Pumping Lemma for CFG and Turing Machines

204213 Theory of Computation

Jittat Fakcharoenphol

Kasetsart University

August 3, 2021

Outline

Non-context-free language

Can you find a CFG describing the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n | n \geq 0\}$?

Non-context-free language

Can you find a CFG describing the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n | n \geq 0\}$?
I bet you can't.

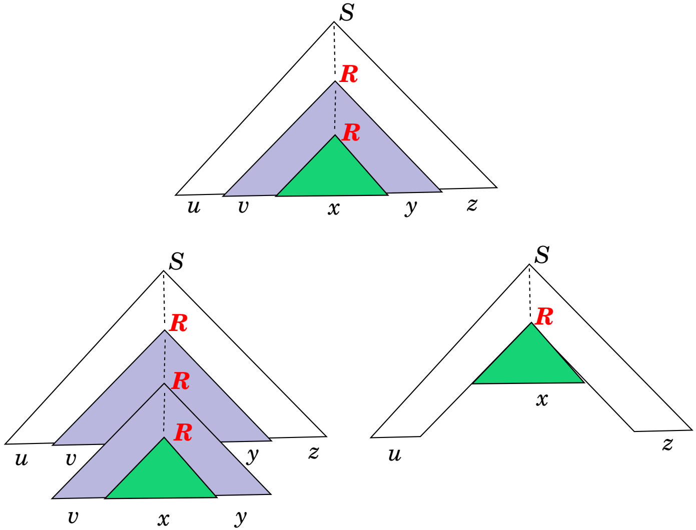
Pumping lemma for CFL

Theorem 1 (pumping lemma for CFL)

If A is a context-free language, then there is a pumping length p such that for any string $s \in A$ of length at least p, s can be divided into 5 pieces $s = uvxyz$ satisfying the following conditions

1. *for each $i \geq 0$, $uv^ixy^iz \in A$,*
2. $|vy| > 0$, *and*
3. $|vxy| \leq p$.

Parse tree for s



$C = \{a^n b^n c^n | n \geq 0\}$ is not context-free (1)

- We'll prove by contradiction. Assume that C is context-free.

$C = \{a^n b^n c^n | n \geq 0\}$ is not context-free (1)

- We'll prove by contradiction. Assume that C is context-free.
- Thus, there exists a pumping length p.

$C = \{a^n b^n c^n | n \geq 0\}$ is not context-free (1)

- We'll prove by contradiction. Assume that C is context-free.
- Thus, there exists a pumping length p.
- Consider $s = a^p b^p c^p \in C$. Note that $|s| \geq p$.

$C = \{a^n b^n c^n | n \geq 0\}$ is not context-free (1)

- We'll prove by contradiction. Assume that C is context-free.
- Thus, there exists a pumping length p.
- Consider $s = a^p b^p c^p \in C$. Note that $|s| \geq p$.
- The pumping lemma states that we can divide $s = uvxyz$, such that $uv^i xy^i z \in C$ for any $i \geq 0$.

$C = \{a^n b^n c^n | n \geq 0\}$ is not context-free (1)

- We'll prove by contradiction. Assume that C is context-free.
- Thus, there exists a pumping length p.
- Consider $s = a^p b^p c^p \in C$. Note that $|s| \geq p$.
- The pumping lemma states that we can divide $s = uvxyz$, such that $uv^i xy^i z \in C$ for any $i \geq 0$.
- We'll show that this leads to a contradiction.

First proof

There are two cases.

- **Case 1**, if each of v and y contains only one kind of alphabets.

First proof

There are two cases.

- **Case 1**, if each of v and y contains only one kind of alphabets. Then consider $s' = uv^2xy^2z$. Note that at least one alphabet appears in s' in fewer times than the others; thus, $s' \notin C$.

First proof

There are two cases.

- **Case 1**, if each of v and y contains only one kind of alphabets. Then consider $s' = uv^2xy^2z$. Note that at least one alphabet appears in s' in fewer times than the others; thus, $s' \notin C$.
- **Case 2**, if v or y contains two kinds of alphabets.

First proof

There are two cases.

- **Case 1**, if each of v and y contains only one kind of alphabets. Then consider $s' = uv^2xy^2z$. Note that at least one alphabet appears in s' in fewer times than the others; thus, $s' \notin C$.
- **Case 2**, if v or y contains two kinds of alphabets. Note that $s' = uv^2xy^2z$ contains alphabets in the wrong order. Again, $s' \notin C$.

First proof

There are two cases.

- **Case 1**, if each of v and y contains only one kind of alphabets. Then consider $s' = uv^2xy^2z$. Note that at least one alphabet appears in s' in fewer times than the others; thus, $s' \notin C$.
- **Case 2**, if v or y contains two kinds of alphabets. Note that $s' = uv^2xy^2z$ contains alphabets in the wrong order. Again, $s' \notin C$.

Note that in either case, s cannot be pumped, and this contradicts the assumption that C is context-free.

Second proof

Note that the first proof doesn't use the 3rd property, stating that $|vxy| \leq p$.

Second proof

Note that the first proof doesn't use the 3rd property, stating that $|vxy| \leq p$. Given that fact, we know that v and y cannot contain all three types of alphabets.

Second proof

Note that the first proof doesn't use the 3rd property, stating that $|vxy| \leq p$. Given that fact, we know that v and y cannot contain all three types of alphabets. Therefore, $s' = uv^2xy^2z$ contains different numbers of a's, b's, or c's, and $s' \notin C$.

Second proof

Note that the first proof doesn't use the 3rd property, stating that $|vxy| \leq p$. Given that fact, we know that v and y cannot contain all three types of alphabets. Therefore, $s' = uv^2xy^2z$ contains different numbers of a's, b's, or c's, and $s' \notin C$. This, again, leads to a contradiction.

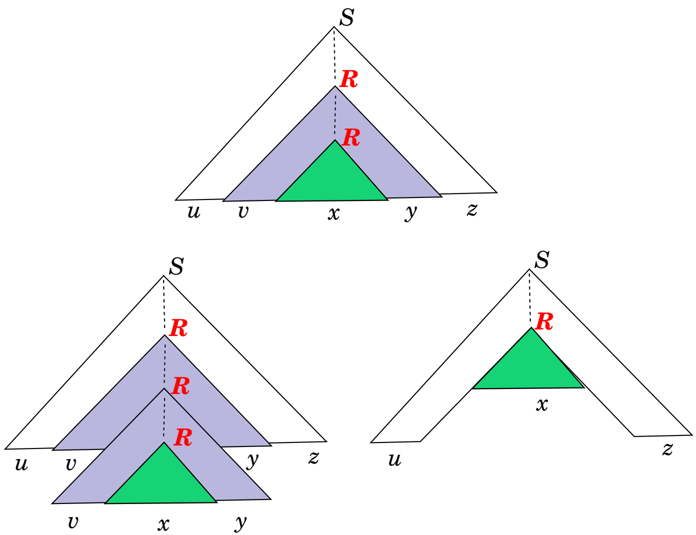
Pumping lemma for CFL

Theorem 2 (pumping lemma for CFL)

If A is a context-free language, then there is a pumping length p such that for any string $s \in A$ of length at least p, s can be divided into 5 pieces $s = uvxyz$ satisfying the following conditions

1. *for each $i \geq 0$, $uv^ixy^iz \in A$,*
2. $|vy| > 0$, *and*
3. $|vxy| \leq p$.

Parse tree for s



Proof Idea

- Recall our proof for the pumping lemma for regular languages.

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?)

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?) Yes, the Pigeon-Hole Principle.

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?) Yes, the Pigeon-Hole Principle.
  - What?

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?) Yes, the Pigeon-Hole Principle.
  - What?
  - Since we have $|V|$ variables, if, on the parse tree, the path from the start variable to some terminal is long enough (how long?)

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?) Yes, the Pigeon-Hole Principle.
  - What?
  - Since we have $|V|$ variables, if, on the parse tree, the path from the start variable to some terminal is long enough (how long?) we should see the same variable twice.

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?) Yes, the Pigeon-Hole Principle.
  - What?
  - Since we have $|V|$ variables, if, on the parse tree, the path from the start variable to some terminal is long enough (how long?) we should see the same variable twice.
  - How can we make sure that the parse tree is very tall?

Tall parse tree: example

G_1

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow 1A0|0A1|\varepsilon \\
B &\rightarrow BB|0|1
\end{aligned}
$$

Tall parse tree: example

G_1

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow 1A0|0A1|\varepsilon \\
B &\rightarrow BB|0|1
\end{aligned}
$$

- What is the longest string whose longest path from S to any terminal is ≤ 4?
- Any bound on the length of the string generated by G_2 that guarantees that the height of its parse tree is at least 5?

Models of computation

- **Finite automata and regular expressions.**
  - Devices with small, limited memory.
- **Push-down automata and context-free languages**
  - Devices with unlimited memory, but have restricted access.

Turing Machines

Turing Machines

- Proposed by Alan Turing in 1936.

Turing Machines

- Proposed by Alan Turing in 1936.
- A finite automaton with an unlimited memory with unrestricted access.

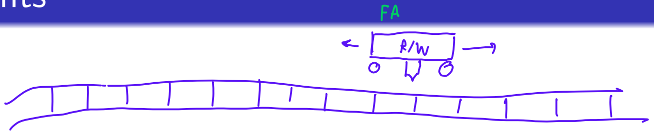
Turing Machines

- Proposed by Alan Turing in 1936.
- A finite automaton with an unlimited memory with unrestricted access.
- Can perform any tasks that a computer can. (we'll see)

Turing Machines

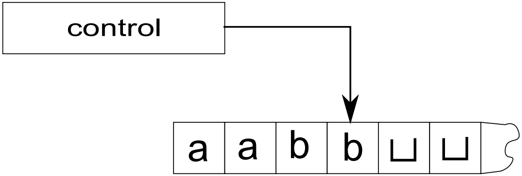
- Proposed by Alan Turing in 1936.
- A finite automaton with an unlimited memory with unrestricted access. → เข้าถึงได้หมด
- Can perform any tasks that a computer can. (we'll see)
- However, there are problems that TM can't solve. These problems are beyond the limit of computation.

Components



- An infinite **tape**. หน่วยความจำ (tape)
- A tape head that can
  - **read and write** to the tape, and
  - **move** around the tape.

Schematic



How Turing machines work

- The tape initialy contains an input string.

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.
- It can write a symbol back and move left or right.

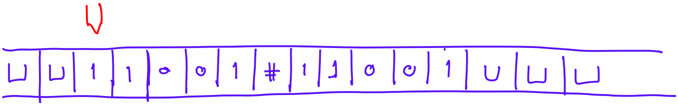
How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.
- It can write a symbol back and move left or right.
- At the end of the computation, the machine outputs accept or reject, by entering accept state of reject state. (After changing, it halts.)

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.
- It can write a symbol back and move left or right.
- At the end of the computation, the machine outputs accept or reject, by entering accept state of reject state. (After changing, it halts.)
- It can go on forever (not entering any accept or reject states).

Example: M_1



- We'll design a TM that recognizes

$$B = \{w\#w \mid w \in \{0,1\}^*\}.$$

Example: M_1 — strategy

- M_1 works by comparing two copies of w.
- M_1 compares two symbols on the corresponding positions.
- It write marks on the tape to keep track of the position.

Example: M_1 — snapshots

0 1 1 0 0 0 # 0 1 1 0 0 0 ⊔

Example: M_1 — snapshots

0 1 1 0 0 0 # 0 1 1 0 0 0 ⊔

x 1 1 0 0 0 # 0 1 1 0 0 0 ⊔

Example: M_1 — snapshots

0 1 1 0 0 0 # 0 1 1 0 0 0 ⊔
x 1 1 0 0 0 # 0 1 1 0 0 0 ⊔
x 1 1 0 0 0 # 0 1 1 0 0 0 ⊔

Example: M_1 — snapshots

```
0 1 1 0 0 0 # 0 1 1 0 0 0 ⊔
x 1 1 0 0 0 # 0 1 1 0 0 0 ⊔
x 1 1 0 0 0 # 0 1 1 0 0 0 ⊔
x 1 1 0 0 0 # x 1 1 0 0 0 ⊔
```

Example: M_1 — snapshots

0	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔

Example: M_1 — snapshots

0	1	1	0	0	0	#	0	1	1	0	0	0	␣
x	1	1	0	0	0	#	0	1	1	0	0	0	␣
x	1	1	0	0	0	#	0	1	1	0	0	0	␣
x	1	1	0	0	0	#	x	1	1	0	0	0	␣
x	1	1	0	0	0	#	x	1	1	0	0	0	␣
x	1	1	0	0	0	#	x	1	1	0	0	0	␣

Example: M_1 — snapshots

0	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔
x	x	1	0	0	0	#	x	1	1	0	0	0	⊔

Example: M_1 — snapshots

0	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔
x	x	1	0	0	0	#	x	1	1	0	0	0	⊔
x	x	1	0	0	0	#	x	x	1	0	0	0	⊔

Example: M_1 — snapshots

state q:

อ่าน input tape

→ เขียนอะไร

→ L, R

0	1	1	0	0	0	#	0	1	1	0	0	0	⊔	จำ 0
x	1	1	0	0	0	#	0	1	1	0	0	0	⊔	ข้าม #
x	1	1	0	0	0	#	0	1	1	0	0	0	⊔	
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔	→ check ตัวหลัง # ที่ยังไม่มา แล้วยังไม่ x ว่า match กับตัวที่จำไว้ไหม
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔	
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔	
x	x	1	0	0	0	#	x	1	1	0	0	0	⊔	
x	x	1	0	0	0	#	x	x	1	0	0	0	⊔	
x	x	x	x	x	x	#	x	x	x	x	x	x	⊔	accept!

Example: M_1 — algorithm

M_1 = "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check if they contains the same symbol.

Example: M_1 — algorithm

M_1 = "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check if they contains the same symbol. If they do not or there is no #, reject.

Example: M_1 — algorithm

M_1 = "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check if they contains the same symbol. If they do not or there is no #, reject. Mark these symbols to keep track of the current position.

Example: M_1 — algorithm

$M_1 =$ "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check if they contains the same symbol. If they do not or there is no #, reject. Mark these symbols to keep track of the current position.
2. After all symbols on the left of # have been marked, check if there're other unmarked symbols on the right of #, if there's any, reject; otherwise accept."

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** (current state) and the (symbol on the tape)

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** current state and the symbol on the tape
  - **Output:** next state, a symbol to be written to the tape, and the new state.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** current state and the symbol on the tape
  - **Output:** next state, a symbol to be written to the tape, and the new state.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** current state and the symbol on the tape
  - **Output:** next state, a symbol to be written to the tape, and the new state.
- So, δ is in the form: $\overset{\text{state}}{Q} \times \overset{\text{tape}}{\Gamma} \rightarrow \overset{\text{state}}{Q} \times \overset{\text{tape}}{\Gamma} \times \overset{\text{moving}}{\{\mathrm{L}, \mathrm{R}\}}$

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** current state and the symbol on the tape
  - **Output:** next state, a symbol to be written to the tape, and the new state.
- So, δ is in the form: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$

  state เขียน เลื่อน
- E.g., if $\delta(q, a) = (r, b, \mathrm{L})$, then if the machine is in state q and reads a, it will change its state to r, write b to the tape and move to the left. เลื่อนเข็มไปทางซ้าย

Definition

Definition (Turing Machine)

A **Turing machine** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the **blank symbol** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$,
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{accept} \neq q_{reject}$.