We have seen that

$A_{TM}$ undecidable
$A_{LBA}$ decidable

However,

$$E_{LBA} = \{ \langle M \rangle \mid M \text{ is an LBA where } L(M) = \emptyset \}$$

is undecidable

We do the proof by the computation history method

Idea: the question of $M$ accepts $w$ can be solved by checking if $L(B) = \emptyset$, where $B$ is an LBA
Then because we assume $E_{\text{LBA}}$ is decidable, we have a decider for $A_{\text{TM}}$

The design of $B$: $B$ recognizes all accepting computation histories for $M$ on $w$

$$M \text{ accepts } w \Rightarrow L(B) \neq \emptyset$$
$$M \text{ rejects } w \Rightarrow L(B) = \emptyset$$

We see that the machine is designed according to the given $w$. This strategy has been used in earlier examples.

Details of $B$: on any input $x$, we check if $x$ is an accepting computation history for $M$ on $w$
Specifically, we check if $x$ is

\[
\# \quad \# \quad \# \cdots \# \quad \#
\]

(1)

and $C_1, \ldots, C_l$ satisfy that

- $C_1$ is the start configuration,
- $C_l$ is an accepting configuration, and
- $C_i$ follows from $C_{i-1}$

The machine looks like
To begin, we check if the input $x$ is in the form of (1).

Next, $C_1$ is $q_0w$, so checking the first condition is easy.

For the third condition, we scan if $C_i$ contains $q_{\text{accept}}$.

Now $C_i$ and $C_{i+1}$ are the same except around the head position.
To compare $C_i$ and $C_{i+1}$, the TM zigzags between them.

The setting looks good, but remember that $B$ is an LBA.

The above discussion seems to show that our operations never go beyond $|x|$. 

On the other hand, if you think extra space is needed, it is fine as long as the space needed is bounded by a constant factor of the length of $\#C_1\# \cdots \#C_i\#$. 

$E_{\text{LBA}}$ Undecidable V
For example, if we copy \( C_i \) and \( C_{i+1} \) to the end for the comparison, the extra space needed is no more than \(|\#C_1\# \cdots \#C_l\#|\)

Thus for input \( x \) we can check if the first half is \( \#C_1\# \cdots \#C_l\# \)

Then the machine never goes beyond \(|x|\). Further, if \( M \) accepts \( w \), then at least one \( x \) is accepted by \( B \)
Earlier we proved that

\[ E_{CFG} = \{ \langle G \rangle \mid G : CFG, L(G) = \emptyset \} \]

is decidable.

Now we show a related problem is undecidable.

\[ ALL_{CFG} = \{ \langle G \rangle \mid G : CFG, L(G) = \Sigma^* \} \]

It checks if \( G \) generates all possible strings.

The proof is still by contradiction.

We assume \( ALL_{CFG} \) is decidable.
Idea: consider a CFG $G$ such that

$$G \text{ generates } \Sigma^* \iff M \text{ does not accept } w$$

This is equivalent to

$$\begin{cases} G \text{ generates } \Sigma^* & \text{if } M \text{ does not accept } w \\ G \text{ fails on some strings} & \text{if } M \text{ accepts } w \end{cases}$$

If we have a decider on $G$, then we have a decider on $A_{TM}$

If $M$ accepts $w$, we let $G$ fail to generate
an accepting computation history for $M$ on $w$

That is, for $G$, the input cannot be

$$
\# \overset{C_1}{\longrightarrow} \# \overset{C_2}{\longrightarrow} \# \cdots \overset{C_l}{\longrightarrow} \# 
$$

where $C_1, \ldots, C_l$ satisfy that

- $C_1$ is the start configuration,
- $C_l$ is an accepting configuration, and
- $C_i$ follows from $C_{i-1}$

Therefore, $G$ generates all strings

that do not start with $C_1$. 
that do not end with an accepting configuration, or

$C_i$ does not yield $C_{i+1}$

Note that it’s “or” because the opposite of $A$ and $B$ and $C$

is

$\neg A$ or $\neg B$ or $\neg C$

On the other hand, if $M$ does not accept $w$, no accepting computation history exists
Then $G$ generates all strings

But how to construct such a CFG?

Let’s generate an equivalent PDA

The PDA nondeterministically checks three branches for the three requirements

For example, the first branch checks if the beginning of the input is $C_1$ and accepts if it is not

The third branch is more complicated

It accepts if $C_i$ does not properly yields $C_{i+1}$

We can push $C_i$ to stack ($\#$ allows us to extract $C_i$)
We pop the stack to compare $C_i$ and $C_{i+1}$

They are the same except around the head position

A problem is that when we pop $C_i$, it is in the reverse order

To enable the comparison, we write the accepting computation history differently

By this way, when we pop $C_2^R$, we get $C_2$ and can do the comparison
This means that for any input $x$, if it is in the form of $#C_1# \cdots #C_l#$, we “treat” the second segment as $C_2^R$ in designing operations.