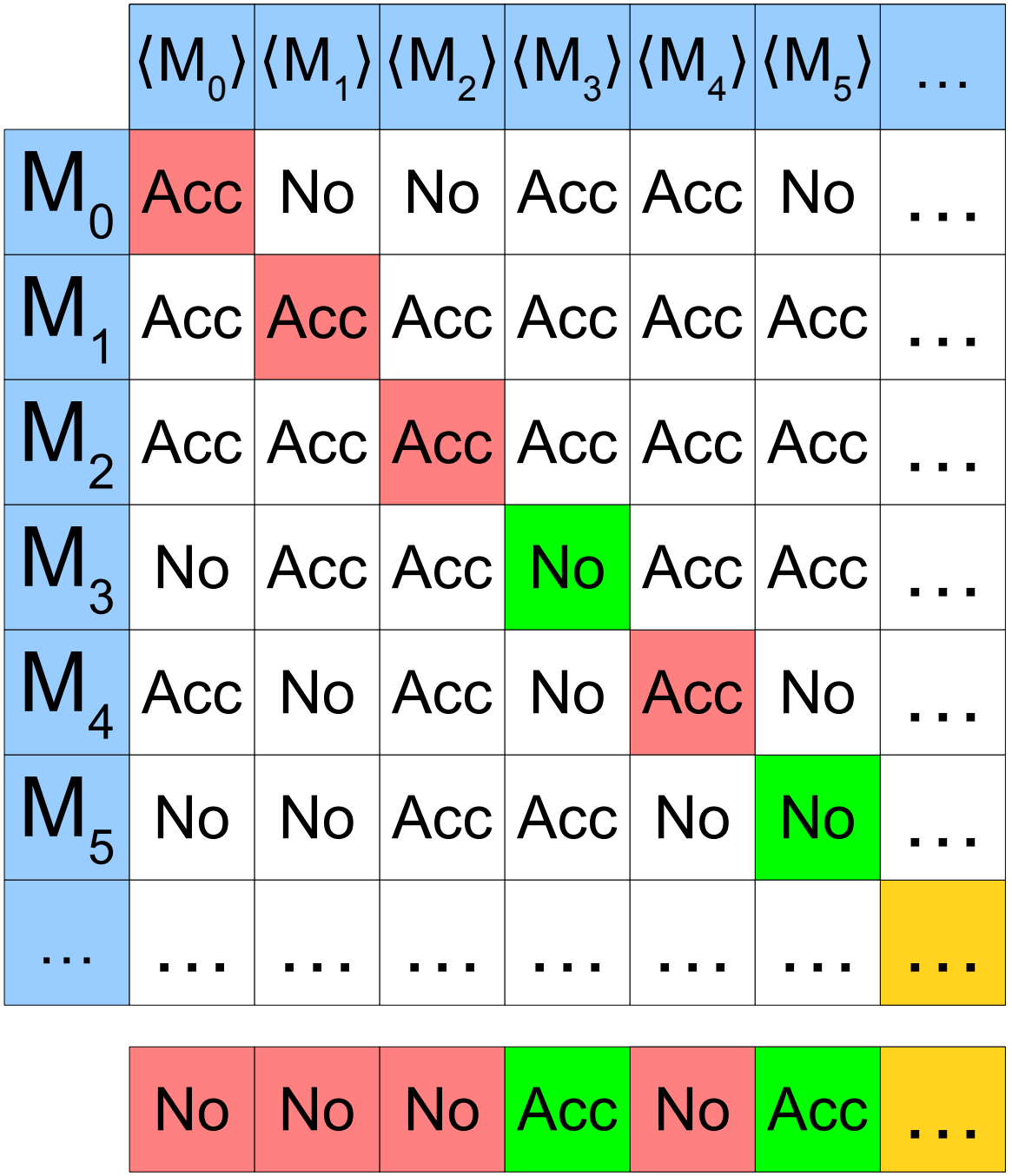
No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…
	No	No	No	Acc	No	Acc	…

“The language of all TMs that do not accept their own description.”

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…
…	…	…	…	…	…	…	…
	No	No	No	Acc	No	Acc	…

{ $\langle M \rangle$ | M is a TM that does not accept $\langle M \rangle$ }



{ ⟨*M*⟩ | *M* is a TM and ⟨*M*⟩ ∉ $\mathscr{L}$(*M*) }

Diagonalization Revisited

- The ***diagonalization language***, which we denote L_D, is defined as

  $$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$$

- That is, L_D is the set of descriptions of Turing machines that do not accept themselves.

$$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$$

Theorem: $L_D \notin$ **RE**.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathcal{L}(M) \}$

Theorem: $L_D \notin$ **RE**.

Proof: By contradiction; assume that $L_D \in$ **RE**.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin \mathbf{RE}$.

Proof: By contradiction; assume that $L_D \in \mathbf{RE}$. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin \mathbf{RE}$.

Proof: By contradiction; assume that $L_D \in \mathbf{RE}$. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin$ **RE**.

Proof: By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

Because $\mathscr{L}(R) = L_D$, we know that a string belongs to one set if and only if it belongs to the other.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin \mathbf{RE}$.

Proof: By contradiction; assume that $L_D \in \mathbf{RE}$. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of L_D, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin$ **RE**.

Proof: By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of L_D, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

$$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$$

Theorem: $L_D \notin$ **RE**.

Proof: By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of L_D, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

We've replaced the left-hand side of this biconditional with an equivalent statement.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin \mathbf{RE}$.

Proof: By contradiction; assume that $L_D \in \mathbf{RE}$. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of L_D, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM M, so in particular it should hold when $M = R$.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin \mathbf{RE}$.

Proof: By contradiction; assume that $L_D \in \mathbf{RE}$. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of L_D, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM M, so in particular it should hold when $M = R$.

A nice consequence of a universally-quantified statement is that it should work in all cases.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin \mathbf{RE}$.

Proof: By contradiction; assume that $L_D \in \mathbf{RE}$. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of L_D, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM M, so in particular it should hold when $M = R$. If we pick $M = R$, we see that

$$\langle R \rangle \notin \mathscr{L}(R) \text{ iff } \langle R \rangle \in \mathscr{L}(R) \qquad (3)$$

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin \mathbf{RE}$.

Proof: By contradiction; assume that $L_D \in \mathbf{RE}$. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of L_D, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM M, so in particular it should hold when $M = R$. If we pick $M = R$, we see that

$$\langle R \rangle \notin \mathscr{L}(R) \text{ iff } \langle R \rangle \in \mathscr{L}(R) \qquad (3)$$

This is clearly impossible.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin$ **RE**.

Proof: By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of L_D, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM M, so in particular it should hold when $M = R$. If we pick $M = R$, we see that

$$\langle R \rangle \notin \mathscr{L}(R) \text{ iff } \langle R \rangle \in \mathscr{L}(R) \qquad (3)$$

This is clearly impossible. We have reached a contradiction, so our assumption must have been wrong.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin$ **RE**.

Proof: By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

From the definition of L_D, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM M, so in particular it should hold when $M = R$. If we pick $M = R$, we see that

$$\langle R \rangle \notin \mathscr{L}(R) \text{ iff } \langle R \rangle \in \mathscr{L}(R) \qquad (3)$$

This is clearly impossible. We have reached a contradiction, so our assumption must have been wrong. Thus $L_D \notin$ **RE**.

$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathscr{L}(M) \}$

Theorem: $L_D \notin$ **RE**.

Proof: By contradiction; assume that $L_D \in$ **RE**. Then there must be some TM R such that $\mathscr{L}(R) = L_D$.

Since $\mathscr{L}(R) = L_D$, we know that if M is any TM, then

$$\langle M \rangle \in L_D \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (1)$$

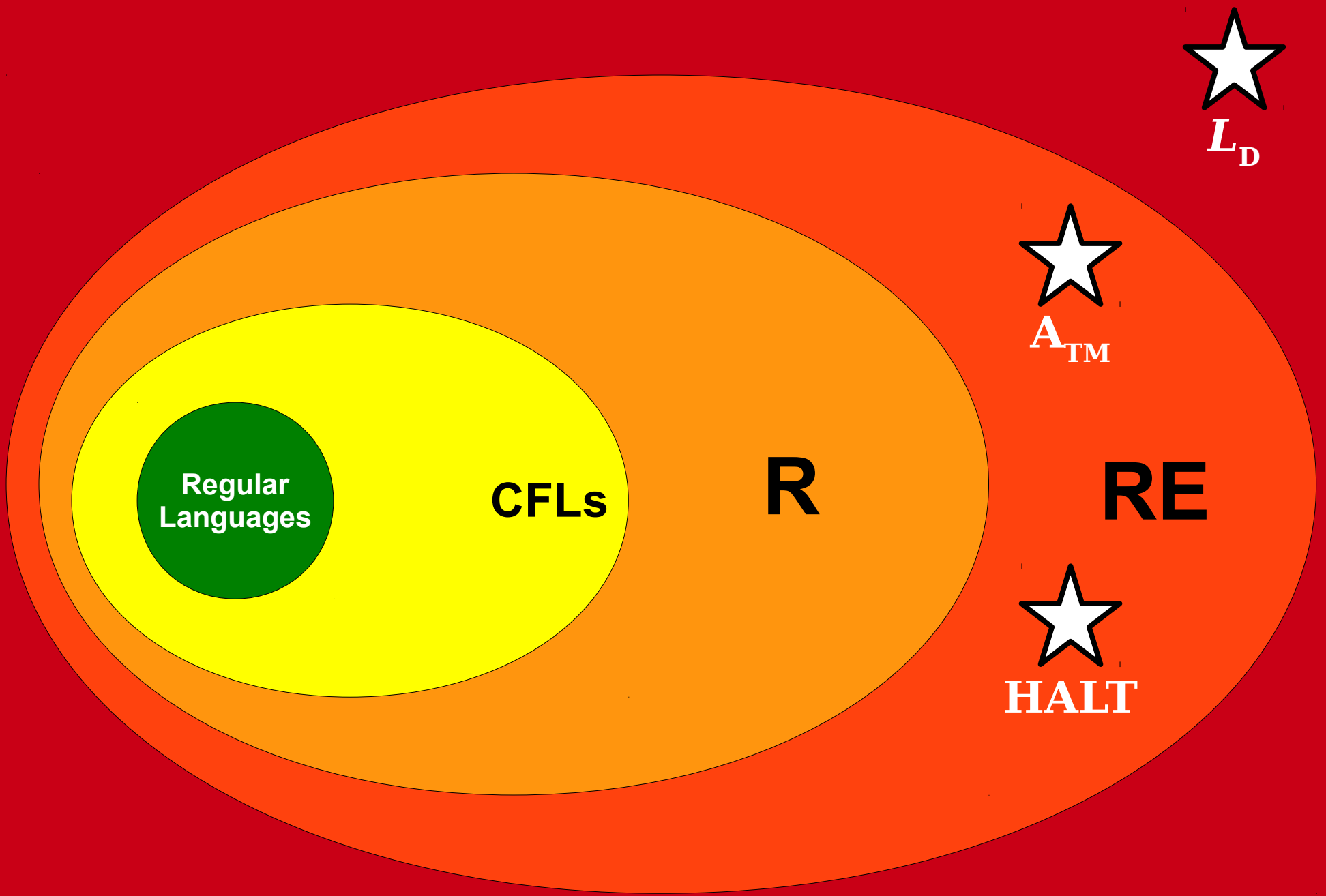
From the definition of L_D, we see that $\langle M \rangle \in L_D$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Combining this with statement (1) tells us that

$$\langle M \rangle \notin \mathscr{L}(M) \text{ iff } \langle M \rangle \in \mathscr{L}(R) \qquad (2)$$

Statement (2) holds for any TM M, so in particular it should hold when $M = R$. If we pick $M = R$, we see that

$$\langle R \rangle \notin \mathscr{L}(R) \text{ iff } \langle R \rangle \in \mathscr{L}(R) \qquad (3)$$

This is clearly impossible. We have reached a contradiction, so our assumption must have been wrong. Thus $L_D \notin$ **RE**. ■


L_D
A_{TM}
Regular Languages
CFLs
R
RE
HALT
All Languages

What This Means

- On a deeper philosophical level, the fact that non-**RE** languages exist supports the following claim:

There are statements that are true but not provable.

- Intuitively, given any non-**RE** language, there will be some string in the language that *cannot* be proven to be in the language.
- This result can be formalized as a result called ***Gödel's incompleteness theorem***, one of the most important mathematical results of all time.
- Want to learn more? Take Phil 152 or CS154!

What This Means

- On a more philosophical note, you could interpret the previous result in the following way:

There are inherent limits about what mathematics can teach us.

- There's no automatic way to do math. There are true statements that we can't prove.
- That doesn't mean that mathematics is worthless. It just means that we need to temper our expectations about it.