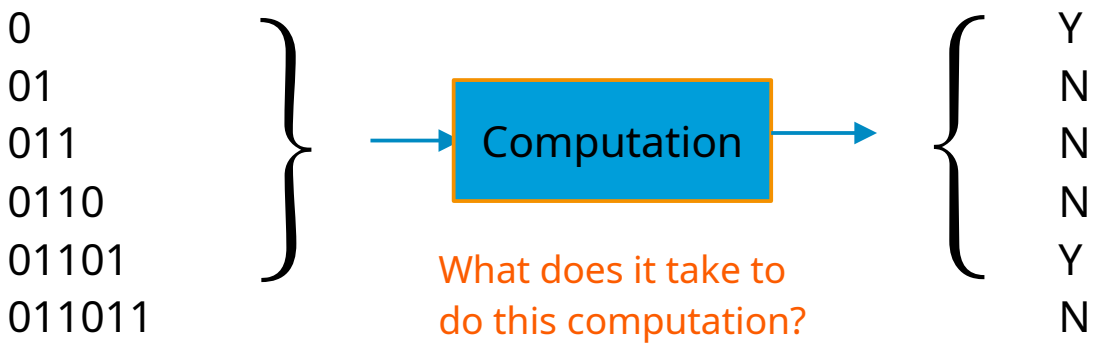
CS 208

Some Computational Problems

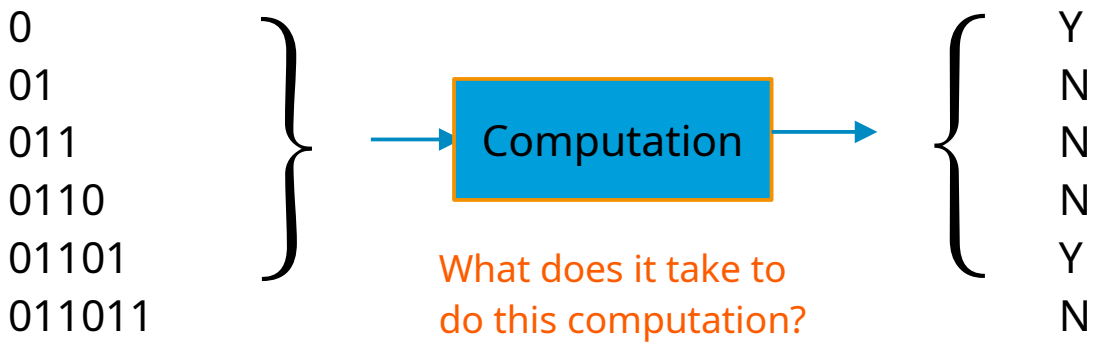
Supratik Chakraborty
Dept of CSE, IIT Bombay

Prob 1: Counting #1‘s modulo k

- Given a stream of 0‘s and 1‘s, output whether the count of 1‘s seen so far is a multiple of k
- Example: k = 3



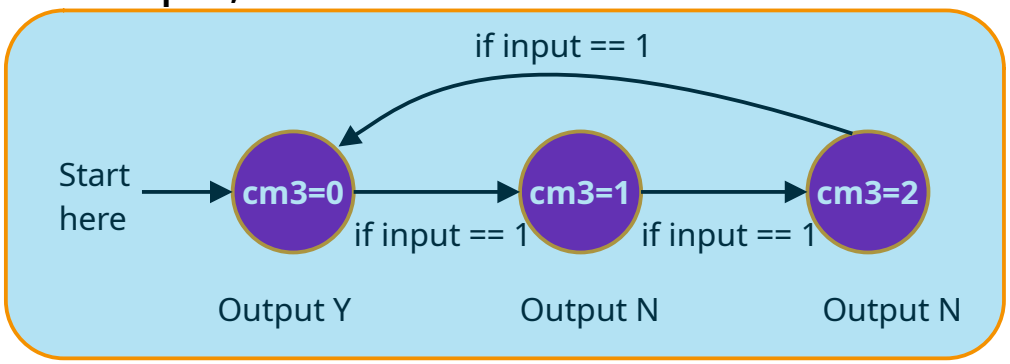
Prob 1: Counting #1‘s modulo k



Intuitive „code“ view

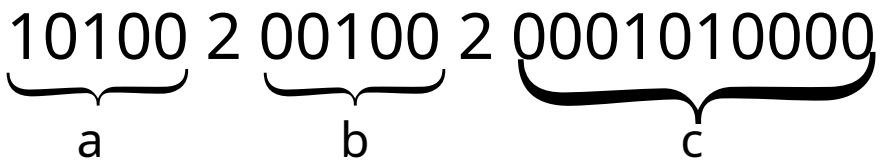
```
cm3 = 0;  // cm3 : count of 1‘s mod 3

Repeat until no symbols to read
   Read next input symbol;

 if (input == 1)
     cm3 = (cm3 + 1) mod 3;

 if (cm3 == 0) output Y
 else output N
```

„State transition“ (automata) view
Compact, mimics intuitive code view

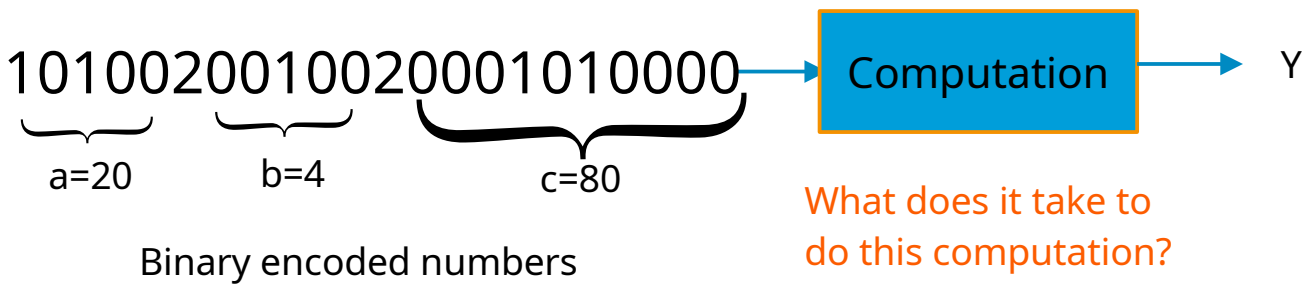


Prob 2: Checking for product triple

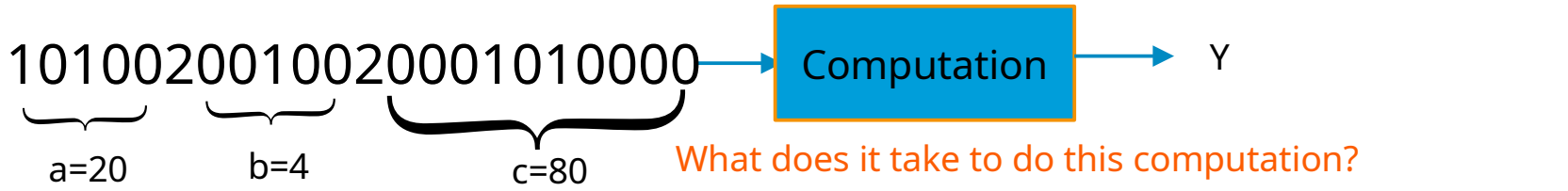
- Given a stream of 0‘s, 1‘s and 2‘s in following format



- Output **Y** if a x b = c and **N** otherwise
- Assume a and b are always k bits long and c is 2k bits long
- Example, k = 5



Prob 2: Checking for product triple

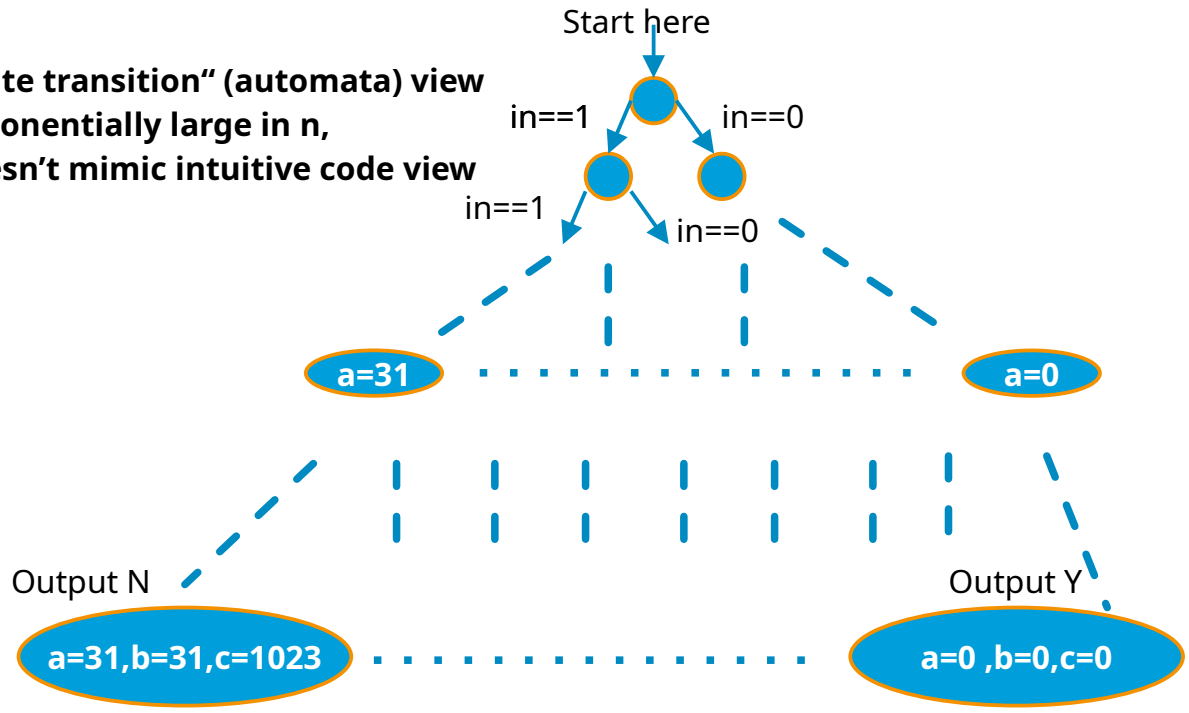


What does it take to do this computation?

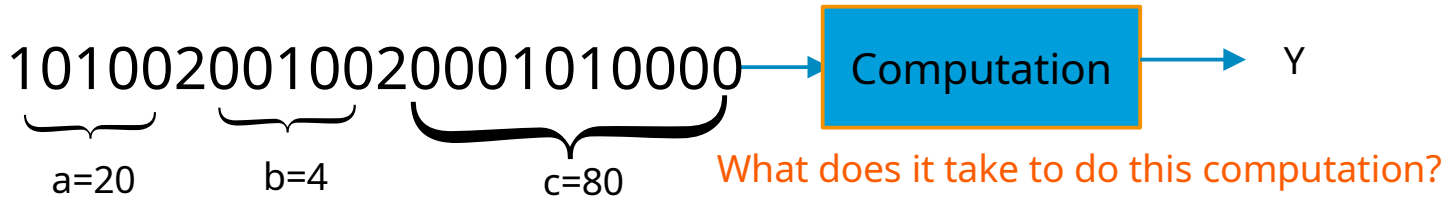
Intuitive „code“ view

```
Read first k symbols;
   Get value of a;
Read next k symbols after 2;
   Get value of b;
Read next k symbols after 3;
   Get value of c;

If (a x b == c) output Y
else output N
```

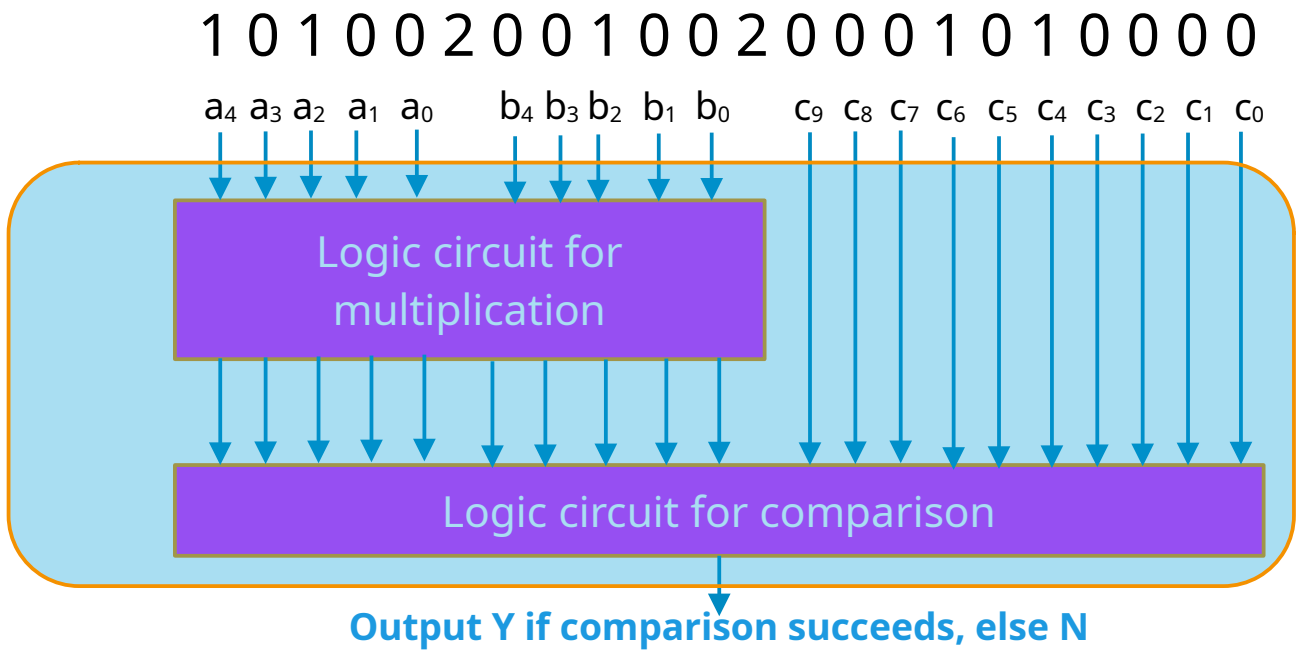
„State transition“ (automata) view
Exponentially large in n,
doesn’t mimic intuitive code view



Prob 2: Checking for product triple



Entire circuit representable by logic formula of size quadratic in n, mimics intuitive code view



Logic and Automata

- State transition centric view: Automata theory
- Logic formula centric view: Logical reasoning
- Both important, and also deeply connected
  - At times, it helps to view computation through lens of automata
    - See Prob 1 in earlier slides
  - At other times, through lens of logic
    - See Prob 2 in earlier slides

Propositional Logic: Syntax and Semantics

Supratik Chakraborty
IIT Bombay

Notation

- Variables: $p, q, r, \ldots$
  - Represent *propositions* or *declarative statements*
- Constants: $\top, \bot$
- Operators or connectives:
  - $\wedge,\ \vee,\ \neg,\ \rightarrow,\ \leftrightarrow,\ (,\ )$
    - We don't need all of them, but convenient to have them ...

Syntax of Propositional Logic

Rules for formuling formulas

- Every variable constitutes a formula
- The constants $\top$ and $\bot$ are formulae.
- If φ is a formula, so are $\neg\varphi$ and (φ)
- If φ_1 and φ_2 are formulas, so is $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$

Propositional formulas as strings and trees

- Alphabet (over which strings are constructed):
  - Set of variable names, e.g. $\{p_1, p_2, q_1, q_2, ...\}$
  - Set of constants $\{\top, \bot\}$
  - Fixed symbols $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow\}$
- Well-formed formula: string formed according to rules on prev. slide
  - $(p_1 \vee \neg q_2) \wedge (\neg p_2 \rightarrow (q_1 \leftrightarrow \neg p_1))$
  - $p_1 \rightarrow (p_2 \rightarrow (p_3 \rightarrow p_4))$
- Well-formed formulas can be represented using trees
  - Consider the rules on prev. slide as telling how to construct a tree bottom-up
  - Parse trees: Obviate the need for '(' and ')'

remove a need/difficulty

Semantics of Propositional Logic

Consider a formula φ with n variables. Let 0 represent "false" and 1 represent "true"

- $[\![\varphi]\!] : \{0,1\}^n \rightarrow \{0,1\}$
- **Semantics is a function**
  - Often represented in tabular form: Truth Table
- Indicates truth value of formula, given truth values of all variables

Rules of semantics

- $[\![\neg\varphi]\!] = 1$ iff $[\![\varphi]\!] = 0$.
- $[\![\varphi_1 \wedge \varphi_2]\!] = 1$ iff $[\![\varphi_1]\!] = [\![\varphi_2]\!] = 1$.
- $[\![\varphi_1 \vee \varphi_2]\!] = 1$ iff at least one of $[\![\varphi_1]\!]$ or $[\![\varphi_2]\!]$ evaluates to 1.
- $[\![\varphi_1 \rightarrow \varphi_2]\!] = 1$ iff at least one of $[\![\varphi_1]\!] = 0$ or $[\![\varphi_2]\!] = 1$.
- $[\![\varphi_1 \leftrightarrow \varphi_2]\!] = 1$ iff both $[\![\varphi_1 \rightarrow \varphi_2]\!] = 1$ and $[\![\varphi_2 \rightarrow \varphi_1]\!] = 1$.

C_1 C_2 ... C_5 Courses

D_1 D_2 ... D_5

S_1
S_2 slots
S_3
S_4

P_{ijk} = C_i in S_k of D_j

$5 \times 5 \times 4 = 100$ variables

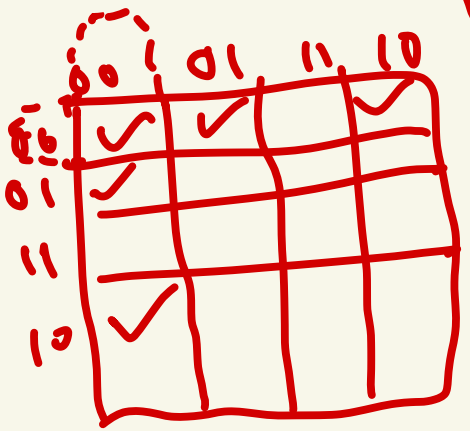
C_1 : $(P_{111} \rightarrow \neg P_{112} \wedge \neg P_{113} \wedge \neg P_{114})$

C_1 scheduled only one slot of Day 1

$\wedge (P_{112} \rightarrow \neg P_{111} \wedge \neg P_{113} \wedge \neg P_{114})$

$\wedge (P_{113} \rightarrow \neg P_{111} \wedge \neg P_{112} \wedge \neg P_{114})$

$\wedge (P_{114} \rightarrow \neg P_{111} \wedge \neg P_{112} \wedge \neg P_{113})$



$$\sum_{k=1}^{4} p_{ijk} \leq 1 \qquad \text{for } i,j \in \{1,2,\dots,s\}$$

$$\sum_{j=1}^{s} p_{ijk} \leq 1 \qquad i \in \{1,2\dots s\} \quad k \in \{1\dots 4\}$$

$$\sum_{k=1}^{4} \sum_{i=1}^{s} p_{ijk} \leq 3 \qquad j \in \{1\dots s\}$$

$$\sum_{i=1}^{s} p_{ijk} \leq 1 \qquad j \in \{1\dots s\} \quad k \in \{1\dots 4\}$$

$$\sum_{k=1}^{4} \sum_{j=1}^{s} p_{ijk} \leq 3$$

Propositional Logic: Syntax and Semantics

Supratik Chakraborty
IIT Bombay

Semantics of Propositional Logic

Consider a formula φ with n variables. Let 0 represent "false" and 1 represent "true"

- $[\![\varphi]\!] : \{0,1\}^n \rightarrow \{0,1\}$
- **Semantics is a function**
  - Often represented in tabular form: Truth Table
- Indicates truth value of formula, given truth values of all variables

Rules of semantics

- $[\![\neg\varphi]\!] = 1$ iff $[\![\varphi]\!] = 0$.
- $[\![\varphi_1 \wedge \varphi_2]\!] = 1$ iff $[\![\varphi_1]\!] = [\![\varphi_2]\!] = 1$.
- $[\![\varphi_1 \vee \varphi_2]\!] = 1$ iff at least one of $[\![\varphi_1]\!]$ or $[\![\varphi_2]\!]$ evaluates to 1.
- $[\![\varphi_1 \rightarrow \varphi_2]\!] = 1$ iff at least one of $[\![\varphi_1]\!] = 0$ or $[\![\varphi_2]\!] = 1$.
- $[\![\varphi_1 \leftrightarrow \varphi_2]\!] = 1$ iff both $[\![\varphi_1 \rightarrow \varphi_2]\!] = 1$ and $[\![\varphi_2 \rightarrow \varphi_1]\!] = 1$.

Semantics example

$[\![(p \vee s) \rightarrow (\neg q \leftrightarrow r)]\!]$

Semantics example

$[\![(p \vee s) \rightarrow (\neg q \leftrightarrow r)]\!]$

Write out the truth table:

p	q	r	s	$p \vee s$	$\neg q$	$\neg q \leftrightarrow r$	$(p \vee s) \rightarrow (\neg q \leftrightarrow r)$

Semantics example

$[\![(p \vee s) \rightarrow (\neg q \leftrightarrow r)]\!]$

Write out the truth table:

p	q	r	s	$p \vee s$	$\neg q$	$\neg q \leftrightarrow r$	$(p \vee s) \rightarrow (\neg q \leftrightarrow r)$
0	0	0	0	0	1	0	1

Semantics example

$[\![(p \vee s) \rightarrow (\neg q \leftrightarrow r)]\!]$

Write out the truth table:

p	q	r	s	$p \vee s$	$\neg q$	$\neg q \leftrightarrow r$	$(p \vee s) \rightarrow (\neg q \leftrightarrow r)$
0	0	0	0	0	1	0	1
0	0	0	1	1	1	0	0

Semantics example

$[\![(p \vee s) \rightarrow (\neg q \leftrightarrow r)]\!]$

Write out the truth table:

p	q	r	s	$p \vee s$	$\neg q$	$\neg q \leftrightarrow r$	$(p \vee s) \rightarrow (\neg q \leftrightarrow r)$
0	0	0	0	0	1	0	1
0	0	0	1	1	1	0	0
0	0	1	0	0	1	1	1
0	0	1	1	1	1	1	1
0	1	0	0	0	0	1	1
0	1	0	1	1	0	1	1
0	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	0
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	1
1	1	0	0	1	0	1	1
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Semantics-based terminology

A formula φ is

- **satisfiable** or **consistent** iff $[\![\varphi]\!] = 1$ for some assign of vars
  - E.g. $p \vee q$, $p \wedge q$

Semantics-based terminology

A formula φ is

- **satisfiable** or **consistent** iff $[\![\varphi]\!] = 1$ for some assign of vars
  - E.g. $p \vee q$, $p \wedge q$
  - Both φ and $\neg\varphi$ may be satisifable.

Semantics-based terminology

A formula φ is

- **satisfiable** or **consistent** iff $[\![\varphi]\!] = 1$ for some assign of vars
  - E.g. $p \vee q$, $p \wedge q$
  - Both φ and $\neg\varphi$ may be satisifable.
- **unsatisfiable** or **contradiction** iff $[\![\varphi]\!] = 0$ for all assign of vars
  - E.g. $p \wedge \neg p$,

Semantics-based terminology

A formula φ is

- **satisfiable** or **consistent** iff $[\![\varphi]\!] = 1$ for some assign of vars
  - E.g. $p \vee q$, $p \wedge q$
  - Both φ and $\neg\varphi$ may be satisifable.
- **unsatisfiable** or **contradiction** iff $[\![\varphi]\!] = 0$ for all assign of vars
  - E.g. $p \wedge \neg p$, $p \wedge ((p \rightarrow q) \wedge \neg q)$

Semantics-based terminology

A formula φ is

- **satisfiable** or **consistent** iff $[\![\varphi]\!] = 1$ for some assign of vars
  - E.g. $p \vee q$, $p \wedge q$
  - Both φ and $\neg\varphi$ may be satisifable.
- **unsatisfiable** or **contradiction** iff $[\![\varphi]\!] = 0$ for all assign of vars
  - E.g. $p \wedge \neg p$, $p \wedge ((p \rightarrow q) \wedge \neg q)$
  - At most one of φ and $\neg\varphi$ can be unsatisfiable.

Semantics-based terminology

A formula φ is

- **satisfiable** or **consistent** iff $[\![\varphi]\!] = 1$ for some assign of vars
  - E.g. $p \vee q$, $p \wedge q$
  - Both φ and $\neg\varphi$ may be satisifable.
- **unsatisfiable** or **contradiction** iff $[\![\varphi]\!] = 0$ for all assign of vars
  - E.g. $p \wedge \neg p$, $p \wedge ((p \rightarrow q) \wedge \neg q)$
  - At most one of φ and $\neg\varphi$ can be unsatisfiable.
- **valid** or **tautology** iff $[\![\varphi]\!] = 1$ for all assign of vars
  - E.g. $p \vee \neg p$,

Semantics-based terminology

A formula φ is

- **satisfiable** or **consistent** iff $[\![\varphi]\!] = 1$ for some assign of vars
  - E.g. $p \vee q$, $p \wedge q$
  - Both φ and $\neg\varphi$ may be satisifable.
- **unsatisfiable** or **contradiction** iff $[\![\varphi]\!] = 0$ for all assign of vars
  - E.g. $p \wedge \neg p$, $p \wedge ((p \rightarrow q) \wedge \neg q)$
  - At most one of φ and $\neg\varphi$ can be unsatisfiable.
- **valid** or **tautology** iff $[\![\varphi]\!] = 1$ for all assign of vars
  - E.g. $p \vee \neg p$, $p \vee \neg(p \wedge q)$

Semantics-based terminology

A formula φ is

- **satisfiable** or **consistent** iff $[\![\varphi]\!] = 1$ for some assign of vars
  - E.g. $p \vee q$, $p \wedge q$
  - Both φ and $\neg\varphi$ may be satisifable.
- **unsatisfiable** or **contradiction** iff $[\![\varphi]\!] = 0$ for all assign of vars
  - E.g. $p \wedge \neg p$, $p \wedge ((p \rightarrow q) \wedge \neg q)$
  - At most one of φ and $\neg\varphi$ can be unsatisfiable.
- **valid** or **tautology** iff $[\![\varphi]\!] = 1$ for all assign of vars
  - E.g. $p \vee \neg p$, $p \vee \neg(p \wedge q)$
  - φ is valid iff $\neg\varphi$ is unsatisfiable.

Semantics-based terminology

A formula φ

- **semantically entails** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ for all assign of vars, where 0 (false) $\preceq$ 1 (true)

Semantics-based terminology

A formula φ

- **semantically entails** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ for all assign of vars, where 0 (false) $\preceq$ 1 (true)
  - E.g. $p \models (p \vee q)$,

Semantics-based terminology

A formula φ

- **semantically entails** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ for all assign of vars, where 0 (false) $\preceq$ 1 (true)
  - E.g. $p \models (p \vee q)$, $\neg p \models (p \rightarrow q)$

Semantics-based terminology

A formula φ

- **semantically entails** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ for all assign of vars, where 0 (false) $\preceq$ 1 (true)
  - E.g. $p \models (p \vee q)$, $\neg p \models (p \rightarrow q)$
  - Denoted $\varphi \models \varphi_1$
  - Equivalently, $\varphi \rightarrow \varphi_1$ is valid

Semantics-based terminology

A formula φ

- **semantically entails** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ for all assign of vars, where 0 (false) $\preceq$ 1 (true)
  - E.g. $p \models (p \vee q)$, $\neg p \models (p \rightarrow q)$
  - Denoted $\varphi \models \varphi_1$
  - Equivalently, $\varphi \rightarrow \varphi_1$ is valid
- is **semantically equivalent** to φ_1 iff $\varphi \models \varphi_1$ and $\varphi_1 \models \varphi$
  - Identical truth tables, (obviously) an equivalence relation

Semantics-based terminology

A formula φ

- **semantically entails** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ for all assign of vars, where 0 (false) $\preceq$ 1 (true)
  - E.g. $p \models (p \vee q)$, $\neg p \models (p \rightarrow q)$
  - Denoted $\varphi \models \varphi_1$
  - Equivalently, $\varphi \rightarrow \varphi_1$ is valid
- is **semantically equivalent** to φ_1 iff $\varphi \models \varphi_1$ and $\varphi_1 \models \varphi$
  - Identical truth tables, (obviously) an equivalence relation
  - E.g. $p \rightarrow q$ and $\neg p \vee q$

Semantics-based terminology

A formula φ

- **semantically entails** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ for all assign of vars, where 0 (false) $\preceq$ 1 (true)
  - E.g. $p \models (p \vee q)$, $\neg p \models (p \rightarrow q)$
  - Denoted $\varphi \models \varphi_1$
  - Equivalently, $\varphi \rightarrow \varphi_1$ is valid
- is **semantically equivalent** to φ_1 iff $\varphi \models \varphi_1$ and $\varphi_1 \models \varphi$
  - Identical truth tables, (obviously) an equivalence relation
  - E.g. $p \rightarrow q$ and $\neg p \vee q$
  - Equivalently, $\varphi \leftrightarrow \varphi_1$ is valid

Semantics-based terminology

A formula φ

- **semantically entails** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ for all assign of vars, where 0 (false) $\preceq$ 1 (true)
  - E.g. $p \models (p \vee q)$, $\neg p \models (p \rightarrow q)$
  - Denoted $\varphi \models \varphi_1$
  - Equivalently, $\varphi \rightarrow \varphi_1$ is valid
- is **semantically equivalent** to φ_1 iff $\varphi \models \varphi_1$ and $\varphi_1 \models \varphi$
  - Identical truth tables, (obviously) an equivalence relation
  - E.g. $p \rightarrow q$ and $\neg p \vee q$
  - Equivalently, $\varphi \leftrightarrow \varphi_1$ is valid
- is **equisatisfiable to** φ_1 iff either both φ and φ_1 are satisfiable or both are unsatisfiable
  - E.g. $p \wedge q$ and $r \vee s$

Semantics-based terminology

A formula φ

- **semantically entails** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ for all assign of vars, where 0 (false) $\preceq$ 1 (true)
  - E.g. $p \models (p \vee q)$, $\neg p \models (p \rightarrow q)$
  - Denoted $\varphi \models \varphi_1$
  - Equivalently, $\varphi \rightarrow \varphi_1$ is valid
- is **semantically equivalent** to φ_1 iff $\varphi \models \varphi_1$ and $\varphi_1 \models \varphi$
  - Identical truth tables, (obviously) an equivalence relation
  - E.g. $p \rightarrow q$ and $\neg p \vee q$
  - Equivalently, $\varphi \leftrightarrow \varphi_1$ is valid
- is **equisatisfiable to** φ_1 iff either both φ and φ_1 are satisfiable or both are unsatisfiable
  - E.g. $p \wedge q$ and $r \vee s$
  - Semantic equivalence implies equisatisfiability, not vice versa

Fun with semantics!

Two syntactically different formulas may be semantically equivalent!

E.g. $\varphi_1 : p \rightarrow (q \rightarrow r)$, $\varphi_2 : (p \wedge q) \rightarrow r$, and $\varphi_3 : (q \wedge \neg r) \rightarrow \neg p$

Fun with semantics!

Two syntactically different formulas may be semantically equivalent!

E.g. $\varphi_1 : p \to (q \to r)$, $\varphi_2 : (p \wedge q) \to r$, and $\varphi_3 : (q \wedge \neg r) \to \neg p$

Truth table way of checking equivalence (or not):

p	q	r	$q \to r$	$p \wedge q$	$q \wedge \neg r$	$\neg p$	φ_1	φ_2	φ_3

Fun with semantics!

Two syntactically different formulas may be semantically equivalent!

E.g. $\varphi_1 : p \rightarrow (q \rightarrow r)$, $\varphi_2 : (p \wedge q) \rightarrow r$, and $\varphi_3 : (q \wedge \neg r) \rightarrow \neg p$

Truth table way of checking equivalence (or not):

p	q	r	$q \rightarrow r$	$p \wedge q$	$q \wedge \neg r$	$\neg p$	φ_1	φ_2	φ_3
0	0	0	1	0	0	1	1	1	1

Fun with semantics!

Two syntactically different formulas may be semantically equivalent!

E.g. $\varphi_1 : p \to (q \to r)$, $\varphi_2 : (p \wedge q) \to r$, and $\varphi_3 : (q \wedge \neg r) \to \neg p$

Truth table way of checking equivalence (or not):

p	q	r	$q \to r$	$p \wedge q$	$q \wedge \neg r$	$\neg p$	φ_1	φ_2	φ_3
0	0	0	1	0	0	1	1	1	1
0	0	1	1	0	0	1	1	1	1
0	1	0	0	0	1	1	1	1	1
0	1	1	1	0	0	1	1	1	1
1	0	0	1	0	0	0	1	1	1
1	0	1	1	0	0	0	1	1	1
1	1	0	0	1	1	0	0	0	0
1	1	1	1	1	0	0	1	1	1

Fun with semantics!

Two syntactically different formulas may be semantically equivalent!

E.g. $\varphi_1 : p \to (q \to r)$, $\varphi_2 : (p \wedge q) \to r$, and $\varphi_3 : (q \wedge \neg r) \to \neg p$

Truth table way of checking equivalence (or not):

p	q	r	$q \to r$	$p \wedge q$	$q \wedge \neg r$	$\neg p$	φ_1	φ_2	φ_3
0	0	0	1	0	0	1	1	1	1
0	0	1	1	0	0	1	1	1	1
0	1	0	0	0	1	1	1	1	1
0	1	1	1	0	0	1	1	1	1
1	0	0	1	0	0	0	1	1	1
1	0	1	1	0	0	0	1	1	1
1	1	0	0	1	1	0	0	0	0
1	1	1	1	1	0	0	1	1	1

Works, but doesn't scale! 2^n rows for n propositions

Semantic reasoning without truth tables?

Yes, **proof rules**

TUTORIAL - 1

1b) (i) $(p \to q) \wedge (q \to r) \to (p \to r)$ True

$\equiv \sim((\sim p \vee q) \wedge (\sim q \vee r)) \vee (\sim p \vee r)$

$(((p \wedge \sim q) \vee (q \wedge \sim r)) \vee (\sim p \vee r)$

(ii) $(p \to q) \wedge (q \to r) \to (r \to p)$ False

(iii) Yes

2b) $P_{ii} \to$ always true

$(P_{ij} \wedge P_{jk}) \to (P_{ik})$ $i \neq j$ $j \neq k$ $i \neq k$

$\sim(P_{ij} \wedge P_{ji})$ $i \neq j$

for $1 \leq i, j, k \leq n$

$Q_R = (P_{11} \wedge P_{22} \wedge \dots \wedge P_{nn}) = \bigwedge_{i=1}^{n} P_{ii}$

$\phi_{m,i} \Rightarrow$ if x_i = maximal

$\phi_m = \phi_{m,1} \vee \phi_{m,2} \vee \dots \vee \phi_{m,n} = \bigvee_{i=1}^{m} \phi_i$

$x_i \leq x_j$ happens only when $i = j$

If $j = i$ $x_i \overset{not}{=} x_j$

$\phi_{m,i} = \bigwedge_{j \neq i} \neg P_{ij} = (\neg P_{i1} \wedge \neg P_{i2} \wedge \dots \wedge \neg P_{i,i-1} \wedge \neg P_{i,i+1} \wedge \dots)$

$\bigvee_{i=1}^{m} \bigwedge_{j=1} \neg P_{ij}$

$\phi \Longrightarrow \phi_m$

ϕ partial order

$x_1 \to s_{11}$

$s_{11} \to x_1$

$x_2 \to s_{12}$

$s_{ij} \wedge x_{i+1} \to s_{i+1\,j+1}$

$\exists$ s_{ij} $j > i$ ✗

$s_{ij} \to s_{i+1,j}$

s_{ij} → $O(nk)$

$\neg x_{i+1} \wedge s_{ij} \to s_{i+1,j}$

redundant

$\neg s_{m,k+1}$

Proof Rules — Normal forms

Introduction

Premises

$\phi_1 \; \phi_2$

Inferences

$\phi_1 \wedge \phi_2$

$\wedge$ $\wedge_i$

Elimination

$$\frac{\phi_1 \wedge \phi_2}{\phi_1} \wedge_e \qquad \frac{\phi_1 \wedge \phi_2}{\phi_2}$$

$\vee$

$$\frac{\phi_1}{\phi_1 \vee \phi_2} \vee_i \qquad \frac{\phi_2}{\phi_1 \vee \phi_2}$$

$\vee_e$

Another formula:

$$\frac{\phi_1 \vee \phi_2 \quad , \; \phi_1 \to \phi_3 \quad , \; \phi_2 \to \phi_3}{\phi_3}$$

$\to_i$

$$\frac{\boxed{\begin{array}{c}\phi_1 \\ \vdots \\ \phi_2\end{array}}}{\phi_1 \to \phi_2}$$

$\to_e$ Modus Ponens

$$\frac{\phi_1 \to \phi_2 \quad \phi_1}{\phi_2}$$

Eg:

$$\frac{\phi_1 \wedge (\phi_2 \wedge \phi_3)}{\phi_1} \wedge_{e_1}$$

$$\frac{\phi_1 \wedge (\phi_2 \wedge \phi_3)}{(\phi_2 \wedge \phi_3)} \wedge_{e_2} \qquad \frac{(\phi_2 \wedge \phi_3)}{\phi_2} \wedge_{e_1}$$

$$\frac{\phi_1 \wedge (\phi_2 \wedge \phi_3)}{\phi_2 \wedge \phi_3} \wedge_{e_2} \qquad \frac{\phi_2 \wedge \phi_3}{\phi_3} \wedge_{e_2}$$

$\phi_1 \wedge \phi_2$ → $(\phi_1 \wedge \phi_2) \wedge \phi_3$ $\wedge_i$

$\neg_i$ ϕ ⋮ $\bot$ → false / $\neg\phi$

$\neg_e$ ϕ $\quad$ $\neg\phi$ / $\bot$

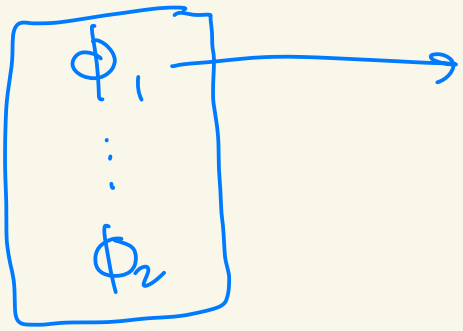
$\bot$ / φ

$\neg\neg_i$ ϕ / $\neg\neg\phi$

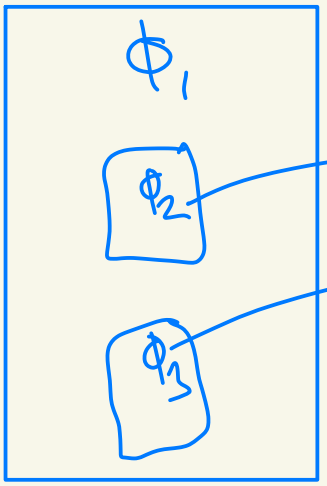
$\neg\neg_e$ $\neg\neg\phi$ / ϕ

$\phi \vee \neg\phi$

$\top$
⋮
$\phi \vee \neg\phi$



Assume that ϕ_1 is true
now see if ϕ_2 can
be achieved from ϕ_1

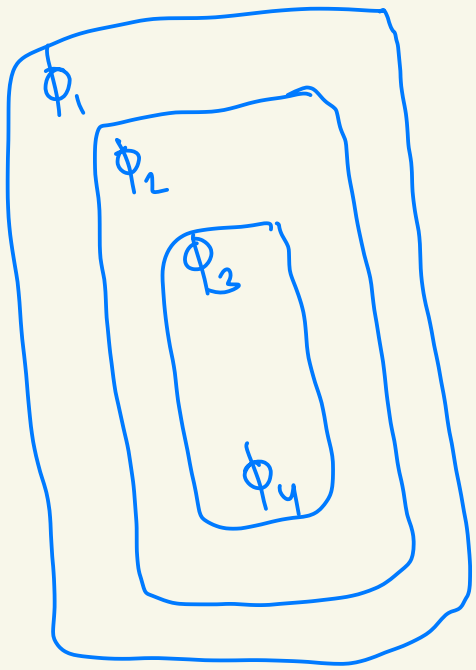


$\phi_1 \& \phi_2$ true
$\phi_1 \& \phi_3$ true

$P \rightarrow --P$

assume
  P .. (1)
  assume
    $\neg P$ .. (2)
    F Not_e using 1,2 ...(3)
  therefore
    $--P$ Not_i using 2,3 ...(4)
therefore
  $P \rightarrow --P$ $\rightarrow$i using (1,2,3,4) .. (5)

$\phi_1, \phi_2, \phi_3 \vdash \phi_4$



$\vdash \varphi \vee \neg\varphi$

$\varphi_0, \varphi_1 \vdash \varphi_2$

φ_0 --- premise

φ_1 --- premise

⋮

φ_2 --- conclusion

$\Sigma \vdash \varphi$

Σ — set of formulas

$\vdash$ — syntactic entailment — looking at the formulas

$\Sigma \models \varphi$

$\models$ — semantic entailment — looking at truth table

syntactic entailment $\longrightarrow$ semantic entailment

Does the reverse also hold true?

$\Sigma \vdash \varphi$ implies $\Sigma \models \varphi$: Soundness of proof rules

$\Sigma \models \varphi$ implies$^{?}$ $\Sigma \vdash \varphi$: Completeness of proof rules

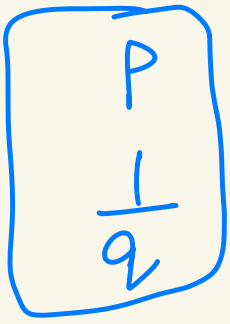
$\Sigma = \{p \to q, \neg q\} \models \neg p$

p	q	$p \to q$	$\neg q$	$\neg p$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	1	0	0

$\neg p, \neg q \vdash ? \; p \to q$

$\neg p$

$\neg q$



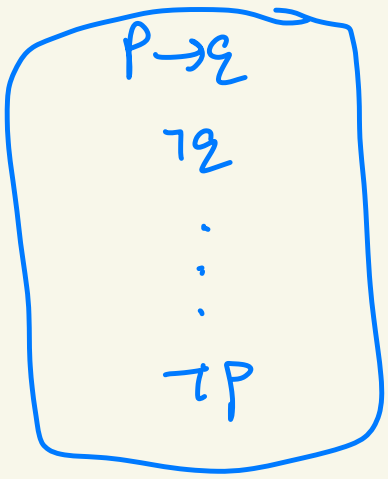
$p \to q$

$\neg p$

$\neg q$

$\vdots$

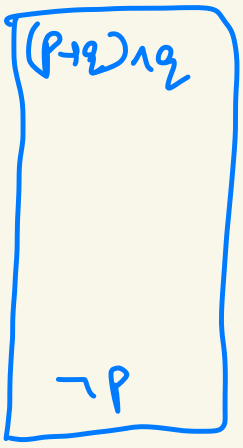
$p \to q$



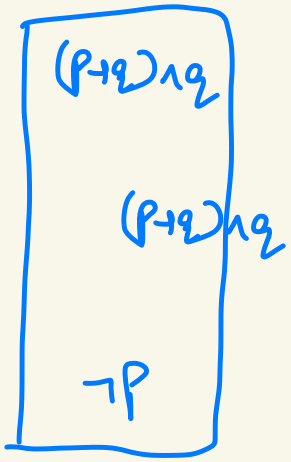
$(p \to q) \to \neg p$

for each row of the truth table, we can do this

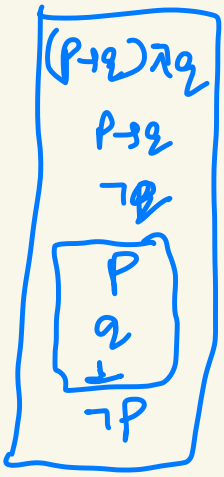
$\neg p$
$\neg q$



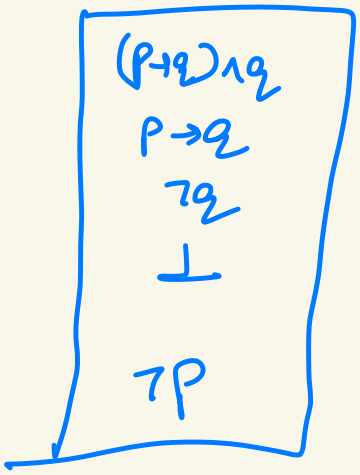
$\neg p$
q



p
$\neg q$



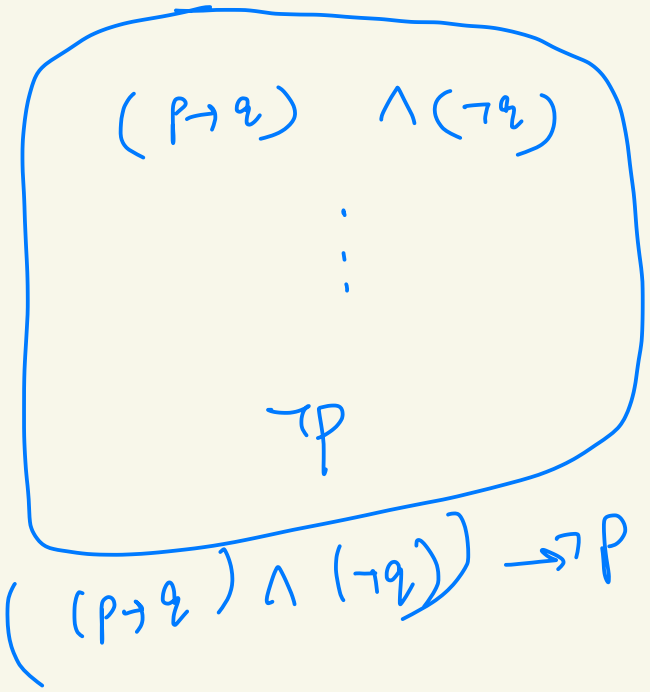
$\neg p$
q



$$(p \to q) \wedge \neg q \to \neg p$$

To prove $p \to q, \neg q \vdash \neg p$

First show $\vdash (p \to q) \wedge (\neg q) \to \neg p$



Step-2 Construct a new proof using above other proof

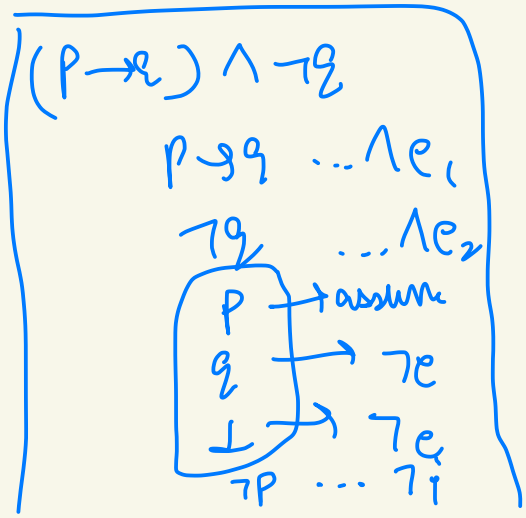
$p \to q$

$\neg q$

⋮

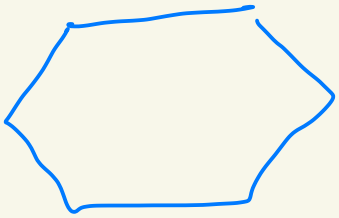
To show $\vdash ((p \to q) \wedge \neg q) \to \neg p$

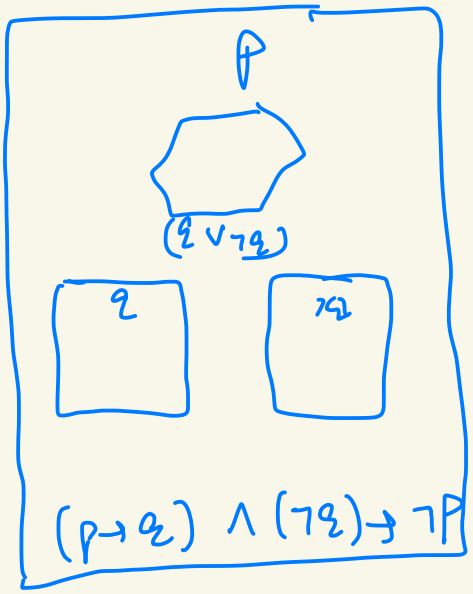
Many proofs possible

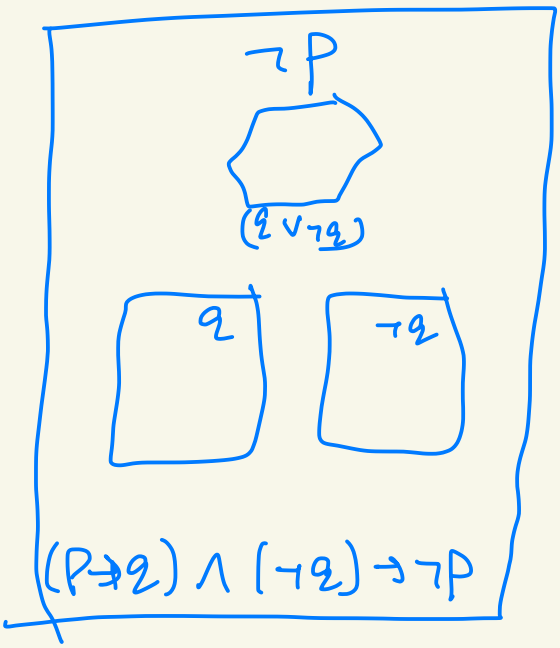


mimic each row of truth table

Brute force approach







$$\Big((p \to q) \wedge (\neg q) \to \neg p \Big) \; \forall_{\mathcal{C}}$$

$\vdash \varphi$: φ is a valid formula

$\vdash \neg\varphi$: ϕ = contradiction

$$\vdash \neg p \to \neg (p \wedge q)$$

$$\vdash (p \to (q \to (p \wedge q)))$$

Satisfiability

ϕ is valid iff $\neg\phi$ is not satisfiable.

$$\phi_1 \wedge (\phi_2 \vee \phi_3) \quad \not\vdash \quad (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)$$

$$\vdash \phi_1 \wedge (\phi_2 \vee \phi_3) \to ((\phi_1 \vee \phi_2) \vee (\phi_1 \wedge \phi_3))$$

$$\phi_1 \wedge (\phi_2 \vee \phi_3) \longleftrightarrow (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)$$

$$\phi_1 \vee (\phi_2 \wedge \phi_3) \longleftrightarrow (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$$

$\wedge, \vee, \neg, \rightarrow$

$(\phi_1 \rightarrow \phi_2) \leftrightarrow (\neg\phi_1 \vee \phi_2)$

Demorgan's rules

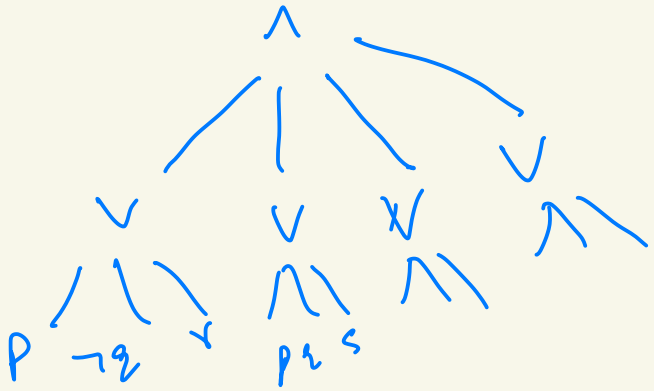
$\neg(\phi_1 \wedge \phi_2) \leftrightarrow \neg\phi_1 \vee \neg\phi_2$

$\neg[\phi_1 \vee \phi_2) \leftrightarrow \neg\phi_1 \wedge \neg\phi_2$

$\neg(p \vee \neg(q \wedge (p \vee \neg r)) \vee \neg q$

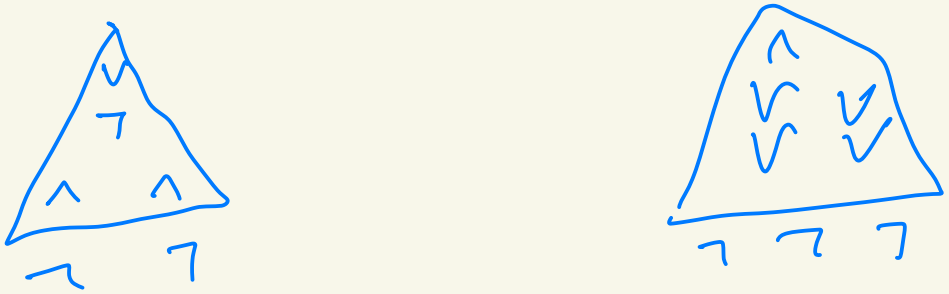
$(\neg p \wedge (q \wedge (p \vee \neg r))) \vee \neg q$

Negation Normal Form (NNF)



Conjunctive Normal Form

Given a DAG representing a prop. logic formula with only $\wedge, \vee, \neg$ nodes, can we efficiently get a DAG representing a semantically equivalent NNF formula?



NNF → not outside only literals, internal nodes have $\vee, \wedge$

Tutorial 2

---

1. A *literal* is a propositional variable or its negation. A *clause* is a disjunction of literals such that a literal and its negation are not both in the same clause, for any literal. Similarly, a *cube* is a conjunction of literals such that a literal and its negation are not both in the same cube, for any literal. A propositional formula is said to be in *Conjunctive Normal Form (CNF)* if it is a conjunction of clauses. A formula is said to be in *Disjunctive Normal Form (DNF)* if it is a disjunction of cubes.

   Let P, Q, R, S, T be propositional variables. An example DNF formula is $(P \wedge \neg Q) \vee (\neg P \wedge Q)$, and an example CNF formula is $(P \vee Q) \wedge (\neg P \vee \neg Q)$. Are they semantically equivalent? Yes

   For the propositional formula $(P \vee T) \rightarrow (Q \vee \neg R) \vee \neg(S \vee T)$, find semantically equivalent formulas in CNF and DNF. Show all intermediate steps in arriving at the final result.

   Hint: Use the following semantic equivalences, where we use $\varphi \Leftrightarrow \psi$ to denote semantic equivalence of φ and ψ:

   - $\varphi_1 \rightarrow \varphi_2 \;\Leftrightarrow\; (\neg\varphi_1 \vee \varphi_2)$
   - *Distributive laws:*
     - $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \;\Leftrightarrow\; (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$
     - $\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \;\Leftrightarrow\; (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$
   - *De Morgan's laws*
     - $\neg(\varphi_1 \wedge \varphi_2) \;\Leftrightarrow\; (\neg\varphi_1 \vee \neg\varphi_2)$
     - $\neg(\varphi_1 \vee \varphi_2) \;\Leftrightarrow\; (\neg\varphi_1 \wedge \neg\varphi_2)$

$(P \wedge \neg Q) \vee (\neg P \wedge Q)$

$((P \wedge \neg Q) \vee \neg P) \wedge ((P \wedge \neg Q) \vee Q)$

$(P \vee \neg P) \wedge (\neg Q \vee \neg P) \wedge (P \vee Q) \wedge (\neg Q \vee Q)$

T T

$(P \vee Q) \wedge (\neg P \vee \neg Q)$

$$(P \vee T) \rightarrow (Q \vee \neg R) \vee \neg(S \vee T)$$

$\neg(P \vee T) \vee (Q \vee \neg R) \vee \neg(S \vee T)$

$(\neg P \wedge \neg T) \vee (Q \vee \neg R) \vee (\neg S \wedge \neg T)$

$(\neg P \wedge \neg T) \vee (Q) \vee (\neg R) \vee (\neg S \wedge \neg T)$ → DNF form

$(\neg P \wedge \neg T) \vee (\neg S \wedge \neg T) \vee Q \vee \neg R$

$((\neg P \vee \neg S) \wedge (\neg T \vee \neg S) \wedge (\neg T \vee \neg T) \wedge (\neg P \vee \neg T)) \vee Q \vee \neg R$

$(\neg T)$

distribute this then this

CNF form

2. Consider the parity function, PARITY : $\{0,1\}^n \mapsto \{0,1\}$, where PARITY evaluates to 1 if and only if an odd number of inputs is 1.

   1. Prove that any CNF representation of PARITY must have n literals (from distinct variables) in every clause.

      [Hint: Take a clause, and suppose the variable v is missing from it. What happens when you flip v?]

   2. Prove that any CNF representation of PARITY must have 2^{n-1} clauses (all the clauses are assumed to be distinct).

      [Hint: What is the relation between CNF/DNF and truth tables?]

   PARITY is ubiquitous across Computer Science; for example, in Coding Theory (see Hadamard codes), Cryptography (see the Goldreich-Levin theorem), and discrete Fourier Analysis (yes, that is a thing)!

CNF representation: $C_1 \wedge C_2 \wedge \dots \wedge C_k$

where C_i are clauses $i \in \{1, 2, \dots, k\}$

Let's suppose the representation is $\underbrace{000\dots\tilde{0}}_{n \text{ times}}$ literal to be considered

$C_i = (x_1 \vee x_2 \vee \dots \vee x_n)$

If x_n is flipped C_i will be flipped hence the result. If x_n is mission, no change but parity can't be same

Hence, contradiction!

One possible representation of CNF $= \underbrace{(x_1 \oplus x_2 \oplus \dots \oplus x_n)}_{\text{has } 2^{n-1} \text{ clauses}}$

3. We have already defined CNF and DNF formulas in Question 1. Recall the definition of *equisatisfiable* formulas taught in class. A **stronger notion of equisatisfiability** is as follows: A formula F with variables $\{f_1, f_2 \cdots f_n\}$ and a formula G with variables $\{f_1, f_2 \cdots f_n, g_1, g_2 \cdots g_m\}$ are **strongly-equisatisfiable** if:

- For any assignment to the f_is which makes F evaluate to true there exists an assignment to the g_is such that G evaluates to true under this assignment with the same assignment to the f_is.

$$AND$$

- For any assignment setting G to true, the assignment when restricted to f_is makes F evaluate to true.

Why does this help us? A satisfying assignment for G tells us a satisfying assignment for F, and if we somehow discover that G has no satisfying assignments, then we know that F also has no satisfying assignments. In other words, the satisfiability of F can be judged using the satisfiability of G. Satisifiability is a central problem in computer science (for reasons you will see in CS218 later) and such a reduction is very valuable. In particular, for a formula F in k-CNF we look for a formula G that has a small number of literals in each clause, and the total number of clauses itself is not too large - ideally a linear factor of k as compared to F.

Now consider an r-CNF formula, i.e a CNF formula with r literals per clause. To keep things simple, suppose $r = 2^k$, for some $k > 0$. We will start with the simplest case: a 2^k-CNF formula containing a single clause $C_k = p_0 \vee p_1 \vee p_2 \cdots p_{2^k-1}$ with 2^k literals.

1. Let's try using 'selector variables' to construct another formula which is strongly-equisatisfiable with C_k. The idea is to use these selector variables to 'select' which literals in our clause are allowed to be false. Use k selector variables $s_1, s_2, s_3 \cdots s_k$ and write a CNF formula that is **strongly-equisatisfiable with** C_k, such that each clause has size $O(k)$.
   [Hint: If you consider the disjunction of $\tilde{s}_1 \vee \tilde{s}_2 \vee \tilde{s}_3 \cdots \tilde{s}_k$, where $\tilde{s}_i$ denotes either s_i or $\neg s_i$, it is true in all but one of the 2^k possible assignments to the s_i variables]
2. What is the number of clauses in the CNF constructed in the above sub-question? How small can you make the clauses by repeatedly applying this technique?
3. Consider a k-CNF formula with n clauses. When minimizing the clause size using the above technique repeatedly, what is the (asymptotic) blowup factor in the number of clauses? Observe that the factor is almost linear, upto some logarithmic factors.
   [Hint: Apply the above reduction repeatedly and see how much net blowup occurs]
4. Can you think of a smarter way to do the reduction that only requires linear blowup?
   [Hint: Consider using different and more auxiliary variables (like the 'selector variables' above) to reduce the size of clauses.]

4 **[Take-away question – solve in your rooms]**

In this question we will view the set of assignments satisfying a set of propositional formulae as a language and examine some properties of such languages.

Let $\mathbf{P}$ denote a countably infinite set of propositional variables $p_0, p_1, p_2, \ldots$. Let us call these variables positional variables. Let Σ be a countable set of formulae over these positional variables. Every assignment $\alpha : \mathbf{P} \to \{0,1\}$ to the positional variable can be uniquely associated with an infinite bitstring w, where $w_i = \alpha(p_i)$. The language defined by Σ - denoted by $L(\Sigma)$ - is the set of bitstrings w for which the corresponding assignment α, that has $\alpha(p_i) = w_i$ for each natural i, satisfies Σ, that is, for each formula $F \in \Sigma$, $\alpha \models F$. In this case, we say that $\alpha \models \Sigma$. Let us call the languages definable this way PL-definable languages.

(a) Show that PL-definable languages are closed under countable intersection, ie if $\mathcal{L}$ is a countable set of PL-languages, then $\bigcap_{L \in \mathcal{L}} L$ is also PL-definable.

(b) Show that PL-definable languages are closed under finite union, ie if $\mathcal{L}$ is a finite set of PL-languages, then $\bigcup_{L \in \mathcal{L}} L$ is also PL-definable.

[Hint: Try proving that the union of two PL-definable languages is PL-definable. The general case can then be proven via induction. If $\mathbf{F} = \{F_1, F_2, \ldots\}$ and $\mathbf{G} = \{G_1, G_2, \ldots\}$ are two countable sets of formulae, then an infinite bitstring $w \in L(\mathbf{F}) \cup L(\mathbf{G})$ if and only if either $w \models$ every F_i or $w \models$ every G_i.]

Tutorial 2

---

1. A *literal* is a propositional variable or its negation. A *clause* is a disjunction of literals such that a literal and its negation are not both in the same clause, for any literal. Similarly, a *cube* is a conjunction of literals such that a literal and its negation are not both in the same cube, for any literal. A propositional formula is said to be in *Conjunctive Normal Form (CNF)* if it is a conjunction of clauses. A formula is said to be in *Disjunctive Normal Form (DNF)* if it is a disjunction of cubes.

   Let P, Q, R, S, T be propositional variables. An example DNF formula is $(P \wedge \neg Q) \vee (\neg P \wedge Q)$, and an example CNF formula is $(P \vee Q) \wedge (\neg P \vee \neg Q)$. Are they semantically equivalent?

   For the propositional formula $(P \vee T) \rightarrow (Q \vee \neg R) \vee \neg(S \vee T)$, find semantically equivalent formulas in CNF and DNF. Show all intermediate steps in arriving at the final result.

   Hint: Use the following semantic equivalences, where we use $\varphi \Leftrightarrow \psi$ to denote semantic equivalence of φ and ψ:

   - $\varphi_1 \rightarrow \varphi_2 \;\Leftrightarrow\; (\neg\varphi_1 \vee \varphi_2)$
   - *Distributive laws:*
     - $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \;\Leftrightarrow\; (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$
     - $\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \;\Leftrightarrow\; (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$
   - *De Morgan's laws*
     - $\neg(\varphi_1 \wedge \varphi_2) \;\Leftrightarrow\; (\neg\varphi_1 \vee \neg\varphi_2)$
     - $\neg(\varphi_1 \vee \varphi_2) \;\Leftrightarrow\; (\neg\varphi_1 \wedge \neg\varphi_2)$

Solution: Propositional formula $(P \vee T) \rightarrow (Q \vee \neg R) \vee \neg(S \vee T)$.

Transformation steps

1. Eliminate Implications:

$$\neg(P \vee T) \vee \big((Q \vee \neg R) \vee \neg(S \vee T)\big)$$

2. Apply De Morgan's Laws:

$$(\neg P \wedge \neg T) \vee \big((Q \vee \neg R) \vee (\neg S \wedge \neg T)\big)$$

Transformation into DNF
Distribute Conjunction over Disjunction:

$$(\neg P \wedge \neg T) \vee (Q) \vee (\neg R) \vee (\neg S \wedge \neg T)$$

Transformation into CNF
Distribute Disjunction over Conjunction:

$$(\neg P \vee Q \vee \neg R \vee \neg S) \wedge (\neg P \vee Q \vee \neg R \vee \neg T)$$
$$\wedge\, (\neg T \vee Q \vee \neg R \vee \neg S) \wedge (\neg T \vee Q \vee \neg R \vee \neg T)$$

2. Consider the parity function, PARITY : $\{0,1\}^n \mapsto \{0,1\}$, where PARITY evaluates to 1 if and only if an odd number of inputs is 1. In all of the CNFs below, we assume that each clause contains any variable at most once, i.e. no clause contains expressions of the form $p \wedge \neg p$ or $p \vee \neg p$. Furthermore, all clauses are assumed to be distinct.

   1. Prove that any CNF representation of PARITY must have n literals (from distinct variables) in every clause.

      [Hint: Take a clause, and suppose the variable v is missing from it. What happens when you flip v?]

   2. Prove that any CNF representation of PARITY must have $\geq 2^{n-1}$ clauses.

      [Hint: What is the relation between CNF/DNF and truth tables?]

   PARITY is ubiquitous across Computer Science; for example, in Coding Theory (see Hadamard codes), Cryptography (see the Goldreich-Levin theorem), and discrete Fourier Analysis (yes, that is a thing)! PARITY is ubiquitous across Computer Science; for example, in Coding Theory (see Hadamard codes), Cryptography (see the Goldreich-Levin theorem), and discrete Fourier Analysis (yes, that is a thing)!

Solution: Let

$$\text{PARITY} := \bigwedge_{i=1}^{m} \left(\vee_{j=1}^{n_i} \ell_{ij} \right)$$

be the CNF representation of PARITY. We want to prove that $n_i = n$ for every i, and $m \geq 2^{n-1}$.

1. Suppose $n_i < n$ for some i. Negate the entire formula to convert the CNF into a DNF. Now, choose an assignment of literals in the i^{th} cube (of the DNF) such that the cube, and hence the whole DNF formula, evaluates to 1. Now, consider a variable v, which doesn't appear in the i^{th} cube, and flip its value. The LHS then becomes 0. However, the i^{th} clause stays 1, leading to a contradiction.

2. Once again, consider the DNF. Note that a conjunction of n literals is satisfied by a unique assignment of variables, and thus, the clauses in the DNF actually encode the assignments that satisfy the formula. Since PARITY is satisfied by 2^{n-1} clauses, we're done.

3. We have already defined CNF and DNF formulas in Question 1. Recall the definition of *equisatisfiable* formulas taught in class. A **stronger notion of equisatisfiability** is as follows: A formula F with variables $\{f_1, f_2 \cdots f_n\}$ and a formula G with variables $\{f_1, f_2 \cdots f_n, g_1, g_2 \cdots g_m\}$ are **strongly-equisatisfiable** if:

   - For any assignment to the f_is which makes F evaluate to true there exists an assignment to the g_is such that G evaluates to true under this assignment with the same assignment to the f_is.

     $$AND$$

   - For any assignment setting G to true, the assignment when restricted to f_is makes F evaluate to true.

   Why does this help us? A satisfying assignment for G tells us a satisfying assignment for F, and if we somehow discover that G has no satisfying assignments, then we know that F also has no satisfying assignments. In other words, the satisfiability of F can be judged using the satisfiability of G. Satisfiability is a central problem in computer science (for reasons you will see in CS218 later), and such a reduction is very valuable. In particular, for a formula F in k-CNF we look for a formula G that has a small number of literals in each clause, and the total number of clauses itself is not too large - ideally a linear factor of k as compared to F.

   Now consider an r-CNF formula, i.e a CNF formula with r literals per clause. To keep things simple, suppose $r = 2^k$, for some $k > 0$. We will start with the simplest case: a 2^k-CNF formula containing a single clause $C_k = p_0 \vee p_1 \vee p_2 \cdots p_{2^k-1}$ with 2^k literals.

   1. Let's try using 'selector variables' to construct another formula which is strongly-equisatisfiable with C_k. The idea is to use these selector variables to 'select' which literals in our clause are allowed to be false. Use k selector variables $s_1, s_2, s_3 \cdots s_k$ and write a CNF formula that is **strongly-equisatisfiable with** C_k, such that each clause has size $O(k)$.
      [Hint: If you consider the disjunction of $\tilde{s}_1 \vee \tilde{s}_2 \vee \tilde{s}_3 \cdots \tilde{s}_k$, where $\tilde{s}_i$ denotes either s_i or $\neg s_i$, it is true in all but one of the 2^k possible assignments to the s_i variables]
   2. What is the number of clauses in the CNF constructed in the above sub-question? How small can you make the clauses by repeatedly applying this technique?
   3. Consider a k-CNF formula with n clauses. When minimizing the clause size using the above technique repeatedly, what is the (asymptotic) blowup factor in the number of clauses? Observe that the factor is almost linear, upto some logarithmic factors.
      [Hint: Apply the above reduction repeatedly and see how much net blowup occurs]
   4. Can you think of a smarter way to do the reduction that only requires linear blowup?
      [Hint: Consider using different and more auxiliary variables (like the 'selector variables' above) to reduce the size of clauses.]

Solution:

1. The idea is very similar to a MUX (or multiplexer) from digital electronics. We consider the 2^k possible disjunctions: $s_1 \vee s_2 \vee \cdots s_k$; $s_1 \vee s_2 \vee \cdots \neg s_k$ etc. We associate a non-negative integer with each such disjunction, where the k-bit binary encoding of the integer is obtained as $b_1 b_2 \ldots b_k$, where $b_k = 1$ if s_k is in the disjunction and $b_k = 0$ otherwise. For example, the integer associated with the disjunction $(s_1 \vee s_2 \vee \cdots s_k)$ is $11 \cdots 1$ or $2^k - 1$, and the integer associated with $(\neg s_1 \vee \neg s_2 \vee \cdots \neg s_k)$ is $00 \cdots 0$ or 0. Let the disjunction corresponding to the number $0 \leq q \leq 2^k - 1$ in binary be denoted D_q. Consider the CNF formula:

   $$\bigwedge_{i=0} (D_i \vee p_i)$$

If any of the p_is is true, the unique assignment setting D_i to false satisfies all the other $2^k - 1$ clauses. On the other hand if all the p_is are false, any assignment to the s_i variables will dissatisfy atleast one D_q. Thus this formula is strongly equisatisfiable - any assignment setting C_k to true has a satisfying assignment to the s_is, and any assignment setting our formula to true must have one of the p_is true, which makes C_k true.

2. The size of clauses is now $k+1$. The number of clauses is 2^k. Thus, we have obtained an exponential reduction in the size of a clause for a linear factor (the initial size was 2^k blowup in number of clauses. We can reduce the size as long as $\lceil \log_2(\text{size}) \rceil + 1 < \text{size}$, which is true as small as size 3. The size CANNOT be reduced below 3 with this method.

3. We obtain logarithmic reduction for linear blowup. Hence, the factor of blowup will look like $k \cdot \log(k) \cdot \log\log(k) \cdot \log\log\log(k) \cdots \log^{\log^*(k)}(k)$. The factors after the k are all logarithmic factors and can be bounded by a polynomial of any degree > 0.

4. Indeed, we do not need to introduce so many selectors for a clause! We can simply use an indicator s_i to indicate that a clause after p_i is satisfied. Thus, the equisatisfiable formula:

   $$(p_1 \vee p_2 \vee s_2) \wedge (\neg s_2 \vee p_3 \vee s_3) \wedge (\neg s_3 \vee p_4 \vee s_4) \cdots$$

   works and has a linear blowup! Think about Tseitin encoding covered in class.

   However this method also fails to lower the size below 3. Indeed we should suspect that it is not possible to reduce the size below 3, because 2-SAT has a linear time solution while 3-SAT is NP-Complete!

4. **[Take-away question – solve in your rooms]**

In this question we will view the set of assignments satisfying a set of propositional formulae as a language and examine some properties of such languages.

Let $\mathbf{P}$ denote a countably infinite set of propositional variables $p_0, p_1, p_2, \ldots$. Let us call these variables positional variables. Let Σ be a countable set of formulae over these positional variables. Every assignment $\alpha : \mathbf{P} \to \{0,1\}$ to the positional variable can be uniquely associated with an infinite bitstring w, where $w_i = \alpha(p_i)$. The language defined by Σ - denoted by $L(\Sigma)$ - is the set of bitstrings w for which the corresponding assignment α, that has $\alpha(p_i) = w_i$ for each natural i, satisfies Σ, that is, for each formula $F \in \Sigma$, $\alpha \models F$. In this case, we say that $\alpha \models \Sigma$. Let us call the languages definable this way PL-definable languages.

(a) Show that PL-definable languages are closed under countable intersection, ie if $\mathcal{L}$ is a countable set of PL-languages, then $\bigcap_{L \in \mathcal{L}} L$ is also PL-definable.

(b) Show that PL-definable languages are closed under finite union, ie if $\mathcal{L}$ is a finite set of PL-languages, then $\bigcup_{L \in \mathcal{L}} L$ is also PL-definable.

[Hint: Try proving that the union of two PL-definable languages is PL-definable. The general case can then be proven via induction. If $\mathbf{F} = \{F_1, F_2, \ldots\}$ and $\mathbf{G} = \{G_1, G_2, \ldots\}$ are two countable sets of formulae, then an infinite bitstring $w \in L(\mathbf{F}) \cup L(\mathbf{G})$ if and only if either $w \models$ every F_i or $w \models$ every G_i.]

Solution:

a) Let $\mathcal{S}$ denote the family of sets of propositional formulae such that $\mathcal{L} = \{L(\sigma) : \sigma \in \mathcal{S}\}$. Consider $\Sigma = \bigcup_{\sigma \in \mathcal{S}} \sigma$. Since each $\sigma \in \mathcal{S}$ is countable, and a countable union of countable sets is countable, Σ is countable as well. Now, for any infinite bitstring w, $w \in L(\Sigma)$ if and only if $w \models F$ for each $F \in \sigma$, for each $\sigma \in \mathcal{S}$ (we abuse notation and denote the assignment corresponding to w by w). This happens if and only if $w \in L(\sigma)$ for each $\sigma \in \mathcal{S}$, ie $w \in \bigcap_{L \in \mathcal{L}} L$.

b) We show that the union of two PL-definable languages is PL-definable. The general case can then be handled via induction. Let $\mathbf{F}$ and $\mathbf{G}$ be two countable sets of formulae. Let $\mathbf{H} = \{F \vee G : F \in \mathbf{F}, G \in \mathbf{G}\}$ (note that this set is countable since $\mathbf{F}$ and $\mathbf{G}$ are countable). An infinite bitstring w lies in $L(\mathbf{H})$ if and only if, for every $F \in \mathbf{F}$ and $G \in \mathbf{G}$, $w \models F \vee G$. Now, if $w \in L(\mathbf{F})$, then $w \models F$ for every $F \in \mathbf{F}$, hence, $w \models F \vee G$ for every $F \in \mathbf{F}$ and every $G \in \mathbf{G}$ and so $w \in L(\mathbf{H})$. Similarly, if $w \in L(\mathbf{G})$, then $w \in L(\mathbf{H})$, ie for any $w \in L(\mathbf{F}) \cup L(\mathbf{G})$, we have $w \in L(\mathbf{H})$. Let us show that these are the only bitstrings in $L(\mathbf{H})$. Assume some infinite bitstring w is such that $w \notin L(\mathbf{F})$ and $w \notin L(\mathbf{G})$. This means that there is some $F \in \mathbf{F}$ and $G \in \mathbf{G}$ such that $w \not\models F$ and $w \not\models G$. This means that $w \not\models F \vee G$, which is a member of $\mathbf{H}$. Hence $w \notin L(\mathbf{H})$. This means that for any infinite bitstring w, $w \in L(\mathbf{H})$ if and only if $w \in L(\mathbf{F}) \cup L(\mathbf{G})$, ie $L(\mathbf{H}) = L(\mathbf{F}) \cup L(\mathbf{G})$, showing that $L(\mathbf{F}) \cup L(\mathbf{G})$ is PL-definable.

4. Countability

1 Motivation

In topology as well as other areas of mathematics, we deal with a lot of infinite sets. However, as we will gradually discover, some infinite sets are bigger than others. Countably infinite sets, while infinite, are "small" in a very definite sense. In fact they are the "smallest infinite sets". Countable sets are convenient to work with because you can list their elements, making it possible to do inductive proofs, for example.

In the previous section we learned that the set $\mathbb{Q}$ of rational numbers is dense in $\mathbb{R}$. In this section, we will learn that $\mathbb{Q}$ is countable. This is useful because despite the fact that $\mathbb{R}$ itself is a large set (it is *uncountable*), there is a countable subset of it that is "close to everything", at least according to the usual topology. Similarly the usual topology on $\mathbb{R}$ contains a lot of sets (uncountably many sets), but as you have already shown on a Big List question without exactly saying so, this topology has a countable basis, meaning that essentially all of the information about the topology can be specified by a "small" collection of open sets. This is not true of the Sorgenfrey Line, for example, as you will later prove.

With regard to this course in particular, the notion of countability comes up quite often in topology and so this set of notes is here to make sure students have a firm grasp of the concepts involved before we start throwing around the words "countable" and "uncountable" all the time.

2 Counting

The subject of countability and uncountability is about the "sizes" of sets, and how we compare those sizes. This is something you probably take for granted when dealing with finite sets.

For example, imagine we had a room with seven people in it, and a collection of seven hats. How would you check that we had the same number of hats as people? The most likely answer is that you would count the hats, then count the people. You would get an answer of 7 both times, you would note that $7 = 7$, and conclude that there are the same number of hats as there are people. You would be correct, of course.

Now imagine I asked you to do this but you had never heard of "7" - that you were *incapable* of describing the size of the collection of hats or the size of the group of people with a number. Could you still prove that there are the same number of hats as people?

One thing you could do is simply start putting hats on people; take your collection of hats and put one on each person's head. After you finish doing this, you would check whether (a) you have no unused hats remaining; and (b) everyone in the room is wearing precisely one hat. If both of these things are true, you would be justified in concluding that there are the same number of hats as there are people. You could even get more specific, and say that if after putting one hat on each person's head there are some left over hats, then there are more hats

than people. On the other hand if after this some people still are not wearing hats, then there are more people than hats.

Surely this all seems trivial, but notice that what we have described here is a way of checking whether two collections of objects are the same size *without counting them*. Without even knowing what numbers are, for that matter. *This* is the notion of comparing sizes of sets that generalizes well to the infinite.

In particular, when dealing with infinite sets you *cannot* just count both sets and describe their sizes with integers you then can compare. The way of comparing sets described in the example above is the only strategy available to you in that situation.

To "mathematize" our above discussion a bit, notice that what you did with the people and hats amounts to constructing a function from the set of people to the set of hats, defined by

$$g(\text{Person X}) = \text{the hat you put on Person X's head.}$$

If you made sure to put at most one hat on each person's head, g is injective. If you put a hat on every person, g is surjective. If g is both injective and surjective (ie. if g is *bijective*), then we concluded that the set of hats and the set of people were the same size.

This is the idea that we use to compare the sizes of all sets.

Definition 2.1. *Given two sets A and B, we say A has the same cardinality as B if there exists a bijection $f : A \to B$. This is usually denoted by $|A| = |B|$.*

Those interested in fancy, old mathematical language might say that two sets of the same cardinality are *equipotent*.

Definition 2.2. *Given two sets A and B, we say that A has cardinality smaller than or equal to B if there exists an injection $f : A \to B$, or equivalently if there exists a surjection $g : B \to A$. This is usually denoted by $|A| \leq |B|$.*

If there is an injection $f : A \to B$ and there is no surjection from A to B, we say that A has smaller cardinality than B, and write $|A| < |B|$.

We will not go too much further into the abstract notion of cardinality, but the following theorem is somewhat satisfying.

Theorem 2.3 (Cantor-Schröder-Bernstein Theorem)**.** *Let A, B be sets. If $|A| \leq |B|$ and $|B| \leq |A|$, then $|A| = |B|$.*

This theorem is much less trivial than it looks. It says that if there is an injection $f : A \to B$ and an injection $g : B \to A$, then there is a bijection $h : A \to B$. Try to prove it yourself, and you will find that it tricky even for finite sets. You are encouraged to look into the proof of this theorem, which is lovely, and the most basic example of a very useful proof technique called a "back and forth argument".

3 Countability

Definition 3.1. *A set A is said to be countably infinite if $|A| = |\mathbb{N}|$, and simply countable if $|A| \leq |\mathbb{N}|$.*

In words, a set is countable if it has the same cardinality as some subset of the natural numbers. In practise we will often just say "countable" when we really mean "countably infinite", when it is clear that the set involved is infinite. Note that $\emptyset$ is countable, since the empty function $f : \emptyset \to \mathbb{N}$ is vacuously an injection.

The prevailing intuition here should be that a set is countable provided that you can list its elements. After all, an injection $f : A \to \mathbb{N}$ is nothing more than a way of assigning a number to each element of A in a reasonable way (ie. in a way such that each element of A gets one number). In particular, a set A is countably infinite provided that you can construct a list

$$a_1, a_2, a_3, a_4, a_5, a_6, \ldots$$

of its elements and prove (a) that no element of A appears on the list more than once; and (b) that the list includes every element of A. (Given such a list, the function defined by $f(n) = a_n$ is a bijection $\mathbb{N} \to A$.) Many of our proofs that sets are countably infinite will just be given as lists like this.

It should already be apparent that this way of comparing sizes produces some results that feel unusual if one is only used to thinking about sizes of finite sets. For example, an obvious fact about finite sets is that if A is a finite set and B is a proper subset of A, then B is smaller than A. That is, if you start with a finite set A and take some things away, what iss left over is smaller than A. This is not the case with infinite sets.

Example 3.2. Let $E \subseteq \mathbb{N}$ be the set of even numbers. Then $|E| = |\mathbb{N}|$. Indeed:

$$2, 4, 6, 8, 10, 12, 14, 16, \ldots$$

4 Simple examples and facts

In this section we will look at some simple examples of countable sets, and from the explanations of those examples we will derive some simple facts about countable sets.

Example 4.1. The set $A = \{\, n \in \mathbb{N} : n > 7 \,\}$ is countable. We can certainly list its elements in a bijective way:

$$8, 9, 10, 11, 12, 13, \ldots$$

or think of the bijection $f : \mathbb{N} \to A$ given by $f(n) = n + 7$.

It should be clear that nothing is special about the number 7 here. We started with a countably infinite set, removed finitely many elements, and were left with something that was still countably infinite. The fact that we removed the *first* seven elements allowed us to define f in a pretty convenient way, but even this was not necessary. This leads us to the following fact:

Proposition 4.2. *If A is a countable set and $B \subseteq A$ is finite, then $A \setminus B$ is countable.*

Proof. This result is obvious if A is finite, so we will treat the case in which A is countably infinite.

It is much easier to explain the idea of this proof than to write it down; to list the elements of $A \setminus B$, start with a list of the elements of A (which we have by the assumption that A is countable), delete the elements of B from the list, then squish everything down to fill the gaps created by the deletions.

For example, we explicitly treat the case in which B has two elements. Let the bijection $f : \mathbb{N} \to A$ witness that A is countable, and suppose $B = \{f(17), f(2523)\}$. Then the following function $g : \mathbb{N} \to A \setminus B$ is a bijection:

$$g(n) = \begin{cases} f(n) & n < 17 \\ f(n+1) & 17 \leq n < 2522 \\ f(n+2) & n \geq 2522 \end{cases}.$$

Check that this is a bijection between $\mathbb{N}$ and $A \setminus B$, and convince yourself of how you would write down a function like this for the general setting. □

We can also use the same idea, but backwards. That is, if we start with a countable set and add finitely many elements, the result is countable.

Example 4.3. The set $A = \{\, n \in \mathbb{Z} : n \geq -7 \,\} = \mathbb{N} \cup \{-7, -6, \ldots, -1, 0\}$ is countable. This is witnessed by the function $f : \mathbb{N} \to A$ defined by $f(n) = n - 8$.

Proposition 4.4. *Let A be a countable set, and B a finite set. Then $A \cup B$ is countable.*

Proof. Again, this result is obvious if A is finite, so assume it is countably infinite. Furthermore we may assume without loss of generality that A and B are disjoint, as any amount of overlap would effectively shrink B and "help" our cause. More formally, $A \cup B = A \cup (B \setminus A)$, and $B \setminus A$ must also be finite, so it suffices to consider disjoint sets A and B.

Let $f : \mathbb{N} \to A$ witness that A is countable, and enumerate $B = \{b_1, b_2, \ldots, b_k\}$ for some k. We can do this since B is finite. Now define:

$$g(n) = \begin{cases} b_n & n \leq k \\ f(n-k) & n > k \end{cases}.$$

In the language of lists, we are simply prepending the elements of B to the beginning of our given list of elements of A, so that $A \cup B$ is listed as:

$$b_1, b_2, \ldots, b_k, f(1), f(2), f(3), f(4), \ldots,$$

then re-numbering this new list with g. □

So adding or removing finitely many elements does not affect countability. However, in the previous section (Example 4.1) we saw that at least in some cases you can remove *infinitely* many elements of a countable set and still be left with a countable set. To reiterate:

Example 4.5. Let $E \subseteq \mathbb{N}$ be the set of even natural numbers. Then E is countable, as witnessed by $f(n) = 2n$.

Again from this example, we can extract a pair of more general facts.

Proposition 4.6. *Let A be a countably infinite set, and B an infinite subset of A. Then B is countable.*

Proof. The idea here is essentially the same as in Proposition 4.2; list the elements of A, delete the elements *not in* B from your list, then squish everything down to fill the gaps. We will take care to be a bit more rigorous with this proof though.

So let $f : \mathbb{N} \to A$ be a bijection witnessing that A is countable. We want to construct a bijection $g : \mathbb{N} \to B$.

What goes on in the proof that follows is essentially this: list the elements of A, then take the element of B with the smallest index in this list, and call that $g(1)$. Then take the element of B with the next smallest index, and call that $g(2)$. Repeat this process inductively, letting $g(k)$ equal the element of B with the k^{th}–smallest index. Then we prove that g is a bijection.

Let $k_1 = \min\{\, k \in \mathbb{N} : f(k) \in B \,\} = \min(f^{-1}(B))$. That is, k_1 is the smallest number that gets mapped into B by f. Define $g(1) := f(k_1)$. We proceed inductively from here.

Assume we have defined $g(1), g(2), \ldots, g(n)$. Let

$$k_{n+1} = \min\{\, k \in \mathbb{N} : f(k) \in B \setminus \{g(1), \ldots, g(n)\} \,\}.$$

That is, k_{n+1} is the smallest number that f maps to an element of B that we have not hit with g so far. (Note that since B is infinite, $B \setminus \{g(1), \ldots, g(n)\}$ is also infinite and in particular nonempty, so this minimum actually exists.)

Then, define $g(n+1) = f(k_{n+1})$. Continuing in this way we construct a function $g : \mathbb{N} \to B$ which is clearly injective. To see that it is surjective, fix $b \in B$, and assume for the sake of contradiction that b is not in the range of g. Define

$$X = \{\, n \in \mathbb{N} : f(n) \in B \text{ but } f(n) \text{ is not in the range of } g \,\}.$$

That is, X is the set of indices (according to f) of elements of B that are missed by g. Then X is nonempty, since $f^{-1}(b) \in X$ at least. Being a nonempty subset of the naturals, X must have

a minimal element, which we call n_0. But then n_0 must be k_N for some $N \leq n$, by definition of the k_i.

To see this more carefully, note that at most n_0 of the k_i's can occur before n_0. Let M be the number of them that occur. In other words:

$$M = |\{1, 2, 3, \ldots n_0\} \cap \{\, k_i \,:\, i \in \mathbb{N} \,\}|$$

This means $k_1, k_2, \ldots, k_M$ are all $\leq n_0$ and there are no other k_i's below n_0. But then:

$$n_0 = \min \{\, n \in \mathbb{N} \,:\, f(n) \in B \setminus \{g(1), \ldots, g(M)\} \,\}.$$

Therefore, by definition of the k_i's, we must have that $n_0 = k_{M+1}$. So $M + 1$ is the N we mentioned earlier.

This is a contradiction, since $g(n_0) = g(k_N) \in B$ by definition of g, but n_0 was a member of X. We conclude that X must be empty and in turn that g is surjective.

□

One way to interpret this result is as a sort of dichotomy theorem. A subset of a countably infinite set must be either finite or infinite, and this result says that if it is not finite, it must be *countably* infinite. In other words, there are no sizes available for a set to be "between" finite and countably infinite.

Just as in the two "matching" propositions we proved earlier, Proposition 4.6 leads to another fact if we go backwards. That is, if we start with a countable set and add countably many elements, the result is countable.

Proposition 4.7. *Let A, B be countable sets. Then $A \cup B$ is countable.*

Proof. If A and B are finite, this is obvious. If just one of them is finite, this is Proposition 4.4. So assume both are countably infinite. And again, without loss of generality, we may assume A and B are disjoint. We can do this since $A \cup B = A \cup (B \setminus A)$, and by the previous proposition $B \setminus A$ must be either finite or countably infinite.

Let $f : \mathbb{N} \to A$ and $g : \mathbb{N} \to B$ be bijections witnessing that A and B are countable. Define $h : \mathbb{N} \to A \cup B$ by:

$$h(n) = \begin{cases} f(k) & \text{if } n = 2k \\ g(k) & \text{if } n = 2k - 1 \end{cases}$$

In the language of lists, take the list of elements of A and the list of elements of B and interweave them.

Convince yourself that h is a bijection. □

Corollary 4.8. *$\mathbb{Z}$ is countable.*

Proof. Notice that $\mathbb{Z} = (\mathbb{N} \cup \{0\}) \cup (-\mathbb{N})$, where $-\mathbb{N}$ is an abuse of notation denoting $\{\, -n \,:\, n \in \mathbb{N} \,\}$, the negative integers. The result then follows from Propositions 4.7 and 4.4. □

Exercise 4.9. Construct an explicit bijection $f : \mathbb{N} \to \mathbb{Z}$.

Exercise 4.10. Prove that any finite union of countable sets is countable. (Hint: Use the previous Proposition, and induction.)

At this point we state a proposition that will free us up just a bit. Every proof that something is countably infinite that we have done so far has involved bijections from $\mathbb{N}$. Bijections are pretty nice functions though, and in particular they have inverses which are also bijections, and the composition of two bijections is again a bijection.

Proposition 4.11. *Let A be an infinite set, and C a fixed countably infinite set. Then the following are equivalent.*

1. *A is countable.*
2. *There exists a bijection $f : \mathbb{N} \to A$.*
3. *There exists a bijection $g : A \to \mathbb{N}$.*
4. *There exists a bijection $h : A \to C$, or $h : C \to A$.*
5. *There exists an injection $i : A \to \mathbb{N}$, or $i : A \to C$.*
6. *There exists a surjection $s : \mathbb{N} \to A$, or $s : C \to A$.*

Proof. **Exercise.** (Just about everything here follows from elementary properties of functions or the Propositions we have proved above. The only one that should require any work to show is equivalent to the others is (6)). □

Exercise 4.12. Later in this course we will learn about the Axiom of Choice. After we have learned about it (or right now if you are already familiar with it) try to spot whether any of the implications between these facts rely on some amount of Choice.

5 More interesting examples and facts

Earlier we proved the odd fact that infinite sets can be the same size as proper subsets of themselves. We even learned that $\mathbb{N}$ and $\mathbb{Z}$ are the same size as one another, even though by most intuitive measures $\mathbb{Z}$ is at least "twice the size" of $\mathbb{N}$. There are, after all, two perfect-looking copies of $\mathbb{N}$ inside $\mathbb{Z}$. This is weird.

We will start off this section by proving that $\mathbb{N}$ is the same size as something with *infinitely many* perfect copies of itself inside it.

Theorem 5.1. *$\mathbb{N} \times \mathbb{N}$ is countable.*

Proof. There are two ways to prove this. One is by drawing a picture and pointing at it, and the other is by explicitly defining a bijection $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. The former way is the best way. Here is the picture:

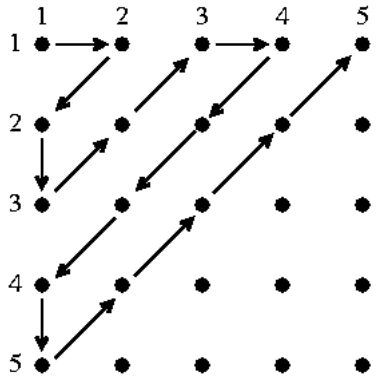


Figure 1: Convince yourself that this is a proof that $\mathbb{N} \times \mathbb{N}$ is countable. (Source: College Math Teaching, 10 April, 2015.)

For a more explicit proof, define a function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ by

$$f(n, m) = 2^n 3^m.$$

This function is injective by the uniqueness of prime decompositions, and so $\mathbb{N}\times\mathbb{N}$ is countable by Proposition 4.11, part 5. □

Corollary 5.2. *Let A and B be countable sets. Then $A \times B$ is countable.*

Proof. **Exercise.** □

Exercise 5.3. Show that the Cartesian product of finitely many countable sets is countable in two different ways. Take note of why each of your two proofs does not extend to even countable products of countable sets.

From Corollary 5.2 follows one of the most important results of this section for our purposes.

Corollary 5.4. *$\mathbb{Q}$ is countable.*

Proof. $\mathbb{Q}$ is obviously infinite, so we will show it is countable by constructing an injection into the set $\mathbb{Z} \times \mathbb{N}$, which is countable by the previous corollary.

Recal that $\mathbb{Q}$ can be expressed as

$$\left\{ \frac{p}{q} \ : \ p \in \mathbb{Z}, q \in \mathbb{N}, \text{ and the fraction is in lowest terms} \right\}.$$

Define a function $f : \mathbb{Q} \to \mathbb{Z} \times \mathbb{N}$ by $f(\frac{p}{q}) = (p, q)$. This map is an injection (check this!) and therefore $\mathbb{Q}$ is countable by Proposition 4.11, part 5. □

We will talk a great deal more about this fact throughout the course, in many different contexts. For now, take a moment to come to terms with it. Thus far you have likely been picturing countable sets as sparse collections of points, like $\mathbb{N}$, $\mathbb{Z}$, etc. $\mathbb{Q}$ on the other hand is

very densely packed. Wherever you look in the real numbers, there are infinitely many rationals there. Still, the set of all rationals is countable.

Another corollary of Theorem 5.1 is the following. At this point you should not be too surprised by it, though it would have been surprising at the beginning of our discussion on countability.

Corollary 5.5. *A countable union of countable sets is countable. That is, if A_n, $n \in \mathbb{N}$ are countable sets, then $\bigcup_{n\in\mathbb{N}} A_n$ is countable.*

Proof. As before, we may assume that the A_n are all infinite and mutually disjoint.

Let $f_n : \mathbb{N} \to A_n$ be a bijection witnessing that A_n is countable, and define $g : \mathbb{N} \times \mathbb{N} \to \bigcup_{n\in\mathbb{N}} A_n$ by $g(n, i) = f_n(i)$.

Exercise 5.6. Show that the function g defined above is a bijection, and conclude that $\bigcup_{n\in\mathbb{N}} A_n$ is countable.

□

Exercise 5.7. Show that the set $\mathcal{B}_{\mathbb{Q}} = \{ (a, b) \subseteq \mathbb{R} : a, b \in \mathbb{Q} \}$ is countable. This means that $\mathbb{R}_{\text{usual}}$ has a countable basis (you showed in Big List 2.2 that $\mathcal{B}_{\mathbb{Q}}$ is a basis for $\mathbb{R}_{\text{usual}}$).

This property of having a countable basis is useful enough (and *invariant* enough, in senses we will discuss later) to earn a name. We will do that at the end of this note.

To summarize what we have learned thus far:

- A subset of a countable set is either finite or countably infinite.
- Finite and countably infinite unions of countable sets are countable.
- Finite Cartesian products of countable sets are countable.

6 Are all sets countable!?

No.

Definition 6.1. *A set A that is not countable is called uncountable.*

By this point, the following should not be surprising.

Proposition 6.2. *If A is uncountable, B is a set, and $f : A \to B$ is a bijection, then B is uncountable.*

Proof. **Exercise.** □

Fine, so what sets are actually uncountable? Lots of them, it turns out. We will give one specific example and its consequences, and one method for producing as many uncountable sets as you want.

The following is a very famous and lovely proof, first given by Cantor in the 1880s. It was quite a surprise at the time. It is often called "Cantor's Diagonalization Proof".

Theorem 6.3 (Cantor)**.** $(0,1) \subseteq \mathbb{R}$ *is uncountable.*

Proof. This is a proof by contradiction. Certainly $(0,1)$ is not finite, so we begin by assuming that $(0,1)$ is countably infinite, and fixing a bijection $f : \mathbb{N} \to (0,1)$ witnessing this. Every real number between 0 and 1 has an infinite decimal expansion of the form:

$$0.a_1a_2a_3a_4a_5a_6a_7\ldots$$

(where we imagine a tail of zeros on a real number with a terminating decimal expansion). In this way, we will index every digit appearing in the decimal expansion of every real number on our list (the list given by f). That is, for $x \in (0,1)$, we have by assumption that there is a unique $n \in \mathbb{N}$ such that $f(n) = x$. So we're going to write:

$$x = f(n) = 0.x_1^n x_2^n x_3^n x_4^n x_5^n x_6^n x_7^n \ldots$$

In your head you should now be imagining that we have constructed an infinite table that looks something like:

$$\begin{array}{rccccccc}
f(1) = & 0. & \color{red}{x_1^1} & x_2^1 & x_3^1 & x_4^1 & x_5^1 & \ldots \\
f(2) = & 0. & x_1^2 & \color{red}{x_2^2} & x_3^2 & x_4^2 & x_5^2 & \ldots \\
f(3) = & 0. & x_1^3 & x_2^3 & \color{red}{x_3^3} & x_4^3 & x_5^3 & \ldots \\
f(4) = & 0. & x_1^4 & x_2^4 & x_3^4 & \color{red}{x_4^4} & x_5^4 & \ldots \\
f(5) = & 0. & x_1^5 & x_2^5 & x_3^5 & x_4^5 & \color{red}{x_5^5} & \ldots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

where I have coloured the diagonal elements in the interesting part of our table red to call attention to them.

We will now construct a real number $y = 0.y_1y_2y_3y_4y_5\ldots$ that cannot be on our list (ie. cannot be in the range of f), contradicting the assumption that f is a bijection.

Indeed, let $y_1 = x_1^1 - 1$, unless $x_1^1 = 0$, in which case let $y_1 = 9$. The specifics of this are not so important, but what is important is that we fix a concrete way of defining y_1 to not equal x_1^1. Now do the same for each y_k. That is, let $y_k = x_k^k - 1$, unless $x_k^k = 0$, in which case let $y_k = 9$. Also, at the end of this process, we have to make sure we did not construct $y = 0 = 0.0000\ldots$ or $y = 1 = 0.9999\ldots$. So at the end, we just make any change to one digit that ensures this.

I claim that $y = 0, y_1y_2y_3y_4y_5\ldots$ is not in the range of f.

To see this, suppose it was in the image of f. Then there would have to exist an $n \in \mathbb{N}$ such that $f(n) = y$. But then by construction of y, the n^{th} decimal place of y differs from the n^{th} decimal place of $f(n)$.

This is a contradiction, proving that y is not in the image of f, which in turn shows that f was not a bijection. Since the f we started with was arbitrary, we conclude that there can be no bijection between $\mathbb{N}$ and $(0,1)$. □

Amazing! I hope some of the beauty of this result is apparent even to the most jaded math specialists reading this. Both of $\mathbb{N}$ and $(0,1)$ are infinite, but $(0,1)$ is *bigger*!

Of course, this also means that $\mathbb{R}$ is uncountable.

Corollary 6.4. *$\mathbb{R}$ is uncountable.*

Proof. All we need to do is exhibit a bijection $h : (0,1) \to \mathbb{R}$. $f(x) = \pi(x - \frac{1}{2})$ is a bijection between $(0,1)$ and $(-\frac{\pi}{2}, \frac{\pi}{2})$ (check this!), and $g(x) = \tan(x)$, when restricted to $(-\frac{\pi}{2}, \frac{\pi}{2})$, is a bijection from this set to $\mathbb{R}$. Their composition $h := g \circ f : (0,1) \to \mathbb{R}$ is therefore a bijection, as required. □

We conclude our discussion of uncountable sets with what is undoubtedly my personal favourite proof in mathematics. I can trace my decision to pursue mathematics professionally directly to when I first read and understood this proof and its implications.

Theorem 6.5. *Let A be a set. There is no surjection $f : A \to \mathcal{P}(A)$. Since $g : A \to \mathcal{P}(A)$ given by $g(x) = \{x\}$ is clearly an injection, this means $|A| < |\mathcal{P}(A)|$. In particular, $\mathcal{P}(\mathbb{N})$ is uncountable.*

Proof. Let A be a set, and let $f : A \to \mathcal{P}(A)$ be a function. We will show that f cannot be surjective. Define a set:

$$D = \{\, x \in A : x \notin f(x) \,\}.$$

Clearly D is a subset of A, so $D \in \mathcal{P}(A)$.

Now suppose for the sake of contradiction that f is surjective. Then in particular its range must contain D, so let $a \in A$ be such that $f(a) = D$.

Question: Is $a \in D$?

Either it is or it is not, surely. We examine both cases.

Case 1: Yes. If $a \in D$, then $a \in f(a)$ (since $f(a) = D$). However looking back at the definition of D, we see that this means a cannot be in D. So this is impossible.

Case 2: No. If $a \notin D$, then $a \notin f(a)$. Looking back at the definition of D again, this precisely means $a \in D$. Again, this is impossible.

In either case we get a contradiction. Since these are the only two cases, the assumption that f is surjective must be false, finishing the argument. □

The implications of this proof are far-reaching. Just for a taste, it implies that if you keep iterating the power set operation, you get sets with strictly larger and larger cardinalities. That is, for example:

$$|\mathbb{N}| < |\mathcal{P}(\mathbb{N})| < |\mathcal{P}(\mathcal{P}(\mathbb{N}))| < |\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N})))| < \cdots$$

So there are *infinitely many sizes of infinity*! I love this stuff.

7 Enough cool proofs, give me some definitions!

Fine, fine. Here are two definitions that we have alluded to as being useful in previous discussions, and that I define here since we have now formalized the notion of countability.

To reiterate some of that discussion, countability is a notion of "smallness" for us. Even though $\mathbb{R}_{\text{usual}}$ is a big, complicated space, the fact that a countable set ($\mathbb{Q}$, for example) is dense in it serves to simplify many things. The space itself is big, but there is a countable (read: small) set that is "close" to everything.

On a similar note, we know that $\mathbb{R}_{\text{usual}}$ has a tonne of open sets that can get very complicated. We already decided that we prefer to specify a simpler basis of open sets rather than specify *every* open set in the topology. But even the usual basis of $\mathbb{R}_{\text{usual}}$—the collection of ϵ-balls around all the points—is big, in the sense that it is uncountable. However, you proved in a Big List problem and also in Exercise 5.7 above that the set of intervals with rational endpoints is a countable basis for the usual topology. This is very useful, because now we know that all the information in this big, complicated topology can be recovered from a small amount of information.

As the course progresses, these ideas will crop up several more times. For the moment, we give them names.

Definition 7.1. *A topological space $(X, \mathcal{T})$ is called separable if there is a countable, dense subset of X.*

Definition 7.2. *A topological space $(X, \mathcal{T})$ is called second countable if there is a countable basis on X that generates $\mathcal{T}$.*

Two notes about second countability:

First, take care to note that the definition does not say that *every* basis of the topology is countable, just that there is at least one countable basis. In particular the usual basis of the usual topology on $\mathbb{R}$ is very much uncountable, but the existence of $\mathcal{B}_{\mathbb{Q}}$ shows that $\mathbb{R}_{\text{usual}}$ is second countable.

Second, it may seem weird to define "second countable" without having defined "first countable". These properties both have terrible names. The concept of "first countable" is something I may have hinted at in lecture, but which we are not quite ready to discuss. You will soon find that it is a subtle and useful property, but only after we have discussed sequences.

Finally, here is one more topological property that involves countability. We will not prove anything with this property at the moment, but you will do a little bit in a Big List problem for this section, and we will revisit this property several times throughout the course.

Definition 7.3. *A topological space $(X, \mathcal{T})$ is said to have the countable chain condition if there are no uncountable collections of mutually disjoint, nonempty, open subsets of X. Another way*

to characterize this property is to say that every collection of mutually disjoint, nonempty, open subsets of X is countable.

Remark 7.4. If $(X, \mathcal{T})$ has the countable chain condition, we will usually say "$(X, \mathcal{T})$ has the ccc" or "$(X, \mathcal{T})$ is ccc".

You proved that $\mathbb{R}_{\text{usual}}$ is ccc with a problem at the end of the first section of the Big List, though of course you did not have this name at the time.

For now we will simply add this property to our growing catalogue of topological properties, but we make special note that the relationships between this property, separability, and second countability will be interesting to explore. Whenever we prove something about one of these three properties, you should think about the other ones in the same context.

CS 208 : Automata Theory and Logic

Spring 2024

Instructor : Prof. Supratik Chakraborty

Disclaimer

Please note this document has not received the usual scrutiny that formal publications enjoy. This may be distributed outside this class only with the permission of the instructor.

Contents

Chapter 1

Propositional Logic

In this course we look at two ways of computation: a state transition view and a logic centric view. In this chapter we begin with logic centered view with the discussion of propositional logic.

Example. Suppose there are five courses $C_1, \ldots, C_5$, four slots $S_1, \ldots, S_4$, and five days $D_1, \ldots, D_5$. We plan to schedule these courses in three slots each, but we have also have the following requirements:

	D_1	D_2	D_3	D_4	D_5
S_1					
S_2					
S_3					
S_4					

- For every course C_i, the three slots should be on three different days.
- Every course C_i should be scheduled in at most one of $S_1, \ldots, S_4$.
- For every day D_i of the week, have at least one slot free.

Propositional logic is used in many real-world problems like timetables scheduling, train scheduling, airline scheduling, and so on. One can capture a problem in a propositional logic formula. This is called as encoding. After encoding the problem, one can use various software tools to systematically reason about the formula and draw some conclusions about the problem.

1.1 Syntax

We can think of logic as a language which allows us to very precisely describe problems and then reason about them. In this language, we will write sentences in a specific way. The symbols used in propositional logic are given in Table 1.1. Apart from the symbols in the table we also use variables usually denoted by small letters $p, q, r, x, y, z, \ldots$ etc. Here is a short description of propositional logic symbols:

- **Variables**: They are usually denoted by smalls ($p, q, r, x, y, z, \ldots$ etc). The variables can take up only true or false values. We use them to denote propositions.
- **Constants**: The constants are represented by $\top$ and $\bot$. These represent truth values true and false.

- **Operators**: $\wedge$ is the conjunction operator (also called AND), $\vee$ is the disjunction operator (also called OR), $\neg$ is the negation operator (also called NOT), $\rightarrow$ is implication, and $\leftrightarrow$ is bi-implication (equivalence).

Name	Symbol	Read as
true	$\top$	top
false	$\bot$	bot
negation	$\neg$	not
conjunction	$\wedge$	and
disjunction	$\vee$	or
implication	$\rightarrow$	implies
equivalence	$\leftrightarrow$	if and only if
open parenthesis	(	
close parenthesis	)	

Table 1.1: Logical connectives.

For the timetable example, we can have propositional variables of the form p_{ijk} with $i \in [5]$, $j \in [5]$ and $k \in [4]$ (Note that $[n] = \{1, \ldots, n\}$) with p_{ijk} representing the proposition 'course C_i is scheduled in slot S_k of day D_j'.

Rules for formulating a formula:

- Every variable constitutes a formula.
- The constants $\top$ and $\bot$ are formulae.
- If φ is a formula, so are $\neg\varphi$ and (φ).
- If φ_1 and φ_2 are formulas, so are $\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \rightarrow \varphi_2$, and $\varphi_1 \leftrightarrow \varphi_2$.

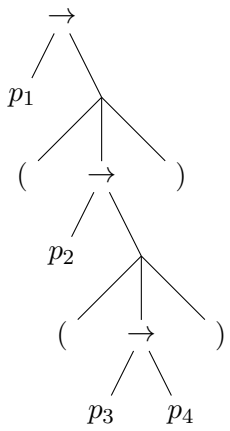
Propositional formulae as strings and trees:

Formulae can be expressed as a strings over the alphabet $\mathbf{Vars} \cup \{\top, \bot, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, (,)\}$. **Vars** is the set of symbols for variables. Not all words formed using the alphabet qualify as propositional formulae. A string constitutes a well-formed formula (wff) if it was constructed while following the rules. Examples: $(p_1 \vee \neg q_2) \wedge (\neg p_2 \rightarrow (q_1 \leftrightarrow \neg p_1))$ and $p_1 \rightarrow (p_2 \rightarrow (p_3 \rightarrow p_4))$.
Well-formed formulas can be represented using trees. Consider the formula $p_1 \rightarrow (p_2 \rightarrow (p_3 \rightarrow p_4))$. This can be represented using the parse tree in figure Figure 1.1a. Notice that while strings require parentheses for disambiguation, trees don't, as can be seen in Figure 1.1b and Figure 1.1c.

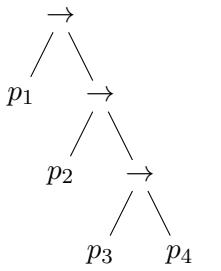
1.2 Semantics

Semantics give a meaning to a formula in propositional logic. The semantics is a function that takes in the truth values of all the variables that appear in a formula and gives the truth value of the formula. Let 0 represent "false" and 1 represent "true". The semantics of a formula φ of n variables is a function

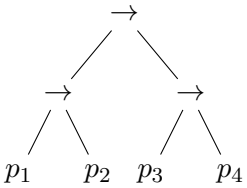
$$[\![\varphi]\!] : \{0,1\}^n \rightarrow \{0,1\}$$



(a)



(b) Parse tree for $p_1 \to (p_2 \to (p_3 \to p_4))$



(c) Parse tree for $(p_1 \to p_2) \to (p_3 \to p_4)$

Figure 1.1: Parse trees obviate the need for parentheses.

It is often presented in the form of a truth table. Truth tables of operators can be found in table Table 1.2.

φ	$\neg\varphi$
0	1
1	0

(a) Truth table for $\neg\varphi$.

φ_1	φ_2	$\varphi_1 \wedge \varphi_2$
0	0	0
0	1	0
1	0	0
1	1	1

(b) Truth table for $\varphi_1 \wedge \varphi_2$.

φ_1	φ_2	$\varphi_1 \vee \varphi_2$
0	0	0
0	1	1
1	0	1
1	1	1

(c) Truth table for $\varphi_1 \vee \varphi_2$.

φ_1	φ_2	$\varphi_1 \to \varphi_2$
0	0	1
0	1	1
1	0	0
1	1	1

(d) Truth table for $\varphi_1 \to \varphi_2$.

φ_1	φ_2	$\varphi_1 \leftrightarrow \varphi_2$
0	0	1
0	1	0
1	0	0
1	1	1

(e) Truth table for $\varphi_1 \leftrightarrow \varphi_2$.

Table 1.2: Truth tables of operators.

Remark. Do not confuse 0 and 1 with $\top$ and $\bot$: 0 (false) and 1 (true) are meanings, while $\top$ and $\bot$ are symbols.

Rules of semantics:

- $[\![\neg\varphi]\!] = 1$ iff $[\![\varphi]\!] = 0$.
- $[\![\varphi_1 \wedge \varphi_2]\!] = 1$ iff $[\![\varphi_1]\!] = [\![\varphi_2]\!] = 1$.
- $[\![\varphi_1 \vee \varphi_2]\!] = 1$ iff at least one of $[\![\varphi_1]\!]$ or $[\![\varphi_2]\!]$ evaluates to 1.

- $[\![\varphi_1 \to \varphi_2]\!] = 1$ iff at least one of $[\![\varphi_1]\!] = 0$ or $[\![\varphi_2]\!] = 1$.
- $[\![\varphi_1 \leftrightarrow \varphi_2]\!] = 1$ iff at both $[\![\varphi_1 \to \varphi_2]\!] = 1$ and $[\![\varphi_2 \to \varphi_1]\!] = 1$.

Truth Table: A truth table in propositional logic enumerates all possible truth values of logical expressions. It lists combinations of truths for individual propositions and the compound statement's truth.

Example. Let us construct a truth table for $[\![(p \vee s) \to (\neg q \leftrightarrow r)]\!]$ (see Table 1.3).

p	q	r	s	$p \vee s$	$\neg q$	$\neg q \leftrightarrow r$	$(p \vee s) \to (\neg q \leftrightarrow r)$
0	0	0	0	0	1	0	1
0	0	0	1	1	1	0	0
0	0	1	0	0	1	1	1
0	0	1	1	1	1	1	1
0	1	0	0	0	0	1	1
0	1	0	1	1	0	1	1
0	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	0
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	1
1	1	0	0	1	0	1	1
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Table 1.3: Truth table of $(p \vee s) \to (\neg q \leftrightarrow r)$.

1.2.1 Important Terminology

A formula φ is said to (be)

- **satisfiable** or **consistent** or SAT iff $[\![\varphi]\!] = 1$ for some assignment of variables. That is, there is at least one way to assign truth values to the variables that makes the entire formula true. Both a formula and its negation may be SAT at the same time (φ and $\neg\varphi$ may both be SAT).
- **unsatisfiable** or **contradiction** or UNSAT iff $[\![\varphi]\!] = 0$ for all assignments of variables. That is, there is no way to assign truth values to the variables that makes the formula true. If a formula φ is UNSAT then $\neg\varphi$ must be SAT (it is in fact valid).
- **valid** or **tautology**: $[\![\varphi]\!] = 1$ for all assignments of variables. That is, the formula is always true, no matter how the variables are assigned. If a formula φ is valid then $\neg\varphi$ is UNSAT.
- **semantically entail** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ for all assignments of variables, where 0 (false) $\preceq$ 0 (true). This is denoted by $\varphi \models \varphi_1$. If $\varphi \models \varphi_1$, then for every assignment, if φ evaluates to 1 then φ_1 will evaluate to 1. Equivalently $\varphi \to \varphi_1$ is valid.

- **semantically equivalent** to φ_1 iff $\varphi \models \varphi_1$ and $\varphi_1 \models \varphi$. Basically φ and φ_1 have identical truth tables. Equivalently, $\varphi \leftrightarrow \varphi_1$ is valid.
- **equisatisfiable** to φ_1 iff either both are SAT or both are UNSAT. Also note that, semantic equivalence implies equisatisfiability but **not** vice-versa.

Term	Example
SAT	$p \vee q$
UNSAT	$p \wedge \neg p$
valid	$p \vee \neg p$
semantically entails	$\neg p \models p \rightarrow q$
semantically equivalent	$p \rightarrow q$, $\neg p \vee q$
equisatisfiable	$p \wedge q$, $r \vee s$

Table 1.4: Some examples for the definitions.

Example. Consider the formulas $\varphi_1 : p \rightarrow (q \rightarrow r)$, $\varphi_2 : (p \wedge q) \rightarrow r$ and $\varphi_3 : (q \wedge \neg r) \rightarrow \neg p$. The three formulas φ_1, φ_2 and φ_3 are semantically equivalent. One way to check this is to construct the truth table.

On drawing the truth table for the above example, one would realise that it is laborious. Indeed, for a formula with n variables, the truth table has 2^n entries! So truth tables don't work for large formulas. We need a more systematic way to reason about the formulae. That leads us to proof rules...
But before that let us get a closure on the example at the beginning of the chapter. Let p_{ijk} represent the proposition 'course C_i is scheduled in slot S_k of day D_j'. We can encode the constraints using the encoding strategy used in tutorial 1 - problem 3. That is, by introducing extra variables that bound the sum for first few variables (sum of i is atmost j). Using this we can encode the constraints as : $\sum_{k=1}^{4} p_{ijk} \leq 1$, $\sum_{j=1}^{5} p_{ijk} \leq 1$, $\sum_{i=1}^{5} p_{ijk} \leq 1$, $\sum_{k=1}^{4}\sum_{i=1}^{5} p_{ijk} \leq 3$, $\sum_{k=1}^{4}\sum_{j=1}^{5} p_{ijk} \leq 3$ and $\neg\big(\sum_{k=1}^{4}\sum_{j=1}^{5} p_{ijk} \leq 2\big)$.

1.3 Proof Rules

After encoding a problem into propositional formula we would like to reason about the formula. Some of the properties of a formula that we are usually interested in are whether it is SAT, UNSAT or valid. We have already seen that truth tables do not scale well for large formulae. It is also not humanly possible to reason about large formulae modelling real-world systems. We need to delegate the task to computers. Hence, we need to make systematic rules that a computer can use to reason about the formulae. These are called as proof rules.
The overall idea is to convert a formula to a normal form (basically a standard form that will make reasoning easier - more about this later in the chapter) and use proof rules to check SAT etc.

Rules are represented as

$$\frac{\text{Premises}}{\text{Inferences}}\ \text{Connector}_{i/e}$$

- **Premise**: A premise is a formula that is assumed or is known to be true.

- **Inference**: The conclusion that is drawn from the premise(s).
- **Connector**: It is the logical operator over which the rule works. We use the subscript i (for introduction) if the connector and the premises are combined to get the inference. The subscript e (for elimination) is used when we eliminate the connector present in the premises to draw inference.

Example. Look at the following rule

$$\frac{\varphi_1 \wedge \varphi_2}{\varphi_1} \wedge_{e_1}$$

In the rule above $\varphi_1 \wedge \varphi_2$ is assumed (is premise). Informally, looking at $\wedge$'s truth table, we can infer that both φ_1 and φ_2 are true if $\varphi_1 \wedge \varphi_2$ is true, so φ_1 is an inference. Also, in this process we eliminate (remove) $\wedge$ so we call this AND-ELIMINATION or $\wedge_e$. For better clarity we call this rule $\wedge_{e_1}$ as φ_1 is kept in the inference even when both φ_1 and φ_2 could be kept in inference. If we use φ_2 in inference then the rule becomes $\wedge_{e_2}$.

Table 1.5 summarises the basic proof rules that we would like to include in our proof system.

Connector	Introduction	Elimination
$\wedge$	$\dfrac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2} \wedge_i$	$\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_1} \wedge_{e_1}$ $\quad$ $\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_2} \wedge_{e_2}$
$\vee$	$\dfrac{\varphi_1}{\varphi_1 \vee \varphi_2} \vee_{i_1}$ $\quad$ $\dfrac{\varphi_2}{\varphi_1 \vee \varphi_2} \vee_{i_2}$	$\dfrac{\varphi_1 \vee \varphi_2 \quad \varphi_1 \to \varphi_3 \quad \varphi_2 \to \varphi_3}{\varphi_3} \vee_e$
$\to$	$\dfrac{\boxed{\begin{array}{c}\varphi_1 \\ \vdots \\ \varphi_2\end{array}}}{\varphi_1 \to \varphi_2} \to_i$	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2} \to_e$
$\neg$	$\dfrac{\boxed{\begin{array}{c}\varphi \\ \vdots \\ \bot\end{array}}}{\neg\varphi} \neg_i$	$\dfrac{\varphi \quad \neg\varphi}{\bot} \neg_e$
$\bot$		$\dfrac{\bot}{\varphi} \bot_e$
$\neg\neg$		$\dfrac{\neg\neg\varphi}{\varphi} \neg\neg_e$

Table 1.5: Proof rules.

In the $\to_i$ rule, the box indicates that we can *temporarily* assume φ_1 and conclude φ_2 using no extra non-trivial information. The $\to_e$ is referred to by its Latin name, *modus ponens*.

Example 1. We can now use these proof rules along with $\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)$ as the premise to conclude $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3$.

$$\frac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_2 \wedge \varphi_3} \wedge_{e_2} \qquad \frac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_1} \wedge_{e_1}$$

$$\frac{\varphi_2 \wedge \varphi_3}{\varphi_2} \wedge_{e_1} \qquad \frac{\varphi_2 \wedge \varphi_3}{\varphi_3} \wedge_{e_2}$$

$$\frac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2} \wedge_i$$

$$\frac{\varphi_1 \wedge \varphi_2 \quad \varphi_3}{(\varphi_1 \wedge \varphi_2) \wedge \varphi_3} \wedge_i$$

1.4 Natural Deduction

If we can begin with some formulas $\phi_1, \phi_2, \ldots, \phi_n$ as our premises and then conclude φ by applying the proof rules established we say that $\phi_1, \phi_2, \ldots, \phi_n$ syntactically entail φ which is denoted by the following expression, also called a *sequent*:

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \varphi.$$

We can also infer some formula using no premises, in which case the sequent is $\vdash \varphi$.
Applying these proof rules involves the following general rule:

> We can only use a formula φ at a point if it occurs prior to it in the proof and if **no box enclosing that occurrence of φ has been closed already**.

Example. Consider the following proof of the sequent $\vdash p \vee \neg p$:

1.	$\neg(p \vee \neg p)$	assumption
2.	p	assumption
3.	$p \vee \neg p$	$\vee_{i_1}$ 2
4.	$\bot$	$\neg_e$ 3,1
5.	$\neg p$	$\neg_i$ 2–4
6.	$p \vee \neg p$	$\vee_{i_2}$ 5
7.	$\bot$	$\neg_e$ 6,1
8.	$\neg\neg(p \vee \neg p)$	$\neg_i$ 1–7
9.	$p \vee \neg p$	$\neg\neg_e$ 8

Example. Proof for $p \vdash \neg\neg p$ which is $\neg\neg_i$, a derived rule

1.	p	premise
2.	$\neg p$	assumption
3.	$\bot$	$\neg_e$ 1, 2
4.	$\neg\neg p$	$\neg_i$ 2–3

Example. A useful derived rule is *modus tollens* which is $p \to q, \neg q \vdash \neg p$:

1.	$p \to q$	premise
2.	$\neg q$	premise
3.	p	assumption
4.	q	$\to_e$ 3,1
5.	$\bot$	$\neg_e$ 4,2
6.	$\neg p$	$\neg_i$ 3–5

Example. $\neg p \wedge \neg q \vdash \neg(p \vee q)$:

1.	$\neg p \wedge \neg q$	premise
2.	$p \vee q$	assumption
3.	p	assumption
4.	$\neg p$	$\wedge_{e_1}$ 1
5.	$\bot$	$\neg_e$ 3,4
6.	$p \to \bot$	$\to_i$ 3–5
7.	q	assumption
8.	$\neg q$	$\wedge_{e_2}$ 1
9.	$\bot$	$\neg_e$ 7,8
10.	$q \to \bot$	$\to_i$ 7–9
11.	$\bot$	$\vee_e$ 2,6,10
12.	$\neg(p \vee q)$	$\neg_i$ 2–11

1.5 Soundness and Completeness of our proof system

A proof system is said to be sound if everything that can be derived using it matches the semantics.

Soundness: $\Sigma \vdash \varphi$ implies $\Sigma \models \varphi$

The rules that we have chosen are indeed individually sound since they ensure that if for some assignment the premises evaluate to 1, so does the inference. Otherwise they rely on the notion of contradiction and assumption. Hence, soundness for any proof can be shown by inducting on the length of the proof.

A complete proof system is one which allows the inference of *every* valid semantic entailment:

Completeness: $\Sigma \models \varphi$ implies $\Sigma \vdash \varphi$

Let's take some example of semantic entailment. $\Sigma = \{p \to q, \neg q\} \models \neg p$.

p	q	$p \to q$	$\neg q$	$\neg p$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	1	0	0

As we can see, whenever both $p \to q$ and $\neg q$ are true, $\neg p$ is true. The question now is how do we derive this using proof rules? The idea is to 'mimic' each row of the truth table. This means that we assume the values for p, q and try to prove that the formulae in Σ imply φ[1]. And to prove an implication, we can use the $\to_i$ rule. Here's an example of how we can prove our claim for the first row:

1.	$\neg p$	given
2.	$\neg q$	given
3.	$(p \to q) \wedge \neg q$	assumption
4.	$\neg p$	1
5.	$((p \to q) \wedge \neg q) \to \neg p$	$\to_i$ 3,4

Similarly to mimic the second row, we would like to show $\neg p, q \vdash ((p \to q) \wedge \neg q) \to \neg p$. Actually for every row, we'd like to start with the assumptions about the values of each variable, and then try to prove the property that we want.

1.	$\neg p$	given
2.	$\neg q$	given
3.	$(p \to q) \wedge \neg q$	assumption
4.	$\neg p$	1
5.	$((p \to q) \wedge \neg q) \to \neg p$	$\to_i$ 3,4

1.	$\neg p$	given
2.	q	given
3.	$(p \to q) \wedge \neg q$	assumption
4.	$\neg p$	1
5.	$((p \to q) \wedge \neg q) \to \neg p$	$\to_i$ 3,4

1.	p	given
2.	$\neg q$	given
3.	$(p \to q) \wedge \neg q$	assumption
4.	$p \to q$	$\wedge_{e_1}$ 3
5.	q	$\to_e$ 1,4
6.	$\bot$	$\neg_e$ 2,5
7.	$\neg p$	$\bot_e$ 6
8.	$((p \to q) \wedge \neg q) \to \neg p$	$\to_i$ 3,4,5,6,7

1.	p	given
2.	q	given
3.	$(p \to q) \wedge \neg q$	assumption
4.	$p \to q$	$\wedge_{e_1}$ 3
5.	$\neg q$	$\wedge_{e_2}$ 3
6.	q	$\to_e$ 1,4
7.	$\bot$	$\neg_e$ 2,6
8.	$\neg p$	$\bot_e$ 7
9.	$((p \to q) \wedge \neg q) \to \neg p$	$\to_i$ 3,4,5,6,7,8

Figure 1.2: Mimicking all 4 rows of the truth table

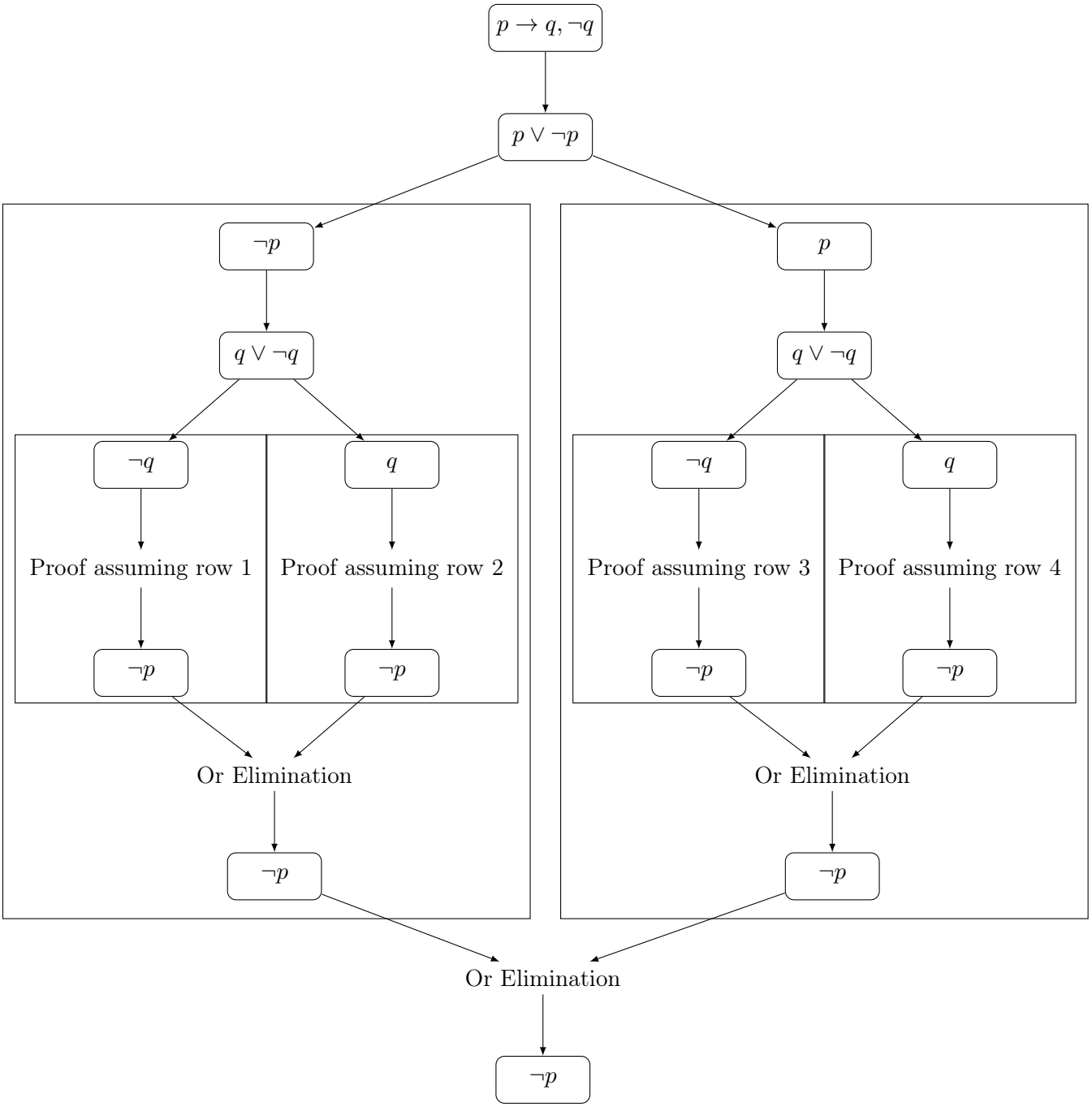
This looks promising, but we aren't done, we have only proven our formula under all possible assumptions, but we haven't exactly proven our formula from nothing given. But note that the reasoning we are doing looks a lot like case work, and we can think of the $\vee_e$ rule. In words, this rule states that if a formula is true under 2 different assumptions, and one of the assumptions is always true, then our formula is true. So if we just somehow rigorously show at least one of our row assumptions is always true, we will be able to clean up our proof using the $\vee_e$ rule.

But as seen above, we were able to show a proof for the sequent $\vdash \varphi \vee \neg\varphi$. If we just recursively apply this property for all the variables we have, we should be able to capture every row of truth table. So combining this result, our proofs for each row of the truth table, and the $\vee_e$ rule, the whole proof is constructed as below. The only thing we need now is the ability to construct proofs for each row given the general valid formula $\bigwedge_{\phi \in \Sigma} \phi \to \varphi$.

This can be done using **structural induction** to prove the following:
Let φ be a formula using the propositional variables $p_1, p_2, \ldots, p_n$. For any assignment to these variables define $\hat{p}_i = p_i$ if p_i is set to 1 and $\hat{p}_i = \neg p_i$ otherwise, then:

$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \varphi$ is provable if φ evaluates to 1 for the assignment
$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\varphi$ is provable if φ evaluates to 0 for the assignment.

[1] Σ semantically entails φ is equivalent to saying intersection of formulae in Σ implies φ is valid


$p \to q, \neg q$
$p \vee \neg p$
$\neg p$
$q \vee \neg q$
$\neg q$
q
Proof assuming row 1
Proof assuming row 2
$\neg p$
$\neg p$
Or Elimination
$\neg p$
p
$q \vee \neg q$
$\neg q$
q
Proof assuming row 3
Proof assuming row 4
$\neg p$
$\neg p$
Or Elimination
$\neg p$
Or Elimination
$\neg p$

Proof Rules

$\wedge_i$ $\dfrac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2}$

$\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_1}$ $\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_2}$ $\wedge_e$

$\vee_i$ $\dfrac{\varphi_1}{\varphi_1 \vee \varphi_2}$ $\dfrac{\varphi_2}{\varphi_1 \vee \varphi_2}$

$\dfrac{\varphi_1 \vee \varphi_2 \quad \varphi_1 \to \varphi_3 \quad \varphi_2 \to \varphi_3}{\varphi_3}$ $\vee_e$

$\to_i$ $\dfrac{\boxed{\begin{array}{c}\varphi_1 \\ \vdots \\ \varphi_2\end{array}}}{\varphi_1 \to \varphi_2}$

$\dfrac{\varphi_1 \to \varphi_2 \quad \varphi_1}{\varphi_2}$ $\to_e$

Modus Ponens

$\neg_i$ $\dfrac{\boxed{\begin{array}{c}\varphi \\ \vdots \\ \bot\end{array}}}{\neg\varphi}$

$\dfrac{\varphi \quad \neg\varphi}{\bot}$ $\neg_e$ $\bot_i$

$\neg\neg_i$ $\dfrac{\varphi}{\neg\neg\varphi}$

$\dfrac{\neg\neg\varphi}{\varphi}$ $\neg\neg_e$

$\dfrac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_1}\wedge_{e_1}$ $\dfrac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_2 \wedge \varphi_3}\wedge_{e_2}$ $\dfrac{\varphi_2 \wedge \varphi_3}{\varphi_3}\wedge_{e_2}$ $\dfrac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_2 \wedge \varphi_3}\wedge_{e_2}$ $\dfrac{\varphi_2 \wedge \varphi_3}{\varphi_2}\wedge_{e_1}$

$\wedge_i$ $\varphi_2 \wedge \varphi_3$

$\wedge_i$ $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3$

$\varphi_1 \wedge (\varphi_2 \wedge \varphi_3) \longrightarrow (\varphi_1 \wedge \varphi_2) \wedge \varphi_3$

$\vdash \varphi \vee \neg\varphi$

$\{\varphi_0, \varphi_1\} \vdash \varphi_2$

φ_0 --- premise

φ_1 --- premise

⋮

φ_2 --- conclusion

$\Sigma \vdash \varphi$

Σ: set of formulas

$\vdash$: syntactic entailment

$p, q \vdash p \wedge q$

$$\frac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2}$$

$\Sigma \models \varphi$

semantic entailment

$\Sigma \vdash \varphi$ implies $\Sigma \models \varphi$ — soundness of proof rules

$\Sigma \not\vdash \varphi$ implies $\Sigma \not\models \varphi$ — completeness of proof rules

$\Sigma = \{p \to q, \neg q\} \not\models \neg p$

p	q	$p \to q$	$\neg q$	$\neg p$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	1	0	0

$(p \to q) \wedge \neg q$

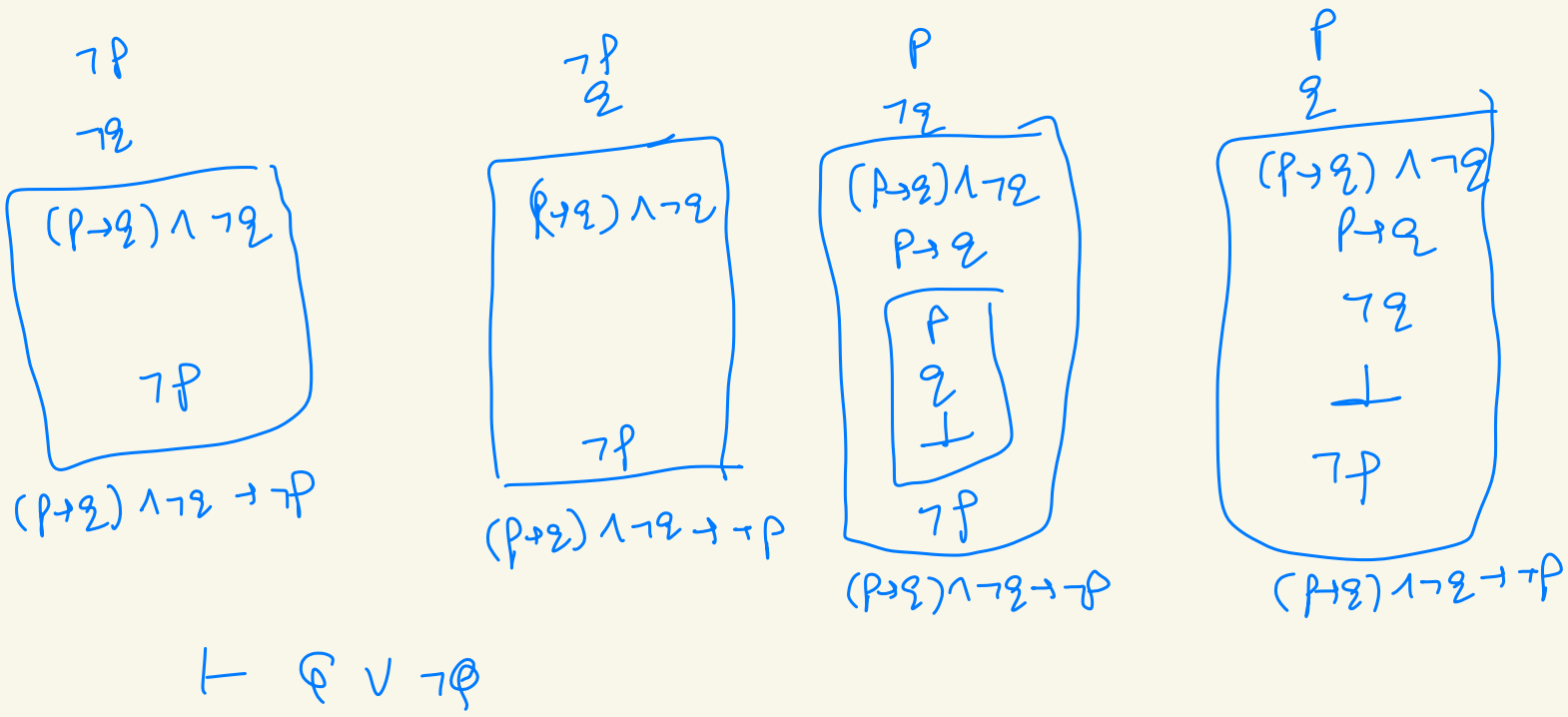
⋮

$\neg p$

$(p \to q) \wedge \neg q \to \neg p$

$\neg p, \neg q \vdash p \to q$

$\neg p, \neg q \vdash \neg q$



$\vdash p \vee \neg p$

To show
$Q_1, Q_2, Q_3 \vdash Q_4$
it suffices to show
$\vdash (Q_1 \wedge Q_2 \wedge Q_3) \rightarrow Q_4$
and vice versa

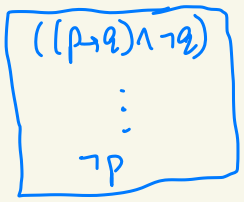
To show
$Q_1, Q_2, Q_3 \not\vdash Q_4$
it suffices to show
$\not\models (Q_1 \wedge Q_2 \wedge Q_3) \rightarrow Q_4$
and vice-versa

$p \rightarrow q, \neg q \models \neg p$ } semantic entailment
implies
$p \rightarrow q, \neg q \vdash \neg p$ } syntactic entailment
implies
$p \rightarrow q, \neg q \models \neg p$ } semantic entailment

} Completeness
} Soundness

To prove: $p \rightarrow q, \neg q \vdash \neg p$

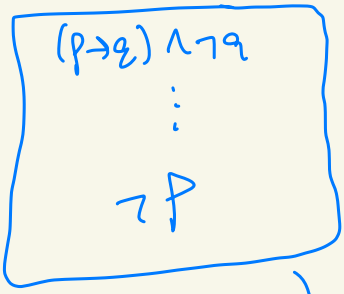
Step 1: First show: $\vdash ((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$



$((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$

Step 2: Construct a new proof using step 1 as sub-proof

$p \to q$
$\neg q$
$(p \to q) \wedge \neg q$ .. $\wedge_i$ rule



$((p \to q) \wedge \neg q) \to \neg p$
$\neg p$ ... $\neg_e$ rule

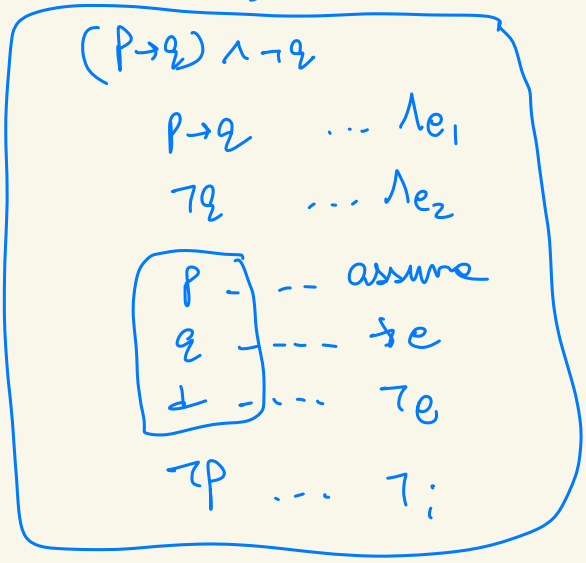
Crucial link = Step 1

Step 1 : To show

$\vdash ((p \to q) \wedge \neg q) \to \neg p$

many proofs possible

mimic each row of truth table
Brute force approach



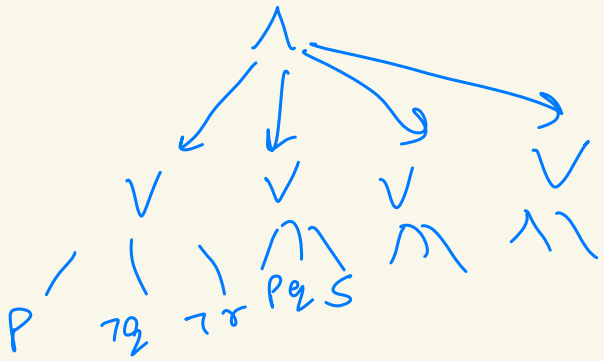
Requires ingenuity to get a short proof

What if a formula is neither valid nor contradiction?
Can we reason directly about satisfiability

φ is valid iff $\neg\varphi$ is not-sat.

Negation Normal form (NNF) → negation is only with literals (not with formulas)
(Use Demorgan's law)

Conjunctive Normal form (CNF)

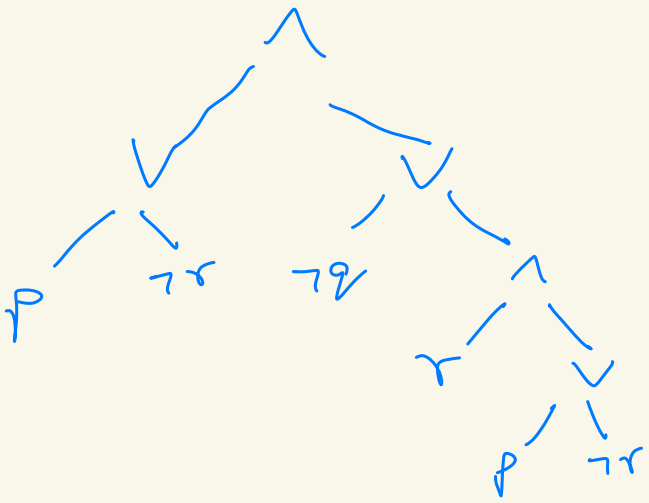


Normal forms

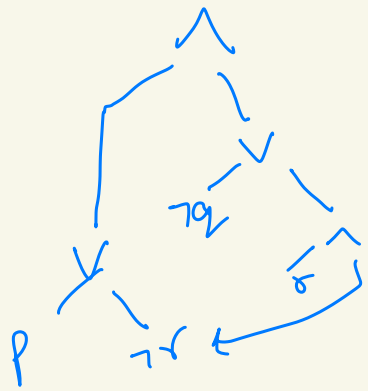
Negation Normal form (NNF) :

$(p \vee \neg r) \wedge (\neg q \vee (r \wedge (p \vee \neg r)))$

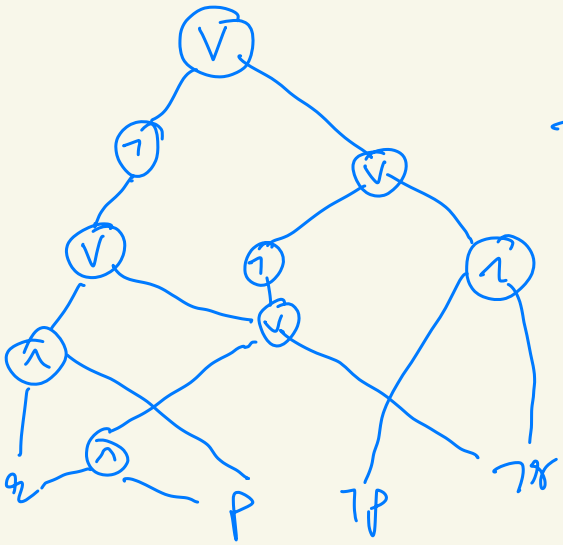
True representation



DAG representation:



Given a DAG representing a prop. logic formula with only $\wedge, \vee, \neg$ nodes, can we efficiently get a DAG representing a semantically equivalent NNF formula?



→ Create a copy with inverted operators and literals (¬ removed)
Then start joining

Literal : variable or its complement

Clause : Disjunction of literals

Cube : Conjunction of literals

Conjunctive Normal form (CNF) : Conjunction of clauses
Product of sums

$(p \vee q \vee \neg r) \wedge (q \vee r \vee \neg s \vee t)$

Disjunctive Normal form (DNF) : Disjunction of clauses
(Sum of products)

$(\neg p \wedge \neg q \wedge r) \vee (\neg q \wedge \neg r \wedge s \wedge \neg t)$

$p, \neg p, q, \neg r$

$(p \vee q \vee \neg r)$, ~~$(q \vee \neg q \vee r)$~~ ✓

$(p \wedge q \wedge \neg r)$, ~~$(q \wedge \neg q \wedge \neg r)$~~ ✗

Validity checking → DNF

Satisfiability checking → CNF

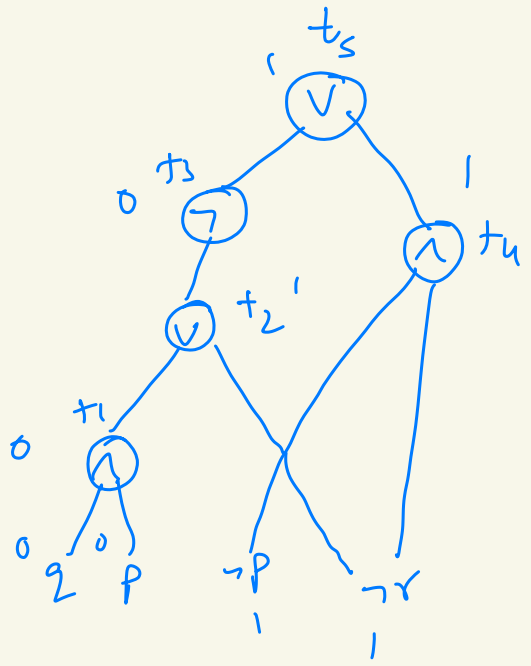
$\varphi(p,q,r)$ → Tseitin Encoding → $\varphi'(p,q,r, t_1, \ldots, t_m)$

Equisatisfiable

not necessarily semantically equivalent

Size of φ' linear in size of φ.

Tseitin encoding



$$(t_1 \leftrightarrow p \wedge q) \wedge$$
$$(t_2 \leftrightarrow t_1 \vee \neg r) \wedge$$
$$(t_3 \leftrightarrow \neg t_2) \wedge$$
$$(t_4 \leftrightarrow \neg p \wedge \neg r) \wedge$$
$$(t_5 \leftrightarrow t_3 \vee t_4) \wedge t_5$$

$p=q=r=0$ ← unique mapping → $p=q=r=0$

$t_1=0,\ t_2=1,\ t_3=0,\ t_4=1$

$t_5=1$

Bijection between satisfiability assignments of φ and φ'

(Stronger than equisatisfiability)

Towards checking satisfiability of CNF

$$(\neg x_3 \vee x_3) \wedge (x_5) \wedge (\neg x_5 \vee x_1)$$
$$(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_4 \vee x_5 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_5)$$

Horn Clause : At most one unnegated var. per clause

$$x_1 \wedge x_2 \rightarrow x_3$$
$$x_4 \wedge x_3 \rightarrow x_5$$
$$x_1 \wedge x_5 \rightarrow \bot$$
$$\top \rightarrow x_5$$
$$x_5 \rightarrow x_1$$

Horn formula

$x_1 = x_5 = x_3 = 1$

$\rightarrow x_2 = x_4 = 0$

n vars, k clauses

Davis Putnam Logemann Loveland (DPLL)

$C_1 \wedge C_2 \wedge \ldots \wedge C_k$

SAT (φ, PA) returns (status, assignment)

status: sat / unsat

assignment: $x_1 = 0, x_2 = 1, x_3 = 0$

0. If $\varphi = T$ return (sat, PA)
   If $\varphi = \bot$ return (unsat, PA)

1. If C_i is a unit clause (clause with single literal l)
   return SAT $(\varphi[l=1], PA \cup \{l=1\})$

   unit propagation — simplify φ after setting $l=1$

2. If a literal l does not appear negated in any clause

   pure literal elimination

   return SAT $(\varphi[l=1], PA \cup \{l=1\})$

3. $x :=$ choose_variable (φ) — decision
   $v :=$ choose_value $() \ldots \in \{0,1\}$
   if SAT $(\varphi[x=v], PA \cup \{x=v\})$.status = sat
   return (sat, $PA \cup \{x=v\}$)
   else if SAT $(\varphi[x=1-v], PA \cup \{x=1-v\})$ status = sat
   return (sat, $PA \cup \{x=1-v\}$)
   else return (unsat, PA) — backtrack

Homework 1 Solutions

Question 1 : Implication "Equations" 15 points

In this question, we will consider "implications" in the same spirit as "equations" between unspecified propositional formulas. You can think of these as "equations" where the unknowns are propositional formulas themselves, and the $=$ symbol has been replaced by $\rightarrow$.

Let φ_1 and φ_2 be unknown propositional formulas over $x_1, x_2, \ldots x_n$. Consider the following implications labelled (1a), (1b) through (na), (nb). A solution to this set of simultaneous implications is a pair of specific propositional formulas (φ_1, φ_2) such that all implications are **valid**, i.e. evaluate to true for all assignments of variables.

(1a)	$(x_1 \wedge \varphi_1) \rightarrow \varphi_2$	(1b)	$(x_1 \wedge \varphi_2) \rightarrow \varphi_1$
(2a)	$(x_2 \wedge \varphi_1) \rightarrow \varphi_2$	(2b)	$(x_2 \wedge \varphi_2) \rightarrow \varphi_1$
	$\vdots$		$\vdots$
(na)	$(x_n \wedge \varphi_1) \rightarrow \varphi_2$	(nb)	$(x_n \wedge \varphi_2) \rightarrow \varphi_1$

In each of the following questions, you must provide complete reasoning behind your answer. Answers without reasoning will fetch 0 marks.

1. *[5 marks]* Suppose we are told that φ_1 is $\bot$. How many semantically distinct formulas φ_2 exist such that implications (1a) through (nb) are valid?
2. *[2 marks]* Answer the above question, assuming now that φ_1 is $\top$.
3. *[5 marks]* If we do not assume anything about φ_1, how many semantically distinct pairs of formulas (φ_1, φ_2) exist such that all the implications are valid?
4. *[3 marks]* Does there exist a formula φ_1 such that there is exactly one formula φ_2 (modulo semantic equivalence) that can be paired with it to make (φ_1, φ_2) a solution of the above implications?

Solution: First note that the entire space of assignments of $x_1, \ldots x_n$ can be partitioned into sub-spaces where x_1 is true, $\neg x_1 \wedge x_2$ is true, $\neg x_1 \wedge \neg x_2 \wedge x_3$ is true, and so on until $\neg x_1 \wedge \cdots \neg x_n$ is true.

1. All implications in the left column reduce to $\bot \rightarrow \varphi_2$. Every such implication is trivially valid (since $\neg\bot \vee \varphi_2$ is true for all φ_2 for all assignments of variables). All implications in the right column reduce to $(x_i \wedge \varphi_2) \rightarrow \bot$. This implication is valid iff $(x_i \wedge \varphi_2)$ evaluates to false for all assignments of the variables. Using the partitioning of variable assignments mentioned above, φ_2 must evaluate to false whenever any of the variables $x_1, \ldots x_n$ evaluates to true. Hence, the semantics of φ_2 is determined for all assignments other than $x_1 = \cdots x_n = 0$. Since φ_2 can have two different truth values (0 or 1) for this assignment, only 2 distinct formulas φ_2 are possible.

2. The right column of implications are trivially valid if $\phi_1 \leftrightarrow \top$ is valid. Using reasoning similar to that above for the left column, once again only 2 distinct formulas φ_2 are possible.

3. Note that $(x_i \wedge \phi_j) \rightarrow \phi_k$ is semantically equivalent to $x_i \rightarrow (\phi_j \rightarrow \phi_k)$. This can be easily checked through truth tables. Therefore, if implications (1a) and (1b) are both valid, then $x_1 \rightarrow (\varphi_1 \leftrightarrow \varphi_2)$ must also be valid. Similarly, for implications (2a) and

(2b), and so on until (na) and (nb). Thus, if all the implications are to be valid, then $\varphi_1 \leftrightarrow \varphi_2$ must evaluate to true for all assignments of variables where at least one of $x_1, \ldots x_n$ is true. In other words, for every φ_1, the semantics of φ_2 can potentially differ from that of φ_1 only for the assignment $x_1 = \cdots x_n = 0$. Therefore, there are only two possible φ_2 for every φ_1. Since the total number of semantically distinct formulas φ_1 on n variables is 2^{2^n} (why?), the total count of semantically distinct pairs (φ_1, φ_2) is $2 \times 2^{2^n} = 2^{2^n+1}$.

4. The above implications allow φ_2 to evaluate to any value in $\{0, 1\}$ when $x_1 = x_2 = \cdots x_n = 0$, regardless of what φ_1 is. Therefore, there are always at least two solutions to the given implications.

Question 2 : Balanced Parentheses 25 points

A *decision problem* is a computational problem that has a "yes" or "no" answer for every given input. An input to a decision problem is often encoded simply as a string of 0's and 1's. Not surprisingly, we can encode *some* decision problems P in propositional logic. Specifically, we construct a propositional logic formula φ_P over as many propositional variables as the count of letters (0s and 1s) in the binary string encoding the input, such that if the binary string is interpreted as an assignment to the propositional variables, then the "yes"/"no" answer to P is obtained from the value given by the semantics of φ_P. If the semantics of φ_P evaluates to 0 (or "false") for the given input, then the output of P for that input is "no"; else, the output is "yes".

Consider the following decision problem of checking if a given string of parentheses is balanced. Given a binary string of length $n, n \geq 1$, where 0 encodes '(' and 1 encodes ')', we say that the string has balanced parentheses if and only if:

- The number of open parentheses, represented by 0s, in the entire string equals the number of closing parentheses, represented by 1s.
- In every proper prefix of the string, the number of open parentheses is at least as much as the number of closing parentheses.

We wish to encode the above problem in propositional logic. Recall from your data structures and algorithms course that checking balanced parentheses can be solved in polynomial time (in fact, with linear time complexity). We want this to be reflected in some aspects of your solution to this problem.

In class, we saw that every propositional formula φ can be represented by a parse tree whose internal nodes are labelled by connectives and whose leaves are labelled by propositional variables. Sometimes, two different sub-trees in a parse tree may be identical. In such cases, it makes sense to represent the formula as a directed acyclic graph (DAG), where syntactically common sub-formulas are represented exactly once. As an example, consider the parse tree and corresponding DAG in Fig. 1, both representing the formula $(r \vee (p \vee q)) \wedge (p \vee q)$. Note

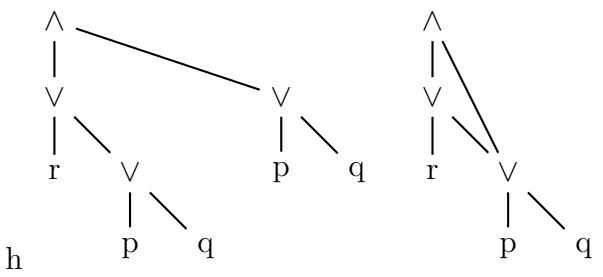


Figure 1: A parse tree and corresponding DAG

that a DAG representation of a formula can be exponentially smaller than a parse tree representation. Furthermore, a DAG representation suffices to evaluate the semantics of a formula, since the semantics of a shared sub-formula needs to be evaluated only once. Thus, if the DAG representation of a formula is small, its semantics can be evaluated efficiently. We define the *DAG size* of a formula to be the number of nodes in its DAG representation. This is also the number of syntactically distinct sub-formulas in the formula.

(a) *[10 marks]* Encode the problem of checking balanced parentheses in a binary string of length n as a propositional logic formula whose DAG size is at most $\mathcal{O}(n^3)$. You must use only n propositional variables corresponding to the n bits in the input string, and the only connectives allowed in your formula are two-input $\vee, \wedge$ and $\neg$. Express the DAG size of your formula as a function of n using big-$\mathcal{O}$ notation, with a clear justification of how you obtained the size expression.

(b) *[5 marks]* Prove that your formula is unsatisfiable for all odd values of n.

(c) *[10 (+ bonus 5) marks]* Suppose your formula is represented as a DAG, as discussed above. Given an input string of 0s and 1s, there is a simple way to evaluate the semantics of the formula. Specifically, we evaluate DAG nodes bottom-up, starting from the leaves (these represent propositional variables whose values are given by the input string) and moving upwards until we reach the root. The value of the root gives the semantics of the formula. Normally, a DAG node (labelled $\wedge, \vee$ or $\neg$) can be evaluated only after all its children have been evaluated. However, if a $\vee$ (resp. $\wedge$) labelled node has a child that has evaluated to 1 (resp. 0), then the node itself can be evaluated to 1 (resp. 0) without evaluating its other child. This can allow us to find the value of the root without evaluating all nodes in the DAG.

What is the worst-case number of DAG nodes (as a function of n in big-$\mathcal{O}$ notation) that need to be evaluated for your formula in part (a), in order to find the value at the root node for any input string of length n? You must give justification for why you really need these many nodes to be evaluated in the worst-case. Answers without justification will fetch 0 marks.

You will be awarded bonus 5 marks if you can show that your formula requires evaluating only $\mathcal{O}(n))$ DAG nodes.

Solution: Let $T_{i,j}$ represent the subformula, which evaluates to $\top$ if the number of open brackets exceeds the number of closed brackets by j in the first i characters of the string. Our formula for balanced parenthesis becomes $T_{n,0}$.

To make this formula well-defined, we must define the complete set of $T_{i,j}$ for all values of i, j. For all values of $i > 0, j < 0$, we have $T_{i,j} = \bot$ and for all values of $j \neq 0$, we have $T_{0,j}$ as $\bot$ and $T_{0,0} = \top$ For all other values of i, j, we have $T_{i,j} = (x_i \wedge T_{i-1,j+1}) \vee (\neg x_i \wedge T_{i-1,j-1})$. This makes the formula well-defined.

Lemma (well-defined): The complete set of propositional variables in the above formula is contained in $\{x_i\}$. We prove this by induction. (Base Cases are $T_{0,j}$ and induction over i)

Lemma (Size): The number of distinct subformulae is in $\mathcal{O}(n^3)$. We prove this by explicitly upper bounding the number of distinct subformulae.

Lemma (Correctness): If $T_{n,0}$ evaluates to $\top$ if and only if the string is balanced. We prove this using induction over even values of n.

(b) In the same way, for part b, we do induction over odd values of n and show that the formula is unsat.

(c) Claim: This formula can be evaluated in $O(n)$ time.

Evaluation Algorithm: We evaluate x_n and, based on whether it is $\top$ or $\bot$, evaluate the appropriate branch in the formula.

We prove the above evaluation is correct using induction and has $O(n)$ time complexity.

Homework 2 Solutions

1. The Case of Dr. Equisemantic and Mr. Irredundant 15 points

Assume we have a countably infinite list of propositional variables $p_1, p_2, \ldots$. For this problem, by "formula", we always mean a finite string representing "syntactically-correct formula". Let Σ be a set of formulae. For any formula φ, we say $\Sigma \models \varphi$ (read as Σ semantically entails φ) if for any assignment α of the propositional variables that makes all the formulae contained in Σ true, α also makes φ true.

Let us call two sets of formulae Σ_1 and Σ_2 *equisemantic* if for every formula φ, we have $\Sigma_1 \models \varphi$ if and only if $\Sigma_2 \models \varphi$. Furthermore, let us call a non-empty set of formulae Σ *irredundant* if no formula σ in Σ is semantically entailed by $\Sigma \setminus \{\sigma\}$.

1. *[5 points]* Show that any set of formulae Σ must always be countable. This implies that we can enumerate the elements of Σ. Assume from now on that $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \ldots\}$.
2. Suppose we define Σ' as follows:

$$\begin{aligned}\Sigma' = \{&\sigma_1,\\ &(\sigma_1) \to \sigma_2,\\ &(\sigma_1 \wedge \sigma_2) \to \sigma_3,\\ &(\sigma_1 \wedge \sigma_2 \wedge \sigma_3) \to \sigma_4,\\ &\vdots\\ \}&\end{aligned}$$

   Suppose we remove all the tautologies from Σ' and call this reduced set Σ''. Prove that Σ'' is irredundant and equisemantic to Σ. You can proceed as follows:

   (a) *[5 points]* Show that a non-empty satisfiable set Γ with $|\Gamma| \geq 2$ is irredundant if and only if $(\Gamma \setminus \{\gamma\}) \cup \{\neg\gamma\}$ is satisfiable for every $\gamma \in \Gamma$.

   (b) *[5 points]* Use the above result to show that Σ'' is irredundant and equisemantic to Σ.

Solution:

1. Recall from your course on Discrete Structures that a set is *countable* if either it is finite, or if there is a bijection between the set and the set of natural numbers. Equivalently, a set is countable if there is an injective function from the set to the set of natural numbers. Recall also that the Cartesian product of two countable sets is countable, and the countable union of countable sets is also countable.

   Now, consider the set S of all finite strings. This set must be countable. To see why this is so, take the set S_1 of all strings of length 1. This set is countable because the number of propositional variables is countable and we only have an additional finite set of non-variable symbols (parentheses, logical and symbol, etc.). Now consider the set S_2 of all strings of length 2. Being the Cartesian product of two countable sets $(S_1 \times S_1)$, this set must also be countable. Inductively, the set S_n of all strings of length n, for any $n \geq 2$, is the Cartesian product of two countable sets $(S_{n-1} \times S_1)$, and hence must be countable. Clearly $S = \cup_{n=1}^{\infty} S_n$. Being a countable union of countable sets, the set S must therefore be countable.

The set of all syntactically-correct formulas is a (strict) subset of the set of all possible finite strings. Being a subset of a countable set, it must be countable. Σ, in turn, is a subset of the set of all syntactically-correct formulas. Being a subset of a countable set, Σ must be countable.

2. (a) Note first that since Γ is satisfiable, so is $\Gamma \setminus \{\gamma\}$ (every α that satisfies all formulas in Γ certainly satisfies all formulas in $\Gamma \setminus \{\gamma\}$). Now, if γ is not semantically entailed by $\Gamma \setminus \{\gamma\}$, there must be some assignment α that makes each formula in $\Gamma \setminus \{\gamma\}$ true but makes γ false. This same assignment α would thus make $\neg\gamma$ true, and hence would be a satisfying assignment for $(\Gamma \setminus \{\gamma\}) \cup \{\neg\gamma\}$. If this holds for every $\gamma \in \Gamma$, this means that no element of Γ is semantically entailed by the rest of the elements. Hence Γ is irredundant.

   To prove the other direction, suppose Γ is irredundant. By definition, no formula $\gamma \in \Gamma$ is semantically entailed by $\Gamma \setminus \{\gamma\}$. In other words, for every $\gamma \in \Gamma$, there exists an assignment α (dependent on γ in general) that satisfies every formula in $\Gamma \setminus \{\gamma\}$, but does not satisfy γ. Hence, this α satisfies all formulas in $(\Gamma \setminus \{\gamma\}) \cup \{\neg\gamma\}$. Since the above argument holds for all $\gamma \in \Gamma$, this proves the statement.

   (b) The fact that Σ'' is equisemantic to Σ is easy to see. We will first show that an assignment α satisfies all formulas in Σ iff it satisfies all formulas in Σ". Clearly, if each of the formulas in Σ evaluates to true for α, then both sides of each implication in Σ' evaluate to true for α. Hence, each of the formulas in Σ' is also true for the same assignment. Conversely, if each of the formulas in Σ' is true for assignment α, then σ_1 is true, and since $(\sigma_1) \rightarrow \sigma_2$ is true, this implies σ_2 is true. Using the same reasoning, it follows by induction that σ_n is true for all $n \geq 1$. Thus, each formula in Σ is true for the assignment α.

   Let S denote the set of satisfying assignments of Σ. We have just shown above that S is also the set of satisfying assignments of Σ'. Now suppose $\Sigma \models \varphi$. This is equivalent to saying that every assignment in S satisfies φ. Since S is also the set of satisfying assignments of Σ', this is equivalent to saying that $\Sigma' \models \varphi$. Hence, if $\Sigma \models \varphi$, then $\Sigma' \models \varphi$ too. A similar reasoning shows the result the other way round. Hence, Σ and Σ' are equisemantic.

   Since Σ'' is just Σ' without the tautologies, and tautologies, being always true, do not change equisemanticness, Σ'' is equisemantic to Σ.

   Let us now look at irredundancy. Let $\Sigma'' = \{\eta_1, \eta_2, \ldots\}$ in order after removing the tautologies from Σ'. Consider any $\eta_n \in \Sigma''$. Since η_n is not a tautology, there must be an assignment α that makes η_n false. By the semantics of $\rightarrow$, this assignment must make each of $\sigma_1, \ldots, \sigma_{n-1}$ true and σ_n false. Since α makes each of $\sigma_1, \ldots \sigma_{n-1}$ true, it satisfies all implications $\eta_1, \ldots \eta_{n-1}$ (both sides of implication having formulas in $\{\sigma_1, \ldots \sigma_{n-1}\}$, must evaluate to true). Similarly, α satisfies all implications $\eta_{n+1}, \ldots$, since the left side of each of these implications has σ_n conjuncted with other formualas, and σ_n evaluates to false under assignment α. Therefore, α is a satisfying assignment for $\Sigma'' \setminus \{\eta_n\}$. Since α also falsifies η_n, it immediately follows that α satisfies all formulas in $(\Sigma'' \setminus \{\eta_n\}) \cup \{\neg\eta_n\}$. Since the abov argument holds for every $\eta_n \in \Sigma$", we conclude that Σ'' has no element that is semantically entailed by the others. Hence Σ" is irredundant.

2. A follow-up of the take-away question of Tutorial 2 — 25 points

To recap from the take-away question of Tutorial 2, we will view the set of assignments satisfying a set of propositional formulae as a language and examine some properties of such languages.

Let $\mathbf{P}$ denote a countably infinite set of propositional variables $p_0, p_1, p_2, \ldots$. Let us call these variables positional variables. Let Σ be a countable set of formulae over these positional variables. Every assignment $\alpha : \mathbf{P} \to \{0, 1\}$ to the positional variable can be uniquely associated with an infinite bitstring w, where the i^{th} bit $w_i = \alpha(p_i)$. The language defined by Σ, also called $L(\Sigma)$, is the set of bitstrings w for which the corresponding assignment α, that has $\alpha(p_i) = w_i$ for each i, satisfies Σ, that is, for each formula $F \in \Sigma$, $\alpha \models F$. In this case, we say that $\alpha \models \Sigma$. Let us call the languages definable in this manner as PL-definable languages.

Example:

Let $\Sigma = \{p_0 \to p_1, p_1 \to p_2, p_2 \to p_3, \ldots\}$.

Then $L(\Sigma) = \{1111\ldots, 0111\ldots, 0011\ldots, 0001\ldots, \ldots, 0000\ldots\}$, or, to be precise, if we denote the infinite bitstring containing only 1s by 1^ω and the infinite bitstring containing only 0s by 0^ω, and the finite bitstring consisting of k 0s by 0^k, then $L(\Sigma) = \{0^k 1^\omega : k \in \mathbb{N}\} \cup \{0^\omega\}$.

(a) *[5 marks]* Show that the language L consisting of all infinite bitstrings except $000\ldots$ (the bitstring consisting only of zeroes) is **not** PL-definable. You may want to prove the following lemma in order to solve this question:

Lemma:

For every PL-definable language L and bitstring $x \notin L$ there exists a finite prefix y of x such that for any infinite bitstring w, $yw \notin L$ (yw refers to the concatenation of y and w).

(b) *[5 + 5 points]* Show that PL-definable languages are closed neither under countable union nor under complementation.

Hint: Try using the result proven in part (a)

(c) *[10 points]* Show that a PL-definable language either contains every bitstring or does not contain uncountably many bitstrings.

Hint: Try using the lemma proven in part (a)

(d) *[Bonus 10 points]* A student tries to extend the definition of PL-languages by allowing the use of "dummy" variables.

Let $\mathbf{X} = \{x_0, x_1, \ldots\}$ denote a countably infinite set of "dummy" variables and let Σ denote a countable set of formulae over both positional and dummy variables. An infinite bitstring w is in the language defined by Σ if and only if there exists an assignment $\alpha : \mathbf{P} \cup \mathbf{X} \to \{0, 1\}$ such that $\alpha \models \Sigma$ and $w_i = \alpha(p_i)$ for each i. Note that the assignment of "dummy" variables in $\mathbf{X}$ are not represented in w. Let us call the languages definable this way extended PL-definable languages, or EPL-definable languages.

Show that EPL and PL are equally expressive, ie every EPL-definable language is a PL definable language and vice versa. This means our attempt to strengthen PL this way has failed. You can use the following theorem without proof:

Theorem:

Let $S_0, S_1, S_2, \ldots$ denote an infinite sequence of non-empty sets of finite bitstrings such that for every $i > 0$ and for every bitstring $x \in S_i$ and every $j \leq i$, there exists a prefix y of x in S_j. Then there exists an infinite bitstring z such that every S_i contains a prefix of z.

Solution:

(a) Let us first prove the lemma mentioned in the question.

Lemma:

For every PL-definable language L and bitstring $x \notin L$ there exists a finite prefix y of x such that for any infinite bitstring w, $yw \notin L$ (yw refers to the concatenation of y

and w).

Proof:

Say $L = L(\Sigma)$ for some countable set of formulae Σ. If $x \notin L(\Sigma)$, then there must be some $F \in \Sigma$ such that $x \not\models F$. Since F is a formula, and all formulas are finite strings by definition, F contains only a finite number of positional variables $p \in \mathbf{P}$. Let the largest index of any positional variable present in F be n. For any infinite bitstring z with first $n+1$ bits being the same as x, the values of all the positional variables present in F are the same in both x and z, which means that $z \not\models F$ as well. This means that, if we let y denote the finite prefix of x consisting of the first $n+1$ bits of x, then for any infinite bitstring w, $yw \not\models F$, by the same argument. This means that $yw \notin L$, proving the lemma. In fact, by a similar argument, the lemma can be strengthened to stating that for any infinite bitstring $x \notin L$, there exists a finite set of positions S, such that for any infinite bitstring y, such that bits of y at the positions in S match those of x, $y \notin L$ as well.

Now, consider L as defined in the problem, ie consisting of every infinite bitstring, except $000\ldots$. Let $x = 000\ldots$. We will show that L is not PL-definable, by contradiction. Assume L is PL-definable. By the lemma we just proved, there exists a finite prefix y of x such that for any infinite bitstring w, $yw \notin L$, ie there are infinitely many bitstrings not in L. This contradicts the fact that $000\ldots$ is the only bitstring not in L. Therefore, L cannot be defined in PL.

(b) Consider the language $L = \{000\ldots\}$ (this language consists only of the infinite bitstring $000\ldots$). This language can be defined in PL as $L(\Sigma)$ where $\Sigma = \{\neg p_0, \neg p_1, \neg p_2 \ldots\}$. However, its complement is the language consisting of all infinite bitstrings other than $000\ldots$, which, as shown earlier, is not PL-definable. Therefore, PL-definable languages are not closed under complementation.

Consider the countably infinite family of languages $L_i = L(\{p_i\})$ for each $i \in \mathbb{N}$. Each L_i consists of strings where the i^{th} bit is 1, and clearly, each L_i is PL-definable. Let $L = \bigcup_{i=0}^{\infty} L_i$. L is the language consisting of all infinite bitstrings other than $000\ldots$, which, as shown earlier, is not PL-definable. Therefore, PL-definable languages are not closed under countable union.

(c) This follows directly from the lemma that was proven earlier. If a PL-definable language does not contain an infinite bitstring x, then there exists a finite prefix y of x such that for every infinite bitstring w, yw is not in the language. Since there are uncountably many infinite bitstrings w, there are uncountably many infinite bitstrings not in the language

(d) Firstly, it is easy to see that every PL-definable is also EPL-definable: PL is a special case of EPL where no dummy variables are used.

We will now show that every EPL-definable language is PL-definable. Say $\Sigma = \{F_0, F_1, \ldots\}$ is a countable set of formulae over the variables in $\mathbf{P} \cup \mathbf{X}$.

Consider $\Sigma' = \{\bigwedge_{j=0}^{i} F_j : i \in \mathbb{N}\} = \{F_0, F_0 \wedge F_1, F_0 \wedge F_1 \wedge F_2, \ldots\}$. We will show that $L(\Sigma) = L(\Sigma')$.

For any word w, $w \in L(\Sigma)$ if and only if there exists an assignment $\alpha : \mathbf{P} \cup \mathbf{X} \to \{0,1\}$ such that $w_i = \alpha(p_i)$ for each natural i and for each $F \in \Sigma$, $\alpha \models F$, ie $\alpha \models F_0$, $\alpha \models F_1$, and so on. This is equivalent to saying $\alpha \models F_0$, $\alpha \models F_0 \wedge F_1$, $\alpha \models F_0 \wedge F_1 \wedge F_2$, so on,

ie $w \in L(\Sigma')$. Therefore, $L(\Sigma) \subseteq L(\Sigma')$. In a similar manner, if $w \in L(\Sigma')$, then there cannot be any F_i such that the corresponding assignment $\alpha \not\models F_i$. Hence, $w \models \Sigma'$, and $L(\Sigma) \subseteq L(\Sigma')$. From the two inclusions proved above, we have $L(\Sigma) = L(\Sigma')$. We will henceforth work with Σ', and denote $F_0 \wedge F_1 \cdots \wedge F_i$ as F'_i. F'_i satisfy a special property - for any assignment α, if $\alpha \models F'_i$, then for every $j \leq i$, $\alpha \models F'_j$.

Some Notation:

- Let $Vars_d(F)$ denote the set of dummy variables whose indices are at most the largest index of a dummy variable in the formula F - for example, if $F = p_0 \vee x_0 \vee x_2$, then $Vars_d(F) = \{x_0, x_1, x_2\}$. We have $Vars_d(F'_i) \subseteq Vars_d(F'_{i+1})$ for every natural i.
- For any EPL formula F, let $Ass_d(F)$ denote the set of possible assignments to the set $Vars_d(F)$ of dummy variables
- For any formula F and assignment α to the dummy variables in F, let $F(\alpha)$ denote the formula obtained by substituting each dummy variable x with its value $\alpha(x)$. Note that $F(\alpha)$ no longer contains any dummy variables and only has positional variables, ie it is a formula in PL.

Define $\Sigma'' = \{ \bigvee_{\alpha \in Ass_d(F'_i)} F'_i(\alpha) : F'_i \in \Sigma' \}$. This is a countable set of PL-formulae. We will show that $L(\Sigma'') = L(\Sigma')$, which will imply that $L(\Sigma'') = L(\Sigma)$.

Say some word $w \in L(\Sigma')$. This means there exists some assignment $\alpha : \mathbf{P} \cup \mathbf{X} \to \{0, 1\}$ such that $\alpha(p_i) = w_i$ and $\alpha \models F'_i$ for each natural i. Now, since $\alpha \models F'_i$, we have $\alpha \models F'_i(\alpha)$, which implies that $\alpha \models \bigvee_{\alpha \in Ass_d(F'_i)} F'_i(\alpha)$ for each natural i, which means $w \in L(\Sigma'')$.

On the other hand, say $w \in L(\Sigma'')$. This means that $w \models \bigvee_{\alpha \in Ass(Vars_d(F'_i))} F'_i(\alpha)$ for each i, ie for each i, there exists an assignment $\alpha_i : Vars_d(F'_i) \to \{0, 1\}$ such that $w \models F'_i(\alpha)$. Let S_i denote the set of such assignments, interpreted as finite bitstrings (eg: $x_0 \to 0, x_1 \to 1, x_2 \to 0$ is interpreted as the bitstring 010). Now, for any $\alpha \in S_i$, $w \models F'_i(\alpha)$, which means that for any $j \leq i$, $w \models F'_j(\alpha)$ as well. This means there is a prefix of the bitstring corresponding to α in each S_j, for each $j \leq i$. Therefore, the theorem can be applied, and hence there is an infinite bitstring such that every S_i contains a prefix of it. This infinite bitstring denotes an assignment to the entire set of dummy variables, and hence there exists an assignment $\alpha : \mathbf{X} \to \{0, 1\}$ such that $w \models F'_i(\alpha)$ for each i, ie there exists an assignment $\alpha' : \mathbf{P} \cup \mathbf{X} \to \{0, 1\}$ such that $\alpha'(p_i) = w_i$ and $\alpha'(x_i) = \alpha(x_i)$ for each i, and, as we have seen, such an α will have $\alpha \models F'_i$ for each i. Therefore, $w \in L(\Sigma')$. This means that $w \in L(\Sigma'')$ if and only if $w \in L(\Sigma')$, and hence $L(\Sigma'') = L(\Sigma') = L(\Sigma)$.

Therefore, every EPL-definable language can also be defined in PL.

$x_1 \oplus x_2 \oplus \dots \oplus x_n$

NNF of size $4n$.

→ alphabet = finite set of symbols.

$\Sigma = \{a, b, c, d\}$

String: finite sequence of letters from Σ

$\Sigma = \{a\}$: Total set of strings = countably infinite

(strings are finite, but length not upper-bounded)

$a \cdot b$

Concatenation of two strings (not commutative) $a \cdot b \neq b \cdot a$

$(a \cdot b) \cdot c = a \cdot (b \cdot c)$

Algebra on strings

Concatenation operator = commutative not associative.

ε = empty string = "" (double quotes with nothing in between)

$\sigma \cdot \varepsilon = \sigma = \varepsilon \cdot \sigma$ for any string σ

No matter what the alphabet is, ε = remains same.

Language = A subset of the set of all finite strings on Σ.

↓ need not be finite

All the subsets of countably infinite → uncountably infinite

Σ^* = set of all finite strings on Σ, including ε

Σ^+ = set of all finite strings on Σ, excluding ε.

Σ = set of alphabets

ε = empty string

If Σ = single letter

Σ^*, Σ^+ countably infinite

(Also true if Σ = set of letters)

$\Sigma = \{a, b, c\}$

Is Σ^* countably infinite?

Use a different base system.

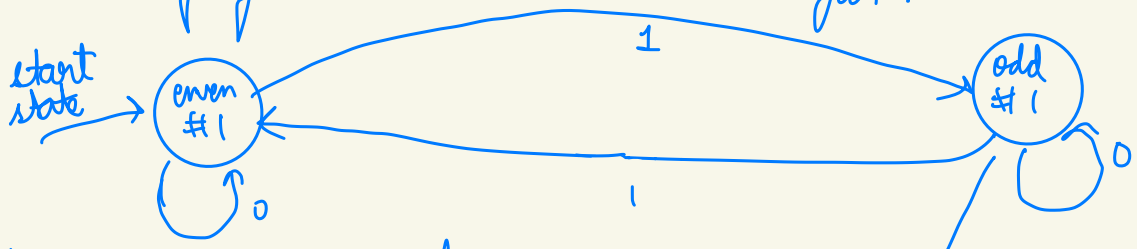
Countable set: injective function to Natural numbers. (surjective not needed)

Example $\Rightarrow$ $\Sigma = \{0, 1\}$

$L = \{ w \mid w \in \Sigma^* . \#1\text{s in } w \text{ is odd} \}$

odd number of 1s

State: summary of what has been seen so far.



Finite Automaton

circles = states

final / accepting state

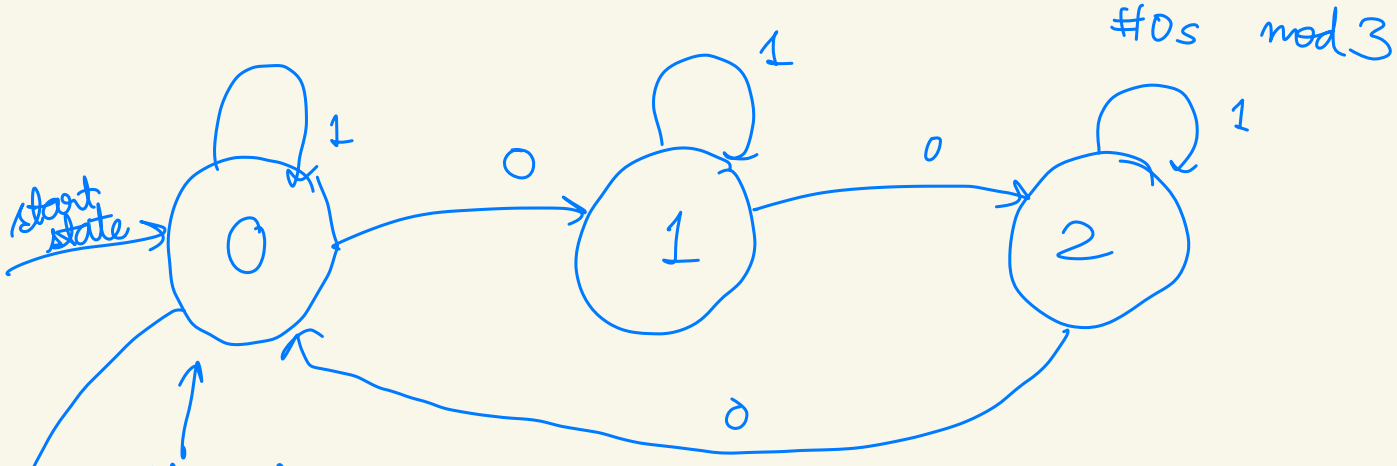
There cannot be two paths for a single string → Deterministic Finite Automaton (DFA)

finite number of states

single path for single string

$L_1 = \{ w \mid w \in \Sigma^*, \#0\text{s is a multiple of } 3 \}$

#0s mod 3



accepting state

if we land at this state at the end string $\in L_1$

* start state can be accepting state

Practice Problems 1

0. A Bit of Warm-Up

(a) A student has written the following propositional logic formula over variables x, y, z.

$$x \to \big((y \to \bot) \vee (\top \to z)\big)$$

Is it possible to write a semantically equivalent formula using only the operators $\neg$ and $\vee$?

(b) Consider the following CNF formula over propositional variables p, q, r, s, t.

$$\varphi(p,q,r,s,t) = (p \vee q \vee r) \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s) \wedge (\neg s \vee t) \wedge (\neg t \vee p)$$

Write a semantically equivalent DNF formula with only one cube. Try to avoid application of the distributive laws, as this can lead to a blow-up of intermediate formula sizes.

(c) We wish to show that $\varphi \models (\neg p \vee (q \wedge r \wedge s \wedge t))$. One way to show this is to show that $\varphi \wedge \neg(\neg p \vee (q \wedge r \wedge s \wedge t))$ is unsatisfiable. Since $\neg(\neg p \vee (q \wedge r \wedge s \wedge t)) = (p) \wedge (\neg q \vee \neg r \vee \neg s \vee \neg t)$, our problem reduces to showing unsatisfiability of $\varphi \wedge (p) \wedge (\neg q \vee \neg r \vee \neg s \vee \neg t)$. Show using repeated application of resolution rules (and nothing else) that the above formula is indeed unsatisfiable.

Solution:

1. Indeed, it's possible. Here are suggested steps.
   - Write every implication $\alpha \to \beta$ as $(\neg\alpha \vee \beta)$. This gives us the formula
     $$\neg x \vee \big((\neg y \vee \bot) \vee (\neg\top \vee z)\big)$$
   - Simplify every $(\alpha \vee \bot)$ to α. Similarly, $\neg\top$ can be replaced by $\bot$. This gives us
     $$\neg x \vee \big((\neg y) \vee (z)\big)$$
   - Finally, apply DeMorgan's laws to replace every $\alpha \vee \beta$ with $\neg(\neg\alpha \wedge \neg\beta)$. This gives
     $$\neg\big(x \wedge (y \wedge \neg z)\big)$$
2. Notice that φ can be written as
   $$(p \vee q \vee r) \wedge (p \to q) \wedge (q \to r) \wedge (r \to s) \wedge (s \to t) \wedge (t \to p)$$
   The five implications above force p, q, r, s, t to have the same value in any satisfying assignment of φ (try to convince yourself why this is the case). Therefore, for φ to be satisfied, we must have at least one of p, q, r to be true (because of the first clause of φ), and all of p, q, r, s, t must have the same value (because of the remaining clauses of φ). This implies that the only satisfying assignment of φ is $p = q = r = s = t = 1$. Hence, the semantically equivalent DNF formula is $p \wedge q \wedge r \wedge s \wedge t$.

3. We don't even need all clauses of φ to show unsatisfiability by resolution. Here are suggested resolution steps (alternative resolution steps are also possible).

Clause id	Clause	How obtained
1	$\neg p \vee q$	Given
2	p	Given
3	q	Resolvent of 1, 2
4	$\neg q \vee \neg r \vee \neg s \vee \neg t$	Given
5	$\neg r \vee \neg s \vee \neg t$	Resolvent of 3, 4
6	$\neg q \vee r$	Given
7	r	Resolvent of 3, 6
8	$\neg s \vee \neg t$	Resolvent of 7,5
9	$\neg r \vee s$	Given
10	s	Resolvent of 7, 9
11	$\neg t$	Resolvent of 8, 10
12	$\neg s \vee t$	Given
13	t	Resolvent of 10, 12
14	Empty clause ()	Resolvent of 11, 13

1. Core of unsatisfiability

A set $\mathcal{S}$ of propositional logic formulae is said to form a *minimal unsatisfiable core* if $\mathcal{S}$ is unsatisfiable (i.e. there is no assignment of values to variables that satisfies all formulas in $\mathcal{S}$), but every proper subset of $\mathcal{S}$ is satisfiable.

1. Given an unsatisfiable set $\mathcal{S}$ of propositional logic formulae, the minimal unsatisfiable core may not be unique. Give an example where $\mathcal{S}$ has at least two minimal unsatisfiable cores $\mathcal{C}_1$ and $\mathcal{C}_2$ such that $\mathcal{C}_1 \cap \mathcal{C}_2 \neq \emptyset$.
2. Show that for every $n > 0$, we can find a set of n propositional logic formulae that form a minimal unsatisfiable core. Thus, we can have minimal unsatisfiable cores of arbitrary finite size.

Solution:

1. This is fairly straightforward. Consider the set $\mathcal{S} = \{p, p \to q, \neg q, p \to r, \neg r\}$. The two minimal unsatisfiable cores are $\mathcal{C}_1 = \{p_1, p_1 \to p_2, \neg p_2\}$ and $\mathcal{C}_2 = \{p_1, p_1 \to p_3, \neg p_3\}$

2. The set of formulas $\mathcal{S}_n = \{p_1, p_1 \to p_2, p_2 \to p_3, \ldots p_{n-2} \to p_{n-1}, \neg p_{n-1}\}$ satisfies the condition of the question. Why is the entire set $\mathcal{S}_n$ is unsatisfiable? This should be easy to figure out.

   Every proper subset of $\mathcal{S}_n$ either has p_1 missing, $\neg p_3$ missing or some implication $p_i \to p_{i+1}$ missing for $1 \leq i \leq n-2$. We show that in each of these cases, the remaining subset of formulas is satisfiable.

   - If p_1 is missing, all the other formulas are satisfied by setting $p_2 = \ldots = p_{n-1} = 0$.
   - If $\neg p_{n-1}$ is missing, all the other formulas are satisfied by setting $p_1 = \ldots = p_{n-2} = 1$.
   - If $p_i \to p_{i+1}$ is missing for $1 \leq i \leq n-2$, then all other formulas are satisfied by setting $p_1 = \ldots p_i = 1$ and $p_{i+1} = \ldots p_{n-1} = 0$.

2. Logical interpolation

Let φ and ψ be propositional logic formulas such that $\models \varphi \rightarrow \psi$ (i.e. $\varphi \rightarrow \psi$ is a tautology). Let $Var(\varphi)$ and $Var(\psi)$ denote the set of propositional variables in φ and ψ respectively. Show that there exists a propositional logic formula ζ with $Var(\zeta) \subseteq Var(\varphi) \cap Var(\psi)$ such that $\models \varphi \rightarrow \zeta$ and $\models \zeta \rightarrow \psi$. This result is also known as *Craig's interpolation theorem* as applied to propositional logic. The formula ζ is called an *interpolant* of φ and ψ.

As a specific illustration of the above result, consider the formulas $\varphi = ((p \rightarrow q) \wedge (q \rightarrow \neg p))$ and $\psi = (r \rightarrow (p \rightarrow s))$, where p, q, r, s are propositional variables. Convince yourself that $\models \varphi \rightarrow \psi$ holds in this example. Note that $Var(\varphi) = \{p, q\}$ and $Var(\psi) = \{p, r, s\}$. Let $\zeta = \neg p$. Then $Var(\zeta) \subseteq Var(\varphi) \cap Var(\psi)$. Convince yourself that $\models \varphi \rightarrow \zeta$ and $\models \zeta \rightarrow \psi$ hold in this example.

Solution: Given a formula φ and a variable $v \in Var(\varphi)$, let $\varphi[v = \top]$ denote the formula obtained by replacing all occurrences of v in φ with $\top$, and then simplifying the resulting formula. By simplification, we mean the obvious ones like $\alpha \wedge \top = \alpha$, $\alpha \vee \top = \top$, $\alpha \wedge \neg\top = \bot$, $\alpha \vee \neg\top = \alpha$ for all sub-formulas α of φ. In a similar manner, we define $\varphi[v = \bot]$.

Note that $Var(\varphi[v = \top]) = Var(\varphi) \setminus \{v\}$ and similarly for $\varphi[v = \bot]$.

Now define the formula $\zeta = \bigvee_{v \in Vars(\varphi) \setminus Vars(\psi)} \bigvee_{a \in \{\bot, \top\}} \varphi[v = a]$. Notice that $Vars(\zeta) \subseteq Vars(\varphi) \cap Vars(\psi)$.

You should now be able to argue that (i) $\varphi \models \zeta$, and (ii) $\zeta \models \psi$. Proving (i) should be straightforward from the definition of ζ. To prove (ii), take any satisfying assignment of ζ. By definition of ζ, this gives an assignment of truth values to variables in $Vars(\varphi) \cap Vars(\psi)$ such that this assignment can be augmented with an assignment of truth values to variables in $Vars(\varphi) \setminus Vars(\psi)$ to satisfy the formula φ. However, since $\varphi \rightarrow \psi$ is a tautology, this (augmented) assignment also satisfies ψ. Recalling that satisfaction of ψ cannot depend on the assignment of truth values to variables not present in ψ, we conclude that the assignment of truth values to variables in $Vars(\varphi) \cap Vars(\psi)$ itself satisfies the formula ψ. Hence $\zeta \models \psi$.

3. DPLL with horns

The Horn-Sat problem entails checking the satisfiability of Horn formulas. We say a formula is a Horn formula if it is a conjunction ($\wedge$) of Horn clauses. A Horn clause has the form $\phi_1 \to \phi_2$ where ϕ_1 is either $\top$ or a conjunction ($\wedge$) of one or more propositional variables. ϕ_2 is either $\bot$ or a **single** propositional variable. In this context, answer the following questions

1. Let's try to solve the Horn-Sat problem using DPLL. We can convert each Horn clause into a CNF clause by simply rewriting $(a \to b)$ as $(\neg a \vee b)$, which preserves the semantics of the formula. The resultant formula is in CNF.
   We have seen in class that if DPLL always chooses to assign 0 to a decision variable before assigning 1 (if needed) to the variable, then DPLL will never need to backtrack when given a Horn formula encoded in CNF as input.
   Suppose our version of DPLL does just the opposite, i.e. it always assigns 1 to a decision variable before assigning 0 (if needed). How many backtracks are needed if we run this version of DPLL on the (CNF-ised version of) following Horn formulas, assuming DPLL always chooses the unassigned variable with the smallest subscript when choosing a decision variable?

   (a)
   $$\Big(\big(\bigwedge_{i=0}^{n-1} x_i\big) \to x_n\Big) \wedge \bigwedge_{i=0}^{n-1} (x_n \to x_i) \wedge \Big(\big(\bigwedge_{i=0}^{n} x_i\big) \to \bot\Big)$$

   (b)
   $$\bigwedge_{i=0}^{n-1} \big((x_i \to x_{n+i}) \wedge (x_{n+i} \to x_i) \wedge (x_i \wedge x_{n+i} \to \bot)\big)$$

2. Since we have a polynomial time formula for Horn-Sat, the next question is if it will allow us to solve the Boolean-SAT problem in polynomial time. Sadly, this is not the case. Find a boolean function that cannot be expressed by a Horn formula. Prove that no Horn formula can represent the given boolean function.

Solution:

1. It's best to write out the implications as clauses when you are trying to apply DPLL.

   In problem (a), no unit clauses or pure literals are obtained until all of x_0 through x_{n-1} are assigned the value 1, one at a time. Once x_{n-1} is assigned 1, unit propagation leads to a conflict – x_n must be assigned both 1 and 0 by unit propagation. This causes a backtrack, which ends up setting x_{n-1} to 0. Once this happens, unit propagation assigns the value 0 to x_n, resulting in the partial assignment $x_{n-1} = x_n = 0$, which satisfies all clauses. Therefore, there is exactly one backtrack.

   In part (b), every time a variable x_i for $0 \le i \le n-1$ is assigned 1, unit propagation causes x_{n+i} to be in conflict (must be assigned both 0 and 1). This induces a backtrack that sets x_i to 0, followed by unit propagation setting x_{n+i} to 0. DPLL will then choose x_{i+1} as the next decision variables, assign it 1 and the above process repeats. So, in this case, DPLL will incur n backtracks, one for each of $x_0, \ldots x_{n-1}$.

2. $F = (a \vee b)$. Any formula ϕ which is equivalent to a Horn formula has the following property: if v_1 and v_2 are two valuations that make ϕ evaluate to $\top$, then the valuation $v_1 \wedge v_2$ also makes ϕ evaluate to $\top$. Now consider $F = (a \vee b)$. F is satisfied by $(\top, \bot)$ and $(\bot, \top)$ but not $(\bot, \bot)$. Hence, no Horn Formula can represent F.

$$\bigwedge_{i=0}^{n-1} \big((x_i \to x_{n+i}) \wedge (x_{n+i} \to x_i) \wedge (x_i \wedge x_{n+i} \to \bot)\big)$$

$$\bigwedge_{i=0}^{n-1} \big((\neg x_i \vee x_{n+i}) \wedge (\neg x_{n+i} \vee x_i) \wedge (\neg x_{n+i} \vee \neg x_i) \big)$$

$$\big((\neg x_0 \vee x_n) \wedge (\neg x_n \vee x_0) \wedge (\neg x_n \vee \neg x_0) \big) \wedge \dots$$

$x_0 \to 1$

$\Rightarrow$ backtracks

$(x_n \wedge \neg x_n)$

4. A Game of Sudoku

Sudoku is a logic-based, combinatorial number-placement puzzle. In classic Sudoku, the objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contains all of the digits from 1 to 9 (and as one can naturally deduce; contain each of the digits exactly once). The puzzle setter provides a partially completed grid, which has a single solution for a well-posed puzzle. Encode the puzzle as a CNF formula. You are free to play around with different encodings; use auxiliary variables and attempt to make succinct (read; "optimised") formulae.

Solution: Let $p(i, j, k)$ be a propositional variable that asserts if the cell in row i and column j has the value k. One can clearly note that $1 \leq i, j, k \leq 9$. We encode the following constraints

- Every cell contains at least one number:

$$\phi_1 = \bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} \bigvee_{k=1}^{9} p(i, j, k)$$

- Every cell contains at most one number:

$$\phi_2 = \bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{y=x+1}^{9} (\neg p(i, j, x) \vee \neg p(i, j, y))$$

- Every row contains every number:

$$\phi_3 = \bigwedge_{i=1}^{9} \bigwedge_{n=1}^{9} \bigvee_{j=1}^{9} p(i, j, k)$$

- Every column contains every number:

$$\phi_4 = \bigwedge_{j=1}^{9} \bigwedge_{n=1}^{9} \bigvee_{i=1}^{9} p(i, j, k)$$

- Every 3×3 box contains every number:

$$\phi_5 = \bigwedge_{r=0}^{2} \bigwedge_{s=0}^{2} \bigwedge_{n=1}^{9} \bigvee_{i=1}^{3} \bigvee_{j=1}^{3} p(3r + i, 3s + j, n)$$

The final formula is as follows

$$\phi = (\phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5)$$

5. Poring over proofs

A student has given the following proof of $\top \vdash x \to \neg x$ What are the sources of problem in this proof (else we would be in serious trouble with true being equivalent to false).

```
1. top
     ------------------------------------------------------
2.   | x                 assumption                       |
     |  ------------------------------------------------  |
3.   |  | neg x          assumption                   |  |
4.   |  | bot            bot introduction rule on 2 and 3 |  |
     |  ------------------------------------------------  |
5.   | neg x             bot elimination rule on 4        |
6.   | bot               bot intro rule on 2 and 5        |
     ------------------------------------------------------
7. neg x                 neg intro rule on 2 -- 6
     ---------------------------------------------
8.   | x                 assumption                |
9.   | bot               bot intro rule on 7 and 8 |
10.  | neg x             bot elim rule on 9        |
     ---------------------------------------------
11.x -> neg x            impl intro rule on 8 to 10
```

Solution: Thankfully, there is an error in the proof.

Step 5 infers $\neg x$ outside the inner box by applying $\bot$-elimination rule on the result of Step 4. However, $\bot$ was derived in the scope of the inner box in Step 4. Deriving $\neg x$ inside the inner box after Step 4 would have been fine, but $\bot$-elimination doesn't allow us to infer $\neg x$ outside the scope of the inner box.

6. The Resolution Proof System

Consider the formula $\bigoplus_{i=1}^{n} x_i$, where $\oplus$ represents xor. It can be shown by induction that this is semantically equivalent to the PARITY function that evaluates to true if and only if an odd number of variables are assigned true.

Show that $\bigwedge_{\substack{S\subseteq\{1,2\ldots n\}\\ |S|=1 \mod 2}} \left(\bigvee_{i\in S} x_i \vee \bigvee_{j\notin S} \neg x_j \right)$ is the only CNF equivalent to φ, upto adding tautological clauses and repeating variables.

Hint: If two CNFs φ and ψ are equivalent, then for every clause α in φ, we have $\psi \vdash \alpha$ and for every clause β in ψ, we have $\varphi \vdash \beta$, where the proofs used are resolution proofs.

As a refresher, the resolution proof system is a system of proof rules for CNFs that is both sound and complete, ie if φ and ψ are CNFs, then $\varphi \models \psi$ if and only if $\varphi \vdash \psi$. We say $\varphi \vdash \psi$ iff for every clause c in ψ, we have $\varphi \vdash c$. The proof rules that can be applied in resolution proofs are:

1. (*Assumption*) For every clause c in φ, $\varphi \vdash c$
2. (*Resolution*) If we have $\varphi \vdash p \vee c_1$ and $\varphi \vdash \neg p \vee c_2$ for any clauses c_1 and c_2 and propositional variable p, then we can deduce $\varphi \vdash c_1 \vee c_2$.
3. (*Or Introduction*) For any clauses c_1 and c_2, if we have $\varphi \vdash c_1$, then we can deduce $\varphi \vdash c_1 \vee c_2$

Other than these, we are implicitly allowed to reorder any of the literals within a clause and any of the clauses within a CNF, and we can remove tautological clauses from the CNF (these are the clauses that contain a variable as well as it's negation).

Solution: An assignment α satisfies the formula if and only if the number of variables set to 1 by α is odd. An assignment does not satisfy the CNF given in the problem (call it φ) if and only if the some clause is falsified. This occurs if and only if an even number of variables are set to true. Therefore, an assignment satisfies φ if and only if an odd number of variables are set to 1. Therefore, φ in the question is semantically equivalent to $\bigoplus_{i=1}^{n} x_i$.

Assume there is some other CNF ψ also equivalent to $\bigoplus_{i=1}^{n} x_i$, where there are no tautological clauses in ψ and no repeated variables in a single clause. φ and ψ will be equivalent CNFs, ie $\varphi \models \psi$ and $\psi \models \varphi$.

From the first condition, we can conclude that for any clause c in ψ, we have $\varphi \vdash c$ by a resolution proof.

The **key idea** here is to note that when two clauses of χ are resolved, only tautological clauses result. Say there are clauses $c_1 = x \cup C_1$ and $c_2 = \neg x \cup C_2$ being resolved with respect to the variable x. There must be some other variable y such that y is present in C_1 and $\neg y$ is present in C_2 (or vice versa). This is as the number of non-negated variables in each clause in χ must be odd. Therefore, the resolvent $C_1 \cup C_2$, will contain both y and $\neg y$ and will therefore be a tautology.

Also note that since each clause of φ already contains every variable, applying *or introduction* on a clause either leaves it unchanged, or results in a tautology.

This means that the only proof rule that can be used to get non-tautological clauses is *Assumption*. Hence, if $\varphi \vdash c$, c must have been a clause of φ in the first place. This means that all the clauses in ψ must have already been present in φ.

Now, we must also have $\psi \vdash c$ for every clause c in φ. Since every clause in ψ is also a clause in φ, we can again conclude that *resolution* on them also produces only tautologies,

and similarly *or introduction* is useless as well. Therefore, if $\psi \vdash c$, then c must have been a clause of ψ. Therefore, every clause in φ must also be present in ψ.

Hence the clauses of ψ and φ must be the same, ie ψ and φ must be the same, meaning that the CNF obtained is unique.

7. A flavour of DFAs

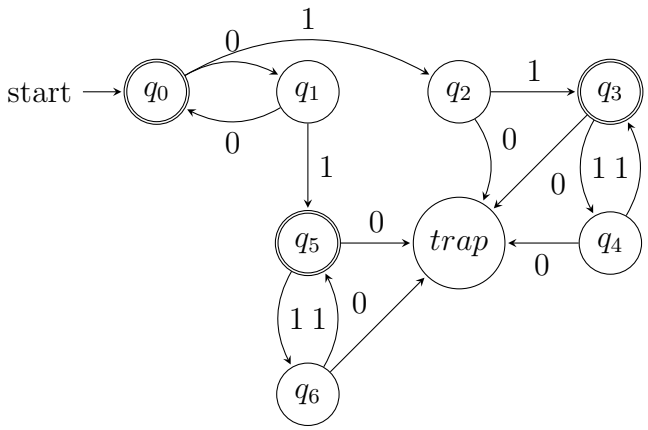
Consider a DFA M with the set of states Q, alphabet $\Sigma = \{0, 1\}$, transition function δ, and accepting state F.

Draw the state diagram for the DFA M that accepts the language $L = \{0^n 1^m \mid m + n \text{ is even}\}$. The DFA should recognize strings where the total number of '0's and '1's is even. Provide the diagram along with the state transitions.

Note: $L(M)$ represents the language accepted by DFA M, and $\sim_L$ is the equivalence relation induced by the language.

Solution:

The state diagram for the DFA M is as follows:



where the set of states $Q = \{q_0, q_1, \ldots q_6, trap\}$, alphabet $\Sigma = \{0, 1\}$, transition function δ, and accepting state $F = \{q_0, q_5, q_3\}$ and initial state is q_0.

State	Input 0	Input 1
q_0	q_1	q_2
q_1	q_0	q_5
q_2	trap	q_3
q_3	trap	q_4
q_4	trap	q_3
q_5	trap	q_6
q_6	trap	q_5

Explanation for the states:

- q0: Even no of 0's
- q1: Odd no of 0's
- q2: Odd no of 1's after even 0's
- q3: Even no of 1's
- q4: Odd no of 1's
- q5: Odd no of 1's after odd 0's
- q6: Even no of 1's
- trap: If 0's seen after 1's

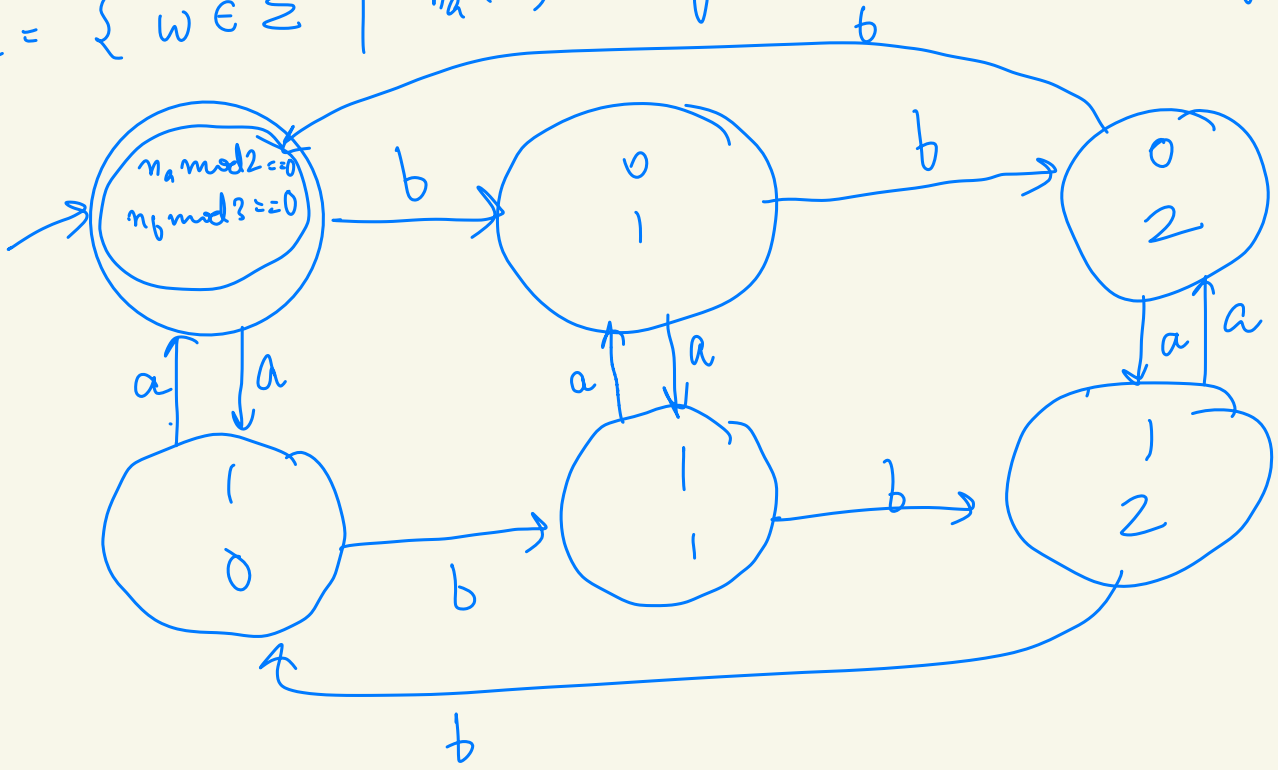
8. More solved problems on DFA

Please look at some solved examples from the textbook *Introduction to Automata Theory, Languages and Computation* by J.E. Hopcroft, R. Motwani and J.D. Ullman (2nd or later editions). For example, you can look at Examples 2.2. 2.4, 2.5 to get a feel of DFAs for simple languages.

Finite Automata

$\Sigma = \{a, b\}$

$L = \{ w \in \Sigma^* \mid n_a(w) \text{ div by } 2 \text{ and } n_b(w) \text{ div. by } 3 \}$



$\Sigma = \{a, b\}$

$L = \{ w \in \Sigma^* \mid n_{ab}(w) = n_{ba}(w) \}$

$w = abaabab$

$n_{ab}(w) = 3$

$n_{ba}(w) = 2$

$\} \; w \notin L$

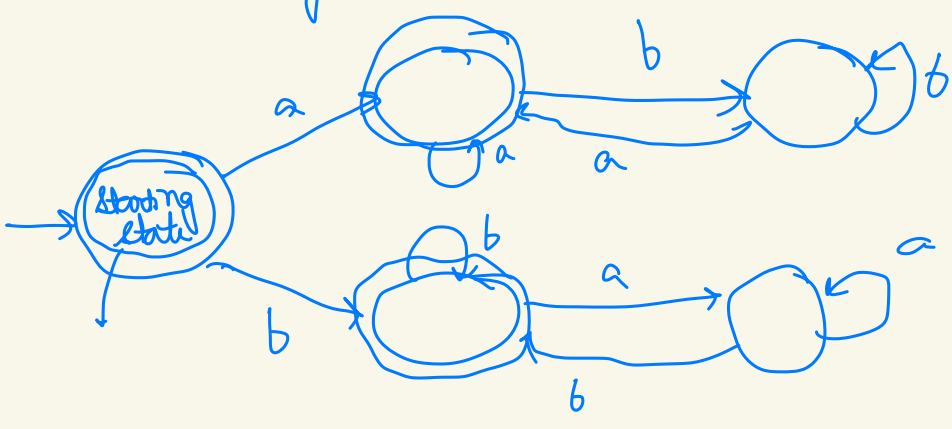
Change from ab to ab will have ba in between ab's and ba's will always alternate

Three states +1, -1, 0

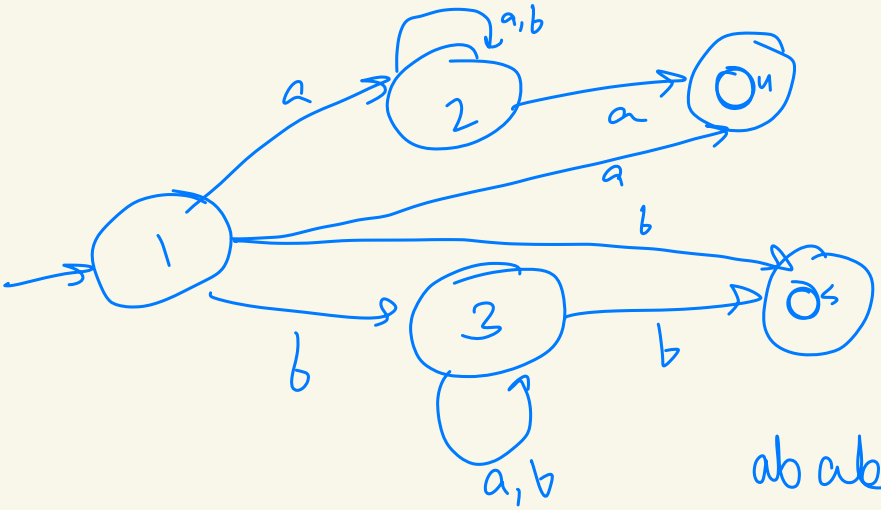
0 accepting state

$L' = \{ w \in \Sigma^* \mid n_a(w) = n_b(w) \}$

→ There is no finite automaton



Two accepting states.



Non-deterministic FA

ababa

If there is atleast one path which completes the string and ends up in an accepting states, we are done!

1 2 2 2 2

1 2 2 2 2 4

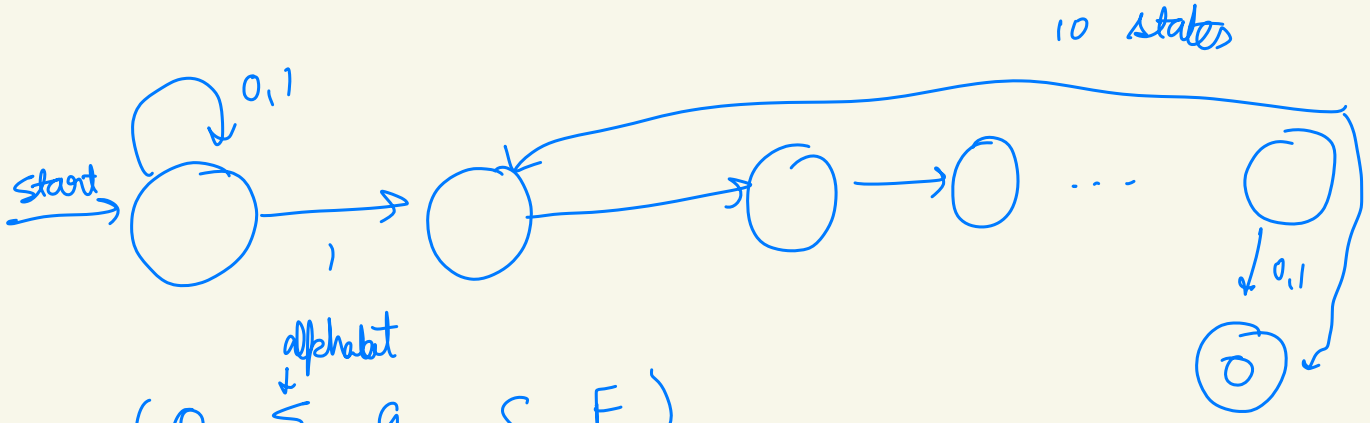
1 4 → stuck

1 2 2 4 → stuck

$\Sigma = \{0,1\}$

$L = \{ w \in \Sigma^* \mid |w| > 10,\ 10^{th} \text{ letter from end is a} \}$

2^{10} states remember 10 length strings



DFA : $(Q, \Sigma, q_0, \delta, F)$

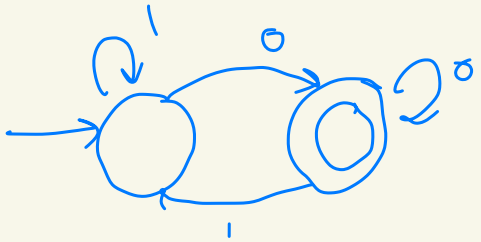
Q : set of states

Σ : alphabet

$q_0 \in Q$: initial state

δ : Transition fn $\delta : Q \times \Sigma \to Q$

$F \subseteq Q$: set of final states

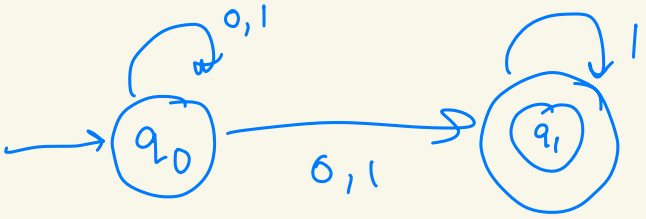


$\{ \{q_0, q_1\}, \{0,1\}, q_0, \delta, \{q_1\} \}$

δ	Q	Σ	Q
	q_0	1	q_0
	q_0	0	q_1
	q_1	0	q_1
	q_1	1	q_0

NFA : $(Q, \Sigma, Q_0 \subseteq Q, \delta, F)$

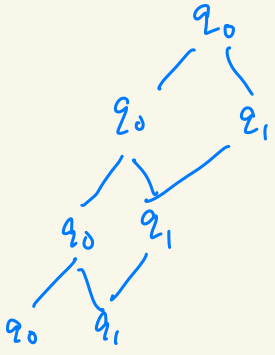
$\delta : Q \times \Sigma \to 2^Q$

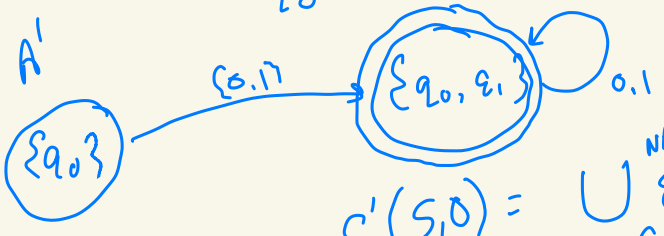


$L(A) = \{ w \in \{0,1\}^* \mid w \text{ is accepted by } A \}$

$L(A) = \Sigma^* \setminus \{\varepsilon\}$

Q	Σ^*	2^Q
q_0	0	$\{q_0, q_1\}$
q_0	00	$\{q_0, q_1\}$
q_0	01	$\{q_0, q_1\}$
q_0	011	





$\delta'(S, 0) = \bigcup_{q \in S} \delta^{NFA}(q, 0)$

New graph - DFA on the power set of the earlier states.

$L(A') = L(A)$ → we need to argue for this.

To show $L(A) = L(A')$

(i) $L(A') \subseteq L(A)$] Use induction

(ii) $L(A) \subseteq L(A')$

A: NFA

A': DFA

We will show

for every $n \geq 0$, for every $w \in \Sigma^*$ s.t $|w| = n$

NFA can reach state $q \in Q$ on reading w iff

DFA A' reaches state $S \subseteq Q$, s.t $q \in S$ (on reading w)

Induction on n

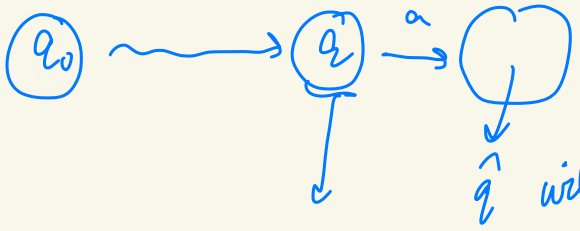
Base: $n = 0$ from definition of initial state of A'

Hypothesis: Claim holds for $0 \leq n < k$

Induction step: To show that claim holds for $n = k$

$|w| = k$ write $w = \underbrace{w'}_{\in \Sigma^*} \underbrace{a}_{\in \Sigma}$ $|w'| = k-1$

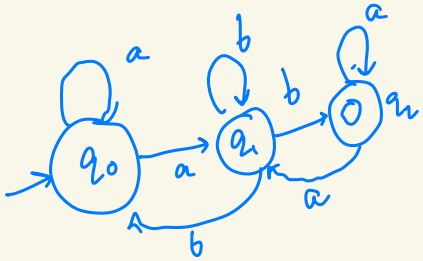
NFA $q_0 \rightsquigarrow q \rightarrow \hat{q}$ iff

DFA 

many states one of them is q

we will take paths from all of them and take union.

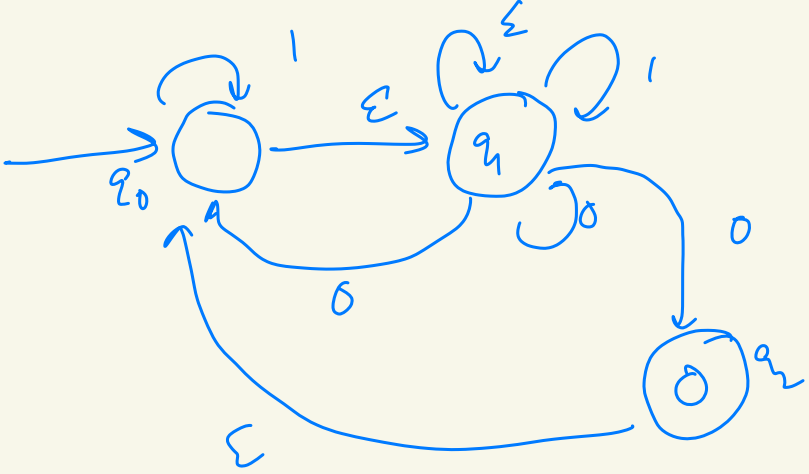
$q \rightarrow \hat{q}$ is there must be a part of the next subset.



ababab

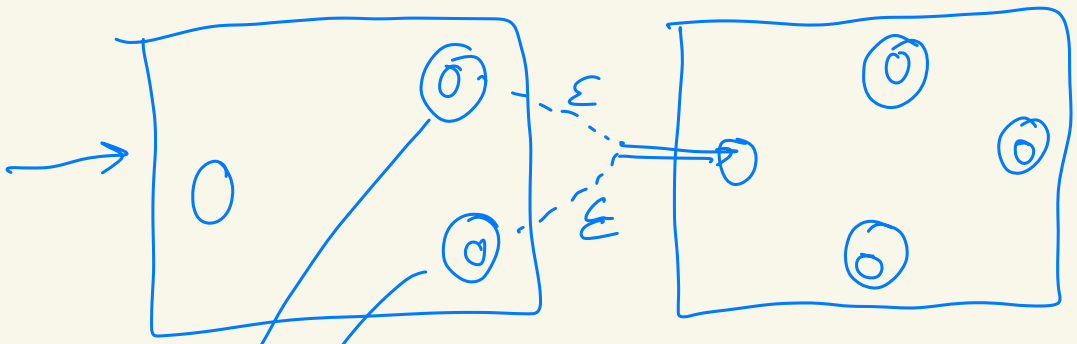
$\{q_0\} \xrightarrow{a} \{q_0, q_1\} \xrightarrow{b} \{q_0, q_1, q_2\} \xrightarrow{a} \{q_0, q_1, q_2\} \xrightarrow{b} \{q_0, q_1, q_2\} \xleftarrow{a} \{q_0, q_1, q_2\} \xleftarrow{b} \{q_0, q_1, q_2\}$

$O(nk)$ — n: size of the automaton, k: length



$10 \notin L$ without ε-edges

$10 \in L$ with ε-edges



ε-transitions can happen both in DFAs and NFAs

If we join DFAs with ε then we will get NFAs

$L_1 \cdot L_2$ using these ε strings

If we do so then these won't remain final.

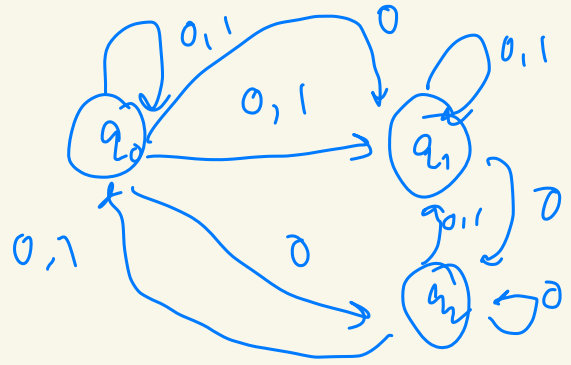
ε-closure

$q_0 \rightsquigarrow \{q_0, q_1\}$ = ε-closure (q_0)

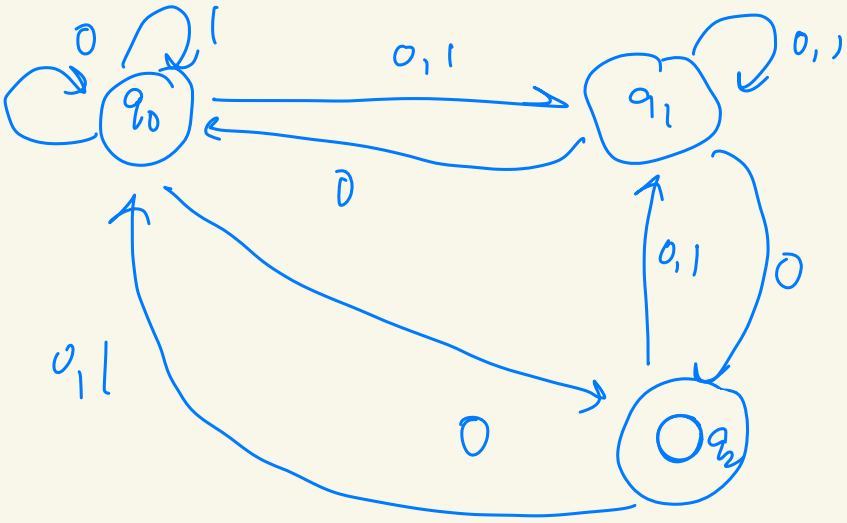
$q_1 \rightsquigarrow \{q_1\}$ = ε-closure (q_1)

$q_2 \rightsquigarrow \{q_0, q_1, q_2\}$ = ε-closure (q_2)

final states = ε closure of final states



Start state = ε-closure of q_0 (or all start states)



We converted ε to normal NFA by adding edges.

By using ε-closure add-direct edges

The final states will also change

All states have to be accepting which have the current final states in their epsilon closure.

NFA with $\varepsilon \equiv$ NFA without $\varepsilon \equiv$ DFA

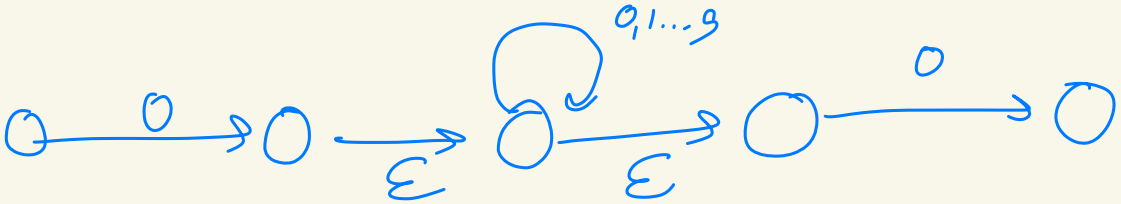
$$\Sigma = \{0, 1, 2, \cdots, 9\}$$

$$L = \{ w \in \Sigma^* \mid w = uvw,\ u = w \in \Sigma^*,\ v \in \Sigma^*,\ |w| \leq 2 \}$$

0000

$u = 0$ $v = 00$ $w = 0$

$u = 00$ $v = \varepsilon$ $w = 00$



Regular expression matching

NFA with ε ≡ NFA without ε ≡ DFA

Regular expressions

Syntax: $\Sigma = \{a, b\}$

a
b
e_1 e_2 →

- $e_1 + e_2$
- $e_1 . e_2$ (Concatenation)
- e_1^*
- (e_1)
- x

→ Also regular expressions

Semantics

$[\![a]\!] = \{a\}$ $\quad [\![b]\!] = \{b\}$

($[\![\cdot]\!]$: meaning)

$[\![e_1 + e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$

$[\![e_1 . e_2]\!] = [\![e_1]\!] \cdot [\![e_2]\!]$

$= \{ u.v \mid u \in [\![e_1]\!], v \in [\![e_2]\!] \}$

$(a.b) + a = \{ab, a\}$

$[\![\varepsilon]\!] = \{\varepsilon\}$

$[\![e_1^*]\!] = \bigcup_{n \geq 0} [\![e_1^n]\!]$

$e_1^n = \underbrace{e_1 . e_1 \cdots e_1}_{n \text{ times}}$

$e_1^0 = \varepsilon$

$e_1 = a + b$ $\quad [\![e_1]\!] = \{a, b\}$

$[\![e_1^3]\!] \Rightarrow$ $a.a.a$, $a.b.a$, $b.b.a$

preference: $*$, $\cdot$, $+$

$[\![\underbrace{(a+b)^*}_{e_1}]\!] = \{ \varepsilon^{e_1^0}, a^{e_1^1}, b, aa^{e_1^2}, ab, ba, bb, \dots \}$

$[\![e_1]\!] = [\![a+b]\!] = \{a, b\}$

$[\![a^* + b^*]\!] = \{ u \in \Sigma^* \mid u = a^n \text{ or } u = b^m, n, m > 0 \}$

$[\![a^* . b^*]\!]$ $L(a^* . b^*)$ ε ✓ a ✓ b ✓ ab ✓ ba ✗ abb ✓ aba ✗ aaa ✓

$L[\![(a^* . b^*)^*]\!]$ ε ✓ ↓ it is in $a^* . b^*$ a ✓ b ✓ ab ✓ bq ✓ abb ✓ aaa ✓ aba ✓

it is in $(a^* . b^*)^2$

$ba \in (a^* . b^*) . (a^* . b^*)$ (εb, $a \varepsilon$)

it is in $(a^* b^*)^2$ ($ab\, a\varepsilon$)

Is $L((a^* b^*)^*) = L((a+b)^*)$?

ε easy

$\supseteq$?

ε is also present in both.

$(a^* b^*)^6$

$abab\,ba$

$(a^* b^*)(a^* b^*)(a^* b^*)(a^* b^*)(a^* b^*)(a^* b^*)$

$a\varepsilon$ εb $a\varepsilon$ εb εb $a\varepsilon$

CS208 Practice Problem Set 2

1. Let $\Sigma = \{0, 1\}$.

   (a) Construct DFAs for each of the following languages. Try to use as few states as you can in your construction. In the following $n_i(w)$ denotes the count of i's in the string w, for $i \in \Sigma$. Similarly, $|w|$ denotes the length of the string w.

      i. $L_1 = \{0^m 1^n \mid m + n = 0 \mod 2 \text{ and } m = 0 \mod 3\}$. Think of L_1 as the set of all even length strings of 0s followed by 1s, where the count of 0s is a multiple of 3.

      ii. $L_2 = \{0^m 1^n 0^k \mid m, n, k > 0 \text{ and } m + n + k = 0 \mod 2\}$. Think of L_2 as the set of all strings obtained by inserting a block of 1s inside a block of 0s such that the length of the overall string becomes even.

      iii. $L_3 = \{w \in \Sigma^* \mid w = u \cdot v, \text{ where } u, v \in \Sigma^* \text{ and } n_0(u) = 0 \mod 2, n_1(v) = 0 \mod 2\}$. Think of L_3 as the set of all strings that can be cut into two parts and given to two applications, one of which expects an even count of 0s while the other expects an even count of 1s.

      iv. $L_4 = \{w \in \Sigma^* \mid |w| + 2.n_1(w) = 0 \mod 3\}$. Think of L_4 as the set of all strings whose length would be a multiple of 3 if we simply repeated each 1 in the string thrice (without caring about the 0s).

   (b) Construct NFAs for each of the following languages. Feel free to use ϵ-transitions. The purpose of constructing NFAs is really to capture the intuitive structure of strings in the language in as direct a way as possible. Feel free to re-use DFAs constructed in the previous part of this question as building blocks of your NFAs.

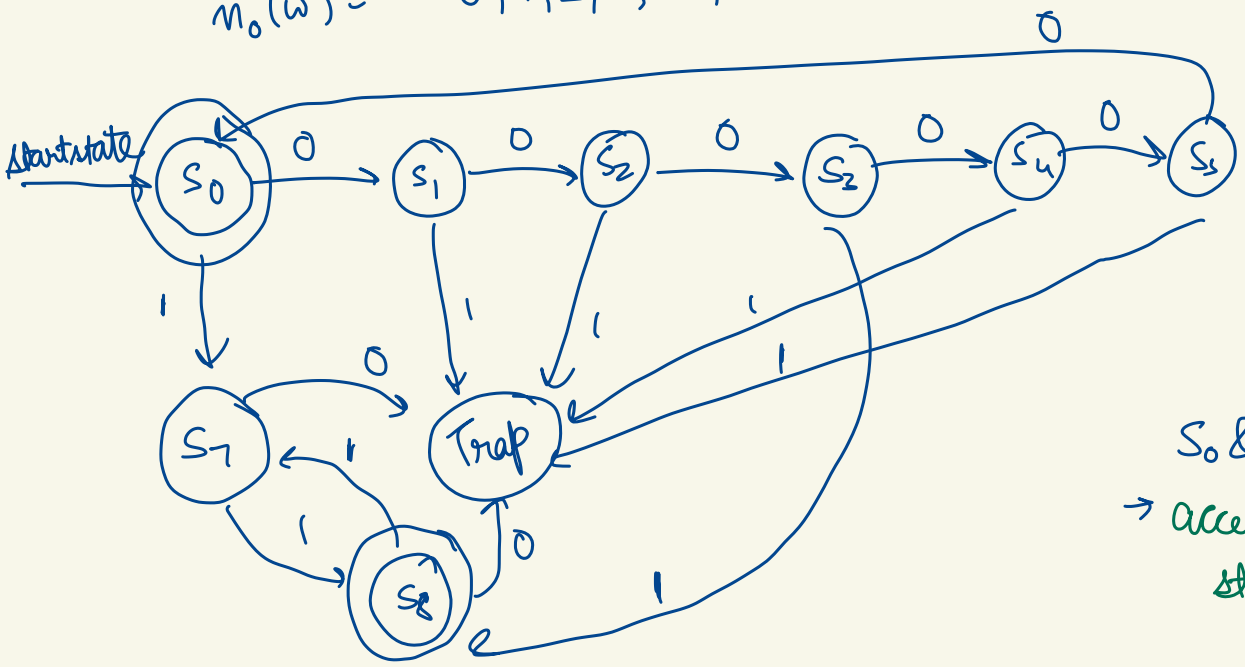
      i. $L_5 = \{w \in \Sigma^* \mid w = u \cdot v \cdot x, \text{ where } u, v, w \in \Sigma^* \text{ and } |u| + 2.|v| + 3.|x| = 0 \mod 4\}$. You can think of a string being

i. $L_1 = \{0^m 1^n \mid m + n = 0 \mod 2 \text{ and } m = 0 \mod 3\}$. Think of L_1 as the set of all even length strings of 0s followed by 1s, where the count of 0s is a multiple of 3.

$0^m 1^n$ $\quad (m+n) = 0 \mod 2$

$m = 0 \mod 3$

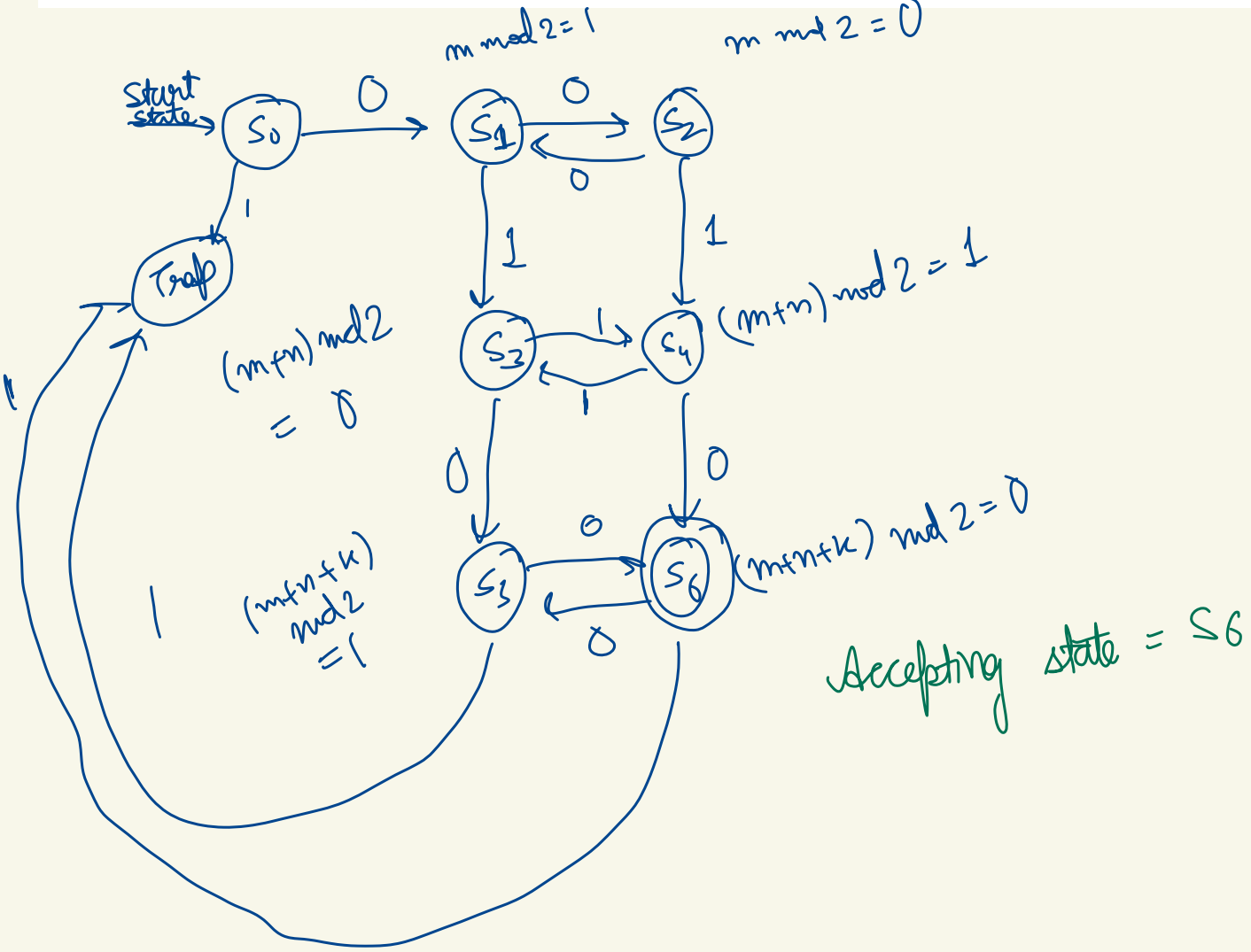
$n_0(w) =$ $\quad 0, 1, 2, 3, 4, 5 \quad \mod 6$



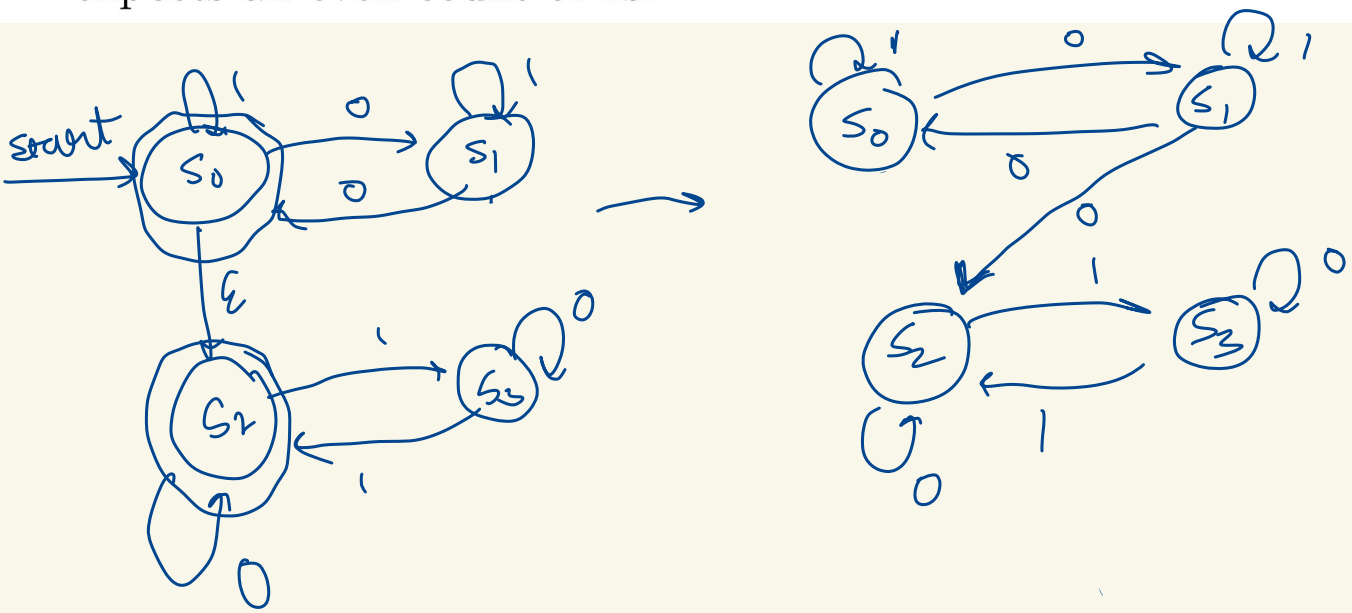
S_0 & S_8

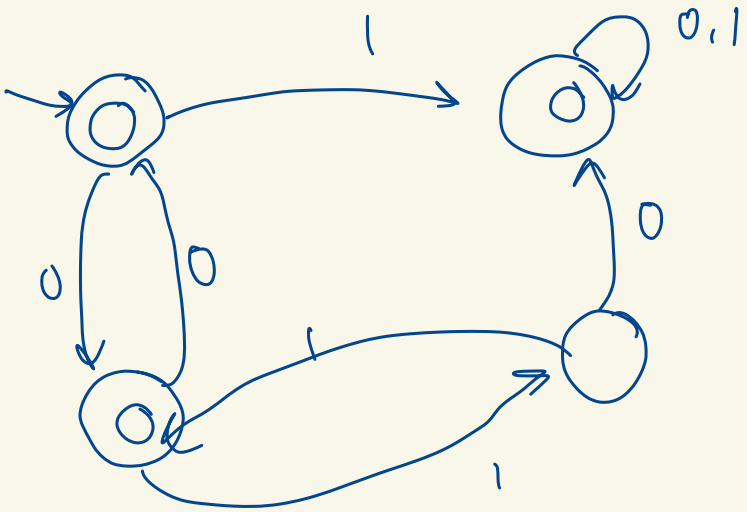
→ accepting states

ii. $L_2 = \{0^m 1^n 0^k \mid m, n, k > 0 \text{ and } m + n + k = 0 \mod 2\}$. Think of L_2 as the set of all strings obtained by inserting a block of 1s inside a block of 0s such that the length of the overall string becomes even.



iii. $L_3 = \{w \in \Sigma^* \mid w = u \cdot v,$ where $u, v \in \Sigma^*$ and $n_0(u) = 0 \mod 2, n_1(v) = 0 \mod 2\}$. Think of L_3 as the set of all strings that can be cut into two parts and given to two applications, one of which expects an even count of 0s while the other expects an even count of 1s.





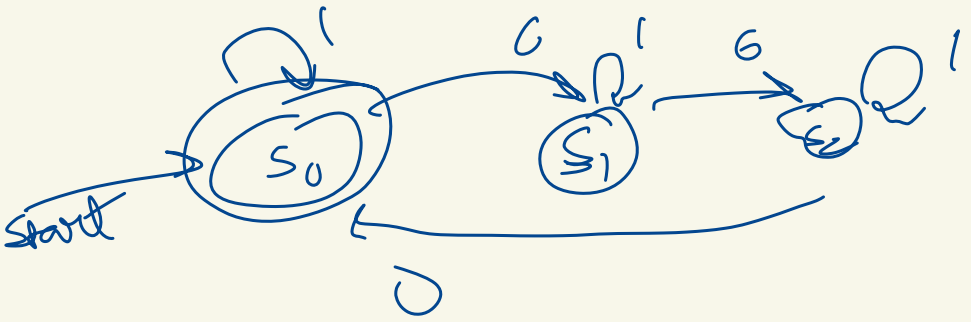
If the number of 1s is 0 mod 2 in the whole string, it is accepted. For odd i.e. 1s are 1 mod 2, prefix with even 0s and odd 1s is necessary

iv. $L_4 = \{w \in \Sigma^* \mid |w| + 2.n_1(w) = 0 \mod 3\}$. Think of L_4 as the set of all strings whose length would be a multiple of 3 if we simply repeated each 1 in the string thrice (without caring about the 0s).

$|w| + 2n_1(w) = 0 \mod 3$

$n_0(w) + n_1(w) + 2n_1(w) = 0 \mod 3$

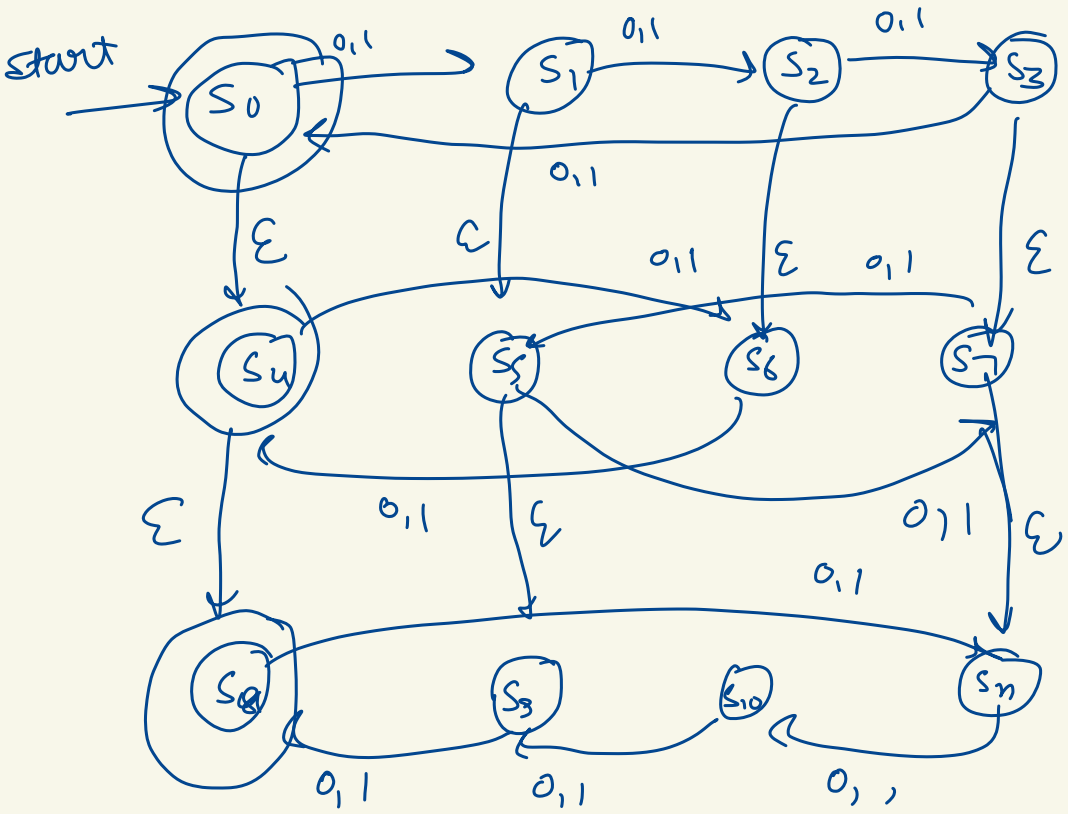
$n_0(w) + \underbrace{3n_1(w)}_{0 \mod 3} = 0 \mod 3 \Rightarrow n_0(w) = 0 \mod 3$



Accepting state = S_0

i. $L_5 = \{w \in \Sigma^* \mid w = u \cdot v \cdot x,$ where $u, v, w \in \Sigma^*$ and $|u| + 2.|v| + 3.|x| = 0 \mod 4\}$. You can think of a string being

$w = u \cdot v \cdot x$



Accepting states = s_0, s_4, s_8

broken into three parts and fed to three applications, such that the first (resep. second and third) application charges Re. 1 (resp. Rs. 2 and Rs. 3) to process each letter in the string. Then L_5 is the set of strings that can be processed by spending a muliple of 4 Rupees.

ii. $L_6 = \{w \in \Sigma^* \mid w = u \cdot v,$ and either $u \in L_1, v \in L_2$ or $u \in L_2, v \in L_1\}$. Thus, L_6 is the set of all strings that can be broken into two parts, such that the first part belongs to L_1 and the second part belongs to L_2, or vice versa.

iii. $L_7 = \{w \in \Sigma^* \mid u_1 \cdot u_2 \cdot \ldots \cdot u_k,$ where $k > 0, k = 0 \mod 2,$ and $u_i \in L_2$ for all $i\}$. Thus, L_7 is the language of all strings that can be broken up into an even number of strings from L_2.

iv. $L_8 = \{w \in \Sigma^* \mid n_1(w') = 0 \mod 2,$ where w' is obtained from w by replacing every alternate letter starting from the second letter by $\varepsilon\}$. Thus if $w = 0101001$, then $w' = 0001$. You can think of the string w as a sequence of bits coming in so fast that a machine can only read every alternate bit. L_8 is then the sequence of strings that can be accepted by such a lossy automaton for L_2.

2. Suppose r_1 and r_2 are regular expressions over the same alphabet Σ. We say $r_1 = r_2$ to denote equality of the languages represented by r_1 and r_2. In other words, every string in the language represented by r_1 is also included in the language represented by r_2 and vice versa. For each of the following pairs of regular expressions over $\Sigma = \{0, 1\}$, either prove that they represent the same language, or give a string that is present in the language of one but not in the language of the other. In the latter case, you must also describe why your solution string is in the language of one regular expression, but not in that of the other.

(a) $r_1 = \mathbf{1}^*(\mathbf{1} + \mathbf{0})^*\mathbf{0}^*$ and $r_2 = (\mathbf{0}^*\mathbf{1}^*)^*$

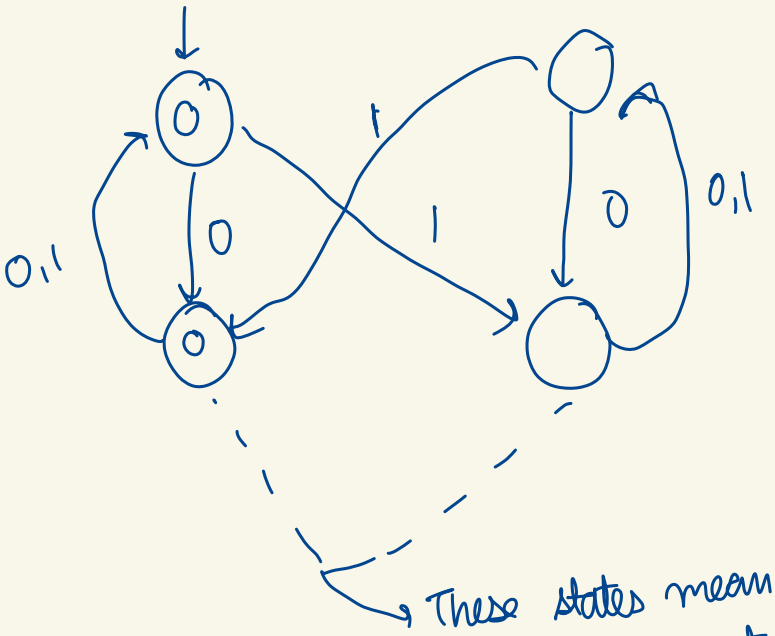
(b) $r_1 = ((\mathbf{0} + \mathbf{1})^*\mathbf{0})^*\mathbf{0}$ and $r_2 = (\mathbf{0} + \mathbf{1})^*\mathbf{0}^*\mathbf{0}$

(c) $r_1 = (\mathbf{0} + \mathbf{1})^*\mathbf{01}(\mathbf{0} + \mathbf{1})^*$ and $r_2 = \mathbf{1}^*(\mathbf{0} + \mathbf{1})^*\mathbf{0}(\mathbf{0} + \mathbf{1})^*\mathbf{1}$

3. Give as small an upper bound as you can of the number of *distinct* languages over $(\mathbf{0} + \mathbf{1})^*$ that can be recognized by DFAs using at most n states. Your answer must be a function of n. You must also clearly explain your reasoning.

(iv)

iv. $L_8 = \{w \in \Sigma^* \mid n_1(w') = 0 \mod 2$, where w' is obtained from w by replacing every alternate letter starting from the second letter by $\varepsilon\}$. Thus if $w = 0101001$, then $w' = 0001$. You can think of the string w as a sequence of bits coming in so fast that a machine can only read every alternate bit. L_8 is then the sequence of strings that can be accepted by such a lossy automaton for L_2.



These states mean that the next character has to be avoided

3. Give as small an upper bound as you can of the number of *distinct* languages over $(\mathbf{0}+\mathbf{1})^*$ that can be recognized by DFAs using at most n states. Your answer must be a function of n. You must also clearly explain your reasoning.

From every state 0 and 1 will depart

n options, n options

n^2 → for every state

Total = $\underbrace{n^2 \times n^2 \times n^2 \times \ldots \times n^2}_{\text{Total } n \text{ states}} = n^{2n}$

Now, we will have to specify start and accepting states:

W.L.O.G → S_1 = start state

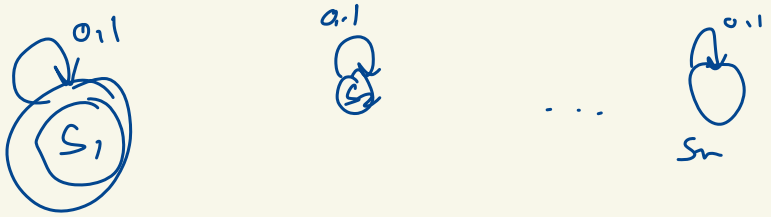
Different sets of accepting states

Excluding S_1 → number of accepting states = 1 to n-1

Including S_1 → number of accepting states = 1 ... n-1

$2(n-1)\, n^{2n}$ = DFAs (valid upper bound)

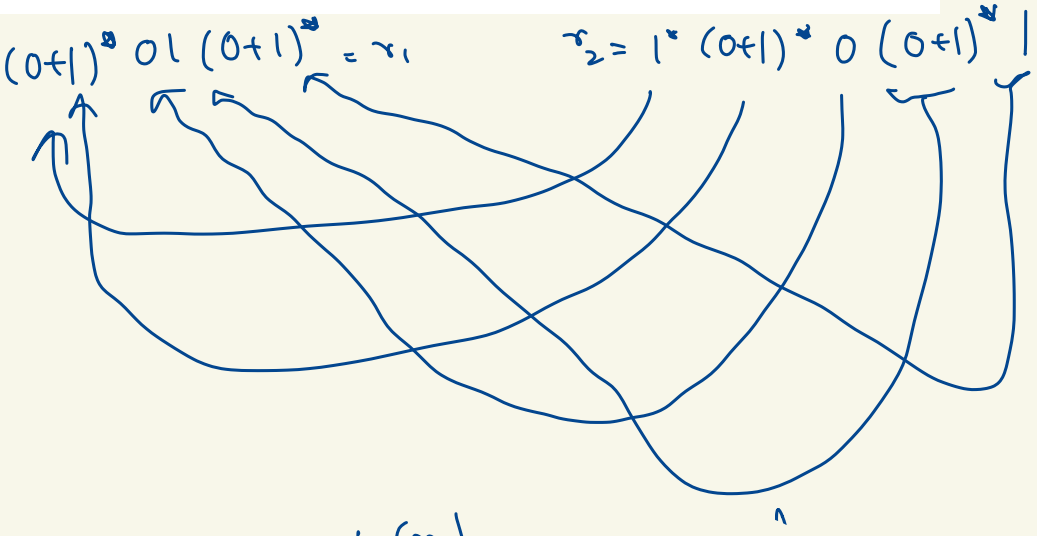
If $(0+1)^*$ = Language



next consider x accepting states and prove

(c) $r_1 = (\mathbf{0}+\mathbf{1})^*\mathbf{01}(\mathbf{0}+\mathbf{1})^*$ and $r_2 = \mathbf{1}^*(\mathbf{0}+\mathbf{1})^*\mathbf{0}(\mathbf{0}+\mathbf{1})^*\mathbf{1}$

(c)



$010 \in \underbrace{L(r_1)}$

$\hookrightarrow \varepsilon 010$

$010 \notin L(r_2)$ as $L(r_2)$ ends with 1

(a) $r_1 = \mathbf{1}^*(\mathbf{1}+\mathbf{0})^*\mathbf{0}^*$ and $r_2 = (\mathbf{0}^*\mathbf{1}^*)^*$

$1^* \; (1+0)^* \; 0^*$

$(1+0)^* \; 0^*$

$(1+0)^*$

Hence these are equivalent

$r_2 = (0^* 1^*)^*$

$= (1+0)^*$

(proved in class)

represents all binary strings of finite length

(b) $r_1 = ((\mathbf{0}+\mathbf{1})^*\mathbf{0})^*\mathbf{0}$ and $r_2 = (\mathbf{0}+\mathbf{1})^*\mathbf{0}^*\mathbf{0}$

$((0+1)^* 0)^* \; 0$

repeating strings also have 0 at the end

0 at the end

$l_1 0 l_2 0 l_3 0 \ldots l_n 0 0$

$(0+1)^* 0^* 0$

$l_1 \; l_2 \; 0$

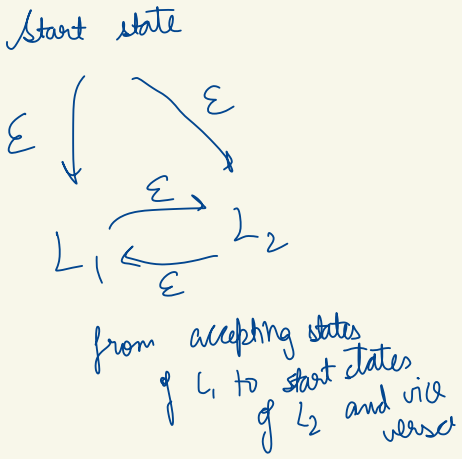
$=$

Consider $1\,0$

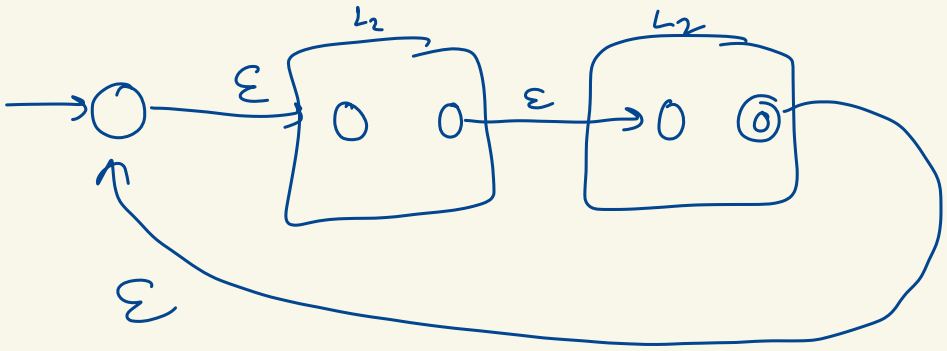
$10 \notin r_1$ but $10 \in r_2$

spending a multiple of 4 Rupees.

ii. $L_6 = \{w \in \Sigma^* \mid w = u \cdot v,$ and either $u \in L_1, v \in L_2$ or $u \in L_2, v \in L_1\}$. Thus, L_6 is the set of all strings that can be broken into two parts, such that the first part belongs to L_1 and the second part belongs to L_2, or vice versa.



iii. $L_7 = \{w \in \Sigma^* \mid u_1 \cdot u_2 \cdot \ldots \cdot u_k,$ where $k > 0, k = 0 \mod 2,$ and $u_i \in L_2$ for all $i\}$. Thus, L_7 is the language of all strings that can be broken up into an even number of strings from L_2.



4. The solution to this problem is specific to your roll number. Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, B\}$. We will say a string $w \in \Sigma^*$ is *embedded* in a string $u \in \Sigma^*$ iff w can be obtained from u by replacing some letters in u with ε. Thus, 014 is embedded in 203421049. To see why this is so, notice that 014 can be obtained as $\varepsilon 0 \varepsilon\varepsilon\varepsilon 1 \varepsilon 4 \varepsilon$. However, 014 is not embedded in 23421049.

22B1003

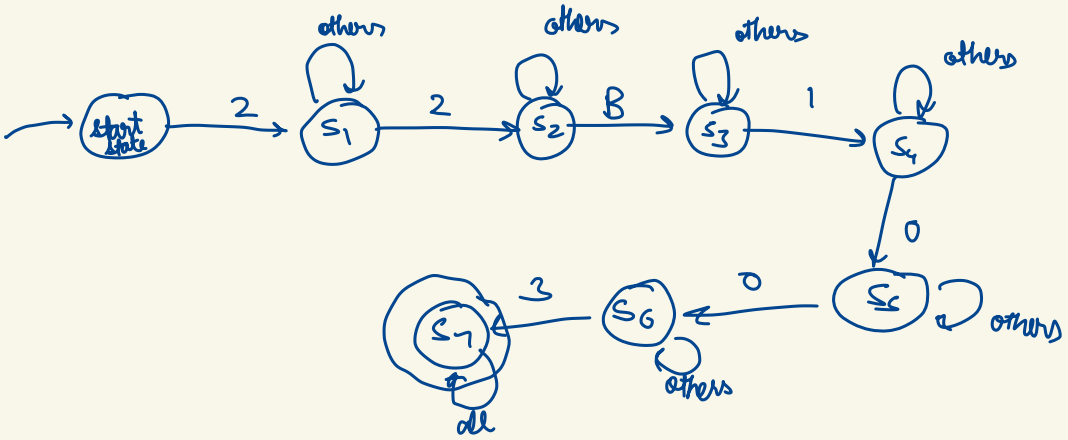
(a) State your roll no. as a string in Σ^*. Let us call this string ρ.

(b) Give a DFA with no more than $|\rho| + 1$ states that accepts the language $L_\rho = \{w \mid w \in \Sigma^*, \rho \text{ is embedded in } w\}$. You must explain what each state in your DFA represents (use a couple of lines of explanation per state), as evidence that you understood the construction of the DFA. Your answer will fetch 0 marks without proper explanation per state.

Note: You can of course start with a NFA, and then use the subset construction to convert it to a DFA. But this is a long and tedious route, and will not give you much intuition to understand what each state in the resulting DFA represents. So you are strongly advised not to follow this route. There is enough structure in the problem to permit a much simpler construction of a DFA, and you are encouraged to think about it.

Ans (b)

(b) Give a DFA with no more than $|\rho|+1$ states that accepts the language $L_\rho = \{w \mid w \in \Sigma^*, \rho \text{ is embedded in } w\}$. You must explain what each state in your DFA represents (use a couple of lines of explanation per state), as evidence that you understood the construction of the DFA. Your answer will fetch 0 marks without proper explanation per state.



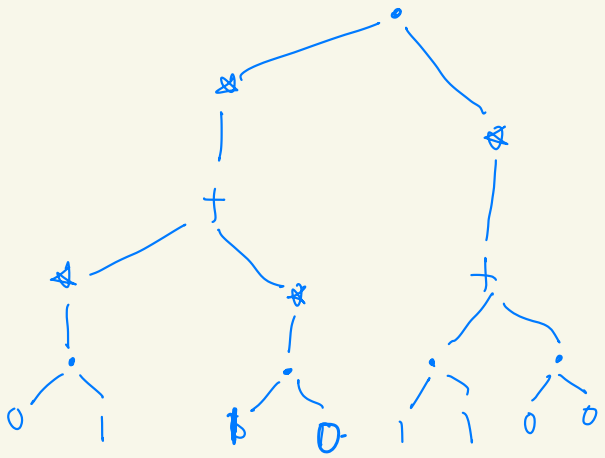
If a string occurs before only, no need to focus on the later part.

* : Kleene closure
Kleene star

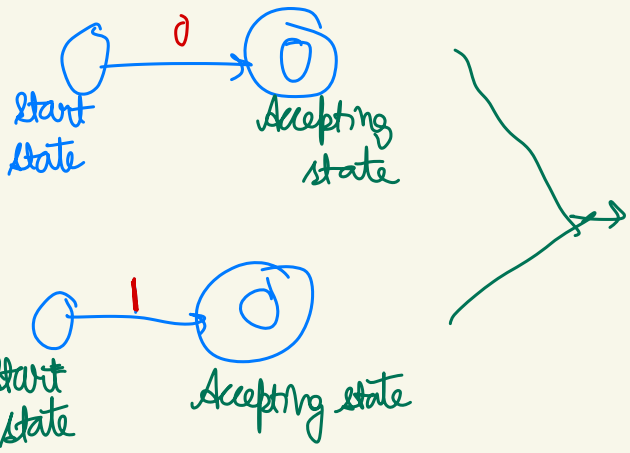
$L(\text{reg exp}) \subseteq L(\text{NFA with } \epsilon) = L(\text{NFA w/o } \epsilon) = L(\text{DFA})$

$\Sigma = \{0, 1\}$

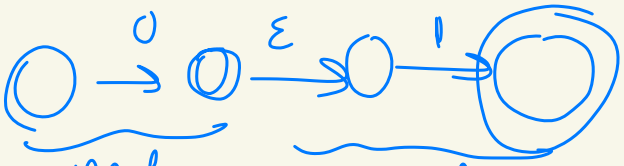
$((01)^* + (10)^*)^* \cdot (11+00)^*$



NFA for 0



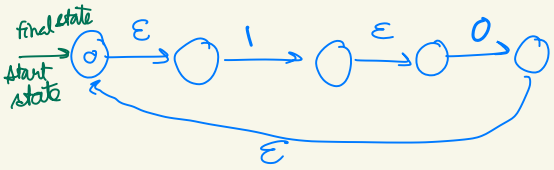
NFA for 1



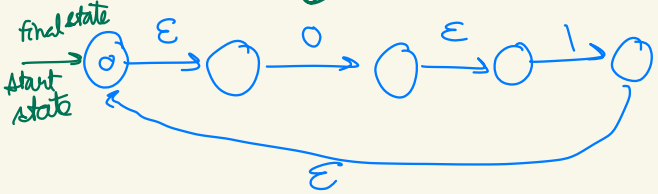
NFA for 0

NFA for 1

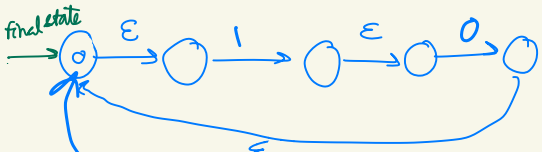
NFA for 0.1

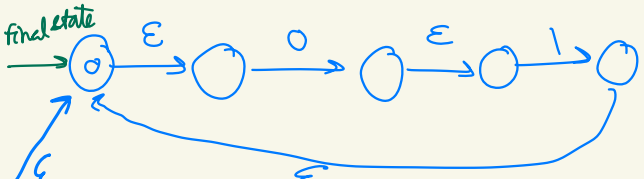


NFA for $(1.0)^*$



NFA for $(0.1)^*$

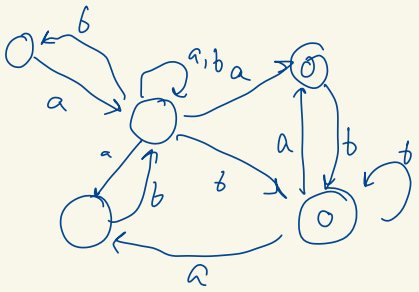


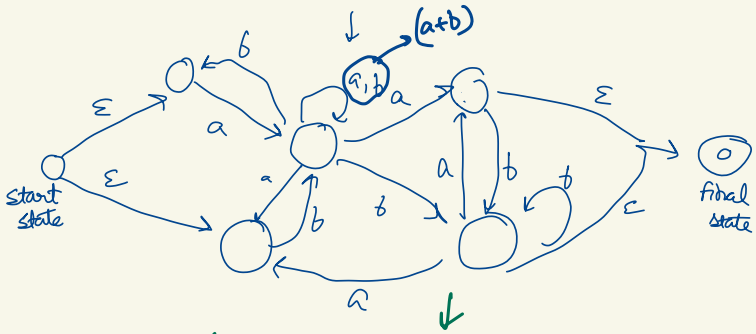


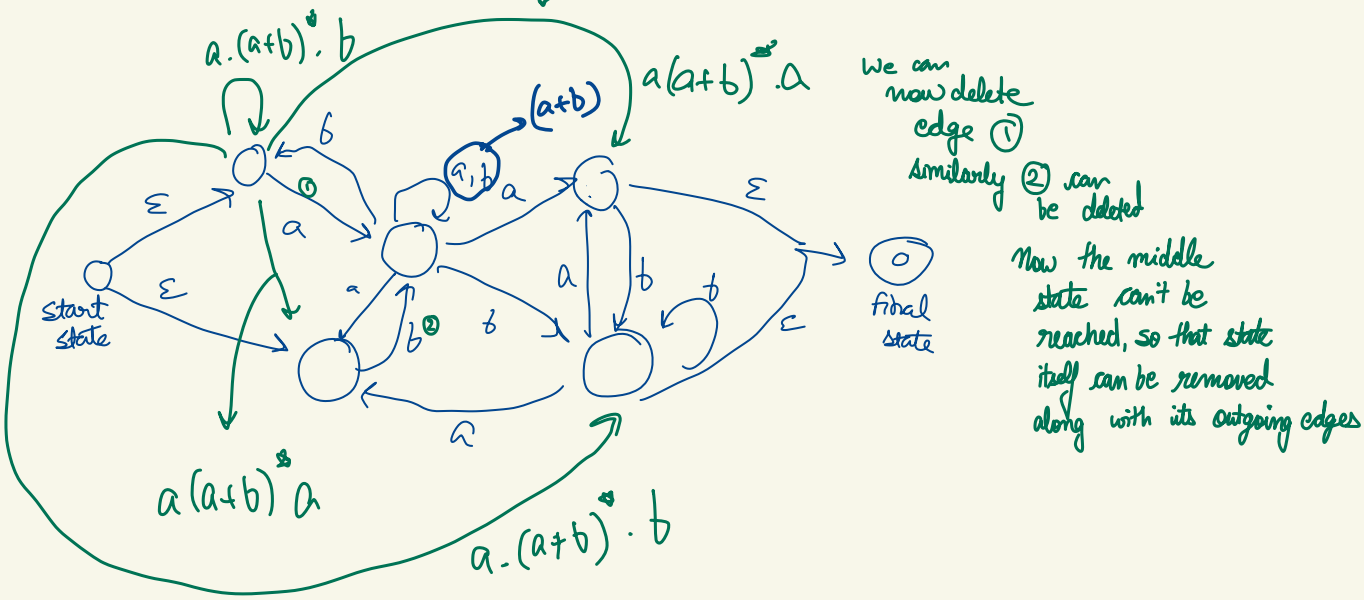
Hence we have shown that

$L(\text{reg exp}) \subseteq L(\text{NFA with } \epsilon)$

NFA for $(0.1)^* + (1.0)^*$





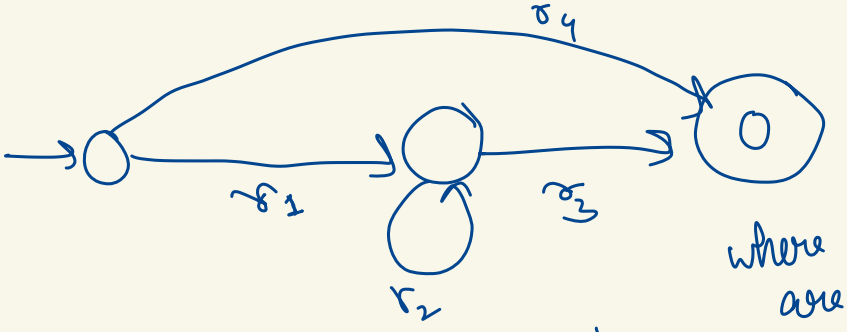


we can now delete edge ①
similarly ② can be deleted

Now the middle state can't be reached, so that state itself can be removed along with its outgoing edges

One by one, we can remove all of the states

Pre-final state :



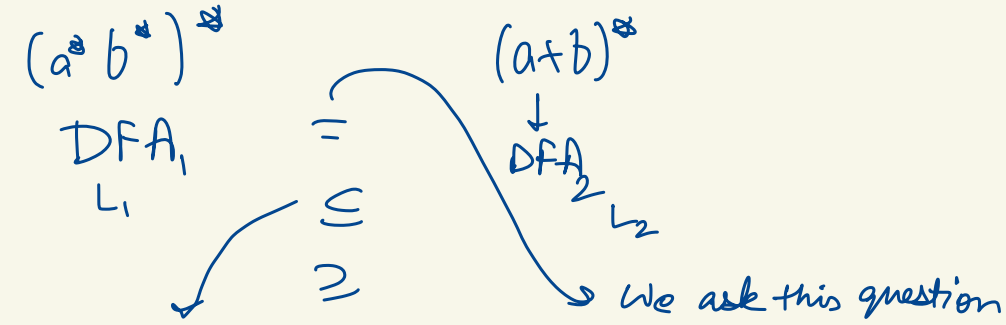
where r_1, r_2, r_3, r_4 are regular expressions

Each step is language preserving

$$\left(\left(r_1 r_2^* r_3 \right) + r_4 \right)$$

Reg Exp ≡ NFA with ε ≡ NFA w/o ε ≡ DFA : Kleene's Theorem

Regular Languages



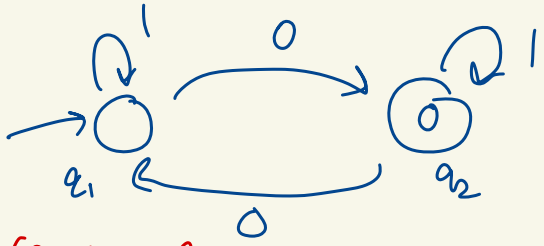
$L_1 \cap (\Sigma^* \setminus L_2) = \phi \equiv L_1 \subseteq L_2$

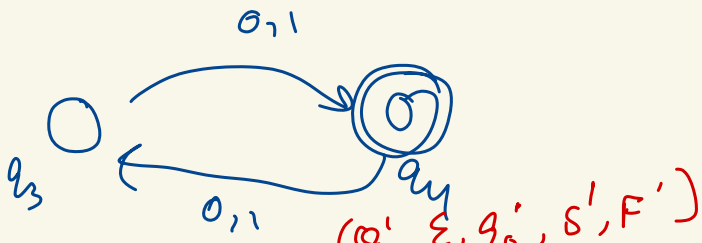
L_2 is reg $\Leftrightarrow$ L_2^* or $\overline{L_2}$ is reg

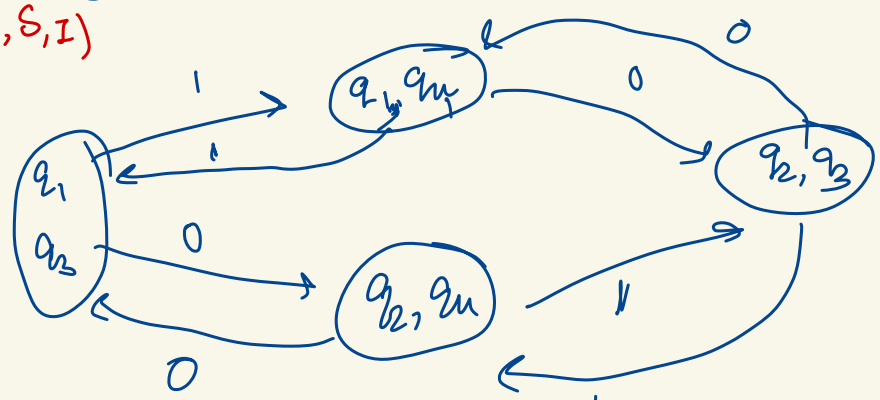
(See from DFA, final → not final, not final → final)

Check if a path from start to final

Now we need to take intersection







Intersection DFA

Two automata running in parallel

Starting state = starting state of both product

Number of states = cartesian product of earlier states

One construction can serve the purpose of all boolean operations of the earlier DFAs.

$\hat{\delta}((q_i, q_j'), a) = (\delta(q_i, a), \delta'(q_j', a))$ Product construction for Automata

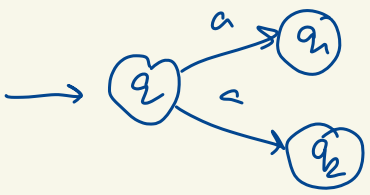
$\downarrow \in \Sigma$

$\hat{\delta} : (Q \times Q') \times \Sigma \to (Q \times Q')$

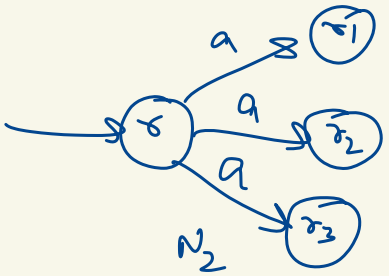
$L_1 = L_2 \iff L_1 \subseteq L_2$ and $L_2 \subseteq L_1$

$L_1 \cap (\Sigma^* \setminus L_2) = \phi$ $\quad$ $L_2 \cap (\Sigma^* \setminus L_1) = \phi$

N_1 and N_2 are NFAs



N_1



N_2

$\hat{\delta}((q,a),a) = \{q_1, q_2\} \times \{r_1, r_2, r_3\}$

$= \delta(q,a) \times \delta'(r,a)$

DFA complement = flip the states of accepting and non-accepting states

For NFA, convert to DFA and then take complement

Closure properties of Regular Languages:

L_1, L_2 = Reg. lang. over $\Sigma_1 = \{a, b\}$

$\overline{L_1}$ $\quad$ $L_1 \cup L_2$, $L_1 \cap L_2$

Reg. Langnage $L_1 \subseteq \Sigma_1^*$ $\quad$ $a^* b^*$

$\Sigma_2 = \{0, 1, 2\}$

Reg. Lang. $L_a \in \Sigma_2^*$, $L_b \in \Sigma_2^*$

$L_a = 0^*(1.2)^* 1^*$

$L_b = 1^*(0+2)^*$

substitute $(L_1, L_a, L_b) = \{ w \in \Sigma_2^* \mid \exists\ v \in L_1$ st $w \in L_{\alpha_1} L_{\alpha_2} \cdots L_{\alpha_k} \}$

$L_1 = a^* b^*$ $\quad L_a = 0^* (1+2)^* 1^* \quad L_b = 1^* (0+2)^*$

Eg: $aabbb \in L_1$

$= u$

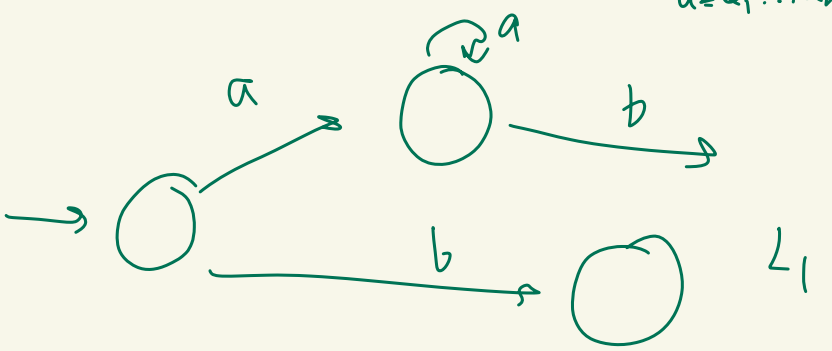
$a \; a \; b \; b \; b$

$\alpha_1 \; \alpha_2 \; \alpha_3 \; \alpha_4 \; \alpha_5$

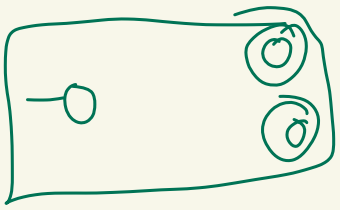
Replace a by L_a

$w \in L_a \cdot L_a \cdot L_b \cdot L_b L_b$?

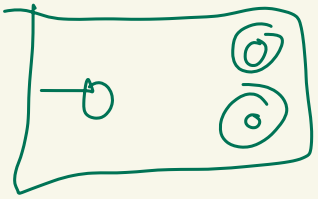
subst $(L_1, L_a, L_b) = \bigcup_{u = \alpha_1 \ldots \alpha_k} L_{\alpha_1} \ldots L_{\alpha_k}$

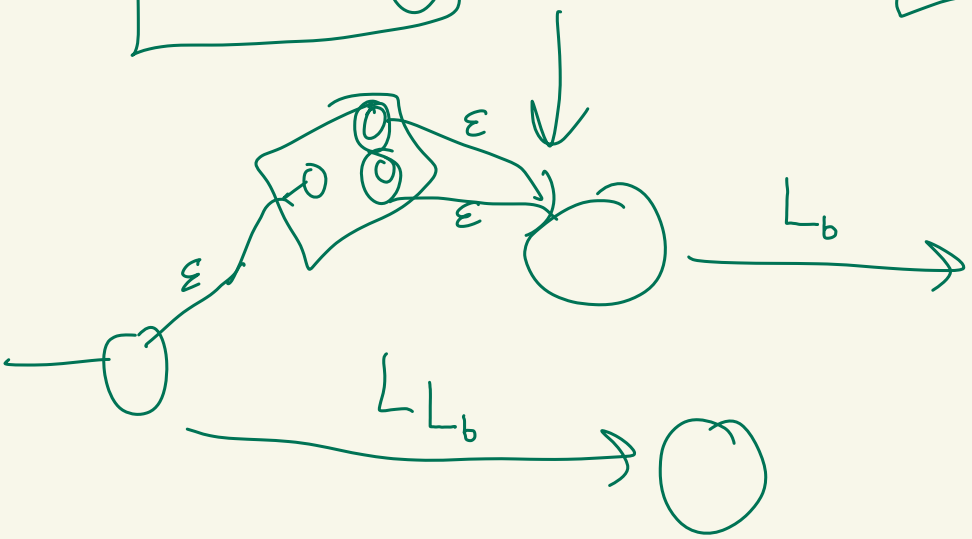


L_a



L_b





$L_a \cdot L_b$

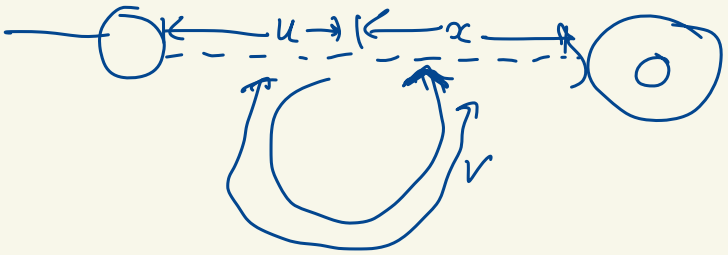
$\Sigma_1 = \{a, b\}$ $L_1 = \{a \cdot b\}$

L_a even number of times
L_b odd number of times

$L' = \{a^n b^m / n = 0 \bmod 2, m = 1 \bmod 2\}$

Use substitution.

Consider $|A| = 20$ states, $A = DFA$, $w \in L(A)$, $|w| = 100$ → Loop must exist



$w = u.v.x$

$u.x \in L$

$u.v.v.x \in L$

$u.v^i x \in L$ for $u \; i \geq 0$

If a DFA of n states accepts a string of length $n+1$, then the language space of that DFA = infinite.

Check between length n and $2n$

Shortest string with this behaviour = $2n$

Else no.

CS208 Tutorial 3: Finite Automata Theory

---

1. Often, we are required to find ways to accept sets of strings with various properties. We've seen in class that DFAs and NFAs are ways to achieve this, with an equivalence between NFAs and DFAs. In this question and the next, we'll specify some properties of sets of strings and you will be required to construct (small) DFAs/NFAs to accept these sets (or languages). We aren't insisting that you should find the smallest (in terms of number of states) automata, but try to use as few states as you can.

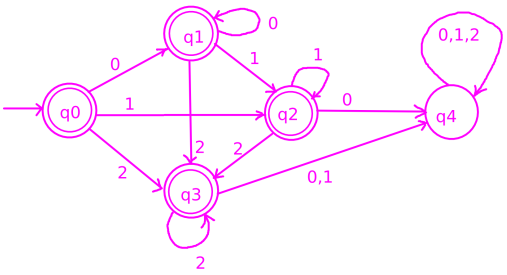
   Let $\Sigma = \{0, 1, 2\}$. Construct DFAs for recognizing each of the following languages.

   1. $L_1 = \{w \in \Sigma^* \mid w$ doesn't have any pair of consecutive decreasing letters (numbers)$\}$. For example $010 \notin L$ but $00012 \in L$ and $222 \in L$.
   2. $L_2 = \{w \in \Sigma^* \mid w = u.v,\ u \in \Sigma^+,\ v \in \Sigma^*,\ \ n_0(u) > n_1(u) + 2$ or $n_1(u) > n_0(u) + 2\}$, where $n_i(u)$ denotes the count of i's in u, for $i \in \{0, 1\}$. For example, $0010221020012 \in L$ but $012012012 \notin L$.
   3. $L_3 = \{0^n \mid n > 0, n^3 + n^2 + n + 1 = 0 \mod 3\}$

Solution:

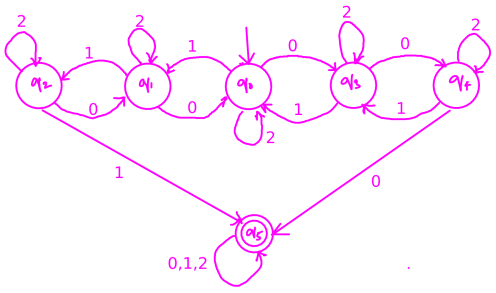
Possible DFAs are given below:

1.



q_1, q_2, q_3 remember that we've not seen any decreasing sequence of letters so far, and the last letter seen was 0, 1, 2, respectively. All of these strings are accepted. q_4 remembers that we have seen a decreasing sequence, and hence anything we see subsequently cannot make us accept the string. So q_4 is a sink or trap state – once you reach there, you can't escape it.

2.



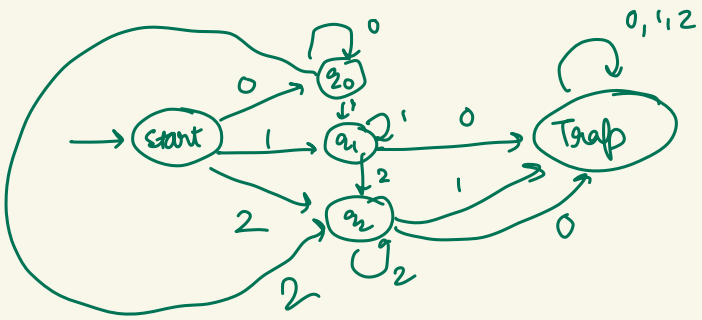
For a prefix u of w seen so far, we need to remember if $n_0(u) - n_1(u)$ is $0, 1, -1, 2, -2$ or greater than 2 or less than -2. Once the prefix u satisfies $n_0(u) - n_1(u) > 2$ or $n_0(u) - n_1(u) < -2$, we can immediately accept the word, regardless of what subsequent letters are seen. States q_0, q_1, q_2, q_3, q_4 remember if $n_0(u) - n_1(u) = 0, -1, -2, 1, 2$ respectively, where u is the (prefix of the) input word seen so far. State q_5 simply remembers whether $|n_0(u) - n_1(u)| > 2$.

3. Notice that $n^3 + n^2 + n + 1 = 0 \mod 3$ is equivalent to $(n^2 + 1) \cdot (n + 1) = 0 \mod 3$. Therefore, $((n^2 + 1) \mod 3) \cdot ((n + 1) \mod 3)$ has to be equal to 0.

   Given that $n \mod 3 \in \{0, 1, 2\}$, it follows that we require $n^2 = 2 \mod 3$ or $n = 2 \mod 3$. However, for no n is $n^2 = 2 \mod 3$. Why so? Because this would require $((n$

1. 1. no consecutive decreasing letters

$\Sigma = \{0,1,2\}$



2. 2. $L_2 = \{w \in \Sigma^* \mid w = u.v,\ u \in \Sigma^+,\ v \in \Sigma^*,\ n_0(u) > n_1(u) + 2 \text{ or } n_1(u) > n_0(u) + 2\}$, where $n_i(u)$ denotes the count of i's in u, for $i \in \{0,1\}$. For example, $0010221020012 \in L$ but $012012012 \notin L$.

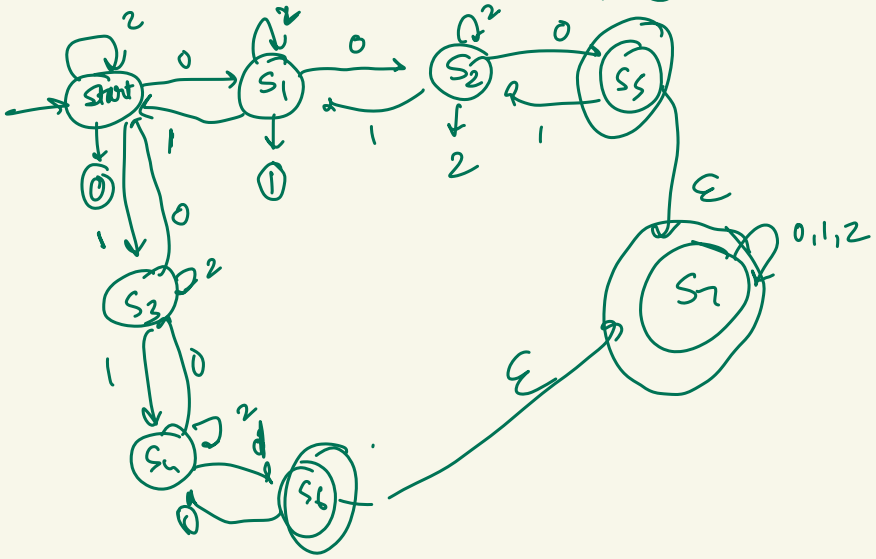
$w = u.v$ $\quad \Sigma = \{0,1,2\}$

$u \in \Sigma^+ \quad v \in \Sigma^*$

$|n_0(u) - n_1(u)| > 2$ } for (w)

after that we transition to V

$n_0(u) - n_1(u) = \ldots -3, -2, -1, 0, 1, 2, 3 \ldots$



we can even avoid S_5, S_6

3b $(n^3 + n^2 + n + 1) \equiv 0 \bmod 3$

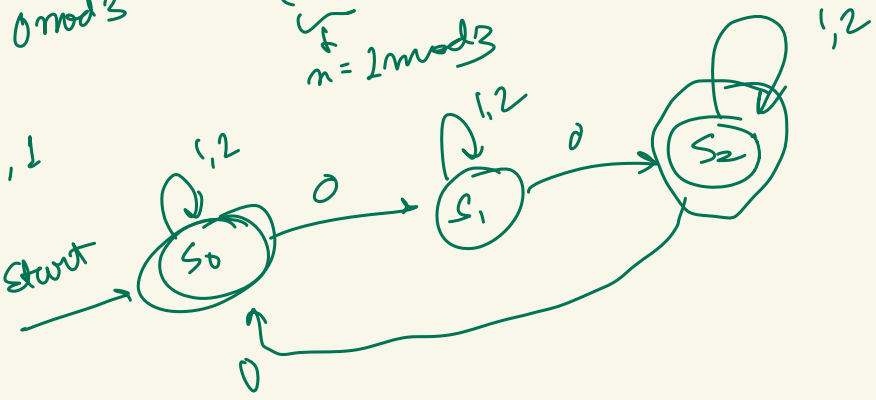
$(n^2+1)(n+1) = 0 \bmod 3$

$(n+1) = 0 \bmod 3$

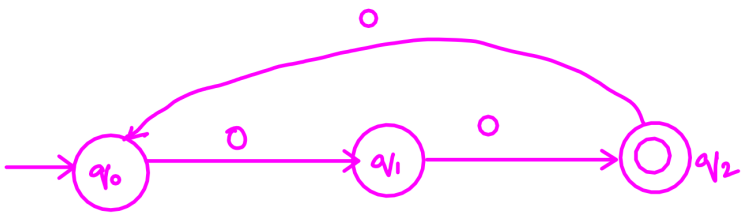
$n = 1 \bmod 3$

$(n^2+1) = 0 \bmod 3$

Possible: 1, 2, 1



$\mod 3) \times (n \mod 3)) = 2 \mod 3$. Since $n \mod 3 \in \{0,1,2\}$, $(n \mod 3) \times (n \mod 3)$ can only be in $\{0,1\} \mod 3$. Therefore, we must have $n = 2 \mod 3$. The DFA is now easily obtained as follows:

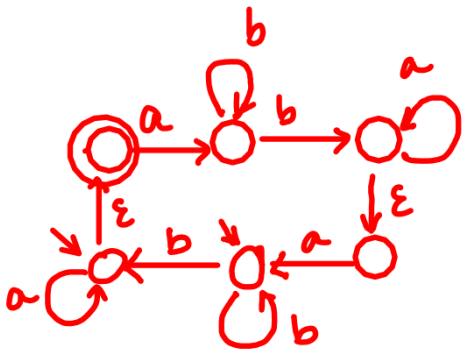


2. Let $\Sigma = \{a, b\}$. Construct NFAs, possibly with ε-transitions for each of the following languages.
   1. $L_4 = \{w \in \Sigma^* \mid n_{ab}(w) \text{ is even}\}$, where $n_{ab}(w)$ denotes the count of times ab appears in w as consecutive letters.
   2. $L_5 = \{w \in \Sigma^* \mid w$ contains the "pattern" abba (as consecutive letters) followed by the "pattern" baba, possibly in an overlapping manner$\}$. For example, $ababababbaabb, bababbabb \notin L$ but $abababbbaba, ababbaabbbaba \in L$.

Now try constructing a DFA that recognizes L_5 using the subset construction and ε-edge removal on the NFA constructed above. Do you see an exponential blow-up in the count of states as you do this conversion?

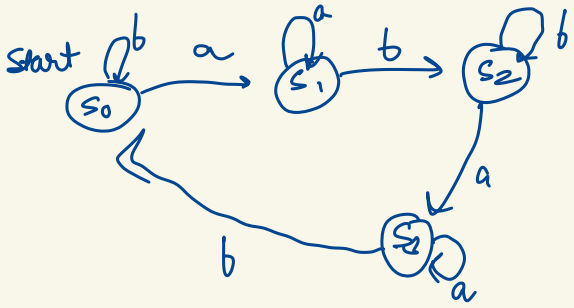
Solution: Possible NFAs are given below. These are not the only solutions. In fact, the first subquestion has a fairly simple DFA that can be directly constructed as well.

1.



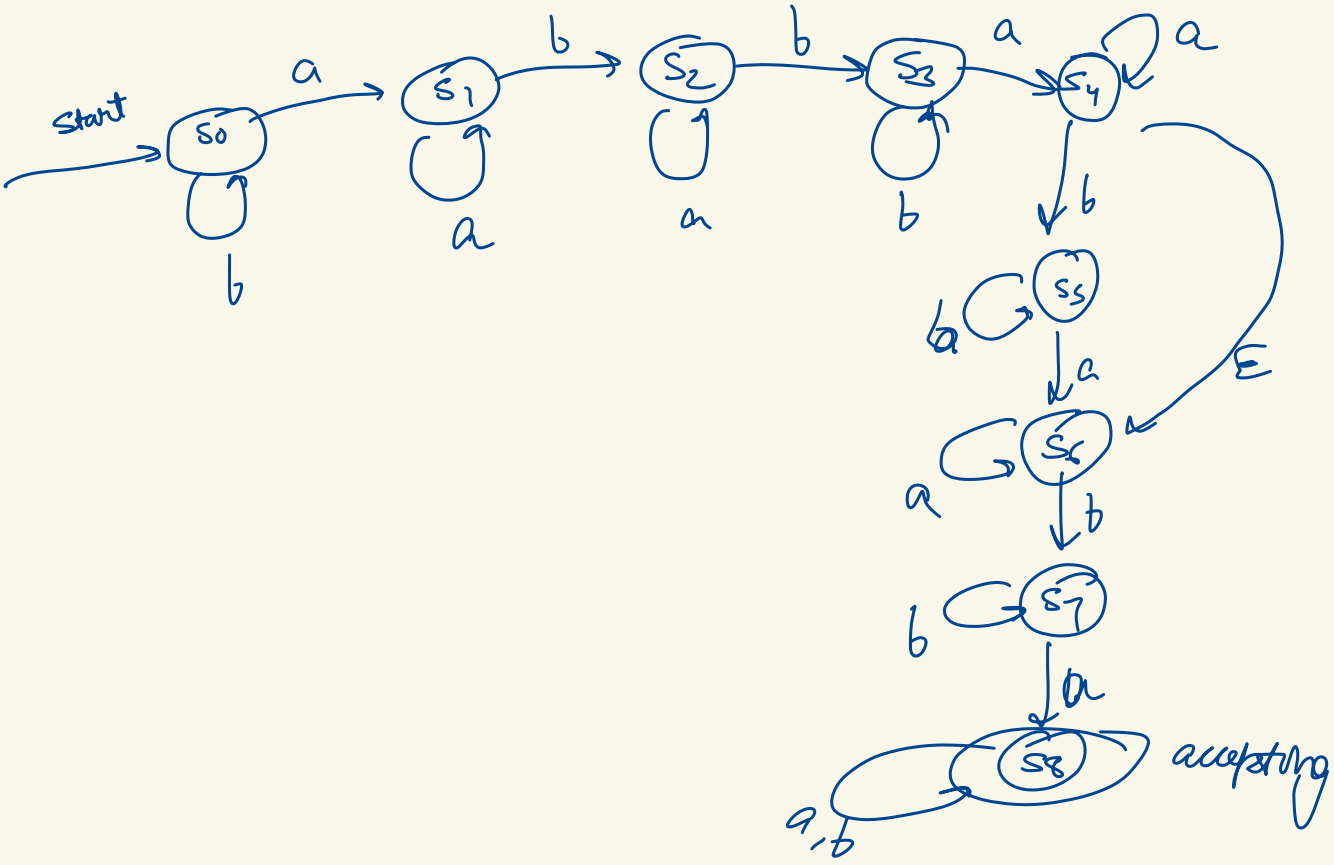
The solution given here illustrates the convenience provided by NFAs with ε-transitions. We have two copies of an NFA that reads a sequence of as and bs with a leading a, a single ab change and possibly a trailing sequence of 0 or more as. These two NFAs can be seen in the top row (read left to right) and in the bottom row (read right to left). We simply connect them using ε-transitions, so that we read words with an even count of ab changes. The acceptance state of the overall NFA is therefore the starting state of the top copy of the NFA. The start states of the overall NFA are chosen so that we can accept a leading sequence of 0 or more as, or even such a sequence of bs followed by as (i.e. strings that don't contain a single ab change).

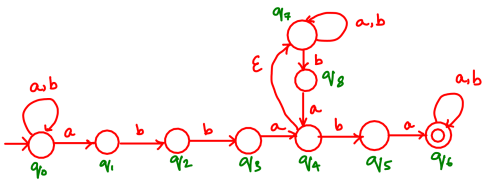
2. 1. $n_{ab}(w)$ = even



2. 2. $L_5 = \{w \in \Sigma^* \mid w$ contains the "pattern" abba (as consecutive letters) followed by the "pattern" baba, possibly in an overlapping manner$\}$. For example, $abababbaabb, bababbabb \notin L$ but $abababbaba, ababbaabbaba \in L$.

abba ba or abba baba



2. 

The NFA is quite self-explanatory. This clearly shows the convenience of NFAs with ε-transitions. You can capture the intent of the problem so clearly that a look at the NFA tells you what it's designed to accept. This is not (so easily) the case if you determinize this NFA.

The NFA to DFA construction is left to you as a standard exercise.

3. **Take-away question: Propositional formulas and NFAs**

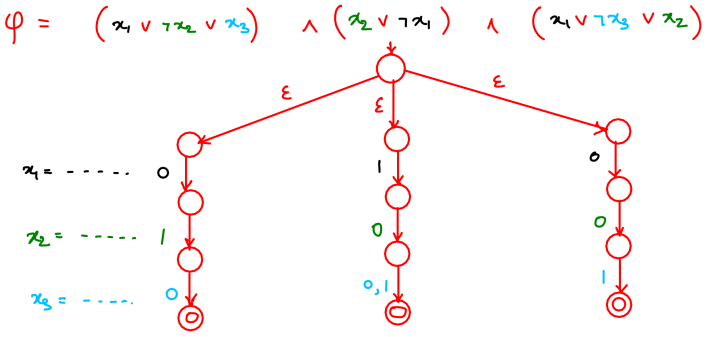
For a CNF formula φ with m variables and c clauses, show that you can construct in polynomial time an NFA with $O(cm)$ states that accepts all **falsifying or non-satisfying assignments**, represented as boolean strings of length m. You can assume that the formula φ is over variables $x_1 x_2 \dots x_m$, and you can assume that the NFA is fed as input the word $v_1 \cdot v_2 \cdot \dots \cdot v_m$, where $v_i \in \{0,1\}$ and v_i is interpreted as the value of propositional variables x_i, for all $i \in \{1, \dots m\}$.

Can you construct an NFA in time polynomial in c and m that accepts all and only **satisfying assignments** of the CNF formula φ?

Solution:

1. On input ϕ, construct an NFA that non-deterministically picks one of the c clauses (via ε-transitions), reads the input of length m, and accepts if it does not satisfy the clause. For each clause, we need exactly m states, so the NFA has $O(cm)$ states. It is clear that we can construct the NFA in polynomial time, hence it is of polynomial size.

   Now consider any falsifying assignment of φ. At least one clause must be falsified by this assignment. Hence all literals of this clause must evaluate to 0. The NFA can non-deterministically choose this clause and accept the assignment string. Conversely, if the NFA accepts an assignment string, then there is a clause of φ that is falsified by the assignment. So, the assignment is a falsifying assignment of φ. Hence, the NFA accepts all and only the nonsatisfying assignments of φ. For example: let $\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_1) \wedge (x_1 \vee \neg x_3 \vee x_2)$, the corresponding NFA is as shown below.



2. Unless we have a polynomial time algorithm for checking satisfiability of CNF formulas (aka **P vs NP** problem), such a construction is not possible. Indeed, if we could do this, then there would be a way to check in time polynomial in c and m whether φ is (un)satisfiable. Why? Simply construct the NFA for φ and check in the graph corresponding to the NFA whether any final state can be reached from any initial state

3. 1. CNF = c clauses, m states input = $v_1 v_2 \cdots v_m$ → $x_1 x_2 \cdots x_m$

non-satisfying assignments → accepted by NFA with $O(cm)$ states

ε transitions to all clause

now all those should evaluate to false

by any path. If so, there is a satisfying assignment, else the formula is unsatisfiable. Note that searching for a path in the graph can be done using any graph search algorithm like BFS or DFS in time that is polynomial in the size of the NFA (viewed as a graph). If the NFA's size is polynomial in c and m, this search (BFS/DFS) will also take time polynomial in c and m.

The pumping lemma for regular languages

Let L be a regular language. Then there exists a constant n (which depends on L) such that every string w in L such that $|w| \geq n$, we can break w into three strings $w = xyz$ such that:

1. $y \neq \varepsilon$
2. $|xy| \leq n$
3. for all $k \geq 0$, the string $xy^k z$ is also in L.

Proof: $L = L(A)$ for some DFA A

$A \rightarrow n$ states

$w = q_1 q_2 \dots q_m \quad m \geq n$

for $i = 0 \dots n$ define $p_i = \hat{\delta}(q_0, q_1, q_2, \dots, q_i)$

$\hat{\delta}$: transition function

$p_0 = q_0$ $\quad q_0$ = start state of A

By pigeonhole principle $p_i = p_j$ for some $i < j$

$w = xyz$

$x = q_1 \dots q_i$

$y = q_{i+1} \dots q_j \rightarrow \neq \varepsilon$

$z = q_{j+1} \dots q_m$

* Regular languages are closed under union, intersection, complement, difference, reversal, closure (star) and concatenation.
* Running time of NFA to DFA conversion, including the case where the NFA has ε-transitions is $O(n^3 2^n)$.
* Automaton → Regex $O(n^3 4^n)$

(i) $R = R_1 + R_2$ $L(R)$ is empty if and only if $L(R_1)$ and $L(R_2)$ are empty.

(ii) $R = R_1 R_2$ $L(R)$ is empty if and only if $L(R_1)$ or $L(R_2)$ is empty

(iii) $R = R_1^*$ $L(R)$ is not empty, it always includes at least ε.

(iv) $R = (R_1)$ $L(R)$ is empty if and only if $L(R_1)$ is empty since they are the same language.

4.1.3 Exercises for Section 4.1

Exercise 4.1.1: Prove that the following are not regular languages.

a) $\{0^n1^n \mid n \geq 1\}$. This language, consisting of a string of 0's followed by an equal-length string of 1's, is the language L_{01} we considered informally at the beginning of the section. Here, you should apply the pumping lemma in the proof.

b) The set of strings of balanced parentheses. These are the strings of characters "(" and ")" that can appear in a well-formed arithmetic expression.

* c) $\{0^n10^n \mid n \geq 1\}$.

d) $\{0^n1^m2^n \mid n$ and m are arbitrary integers$\}$.

e) $\{0^n1^m \mid n \leq m\}$.

f) $\{0^n1^{2n} \mid n \geq 1\}$.

(a) Let DFA (L) has k states

$w = 0^k 1^k$

$|xy| \leq k \rightarrow z$ has $0^r 1^k$ $\quad r \geq 0$

y has some zeroes $(y \neq \epsilon)$

$xz \notin L$ Pumping Lemma violated

(b) Consider '(' = 0 ')' = 1

DFA(L) has k states

$w =$ (((( )))) (k times each)

$|xy| \leq k$ but $xz \notin L$

total number of closed ≠ total number of open

! **Exercise 4.1.2:** Prove that the following are not regular languages.

* a) $\{0^n \mid n \text{ is a perfect square}\}$.

b) $\{0^n \mid n \text{ is a perfect cube}\}$.

c) $\{0^n \mid n \text{ is a power of 2}\}$.

d) The set of strings of 0's and 1's whose length is a perfect square.

e) The set of strings of 0's and 1's that are of the form ww, that is, some string repeated.

f) The set of strings of 0's and 1's that are of the form ww^R, that is, some string followed by its reverse. (See Section 4.2.2 for a formal definition of the reversal of a string.)

g) The set of strings of 0's and 1's of the form $w\overline{w}$, where $\overline{w}$ is formed from w by replacing all 0's by 1's, and vice-versa; e.g., $\overline{011} = 100$, and 011100 is an example of a string in the language.

h) The set of strings of the form $w1^n$, where w is a string of 0's and 1's of length n.

! **Exercise 4.1.3:** Prove that the following are not regular languages.

a) The set of strings of 0's and 1's, beginning with a 1, such that when interpreted as an integer, that integer is a prime.

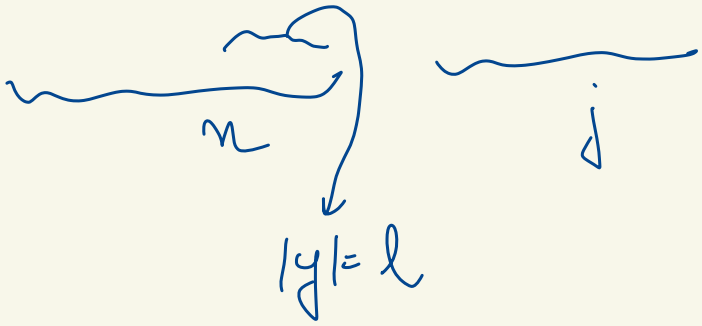
b) The set of strings of the form 0^i1^j such that the greatest common divisor of i and j is 1.

4.1.3 (b) $0^i 1^j$ $\gcd(i,j) = 1$

$\exists n$ s.t. $|xy| \leq n$ $y \neq \epsilon$ $xy^k z \in L$

Consider $i = n$

$\gcd(n, j) = 1$



$|y| = l$

$(n + \lambda l)$ → can be made multiple of j

Hence for that λ, the string will not belong to L

⇒ Contradiction!

4.1.2 (c) 0^n $n = 2^k$ $k \geq 0$

Assume regular $\exists\ l$ s.t $|xy| \leq l$ $y \neq \varepsilon$

$xy^k z \in$

$|xyz| = 2^k$

$2^k + r\lambda \neq$ power of two

$\forall$ every r

$\lambda \neq 0$

(f) ww^R $\exists\ n$ s.t $|xy| \leq n$ $y \neq \varepsilon$

Consider $|w| = n$

$\underbrace{0\,0\,0\,0 \ldots 1}_{w}\ \underbrace{1\,0\,0\,0}_{w^R}$

(i) y has 1 → then reverse not possible

(ii) y has not 1 → then not-satisfiable for a k

(h) $w\,1^n$ $|w| = n$ $w = \{0,1\}^n$

$\exists k$ $|xy| \leq k$ $y \neq \varepsilon$ $xy^k z \notin L$

Let $n = k$

$|w| = k$

1^k

$\underbrace{- - - - - -}_{w}\ \underbrace{1 \ldots 1}_{k \text{ times}}$

Consider $w = \underbrace{0\,0\,0 \ldots 0}_{k \text{ times}}$

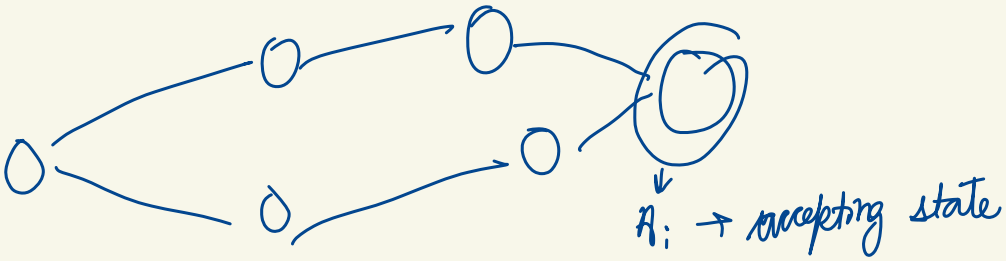
hence not possible.

*! **Exercise 4.2.2:** If L is a language, and a is a symbol, then L/a, the *quotient* of L and a, is the set of strings w such that wa is in L. For example, if $L = \{a, aab, baa\}$, then $L/a = \{\epsilon, ba\}$. Prove that if L is regular, so is L/a. *Hint*: Start with a DFA for L and consider the set of accepting states.

confirm correctness

! **Exercise 4.2.3:** Lf L is a language, and a is a symbol, then $a\backslash L$ is the set of strings w such that aw is in L. For example, if $L = \{a, aab, baa\}$, then $a\backslash L = \{\epsilon, ab\}$. Prove that if L is regular, so is $a\backslash L$. *Hint*: Remember that the regular languages are closed under reversal and under the quotient operation of Exercise 4.2.2.

4.2.2



for all accepting states A_i, search for any incoming edges labelled 'a'. Remove all those edges and make those states as accepting states Then remove A_i and all other incoming edges mapped to those states themselves.

4.2.3

If L then $a\backslash L$ ↑ reverse back

reverse L^R → quotient $L^R \backslash a$

! **Exercise 4.2.6:** Show that the regular languages are closed under the following operations:

a) $min(L) = \{w \mid w \text{ is in } L, \text{ but no proper prefix of } w \text{ is in } L\}$.

b) $max(L) = \{w \mid w \text{ is in } L \text{ and for no } x \text{ other than } \epsilon \text{ is } wx \text{ in } L\}$.

c) $init(L) = \{w \mid \text{for some } x, wx \text{ is in } L\}$. ← can x be ε (doubt)

Hint: Like Exercise 4.2.2, it is easiest to start with a DFA for L and perform a construction to get the desired language.

(a) There should not be any path from one accepting to the other. Neither there should be any self-edge on any of the accepting states.

Remove all the outgoing edges of all the accepting states of the DFA.

(c)

*!! **Exercise 4.2.8:** Let L be a language. Define *half*(L) to be the set of first halves of strings in L, that is, $\{w \mid$ for some x such that $|x| = |w|$, we have wx in $L\}$. For example, if $L = \{\epsilon, 0010, 011, 010110\}$ then *half*$(L) = \{\epsilon, 00, 010\}$. Notice that odd-length strings do not contribute to *half*(L). Prove that if L is a regular language, so is *half*(L).

doubt

$w \rightarrow |w| = |x|$ and $wx \in L$

! **Exercise 4.2.13:** We can use closure properties to help prove certain languages are not regular. Start with the fact that the language

$$L_{0n1n} = \{0^n 1^n \mid n \geq 0\}$$

is not a regular set. Prove the following languages not to be regular by transforming them, using operations known to preserve regularity, to L_{0n1n}:

* a) $\{0^i 1^j \mid i \neq j\}$. → complement $\Sigma^* \setminus L$

b) $\{0^n 1^m 2^{n-m} \mid n \geq m \geq 0\}$.

$0^n \{1+2\}^n$ → not regular

* **Exercise 4.3.1:** Give an algorithm to tell whether a regular language L is infinite. *Hint:* Use the pumping lemma to show that if the language contains any string whose length is above a certain lower limit, then the language must be infinite.

Exercise 4.3.2: Give an algorithm to tell whether a regular language L contains at least 100 strings.

Exercise 4.3.3: Suppose L is a regular language with alphabet Σ. Give an algorithm to tell whether $L = \Sigma^*$, i.e., all strings over its alphabet.

→ search for all strings w such that $n \leq |w| \leq 2n$

where n = number of states in the DFA of L

CS 208 Tutorial 4

1. Let $\Sigma = \{0,1\}$. In each of the following problems, you have to prove the regularity of a desired language.

   1. For every word w in Σ^*, we say $w' \in \Sigma^*$ is a subword of w iff w' is obtained by replacing some letters in w with ε. Thus, 011 is a subword of 0010111, but is not a subword of 0001. Furthermore, for $w \in \Sigma^*$, define w^R as the string w read in reverse. For example, 011^R is 110. Now, let L be a regular language over Σ. Define

      $$\text{Lossy}(L) := \{w' : w' \text{ is a subword of some word in } L\}$$

      The need for such languages arise in real life: Imagine a channel on some network, on which we are transmitting bit streams. However, the channel is lossy, and some bits are lost in transmission. Therefore, if we transmit a word w, we may receive a subword of w. In particular, if we want to transmit words from a regular language L [1], we'll actually end up receiving subwords of words in L [2].

      Show that if L is regular, so is $\{w^R : w \in \text{Lossy}(L)\}$. by constructing an **NFA** for this language.

   2. Let $L_1 := L((0+1)^*01(0+1)^*10)$. Define

      $$L_2 := \bigcup_{a_1 a_2 \cdots a_k = w \in L_1} S(a_0) \cdot S(a_1) \cdots S(a_k)$$

      where $a_i \in \{0,1\}$ are the alphabets of the word $w \in L_1$, and $S(0) := \{0101\}, S(1) := \{1010\}$ are functions mapping each letter of Σ to a singleton language. Construct a **DFA** for L_2.

      Motivation: Such transformations are used in networks, where we encode individual bits into larger chunks for redundancy in our transformation, so that even if a few bits are corrupted, the transmitted word can still be recovered. Also look up string homomorphisms (and inverse homomorphisms) from Hopcroft, Motwani and Ullmann.

   3. Let L_1, L_2 be regular languages over Σ. Define

      $$\text{interleave}(L_1, L_2) := \{\alpha_1\beta_1\alpha_2\beta_2 \ldots \alpha_k\beta_k : \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_k \in \Sigma^*, \alpha_1\alpha_2 \ldots \alpha_k \in L_1, \beta_1\beta_2 \ldots \beta_k \in L_2\}$$

      In the above definition, $\alpha_1\alpha_2 \ldots \alpha_k$ represents the concatenation of strings $\alpha_1, \alpha_2, \ldots, \alpha_k$, and similarly for $\beta_1\beta_2 \ldots \beta_k$.

      Prove that if L_1, L_2 are regular, then $\text{interleave}(L_1, L_2)$ is regular.

      Motivation: Imagine a network where packets are arriving from two different channels (look up TDM channels). Then what is received on the other side is an interleaving of the packets received from both the channels.

[1] Why might we want to do so? Recall that regular expressions and hence regular languages can be used to describe sets of words containing specific patterns.

[2] If you are more interested about such things, you can look up *erasure codes* for further reading.

31. $M_1 = (P, \Sigma, \delta_1, P_0, F_1)$
$M_2 = (Q, \Sigma, \delta_2, Q_0, F_2)$
} DFAs accepting L_1 and L_2 respectively

$M = (P \times Q, \Sigma, \delta, \{(P_0, Q_0)\}, F_1 \times F_2)$

NFA

$\delta((p,q), a) = \{(\delta(p,a), \varepsilon), (p, \delta_2(q,a))\}$

Claim: $L(M)$ interleave (L_1, L_2)

Proof: $\hat{\delta}(\{(P_0, Q_0)\}, x) = \{(\hat{\delta}_1(P_0, x), \hat{\delta}_2(q_0, x))\}$

2. Recall the Pumping Lemma described in class (without mentioning the name as such). To recall: Let L be a regular language. Then there is an integer $p \geq 1$ (depending only on L) such that for any $w \in L$ such that $|w| \geq p$, we can write $w = xyz$, such that:

   1. $|y| \geq 1$.
   2. $|xy| \leq p$. Note that x *may* be ε.
   3. $xy^n z \in L$ for all $n \geq 0$. In particular, $xz \in L$.

   More informally, for any regular language, and for any long enough word in it, after removing some small enough prefix and some suffix, the rest of the word is just some small enough word repeated over and over again, i.e. 'pumped' (that's where the lemma gets its name from).

   The Pumping Lemma is extremely useful to show that a given language is **not regular**. Basically, if we show that a given language L doesn't satisfy the Pumping Lemma, then it can't be regular (since every regular language satisfies the Pumping Lemma).

   The strategy to show a language L non-regular using Pumping Lemma is best viewed as a turn-based game between an adversary (who wants to show that the language is not regular) and a believer (who believes that the language is regular). The game goes as follows:

   - The believer chooses an integer $p > 0$. She claims this is the count of states in the DFA that she believes recognizes the language.
   - The adversary chooses a (often carefully constructed) string $w \in L$ such that $|w| > p$.
   - The believer then splits w into three parts $w = x \cdot y \cdot z$, where $|xy| \leq p$. The believer thinks this is the shortest prefix of w that ends up looping back to a state of the p-state DFA.
   - The adversary now chooses an (often carefully constructed) integer $n \geq 0$ such that $xy^n z \notin L$, thereby winning the game

   If the **adversary can win** the above game (i.e. can choose w in step 2 and n in step 4) *for every choice of p and for every decomposition of w and $x \cdot y \cdot z$ chosen by the believer*, then the language L must be non-regular. Why so? Because if L was indeed regular, there must exist a DFA with a certain number of states accepting L. If the believer chooses p to be this count of states in step 1, then we know from the Pumping Lemma that for every $w \in L$ such that $|w| > p$ which the adversary can choose in step 2, there must exist a decomposition $x \cdot y \cdot z$ with $|xy| \leq p$ such that $xy^n z \in L$ for all $n \geq 0$.

   Take a few moments to understand the above sequence of steps in the game.

   Using the above idea, show that the following languages are **not** regular:

   1. $\{0^n 1^m : n \geq m \geq 0\}$. Consider $n = p$ $m = p$
   2. $\{0^{n^2} : n \geq 0\}$. Consider $n = p$ $n^2 - n > (n-1)^2$ → Hence $\alpha 2 \notin L$

   It is important to note that the pumping lemma is not and "if and only if" statement: If a language is regular, it satisfies the lemma, but not necessarily the other way arond. Indeed, there are languages that are not regular, yet satisfy the conditions of the pumping lemma.

   Try to prove that the following language is not regular (with knowledge of the fact that $\{a^n b^n \mid n \geq 0\}$ is not regular – a fact that can be proved again using the Pumpling Lemma), yet can be pumped.

$$L := \{c^m a^n b^n : m \geq 1, n \geq 0\} \cup a^* b^*$$

$L_1 = \{c a^n b^m\}$ $L_2 = c a^* b^*$

L_1 ≠ regular L_2 = regular $L_2 \cap L = L_1$

If L were regular then L_1 = regular

3. **Takeaway**: In this question, we'll see an almost templatized way of proving that certain transformations of regular languages are regular.
Let $\mathcal{L}$ be a regular language, and let $\mathcal{A} := (Q, \Sigma, \delta, q_0, \mathcal{F})$ be a DFA recognizing $\mathcal{L}$. Note that Q is the set of states of $\mathcal{A}$, $q_0 \in Q$ is the start state, $\mathcal{F} \subseteq Q$ is the set of final states, Σ is our alphabet, and $\delta : Q \times \Sigma \mapsto Q$ is the transition function.
For any $\alpha \in Q, B \subseteq Q$, define $\mathcal{L}(\alpha, B)$ to be the regular language recognized by the DFA $(Q, \Sigma, \delta, \alpha, B)$, i.e. we take $\mathcal{A}$ and change the starting state to α and the end state(s) to B.
Let $\mathcal{L}$ be a regular language. Prove that the following languages are regular:

1. $\text{init}(\mathcal{L}) := \{w : wx \in \mathcal{L} \text{ for some } x \in \Sigma^*\}$ $\alpha = q_0$ $B \rightarrow$ set of all states which can reach F will work.
2. $\text{CubeRoot}(\mathcal{L}) := \{w : w^3 \in \mathcal{L}\}$

[Hint: Try to express these languages as union/intersections/concatenations of $L(\alpha, B)$'s for various α, B.]
See section 4.2 of Hopcroft, Motwani, and Ullmann for more problems of this type.

These questions can also be solved by converting the DFA of $\mathcal{L}$ to NFAs accepting the given languages. However, many of those conversions are clumsy, and it requires some care to show that the transformed automaton accepts precisely the desired language. The above method, however, is short and much more transparent.

Consider two states in between a, b

$q_0 \rightarrow a$
$a \rightarrow b$
$b \rightarrow F$
} → Intersection of these three languages

Union over all a, b

4. **Takeaway**: Consider the 8-letter alphabet

$$\Sigma_3 = \{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)\}$$

Each letter of the alphabet is a 3-tuple of bits. Consider the language formed by words in this language that satisfy the following property: If all the X-coordinates of the letters are considered as a binary number N_X in reverse, and all the Y-coordinates of the letters are considered as a binary number N_Y in reverse, and all the Z-coordinates of the letters are considered as a binary number N_Z in reverse, then $N_X + N_Y = N_Z$.

For example, the word $(0,0,0)(0,1,1)(1,0,1)$ is in the language, since we have $N_X = 100_2 = 4$, $N_Y = 010_2 = 2$ and $N_Z = 110_2 = 6$. However, the word $(0,0,0)(0,1,1)(1,1,0)$ is not in the lanuage, since $N_X = 100_2 = 4$, $N_Y = 110_2 = 6$ but $N_Z = 010_2 = 2$.

1. Prove that this language is regular, and draw a DFA for it. You can think of words accepted by this DFA as denoting the "solutions" to the equation $N_X + N_Y = N_Z$.

Consider any "formula" which is written exclusively using the symbols $+, =, \vee, \neg$, variable symbols, and constant symbols 0 and 1 along with appropriate parenthesis. One such expression is $((x + y = z) \vee (x - y = 1 + 1 + 1)) \wedge (z = 1 + 1 + x + x)$. What are the set of solutions of the expression? For the above expression, the possible satisfying solutions are all tuples $\{(a, a-3, 2+2a) : a \geq 3\}$, and all tuples $\{(a, a+2, 2+2a) : a \geq 0\}$ Here, we investigate whether we can come up with a simple way to check for solutions to such an equation.

1. Prove that the set of satisfying solutions of such a formula in k variables $F(x_1, x_2 \cdots x_k)$ can be written as a regular language over Σ_k where each coordinate is interpreted in reverse binary. Does this give you an algorithm to decide if any such formula has a solution? Could you do such a thing if multiplication was also allowed?

5. **Takeaway**: A student claims that every regular language over a unary alphabet (say, $\Sigma = \{0\}$) is a finite union of languages of the form $Ł_{c,d} = \{0^{c.n+d} \mid n \geq 0\}$ for constant integers $c, d \geq 0$. Either prove the student wrong by providing a regular language over $\Sigma = \{0\}$ that can't be expressed in the above form (you must give a proof of this), or prove that the student's claim is correct.

$L = \{ 0^{a_1} \; 0^{a_2} \; \dots \; 0^{a_k} \; \dots \}$ $\quad a_i > 0$

↳ regular language over unary alphabet

Now we need to find $(c_1, d_1)\ (c_2, d_2) \dots (c_r, d_r)$

such that $\forall i$, $a_i = c_j n + d_j$ for some j

Case-1 : Language has finite strings then :

$a_1, a_2, \dots, a_n$

$d_1 \; d_2 \; \dots \; d_n$ → finite union.

$c_i = 0 \quad \forall i$

Case-2 : Language has infinitely many strings.

Pumping Lemma → $xy^n z$ → $a + b = d$

$x = a \quad z = b$

$y = c$

Steps for converting NFA with ε to NFA without ε :

(1) Write ε-closure for all states q. (transitive closure)

(2) For every state q, find the union of all of the states that can be reached from ε-closure (q) states on each bit x and add edges from q to those states in the union set, if they are not present.

(3) Mark all the states whose ε-closure contains accepting states as accepting.

(4) Remove all the ε-edges.

$* \quad \exists p>0 \quad \forall w \quad (w \in L) \wedge (|w| \geq p) \quad \exists x,y,z \; (w = x.y.z) \quad \forall n \; (n \geq 0) \rightarrow xy^n z \in L$

believer — adversary — believes — adversary

If L is regular then this holds.

believer thinks that L is regular

Adversary thinks that L is not regular.

$\forall p>0 \; \exists w \; (w \in L) \wedge (|w| \geq p) \; \forall x,y,z \; (w = x.y.z) \; \exists n \; (n \geq 0) \wedge xy^n z \notin L$

↳ Negation of above implies that L is not regular. (adversary wins)

1. L is regular # states in DFA for $L = n$

Is L infinite ?

If DFA given: Find a cycle which is reachable from initial state and we can reach the final state from that cycle.

Feed all words w, $n < |w| \leq 2n$ to oracle. If $w \in L$ for some such w, then L is infinite.

If $w \notin L$ for every such w, then L is finite

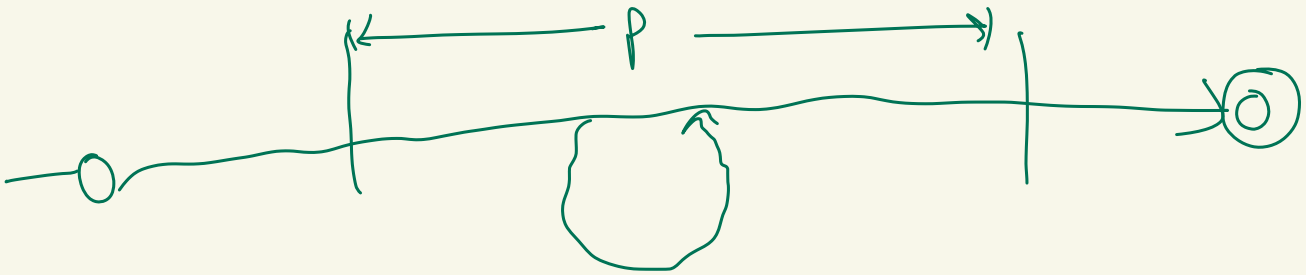
$\sum_{i=n+1}^{2n} |\Sigma|^i$ → maximum number of trials for w → finite

Now remove the y part

$\rightarrow xz \in L$

Start with a string and keep chopping off such cycles.

Angluin's L^* algorithm for learning DFAs



Pumping lemma - can be applied from any point.

$(0+1)^* \; 0 \; 1111$

#1's is prime

Eg: 001 0 1111

$x = \varepsilon \quad y = 0$

$0^i \; 01 \; 0 \; 1111$

This part is completely untouched because of $(0+1)^*$

Consider $01^k \quad k \geq p$

number of states in DFA

Apply pumping lemma to the last part

Then we will be pumping 1s → no longer prime

→ Not regular

→ Another version

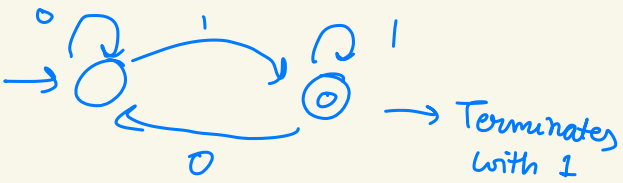
If this language is regular, then the reverse will also be regular, now apply normal pumping lemma

NFA w/ ε

NFA w/o ε

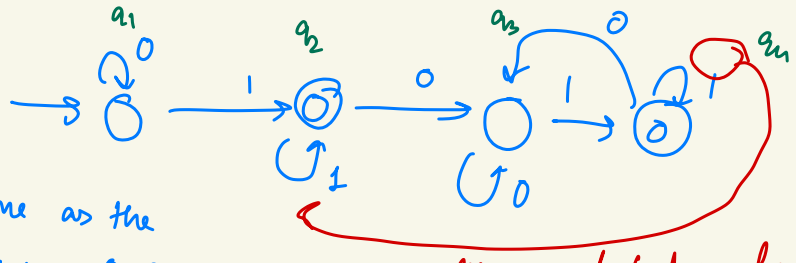
DFA ——————

RE



$(0+1)^* 1$

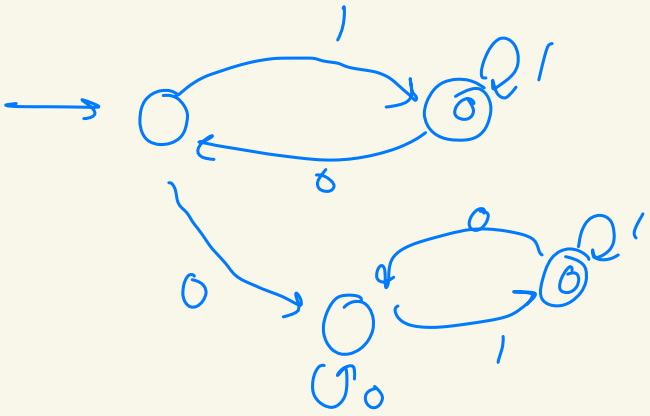
Are there 2-state 4-state DFAs accepting the same language



Same as the previous one

move 1 to here for another 4-state DFA

for 4 state, there are multiple DFAs accepting the same language



Is one-state DFA possible? No !! Consider that state to be accepting / not.

for this language = minimum states in DFA = 2

Claim: for every language numbers of minimum state DFA = 1

Indistinguishability : $\equiv$

$q_i \equiv q_j \quad \forall w \in \Sigma^* \quad q_i \xrightarrow{w} q_i' \quad q_j \xrightarrow{w} q_j'$

either both q_i' and $q_j' \in F$ or both $\notin F$

{ we are basically changing the start states }

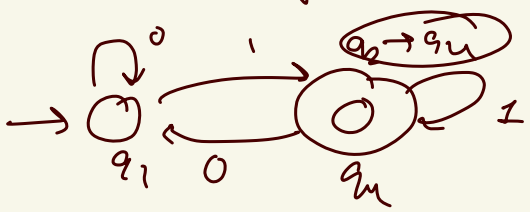
* Reflexive relation
* Symmetric
* Transitive $\quad q_i \equiv q_j \wedge q_j \equiv q_k \Rightarrow q_i \overset{?}{\equiv} q_k$

Do by contradiction, $\exists w$ such that $q_i' \in F$ but $q_k' \notin F$ (or $q_i' \notin F$ but $q_k' \in F$) → Contradiction!

∴ Indistinguishability → equivalent relation

In the above example the equivalence classes are $\{q_1, q_3\}$ $\{q_2, q_4\}$

we can use one from each of the classes



On 1 from q_1, we go to q_2, but we can go to q_4 due to equivalence

Similarly from q_4 to q_3

* An accepting cannot be indistinguishable from non-accepting state (Consider $w = \epsilon$).

(So, mark q_4 as accepting)

* An initial and non-initial state can be clubbed

(some questions arise)

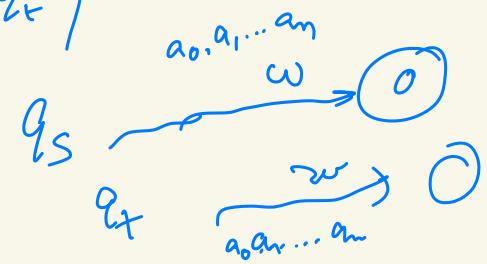
① How do we calculate the equivalence classes of this relation?

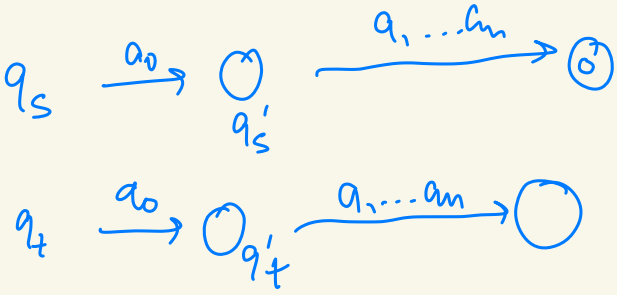
Initially $q_i \equiv q_i \quad \forall q_i \in Q$ $|Q|$ classes initially

$q_i \not\equiv q_j \quad \forall q_i, q_j \in F \times (Q \setminus F)$

Accepting and non-accepting are distinguishable

Suppose these $\begin{pmatrix} q_s \\ q_t \end{pmatrix}$ are distinguishable

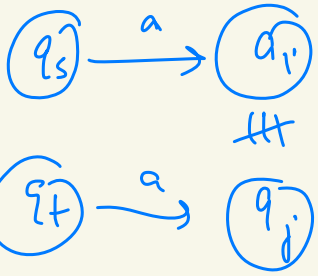




Clearly q_s' and q_t' are distinguishable for $w' = a_1 \ldots a_n$

- $\forall (q_i, q_j)$ s.t. $q_i \notin q_j$

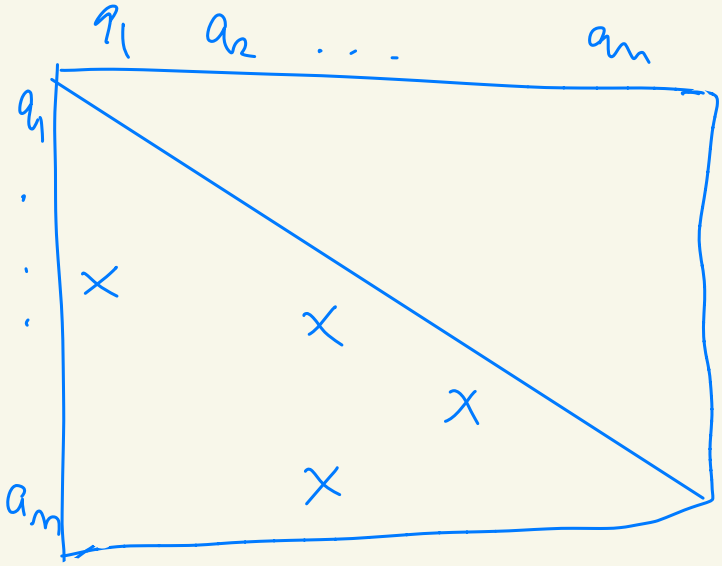
  if $\forall (q_s, q_t) \wedge \exists a \in \Sigma$ such that $q_s \xrightarrow{a} q_i$; $q_t \xrightarrow{a} q_j$ then (q_s, q_t) must be in $\neq$



It is easy to find distinguishable pairs rather than indistinguishable pairs.

↓

Initialize set with final – nonfinal states and then keep on increasing

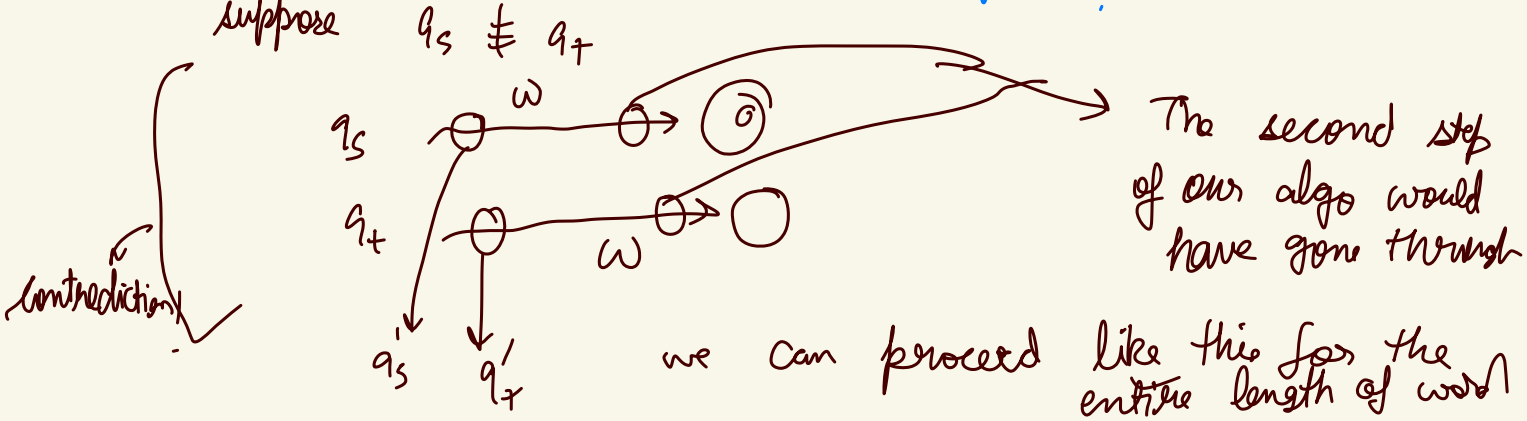


distinguishability → symmetric

start with final – non final and continue the process

stop when no more crosses can be inserted.

Q. Can we still find a pair of distinguishable states, but not detected yet after the algo finishes?



This is an inductive argument.

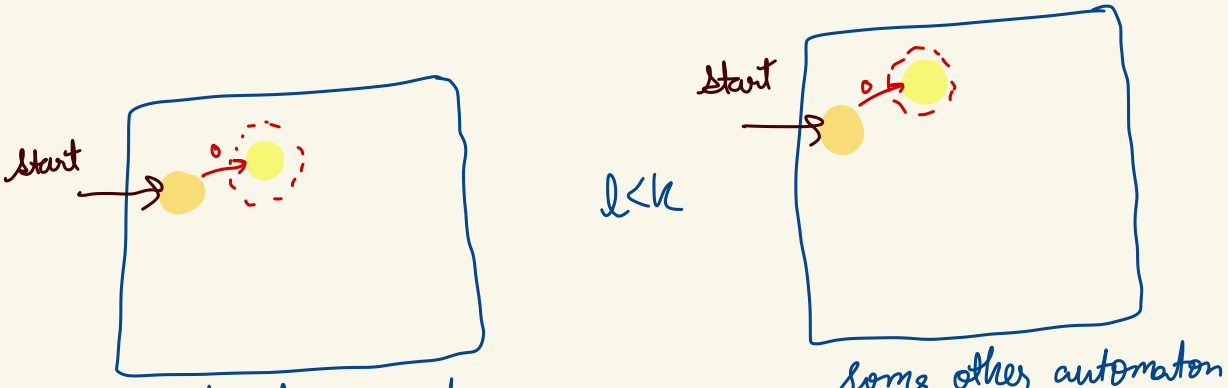
* Algo is not exponential.
* Not filled with crosses → indistinguishable states

Distinguishability → reflexive ✗ symmetric ✓
transitive ✗

Q1 Is it possible to compress further?

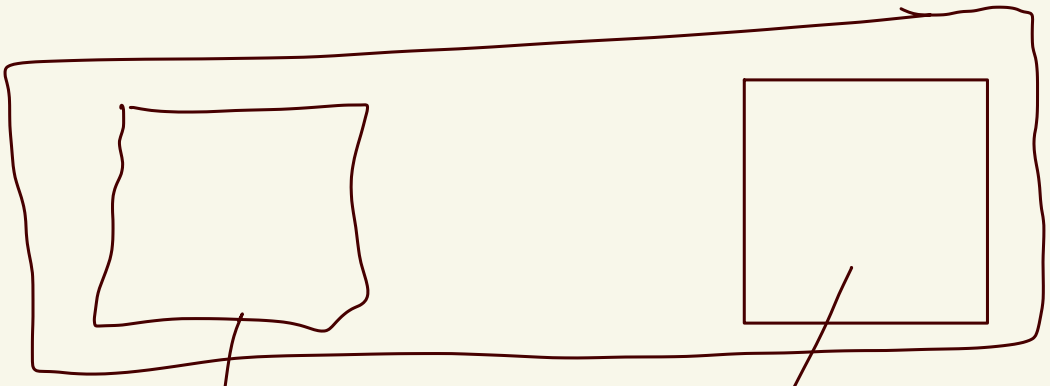
Q2 Any other minimal DFA?

$|\equiv|$: no. of equivalent classes of $\equiv$



Reduced automaton using our method
k states

Some other automaton recognizing same language (DFA)
l states



The two starts of the two automaton have to be indistinguishable (because both of them accept the same language).

every pair distinguishable (because our algo guarantees so)

every pair distinguishable (else we can reduce further)

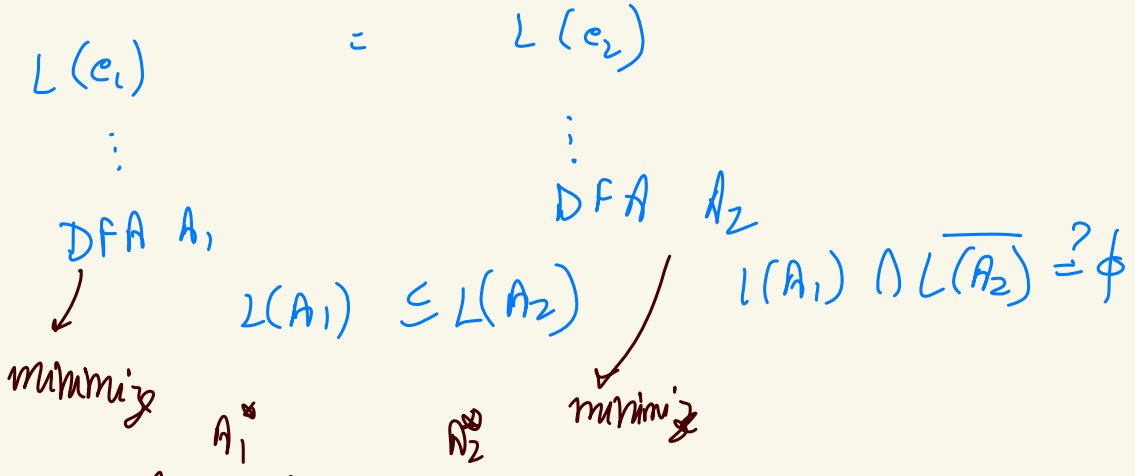
The two yellow-highlighted states are also indistinguishable
(suppose they are distinguishable, then find such word w,
$w' = 0+w$, but start states are indistinguishable',)

Continue this process. (every state is reachable from the start state, else we can throw out that state.)

we have a map from one to another automaton.
It is injective and a function
(because of transitivity of indistinguishability).
$\Rightarrow$ Bijection between two finite sets $\rightarrow$ same cardinality
$l = k$

$\Rightarrow$ not just same cardinality, but isomorphic to each other

$\Rightarrow$ Minimum state DFA = unique



If the languages are same, then they are isomorphic DFAs

$\Sigma = \{0,1\}$ $\quad L = (0+1)^* 1$

$\forall w_1, w_2 \in \Sigma^*$ $\quad w_1 \sim_L w_2$ iff: $\forall x \in \Sigma^*$ $w_1 x$ & $w_2 x$ are both in L or both not in L

Now, we have defined a relation on strings instead of states.

* It is reflexive
* Symmetric
* Transitive

$\rightarrow$ Equivalent relation

$L = \{0^n 1^n \mid n \geq 1\}$

Take two words w_1, w_2 which are related. From start state, we reach q_1, q_2 on seeing w_1, w_2 then $q_1 \equiv q_2$.

* Hence, the number of equivalence classes of this relation = number of states in a minimal DFA.

(Prove by contradiction!)

(It can't be lesser, else we could have reduced the number of states of the DFA!)

$q_i \equiv q_j$ $\quad |\equiv| = 20$

$w_1 \sim_L w_2$

$|\sim_L| \stackrel{?}{=} 21$

→ suppose this was 21, then take a word from i^{th} and j^{th} equivalence class, then the states will be indistinguishable

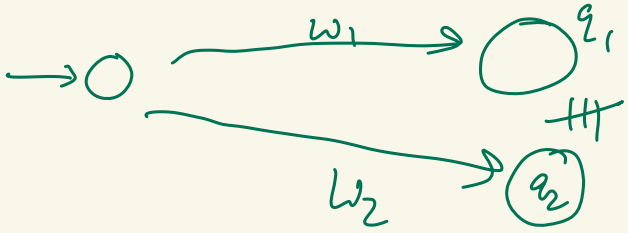
$\exists x \in \Sigma^*$

$w_1 x \in L$

$w_2 x \notin L$

$w_1 \in i^{th}$

$w_2 \in j^{th}$



These states also have to be distinguishable

we can do this for every pair of equivalent classes.

⇒ we can find 21 indistinguishable classes of $\equiv$

but this is not possible

$|\sim_L| \leq 20$

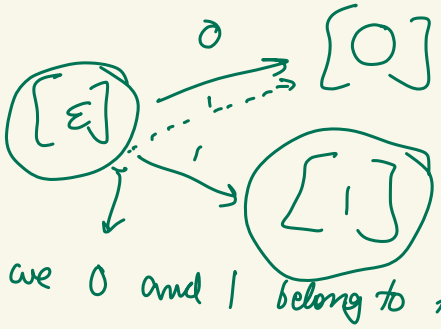
$|\sim_L| \leq |\equiv|$

$\sim_L$

$[\varepsilon]$ → equivalence class containing ε

$\Sigma = \{0, 1\}$

$\sim_L \in \Sigma^* \times \Sigma^*$



Continue this process.

If we 0 and 1 belong to the same class

If we continue, this process, we won't be able to add more circles (finite equivalence classes), we will get an automaton

whichever word is in the language, mark that state to be accepting.

L : any language $\subseteq \Sigma^*$

Myhill - Nerode Theorem

$\sim_L$: $|\sim_L|$ = finite $\Leftrightarrow$ L = regular $\Rightarrow$ PL holds

[Test for regularity of a language]

$\{0^n 1^n \mid n \geq 0\}$

$(^n)^n$

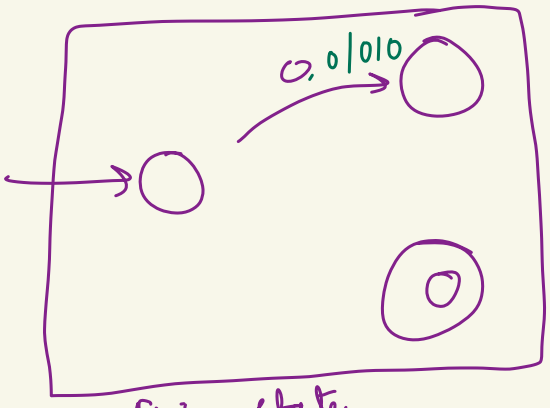
( ) ( ) (( ) ( ))

L: balanced parenthesis

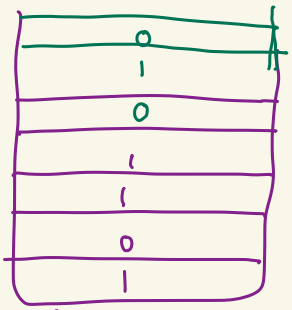
$L \cap (^*)^*$

$= (^n)^n \Leftrightarrow \{0^n 1^n\}$

(Another proof, else use pumping lemma).



Finite state automaton



Stack → we can only access the top element push and pop

Σ, Γ

$(Q, \Sigma, \Gamma, q_0, \delta, F)$ → new representation

Γ: stack elements

$\delta: Q \times \Sigma \to Q$ (earlier)

$\delta: Q \times \Sigma \times \Gamma \to Q \times \Gamma^*$

Γ: we have to check what is present on stack

Γ^*: stack's new stack

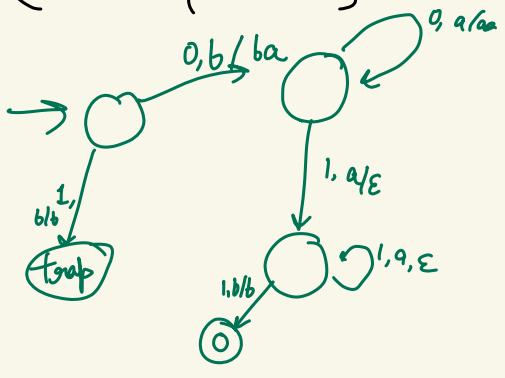
$\delta: q_0, 0, 0, \to q_1, 010$

one more stack / queue → every language can be represented.

$L = \{0^n 1^n \mid n \geq 0\}$

$\Sigma = \{0, 1\}$

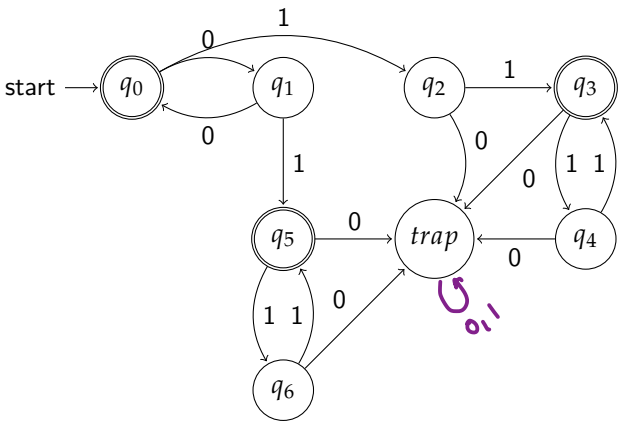
$\Gamma = \{a, b\}$ (b: bottom of the stack, a: top of the stack)

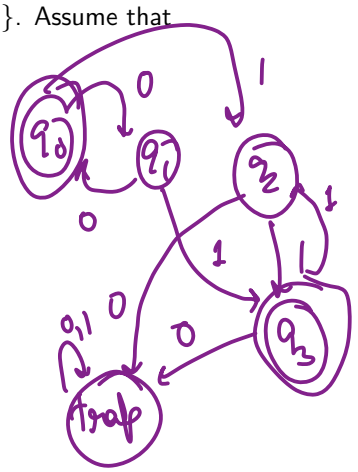


CS208 Tutorial 5: More on automata theory

---

1. Consider the DFA shown below that accepts the language $\{0^n 1^m \mid n+m \text{ is even}\}$. Assume that the trap state loops back to itself on all letters of Σ.

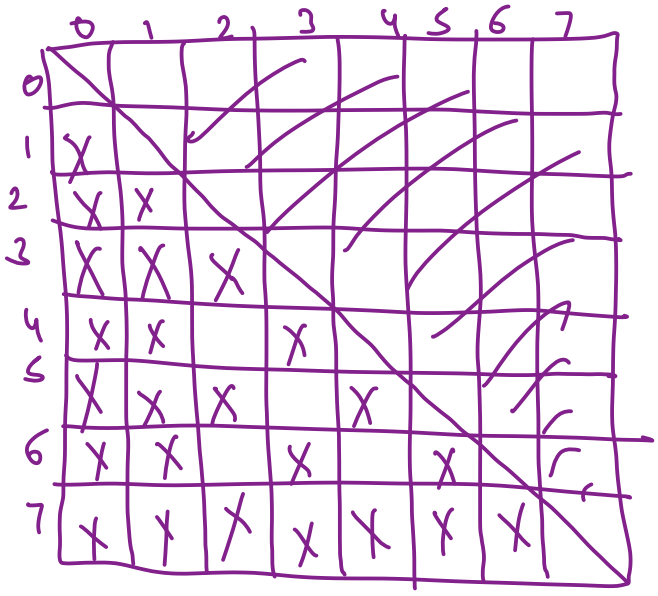




   (a) Using the method discussed in class, find all distinguishable and indistinguishable pairs of states in the above DFA. You can record this by constructing an upper-triangular (or lower-triangular) matrix with 8 rows and 8 columns (corresponding to 8 states of the DFA), as discussed in class.

   (b) Find all equivalence classes of the indistinguishability relation obtained above.

   (c) Using one state from each equivalence class to represent all states of the class, construct a minimal DFA for the language represented by the above DFA.



2. Consider a language $L \subseteq \Sigma^*$ for some finite alphabet Σ. As discussed in class, the *Nerode equivalence* $\sim_L$ is an equivalence relation over Σ^* such that for any $x, y \in \Sigma^*$, $x \sim_L y$ if and only if for every $z \in \Sigma^*$, $xz \in L \iff yz \in L$. The relation $\sim_L$ partitions Σ^* into equivalence classes of words. Hence, each equivalence class of $\sim_L$, viewed as a set of words, is a language by itself.

   Recall further from our discussion in class:

   - The Nerode equivalence is well-defined for every (regular or non-regular) language L over Σ.
   - The *Myhill-Nerode Theorem* states that L is regular if and only if the number of equivalence classes of $\sim_L$ is finite.
   - If the number of equivalence classes of $\sim_L$ equals $k \in \mathbb{N}$, then the unique (upto isomorphism) minimal DFA recognizing L has k states.

   In this problem we will explore the Nerode equivalence and some of its variants.

   (a) For each of the following languages L, describe (in any suitable form) the equivalence classes of $\sim_L$ as languages over $\{0, 1\}$.

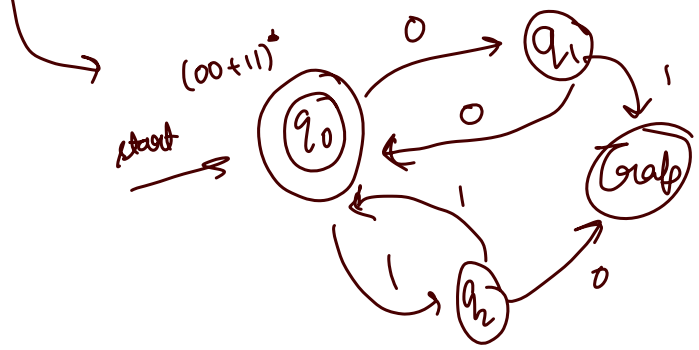
   (a) L is the language corresponding to $(\mathbf{00} + \mathbf{11})^*$

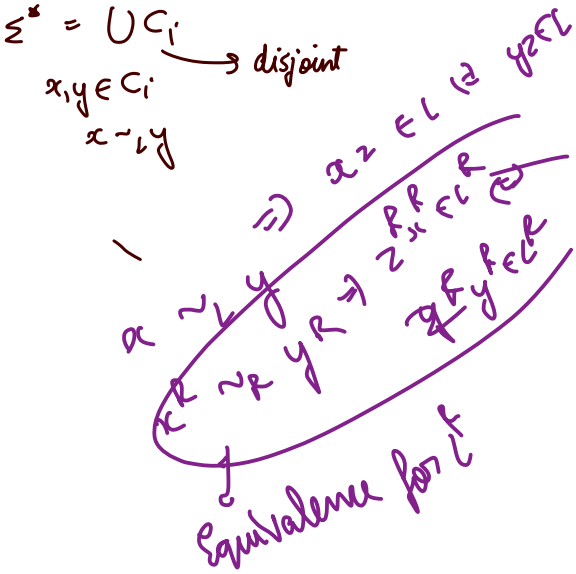
   (b) L is the language $\{0^i 1^j \mid i \leq j\}$

   (b) Define an equivalence relation $\sim_R$ such that for any $x, y \in \Sigma^*$, $x \sim_R y$ if and only if for every $z \in \Sigma^*$, $zx \in L \iff zy \in L$. *Note the difference of $\sim_R$ from the Nerode equivalence $\sim_L$.*

   (i) Show that the number of equivalence classes of $\sim_R$ is finite if and only if L is regular.

   (ii) Let L_{rev} denote the language formed by reversing each string in L. Show that if the number of equivalence classes of $\sim_R$ is k, then the size of the unique minimal DFA recognizing L_{rev} is also k.





(a)(b) $C_1 = L$

$C_2 = (0^* 1^*)^C$ → complement

$C_3 = \bigcup_{k=1}^{\infty} \{ 0^{i+n} 1^k \mid i \geq 0 \}$

(b) $x \sim_R y \iff (\forall z \in \Sigma^*$

$zx \in L \iff zy \in L)$

$\forall z \in \Sigma^*$ $x^R z^R \in L_{rev} \iff y^R z^R \in L_{rev}$

z replaces by

$x^R \sim y^R$ in L_{rev}

$F([x]_{\sim_R}) = [x^R]_{\sim}$

bijection

same cardinality

3. A hacker must figure out what a language L is in order to break into a top-secret system. The hacker knows that the language L is regular and that it is over the alphabet $\{0,1\}$. However, no other information about L is directly available. Instead, an oracle is available that only answers "Yes" or "No" in response to specific types of queries, labeled Q1 and Q2 below.

   Q1 **Does there exist any DFA with n states that recognizes L?**
   For every $n > 0$, the oracle truthfully responds "Yes" or "No" to this query.

   Q2 **Does word w belong to L?**
   For every $w \in \{0,1\}^*$, the oracle truthfully responds "Yes" or "No" to this query.

   We are required to help the hacker re-construct a minimal DFA for L. Towards this end, we will proceed systematically as follows.

   (a) Show that if the minimal state DFA for L has N states, then N can be determined using a sequence of $\log_2 N$ Q1 queries. exponential then binomial
   Hint: Use galloping (or exponential) search.

   (b) Show that it is possible to find a word $w \in L$ or determine that $L = \emptyset$ using at most 2^N Q2 queries. check for all words $2^{N-1} + 2^{N-2} + \ldots + 0$
   Hint: Consider any word in L and repeatedly apply the Pumping Lemma to remove loops in the path from the initial state to an accepting state.

   (c) Once we know the minimal count of states, say N, for a DFA for L, we will construct the Nerode equivalence classes $\sim_L$ for L. Recall from our discussion in class that there are exactly N of these, and each equivalence class can be uniquely identified with a state of the minimal DFA recognizing L.
   For any two distinct equivalence classes of $\sim_L$, show the following:

      (i) There exist words $w_1, w_2 \in \Sigma^*$, where $|w_1| \leq N-1$ and $|w_2| \leq N-1$ such that w_1 belongs to the first equivalence class and w_2 to the second. We will use $[w_1]$ to denote the first equivalence class and $[w_2]$ to denote the second, in the discussion below.

      (ii) For $[w_1] \neq [w_2]$, there is a word $x \in \Sigma^*$ of length $\leq N \times (N-1) - 1$ such that $w_1 \cdot x \in L$ and $w_2 \cdot x \notin L$ or vice versa.

      (iii) For $[w_1] \neq [w_2]$, there exists an edge labeled 0 (resp. 1) from the state corresponding to $[w_1]$ to the state corresponding to $[w_2]$ iff for all $x \in \Sigma^*$, where $|x| \leq N \times (N-1) - 1$, $w_1 \cdot 0 \cdot x$ (resp. $w_1 \cdot 1 \cdot x$) and $w_2 \cdot x$ are either both in L or both not in L.

   Using all the above results, design an algorithm that helps the hacker reconstruct the minimal DFA for L. Give an upper bound on the count of Q2 queries needed for this re-construction, in terms of the count N of the states of the minimal DFA for L.

4. **Takeaway**: You can view this question as a continuation of Question 1 on Nerode equivalences and their variants. Define an equivalence relation $\sim_S$ such that for any $x, y \in \Sigma^*$, $x \sim_R y$ if and only if for every $u, v \in \Sigma^*$, $uxv \in L \iff uyv \in L$.

   (i) Show that the number of equivalence classes of $\sim_S$ is finite if and only if L is regular.

   (ii) Assuming that L is regular, if the minimal DFA recognizing L has k states, show that the number of equivalence classes of $\sim_S$ is at most k^k

5. **Takeaway**: Let $\Sigma = \{a\}$.

   (i) Show that for every language L (regular or not) over Σ, the language $L^* = \bigcup_{i=0}^{\infty} L^i$ is regular.

   (ii) Show that for every regular language L over Σ, there exist two finite sets of words S_1 and S_2 and an integer $n > 0$ such that $L = S_1 \cup S_2 \cdot (a^n)^*$

6. **Takeaway**: The *star-height* of a regular expression $\mathbf{r}$, denoted $\mathsf{SH}(\mathbf{r})$, is a function from regular expressions to natural numbers. It is defined inductively as follows:

   - $\mathsf{SH}(\mathbf{0}) = \mathsf{SH}(\mathbf{0}) = \mathsf{SH}(\boldsymbol{\varepsilon}) = \mathsf{SH}(\boldsymbol{\Phi}) = 0$.
   - $\mathsf{SH}(\mathbf{r_1} + \mathbf{r_2}) = \mathsf{SH}(\mathbf{r_1} \cdot \mathbf{r_2}) = \max(\mathsf{SH}(\mathbf{r_1}), \mathsf{SH}(\mathbf{r_2}))$
   - $\mathsf{SH}(\mathbf{r}^*) = \mathsf{SH}(\mathbf{r}) + 1$

   Give a regular expression $\mathbf{r}$ over $\Sigma = \{0, 1\}$ such that the following hold:

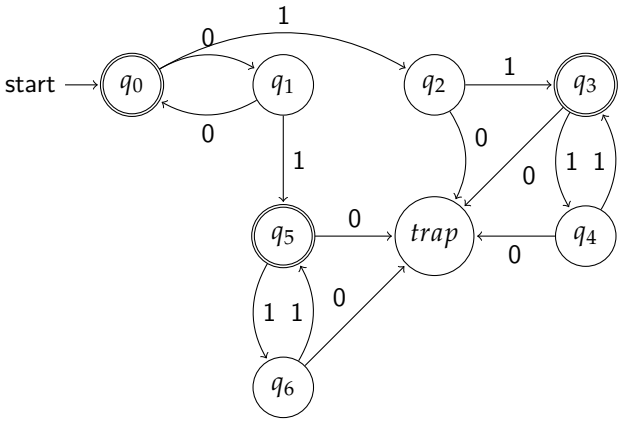
   - $\mathsf{SH}(\mathbf{r}) > 0$, and
   - Every regular expression with star-height $< \mathsf{SH}(\mathbf{r})$ represents a language different from that represented by $\mathbf{r}$.

   You must give brief justification why no regular expression with lesser star-height can represent the same language.

CS208 Tutorial 5: More on automata theory

---

1. Consider the DFA shown below that accepts the language $\{0^n1^m \mid n+m \text{ is even}\}$. Assume that the trap state loops back to itself on all letters of Σ.



   (a) Using the method discussed in class, find all distinguishable and indistinguishable pairs of states in the above DFA. You can record this by constructing an upper-triangular (or lower-triangular) matrix with 8 rows and 8 columns (corresponding to 8 states of the DFA), as discussed in class.

   (b) Find all equivalence classes of the indistinguishability relation obtained above.

   (c) Using one state from each equivalence class to represent all states of the class, construct a minimal DFA for the language represented by the above DFA.

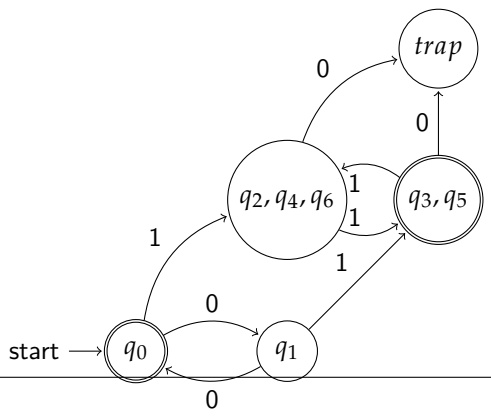
Solution:

(a) Table as shown (State 7 is trap). Each entry is a distinguishing string, if it exists

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	X	ϵ	ϵ	01	ϵ	01	ϵ	ϵ
1		X	0	ϵ	0	ϵ	0	0
2			X	ϵ		ϵ		1
3				X	ϵ		ϵ	ϵ
4					X	ϵ		1
5						X	ϵ	ϵ
6							X	1
7								X

(b) The equivalence classes are $\{q_0, q_1, (q_2, q_4, q_6), (q_3, q_5), trap\}$

(c) The minimized DFA is as shown below:



2. Consider a language $L \subseteq \Sigma^*$ for some finite alphabet Σ. As discussed in class, the *Nerode equivalence* $\sim_L$ is an equivalence relation over Σ^* such that for any $x, y \in \Sigma^*$, $x \sim_L y$ if and only if for every $z \in \Sigma^*$, $xz \in L \iff yz \in L$. The relation $\sim_L$ partitions Σ^* into equivalence classes of words. Hence, each equivalence class of $\sim_L$, viewed as a set of words, is a language by itself.

   Recall further from our discussion in class:

   - The Nerode equivalence is well-defined for every (regular or non-regular) language L over Σ.
   - The *Myhill-Nerode Theorem* states that L is regular if and only if the number of equivalence classes of $\sim_L$ is finite.
   - If the number of equivalence classes of $\sim_L$ equals $k \in \mathbb{N}$, then the unique (upto isomorphism) minimal DFA recognizing L has k states.

   In this problem we will explore the Nerode equivalence and some of its variants.

   (a) For each of the following languages L, describe (in any suitable form) the equivalence classes of $\sim_L$ as languages over $\{0, 1\}$.
      (a) L is the language corresponding to $(\mathbf{00} + \mathbf{11})^*$
      (b) L is the language $\{0^i 1^j \mid i \leq j\}$

   (b) Define an equivalence relation $\sim_R$ such that for any $x, y \in \Sigma^*$, $x \sim_R y$ if and only if for every $z \in \Sigma^*$, $zx \in L \iff zy \in L$. *Note the difference of $\sim_R$ from the Nerode equivalence $\sim_L$.*
      (i) Show that the number of equivalence classes of $\sim_R$ is finite if and only if L is regular.
      (ii) Let L_{rev} denote the language formed by reversing each string in L. Show that if the number of equivalence classes of $\sim_R$ is k, then the size of the unique minimal DFA recognizing L_{rev} is also k.

Solution:

(a) (a) For a regular language like the one in this question, one way to obtain the Nerode equivalence classes is to first construct the *minimal* DFA for the language, and then for each state q in the minimal DFA, list down the set of strings that bring you from the start state of the DFA to state q. Clearly, all such strings w must be in the same Nerode class, since for every string $x \in \Sigma^*$, whether $wx \in L$ or not simply depends on whether you can reach the accepting state of the minimal DFA from q on reading x. Similarly, every string w' that doesn't bring you from the start state to q can't be Nerode equivalent to w since w' must be brining you to a different state q', starting from the start state. However, since q and q' are two different states in the minimal DFA for L, there is a distinguishing string x' such that x' is accepted starting from q and not accepted starting from q' or vice versa. It follows that one of $w.x'$ and $w'.x'$ is in L and the other isn't. Hence, w and w' can't be in the same Nerode class.

We leave it as an exercise for you to construct the minimal DFA for the given language L. Once you do that, it is easy to follow the steps outlined above to find the following Nerode equivalence classes:

$S_1 = L$ (set of strings that bring you to the accepting state of the DFA), $S_2 = (00 + 11)^*0, S_3 = (00 + 11)^*1, S_4 = \Sigma^* - \{S_1 \cup S_2 \cup S_3\}\}$

(b) Since this language is not regular (why? Try using the Pumping Lemma for regular languages), we can't use the above method for finding the Nerode equivalence classes. Hence, we have to look into the specifics of the language and try to construct infinitely many Nerode equivalence classes (Myhill-Nerode theorem guarantees that there are infinitely many Nerode equivalence classes for a non-regular language).

It is easy to see that all strings w not belonging to 0^*1^* are equivalent, since no matter what string x you concatenate to w, the string $w.x$ is not in 0^*1^*, and hence not in L. So all strings not in 0^*1^* form one Nerode equivalence class, say C_1

Let us now focus on strings in 0^*1^*. Consider one such string $w = 0^i1^j$, where $j \geq i$ and $w \neq \varepsilon$ (i.e. it is not the case that $j = i = 0$). Notice that for every string $x \in 1^*$, $w.x \in L$. Similarly, for every string $x \notin 1^*$, $w.x \notin L$. Hence, all strings $w = 0^i1^j$, where $j \geq i$ and $w \neq \varepsilon$ are in the same Nerode equivalence class, say C_2 (they cannot be distinguished by any string $x \in \Sigma^*$). Moreover, C_1 is different from C_2, since ε distinguishes any string in C_1 from any string in C_2. What if $w = \varepsilon$?. The string $x = 01$ distinguishes w from every string in $C_1 \cup C_2$. Hence ε is in a class, say C_3, by itself.

What about strings of the form 0^i1^j, where $i > j$. Consider any such string $w = 0^i1^j$. The string $x = 1^{i-j}$ distinguishes w from every string in C_1. The string ε distinguishes w from every string in C_2. The string $x = 01$ distinguishes w from ε. The string $x = 1^{\min(i-j,i'-j')}$ distinguishes w from every string $w' = 0^{i'}1^{j'}$, where $i' > j'$ and $i - j \neq i' - j'$. Finally, for every string $w' = 0^{i'}1^{j'}$, where $i' > j'$ and $i - j = i' - j'$, no string x can distinguish w from w'. Therefore, all strings $0^{i+k}1^i$ belong to the same Nerode equivalence class for each $k > 0$, and the classes corresponding to k_1, k_2, where $k_1 \neq k_2$ are distinct.

Summarizing all the above cases, the Nerode equivalence classes are: $C_1 = L(0^*1^*)^c, C_2 = L \setminus \{\varepsilon\}, C_3 = \{\varepsilon\}, C_{3+k} = \{0^{i+k}1^i | i \geq 0\}$, for every $k \geq 1$.

(b) Let $\sim_N$ denote the Nerode equivalence relation for the language L_{rev} formed by reversing each string in L. Now, $x \sim_R y$ if and only if for every $z \in \Sigma^*$, $zx \in L \iff zy \in L$, ie $x^Rz^R \in L_{rev} \iff y^Rz^R \in L_{rev}$ for every $z \in \Sigma^*$, ie $x^R \sim_N y^R$.

Consider the function f mapping equivalence classes of $\sim_R$ to equivalence classes of $\sim_N$ such that for any $x \in \Sigma^*$, $f([x]_R) = [x^R]_N$. Since for any $x, y \in \Sigma^*$, $x \sim_R y$ if and only if $x^R \sim_N y^R$, f is well defined and also an injection, and since $(x^R)^R = x$ for any $x \in \Sigma^*$, $[x]_N = f([x^R]_R)$, ie f is a surjection as well. Hence there exists a bijection between the equivalence classes of $\sim_R$ and the equivalence classes of $\sim_N$, ie they have the same cardinality.

Now, by the *Myhill-Nerode Theorem*, the number of equivalence classes of $\sim_N$ is finite if and only if L_{rev} is regular. Now, for any language L, L_{rev} is regular if and only if L is regular (this can be seen by reversing the transitions in the DFA recognizing L to get an NFA recognizing L_{rev}, and also noting that $(L_{rev})_{rev} = L$).

Considering this, along with the fact the set of equivalence classes of $\sim_R$ has the same cardinality as the set of equivalence classes of $\sim_N$, we get that $\sim_R$ has a finite number of equivalence classes if and only if L is regular. Moroever, if $\sim_R$ has k equivalence classes, then so does $\sim_N$, which, by the *Myhill-Nerode Theorem* means that the unique minimal DFA recognizing L_{rev} has k states.

3. A hacker must figure out what a language L is in order to break into a top-secret system. The hacker knows that the language L is regular and that it is over the alphabet $\{0,1\}$. However, no other information about L is directly available. Instead, an oracle is available that only answers "Yes" or "No" in response to specific types of queries, labeled Q1 and Q2 below.

   Q1 **Does there exist any DFA with n states that recognizes L?**
   For every $n > 0$, the oracle truthfully responds "Yes" or "No" to this query.

   Q2 **Does word w belong to L?**
   For every $w \in \{0,1\}^*$, the oracle truthfully responds "Yes" or "No" to this query.

   We are required to help the hacker re-construct a minimal DFA for L. Towards this end, we will proceed systematically as follows.

   (a) Show that if the minimal state DFA for L has N states, then N can be determined using a sequence of $\mathcal{O}(\log_2 N)$ Q1 queries.
   Hint: Use galloping (or exponential) search.

   (b) Show that it is possible to find a word $w \in L$ or determine that $L = \emptyset$ using at most 2^N Q2 queries.
   Hint: Consider any word in L and repeatedly apply the Pumping Lemma to remove loops in the path from the initial state to an accepting state.

   (c) Once we know the minimal count of states, say N, for a DFA for L, we will construct the Nerode equivalence classes $\sim_L$ for L. Recall from our discussion in class that there are exactly N of these, and each equivalence class can be uniquely identified with a state of the minimal DFA recognizing L.
   For any two distinct equivalence classes of $\sim_L$, show the following:
      (i) There exist words $w_1, w_2 \in \Sigma^*$, where $|w_1| \leq N-1$ and $|w_2| \leq N-1$ such that w_1 belongs to the first equivalence class and w_2 to the second. We will use $[w_1]$ to denote the first equivalence class and $[w_2]$ to denote the second, in the discussion below.
      (ii) For $[w_1] \neq [w_2]$, there is a word $x \in \Sigma^*$ of length $\leq N \times (N-1) - 1$ such that $w_1 \cdot x \in L$ and $w_2 \cdot x \notin L$ or vice versa.
      (iii) For $[w_1] \neq [w_2]$, there exists an edge labeled 0 (resp. 1) from the state corresponding to $[w_1]$ to the state corresponding to $[w_2]$ iff for all $x \in \Sigma^*$, where $|x| \leq N \times (N-1) - 1$, $w_1 \cdot 0 \cdot x$ (resp. $w_1 \cdot 1 \cdot x$) and $w_2 \cdot x$ are either both in L or both not in L.

   Using all the above results, design an algorithm that helps the hacker reconstruct the minimal DFA for L. Give an upper bound on the count of Q2 queries needed for this re-construction, in terms of the count N of the states of the minimal DFA for L.

Solution:

(a) We make Q1 queries using powers of 2 $(1, 2, 4, \ldots)$ until it returns "yes". Say, it returns yes for 2^{k+1}. Then, we know the smallest DFA representing the language has size between $2^k + 1$ and 2^{k+1}. We perform binary search over this space. Total number of queries are $k + 2 + \mathcal{O}(\log(2^{k+1} - 2^k)) = 2k + 2 \in \mathcal{O}(\log_2 N)$

(b) **Claim**: A DFA whose language is non-empty having N states accepts a word of length at most $N - 1$. Proof is left as an exercise to the reader (Use ideas similar to Pumping Lemma). Hence, we can make Q2 queries over all possible words having length less than or equal to $N - 1$. Either we conclude that the language is empty or find a word belonging to the language in at most $2^0 + 2^1 + \ldots 2^{N-1} = 2^N - 1$ Q2 queries

(c) (i) If q_1 and q_2 are the states corresponding to these equivalence classes in the minimal DFA (and q_0 is the initial state), then a word w is in the equivalence class of q_1 iff $\hat{\delta}(q_0, w) = q_1$ (and similarly for q_2). Since equivalence classes are by definition non-empty, such a word w_1 necessarily exists. We can remove cycles in the path this word takes from q_0 to q_1 word to ensure that it's length is at most $N-1$ (Similarly for w_2).

(ii) Let q_1 be the state corresponding to $[w_1]$ and q_2 the state corresponding to $[w_2]$. Since $[w_1]$ and $[w_2]$ are distinct equivalence classes, there must exist a string x such that exactly one of w_1x and w_2x are in L. We will show that there exists such an x with length at most $N-2$.

Consider an equivalence relation $\sim_k$ over the state set Q of the minimal DFA where $q_1 \sim_k q_2$ iff for every string x of length at most k, $\hat{\delta}(q_1, x) \in F \iff \hat{\delta}(q_2, x) \in F$. Some observations:

i. $q_1 \sim_0 q_2$ iff $q_1 \in F \iff q_2 \in F$

ii. $q_1 \sim_{k+1} q_2 \implies$ q_1 $\sim_k q_2$, ie the equivalence classes of $\sim_{k+1}$ are subsets of those of $\sim_k$

By the second observation, $\sim_{k+1}$ has at least as many equivalence classes as $\sim_k$, and if the number of equivalence classes is the same, then $\sim_{k+1}$ and $\sim_k$ are identical. Note that $\sim_0$ has 2 equivalence classes. This means that in the number of equivalence classes of the sequence $\sim_0, \sim_1, \sim_2, \ldots$ keeps increasing from 2, until some k where $\sim_k = \sim_{k+1}$, after which it remains constant. Since the number of equivalence classes of $\sim_0$ is 2, and the number of equivalence classes of $\sim_k$ is at most N, k can be at most $N-2$.

Now, for distinct q_1, q_2, since the DFA is minimal, there exists some string x such that exactly one of $\hat{\delta}(q_1, x)$ and $\hat{\delta}(q_2, x)$ lies in F. Say $|x| = p$. Then $q_1 \nsim_p q_2$. If $p > N-2$, then $\sim_p = \sim k$, ie $q_1 \nsim_k q_2$, ie there is some x' of length at most k such that exactly one of $\hat{\delta}(q_1, x')$ and $\hat{\delta}(q_2, x')$ is in F. Since k is at most $N-2$, this means for any distinct q_1 and q_2 there will exist a string x of length at most $N-2$ such that exactly one of $\hat{\delta}(q_1, x)$ and $\hat{\delta}(q_2, x)$ lies in F. This means that for distinct $[w_1]$ and $[w_2]$ there will exist an x of length at most $N-2$ such that exactly one of w_1x and w_2x is in L.

Algorithm:
Find the value of N (Part a). Consider all words having length less than or equal to $N-1$. Find equivalence classes over these words by taking pairs at a time and iterating over all words having length less than or equal to $N-2$. If you find a distinguisher, they're in different equivalence classes, else they are in the same equivalence class. Each equivalence class now represents a state. Accepting state is simply found by using Q2 on one word in each equivalence class. To find the transition function, we can simply consider the shortest words in each equivalence class and use their prefixes to construct the path from the starting state. The starting state is the class which contains epsilon.

4. **Takeaway**: You can view this question as a continuation of Question 1 on Nerode equivalences and their variants. Define an equivalence relation $\sim_S$ such that for any $x, y \in \Sigma^*$, $x \sim_R y$ if and only if for every $u, v \in \Sigma^*$, $uxv \in L \iff uyv \in L$.

(i) Show that the number of equivalence classes of $\sim_S$ is finite if and only if L is regular.

(ii) Assuming that L is regular, if the minimal DFA recognizing L has k states, show that the number of equivalence classes of $\sim_S$ is at most k^k

Solution: Say L is regular and is recognized by minimal DFA $(Q, \Sigma, \delta, q_0, F)$.

If $x \sim_S y$, then for every state $q \in Q$ we must have $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$. To see this, note that since $(Q, \Sigma, \delta, q_0, F)$ is the minimal DFA recognizing L, for every $q \in Q$ there exists $u \in \Sigma^*$ such that $\hat{\delta}(q_0, u) = q$. Furthermore, if $q_1 \neq q_2$ are distinct states in Q, then there exists $v \in \Sigma^*$ such that exactly one of the following are true:

- $\hat{\delta}(q_1, v) \in F$
- $\hat{\delta}(q_2, v) \in F$

(otherwise the states q_1 and q_2 could be merged). Now, if $x \sim_S y$, but if there is some q such that $\hat{\delta}(q, x) \neq \hat{\delta}(q, y)$ (call these q_1 and q_2), then there exists $u \in \Sigma^*$ such that $\hat{\delta}(q_0, u) = q$ and there exists $v \in \Sigma^*$ such that (WLOG) $\hat{\delta}(q_1, v) \in F$ and $\hat{\delta}(q_2, v) \notin F$. This means that there exist $u, v \in \Sigma^*$ such that $\hat{\delta}(q_0, uxv) \in F$ but $\hat{\delta}(q_0, uyv) \notin F$, which means $uxv \in L$ but $uyv \notin L$, contradicting the definition of $\sim_S$.

Moreover, if $\hat{\delta}(q, x) = \hat{\delta}(q, y)$ for every $q \in Q$, then for every $u, v \in \Sigma^*$, $\hat{\delta}(q_0, uxv) = \hat{\delta}(q_0, uyv)$, ie $uxv \in L \iff uyv \in L$, ie $x \sim_S y$. Therefore, for any $x, y \in \Sigma^*$, $x \sim_S y$ if and only if for every $q \in Q$, $\hat{\delta}(q, x) = \hat{\delta}(q, y)$.

Consider the set of functions from Q to itself, denoted by Q^Q. Consider the function f mapping equivalence classes of $\sim_S$ to elements of Q^Q such that for every $q \in Q$, $f([x]_S)(q) = \hat{\delta}(q, x)$. By the previous result, f is well defined and an injection. Therefore, there exists an injection from the equivalence classes of $\sim_L$ to Q^Q, a finite set.

Therefore, if L is regular, then the number of equivalence classes of $\sim_L$ must be finite, and is at most $\left|Q^Q\right|$, where Q is the set of states of the minimal DFA recognizing L. If $|Q| = k$, then we get that the number of equivalence classes is at most k^k.

It is easier to show the other direction of the implication, ie if the number of equivalence classes of L is finite, then L must be regular. This can be done by constructing a DFA recognizing L. Consider the DFA whose set of states $Q = \{[x]_S : x \in \Sigma^*\}$ (ie the set of states is the set of equivalence classes of $\sim_S$), and transitions are of the form $[x]_S \xrightarrow{a} [xa]_S$, for any $a \in \Sigma$ (ie $\delta([x], a) = [xa]$). This transition function is well defined, since if $x \sim_S y$, then $xa \sim_S ya$ for any $a \in \Sigma$. The initial state is taken to be $q_0 = [\varepsilon]_S$ and the set of final states is $F = \{[x]_S : x \in L\}$. It can be shown that the language recognized by this automaton is precisely L (note that $\hat{\delta}(q_0, x) = [x]$ and if $x \sim_S y$ then $x \in L \iff y \in L$).

5. **Takeaway**: Let $\Sigma = \{a\}$.

   (i) Show that for every language L (regular or not) over Σ, the language $L^* = \bigcup_{i=0}^{\infty} L^i$ is regular.

   (ii) Show that for every regular language L over Σ, there exist two finite sets of words S_1 and S_2 and an integer $n > 0$ such that $L = S_1 \cup S_2 \cdot (a^n)^*$

> **Solution:** (a) To those interested, please read here
>
> (b) Refer to the solution of Tut 4, Question 5

6. **Takeaway**: The *star-height* of a regular expression $\mathbf{r}$, denoted $\mathsf{SH}(\mathbf{r})$, is a function from regular expressions to natural numbers. It is defined inductively as follows:

   - $\mathsf{SH}(\mathbf{0}) = \mathsf{SH}(\mathbf{1}) = \mathsf{SH}(\boldsymbol{\varepsilon}) = \mathsf{SH}(\boldsymbol{\Phi}) = 0$.
   - $\mathsf{SH}(\mathbf{r_1} + \mathbf{r_2}) = \mathsf{SH}(\mathbf{r_1} \cdot \mathbf{r_2}) = \max(\mathsf{SH}(\mathbf{r_1}), \mathsf{SH}(\mathbf{r_2}))$
   - $\mathsf{SH}(\mathbf{r}^*) = \mathsf{SH}(\mathbf{r}) + 1$

   Give a regular expression $\mathbf{r}$ over $\Sigma = \{0, 1\}$ such that the following hold:

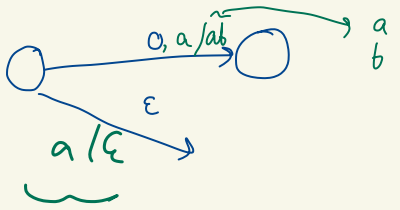
   - $\mathsf{SH}(\mathbf{r}) > 0$, and
   - Every regular expression with star-height $< \mathsf{SH}(\mathbf{r})$ represents a language different from that represented by $\mathbf{r}$.

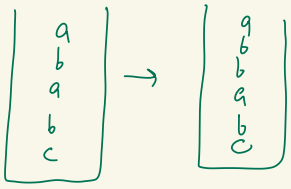
   You must give a brief justification why no regular expression with lesser star-height can represent the same language.

   Solution: The answer to this specific question is really simple if you think about the definition of star height. However, the study of star heights of regular expressions and about the hierarchy of languages corresponding to increasing star heights is very interesting. For those interested in knowing more about star heights, a good starting point is here

   For this specific question, you can simply take the regular expression $\mathbf{0}^*$, which has star height 1. What are the regular expressions with star height < 1. These are $\mathbf{1}, \mathbf{0}, \boldsymbol{\varepsilon}, \boldsymbol{\Phi}$ and combinations of these regular expressions using $+$ and $\cdot$. All of these represent languages with finitely many words, while $\mathbf{0}^*$ represents a language with infinitely many words.

   However, the study of the star height hierarchy is not just limited to star heights of 0 and 1. It extends to all star heights (see here for more details).





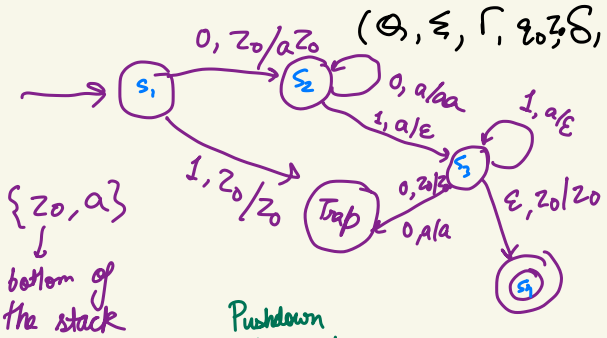
$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$ for stack NFA

$Q \times (\Sigma \cup \{\varepsilon\}) \to 2^{Q}$ NFA

Just popping

$\{0^n 1^m \mid n \geq 1\}$

$(Q, \Sigma, \Gamma, q_0, \delta, F)$

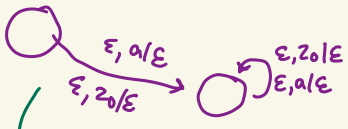


$\Gamma = \{Z_0, a\}$ bottom of the stack

Pushdown Automaton (PDA)

Just make the stack redundant to accept regular languages

To emptying stack:



Acceptance by final state $L(A)$

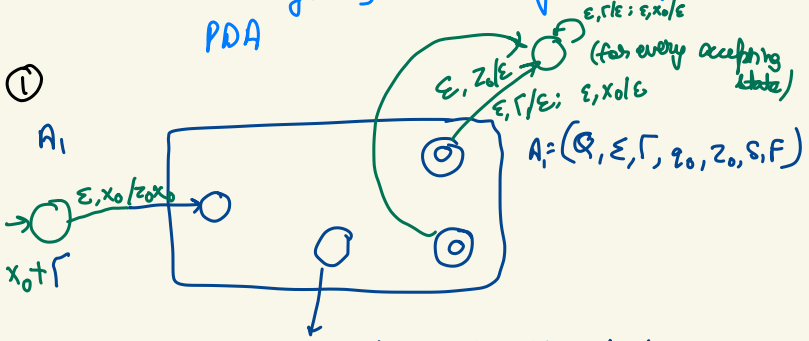
Acceptance by empty stack / null state $N(A)$

Using this we can transform

$\{0^n 1^m \mid n \geq 1, m \geq n\}$

Just change S_3 to accepting in the previous PDA

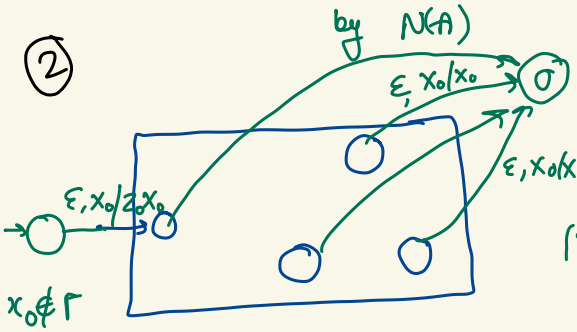
① A_1



$A_1 = (Q, \Sigma, \Gamma, q_0, Z_0, S, F)$

non-accepting but empty stack

But in new case, X_0 will be there, hence not accepted by NFA)

② 

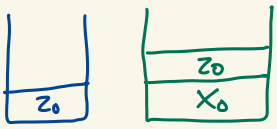
(To be done for every state)

Seeing X_0 → stack empty originally

Given PDA A_1,

✓ ① Does ∃ PDA A_2 s.t. $L(A_1) = N(A_2)$?

✓ ② Does ∃ PDA A_3 s.t $N(A_1) = L(A_3)$?



→ whatever the PDA does, it won't get empty

Acceptance by PDA using final states

For every A ∃ A' ||| For every B ∃ B'

Acceptance by PDA using empty stack

Context-free languages (CFL)

DPDA ⊂ ? NPDA

DPDA: Deterministic PDA

NPDA: Non-deterministic PDA

we can't use subset-construction because of stack

CS 208 : Automata Theory and Logic Spring 2024

Homework 3

Due: 24^{th} Mar, 2024 *Max Marks: 60*

Instructions:

- *Please start writing your* ***solution to each homework problem on a fresh page.***
- *For each homework problem, you must scan your solution and upload a separate PDF file on Moodle. Please check Moodle for detailed instructions on file naming and uploading instructions.*
- *Be brief, complete, and stick to what has been asked.*
- *Untidy presentation of answers, and random ramblings will be penalized by negative marks.*
- *Unless asked for explicitly, you may cite results/proofs covered in class without reproducing them.*
- ***If you need to make any assumptions, state them clearly.***
- ***Do not copy solutions from others. All detected cases of copying will be reported to DADAC with names and roll nos. of all involved. The stakes are high if you get reported to DADAC, so you are strongly advised not to risk this.***

---

1. Many roads to non-regularlity 20 points

In this question, we will look at two different ways of proving languages as non-regular. Consider the following two languages over $\Sigma = \{0, 1\}$:

$L_1 = \{0^i 1^j \mid i, j \in \mathbb{N},\ i + j^2 \text{ is a prime number}\}$. For example, $011, 00111 \in L_1$, but $0011, 0111 \notin L_1$

$L_2 = \{w \mid w \text{ differs from } w^R \text{ in exactly two positions}\}$, where w^R is the reverse of the word w. For example, $1110, 1010111 \in L_2$, but $1010, 10100101 \notin L_2$.

1. *[5 + 5 marks]* Show that L_1 and L_2 are not regular by application of the Pumping Lemma (the version mentioned in Tutorial 4, question 2).
2. *[5 + 5 marks]* Show that L_1 and L_2 are not regular by application of the Myhill-Nerode theorem. In each case, you must provide an infinite set of words (suitably described) such that for every pair of words w_1, w_2 in your set, $w_1 \not\sim_{L_i} w_2$, where $\sim_{L_i}$ is the Nerode equivalence for L_i. You must also indicate a distinguishing word x for each w_1, w_2 above such that $w_1 \cdot x \in L_i$ and $w_2 \cdot x \notin L_i$, or vice versa. Answers without the distinguishing words will fetch no marks.

2. Grammatically speaking ... 20 points

[Please solve this problem after attending Monday's (Mar 18) class.]

Consider the context-free grammar G shown below, where $\{S, A, B\}$ is the set of non-terminals, S is the start symbol and $\{0, 1\}$ is the set of terminals.

$$\begin{aligned} S &\rightarrow AB \mid SS \mid 1S \mid 1 \\ A &\rightarrow 0S \mid 1B1 \mid \varepsilon \\ B &\rightarrow 1S \mid 1B \end{aligned}$$

Let $L(G)$ denote the language generated by G.

1. $L_1 = 0^i 1^j$ $\quad i+j^2 =$ prime

$\exists n$ such that $\quad w \in L_1 \quad w = x.y.z \quad |xy| \le n \quad |y| \ge 1$

$xy^k z \in L$

Consider $w = 0^i 1^j$

where $i > n$

$i+j^2 =$ prime $\Rightarrow w \in L_1$

Therefore, y will have all 0s

$i+(k-1)|y|+j^2$ $\neq$ prime

consider $(k-1) = (i+j^2)$

Then $\quad i' = i+(k-1)|y|$

$j' = j$

$$i'+j'^2 = i+(k-1)|y|+j^2$$
$$= (i+j^2)(|y|+1) \longrightarrow \text{No longer prime}$$

Pumping Lemma not followed $\rightarrow L_1$ not regular

$L_2 =$ w and w^R differ at exactly two places.

does not work

$w = 0^k 01 0^k$
$w^R = 0^k 10 0^k$] differ at exactly two places

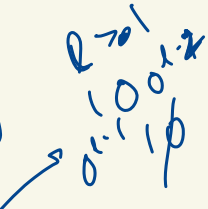
Pumping lemma's n

$w = 0^n 01 0^n$

$w^R = 0^n 10 0^n$

$|xy| \le 0^n$

y has all zeroes and $|y| \ge 1$

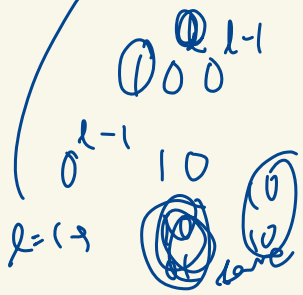


Suppose $|y| = l > 1$

$w' = xz = 0^{n-l} 01 0^n = 0^{n-l} 01 0^l 0^{n-l}$

$(w')^R = 0^n 10 0^{n-l}$

$= 0^{n-l} 0^l 10 0^{n-l}$



$w = 0^n 1$

$w^R = 1 0^n$

~~at two places~~ ~~no benefit~~

$0^p + p \le 0, p 0^p$

$0^p, p 0 \; 0^{p+p \le}$

$w = 0^p 0 1^p 0^p$

$w^R = 0^p 1^p 00^p$ } two places

$xy^{(i+1)}z$ → Consider this

Claim $p > 2$

$i = p$ → can be considered

2. $0^i 1^j \quad i \in j^2 =$ prime

→ Infinite equivalence classes we need to form.

$C_{i,j}$ for every i,j possible.

$C_{i,j} \not\equiv_L C_{i',j'}$ for every i,j

such that

$(i + (j+x)^2)$ prime but

$(i + (j'+x)^2)$ not prime

01 $\quad k=1$ $\quad (1 + (1+x)^2)$

001 $\quad (2 + (1+x)^2)$ ✗

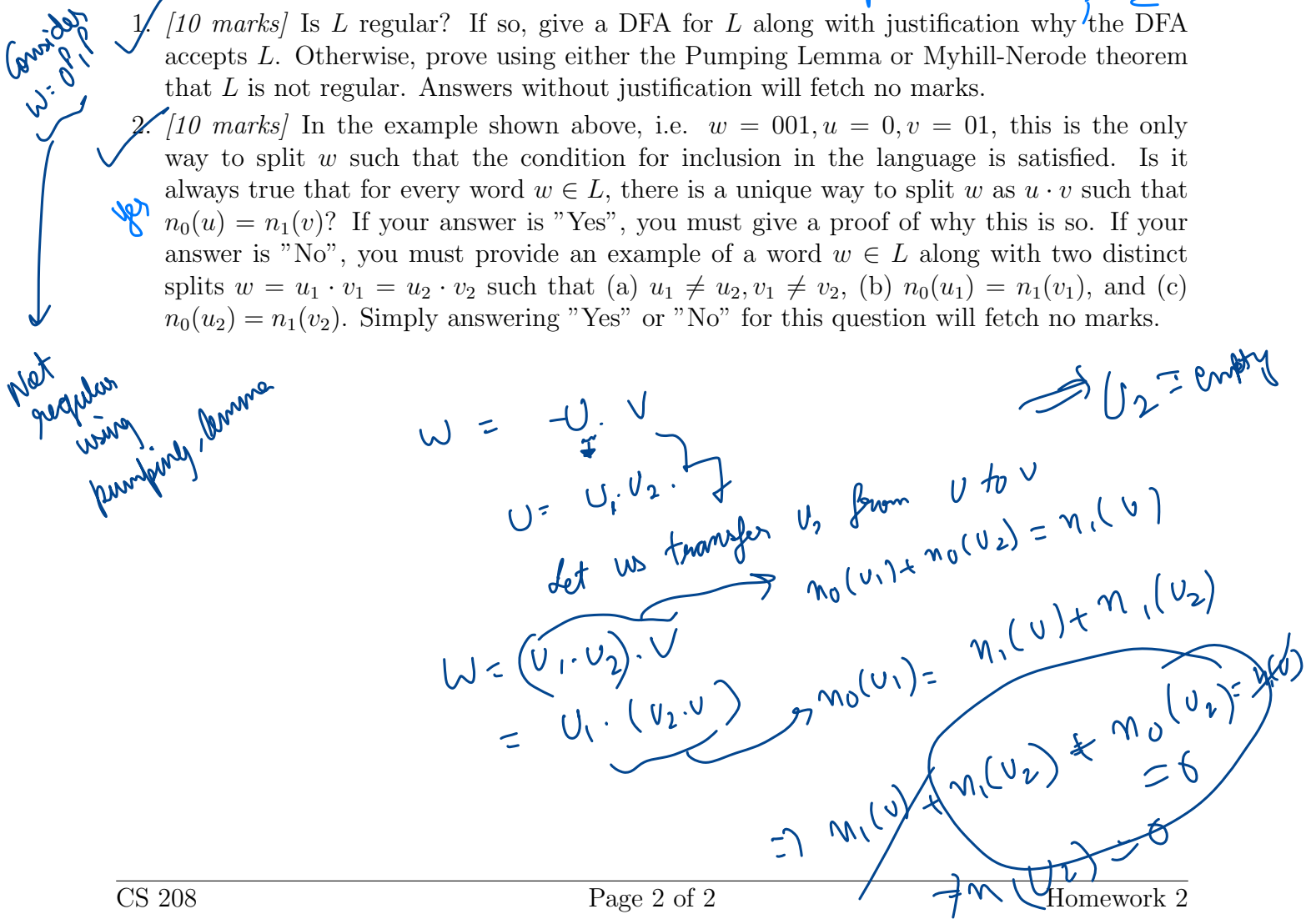
$i \; j$

1. *[2.5+2.5 marks]* Give words $w_1, w_2 \in \{0,1\}^*$ such that (a) each of w_1 and w_2 contains at least one occurrence of 0 and one occurrence of 1, (b) $w_1 \in L(G)$, and (c) $w_2 \notin L(G)$. For w_1, you must give a derivation tree to support your claim that $w_1 \in L(G)$. For w_2, you must give justification why $w_2 \notin L(G)$. Simply giving w_1 and w_2 without justification will fetch no marks.
2. *[5 marks]* Construct another context-free grammar G' that uses only a single non-terminal symbol S and terminal symbols $\{0,1\}$ such that $L(G) = L(G')$. You must provide justification why you think $L(G) \subseteq L(G')$ and $L(G') \subseteq L(G)$.
[Hint: Think of the language generated by each of the non-terminal symbols in the given grammar. Try to see if you can replace some of these non-terminals in the given production rules by something else that refers to only the start symbol and terminal symbols.]
3. *[10 marks]* We wish to construct a PDA with one state and one symbol in the stack alphabet that accepts $L(G)$ *by the empty stack.* If you think this is possible, list the transitions of the PDA assuming the unique state is q and the unique stack symbol is X (also the initial symbol on stack). In this case, you must also give justification why your PDA, say P, accepts the same language as $L(G)$, i.e. $N(P) \subseteq L(G)$ and $L(G) \subseteq N(P)$. If you think this is not possible, give justification in support of your answer. Feel free to use non-deterministic PDAs for this question.
[Hint: Think of the grammar with one non-terminal symbol that you constructed in part (2) of the question. Think also about the relation between count of 0*s and count of* 1*s in words in* $L(G)$*.]*

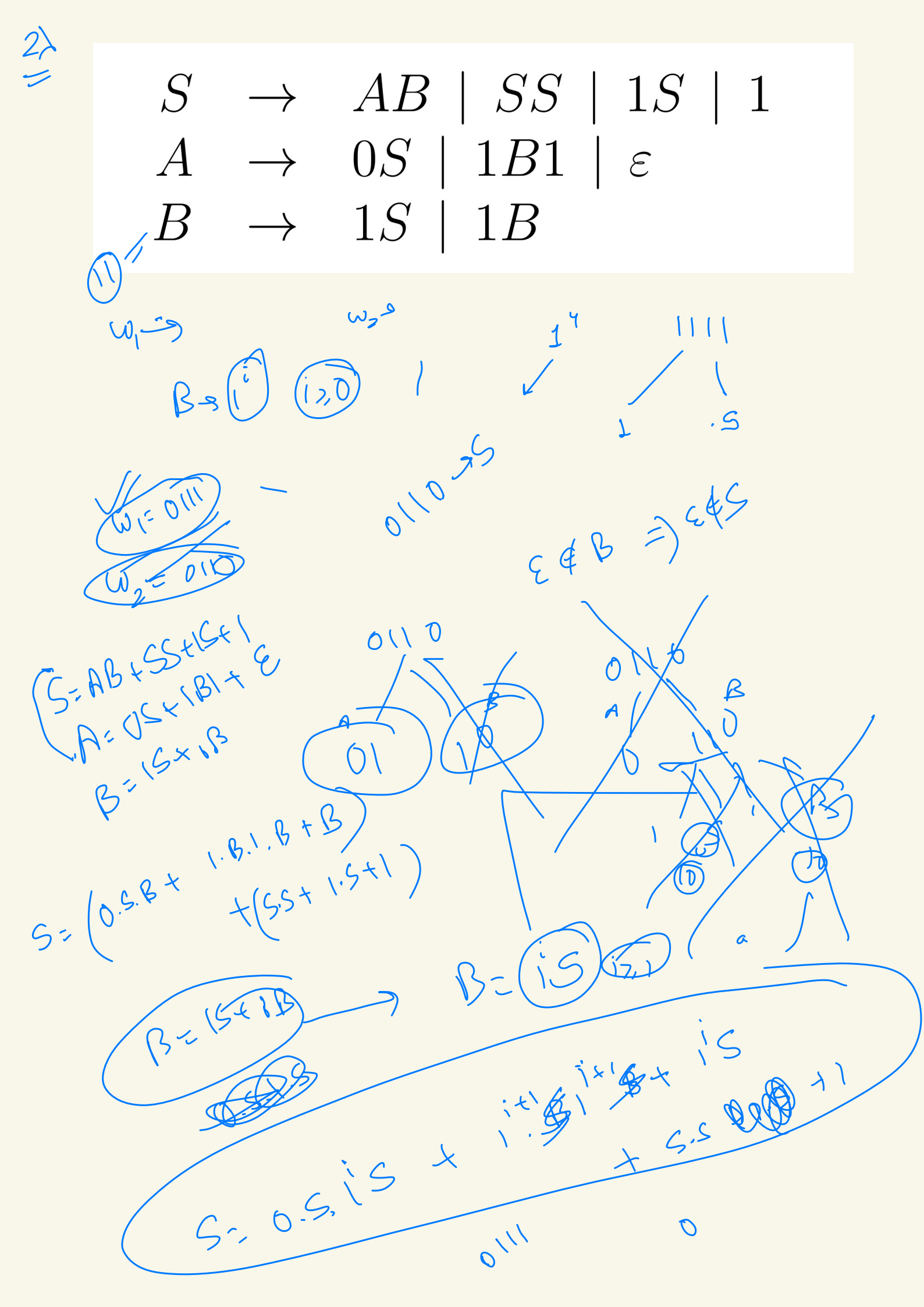
3. To be or not to be regular — 20 points

Let $\Sigma = \{0,1\}$. Consider the language $L = \{w \in \Sigma^* \mid \exists u, v \in \Sigma^*,\ w = u \cdot v \text{ and } n_0(u) = n_1(v)\}$, where $n_i(x)$ denotes the count of times the letter i appears in the word x. For example, $001 \in L$ because we can write $001 = 0 \cdot 01$ and $n_0(0) = 1 = n_1(01)$.

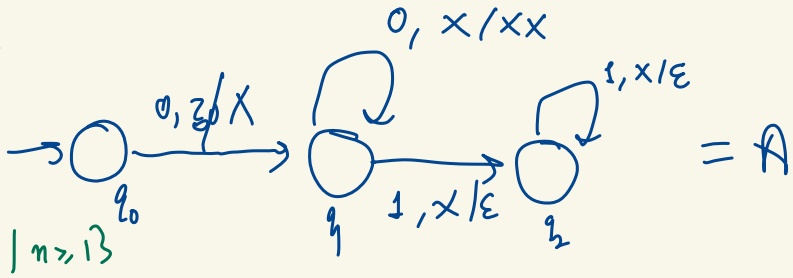
1. *[10 marks]* Is L regular? If so, give a DFA for L along with justification why the DFA accepts L. Otherwise, prove using either the Pumping Lemma or Myhill-Nerode theorem that L is not regular. Answers without justification will fetch no marks.
2. *[10 marks]* In the example shown above, i.e. $w = 001, u = 0, v = 01$, this is the only way to split w such that the condition for inclusion in the language is satisfied. Is it always true that for every word $w \in L$, there is a unique way to split w as $u \cdot v$ such that $n_0(u) = n_1(v)$? If your answer is "Yes", you must give a proof of why this is so. If your answer is "No", you must provide an example of a word $w \in L$ along with two distinct splits $w = u_1 \cdot v_1 = u_2 \cdot v_2$ such that (a) $u_1 \neq u_2, v_1 \neq v_2$, (b) $n_0(u_1) = n_1(v_1)$, and (c) $n_0(u_2) = n_1(v_2)$. Simply answering "Yes" or "No" for this question will fetch no marks.



$$
\begin{aligned}
S &\rightarrow AB \mid SS \mid 1S \mid 1 \\
A &\rightarrow 0S \mid 1B1 \mid \varepsilon \\
B &\rightarrow 1S \mid 1B
\end{aligned}
$$

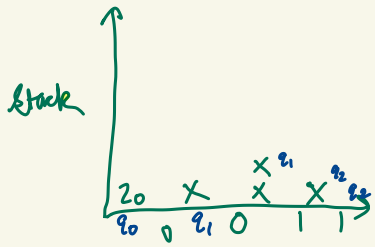


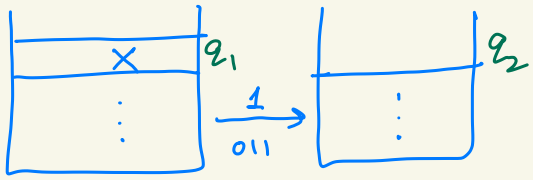
$\Gamma = \{Z_0, X\}$



$= A$

$N(A) = \{0^n 1^n \mid n \geq 1\}$





00111

can be replaced by 011

00 (011) 1

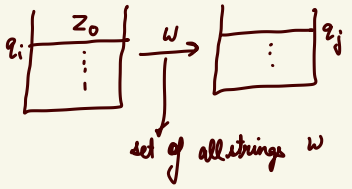
$1 \in L_{q_1 X q_2}$

$011 \in L_{q_1 X q_2}$

$00111 \in L_{q_2 Z_0 q_2}$

$1 \in L_{q_2 X q_2}$

$L_{q_i Z_0 q_j}$ = $\{w \mid$



000111

$Z_0 \xrightarrow{0} X \xrightarrow{0} XX \xrightarrow{0} XXX \xrightarrow{1} XX \xrightarrow{1} X \xrightarrow{1} -$

(q_1) (q_2) (q_2) (q_2)

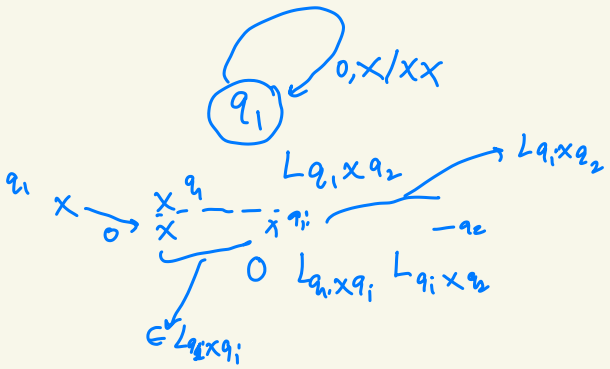
$L_{q_0 Z_0 q_2}$ → Language accepted by N(A)

$\cup$

$L_{q_0 Z_0 q_1}$

$\cup$

$L_{q_0 Z_0 q_0}$



$L_{q_1 X q_2} = 0 . L_{q_1 X q_1} . L_{q_1 X q_2}$

$\cup$

$0 L_{q_1 X q_1} . L_{q_1 X q_2}$

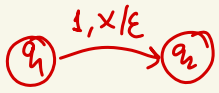
$\cup$

$0 L_{q_1 X q_2} . L_{q_2 X q_2}$

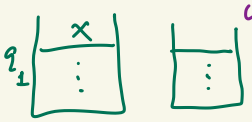
n states, k stack symbols

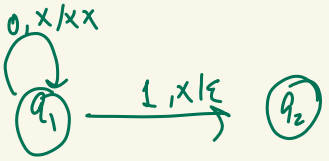
$n^2 k$ = number of such languages

The recurrence relations capture what the PDA is using.



$L_{q_1 X q_2}$





$L_{q_1 X q_2} \Rightarrow 1$

$L_{q_1 X q_2}$

$= 0 . L_{q_1}$

$L_1 \rightarrow 1 \mid 0 . L_2 . L_3 \mid 0 . L_4 . L_1$

$L_2 \rightarrow \cdots$

$\vdots$

$L_{n^2 k} \rightarrow \cdots$

Context free grammars as operation

$L_1 \rightarrow 01 \mid 0 . L_1 . 1$

solution for $L_1 = 0^n 1^n \quad n \geq 1$

$\Sigma^* = \{0,1\}^*$ also works

smallest / best solution

largest solution

$L_1 \supseteq \{1\}$

$\supseteq \{0 . L_2 . L_3\}$

$\supseteq \{0 . L_4 . L_1\}$

$\vdots$

$S \to A.S \mid \varepsilon$

$A \to A1 \mid 0A1 \mid 01$

Non-terminals: S, A

Terminals: 0, 1, ε

Production rules
$S \to A.S$
$S \to \varepsilon$
$A \to A.1, \ldots$

Start symbol: S

Context-free grammar (CFG)

Context-free languages (S and A) (CFL)

$A \to$ 01, 011, 00111, 0011 ... $\in L$ corresponding to A.

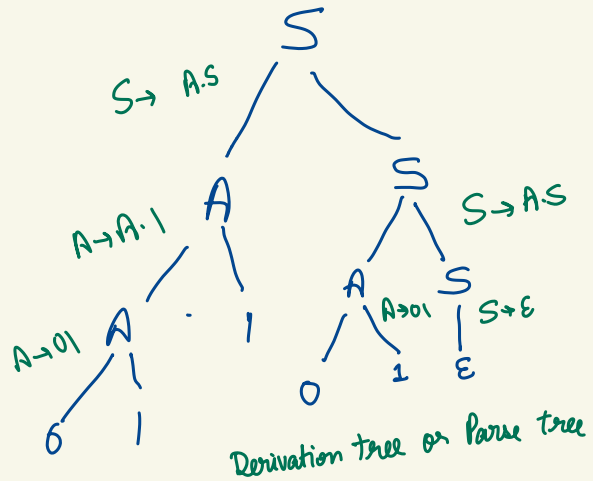
$0^i 1^k \quad k \geq i$

ε, 01, 01101, 011, 0011, 01101 ... $\in L$ corresponding to S

$S = A^*$

$G = (V, \Sigma \cup \{\varepsilon\}, P, S)$

V: non terminals; $\Sigma \cup \{\varepsilon\}$: Terminals; P: ∏; S



Derivation tree or Parse tree

Inherently ambiguous language
↳ have string with ≥2 parse trees for every grammar

Ambiguous grammars

Different trees → same string

$A \to 0A1 \mid 01 \mid AB$
$B \to 1B \mid \varepsilon$

different grammar for the same language

$S \to C.S \mid \varepsilon$
$C \to A.B$
$A \to 0A1 \mid 01$
$B \to 1B \mid \varepsilon$

unambiguous grammar

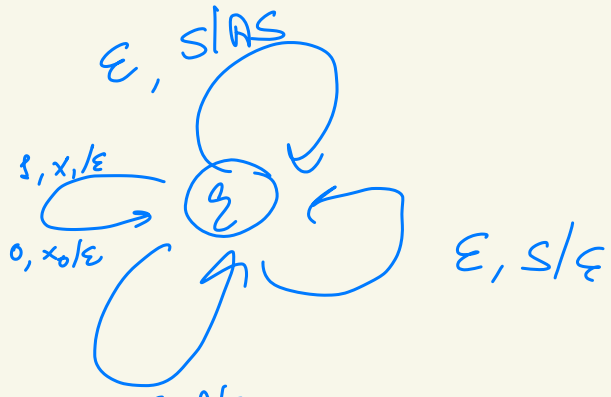
PDA ≡ CFG

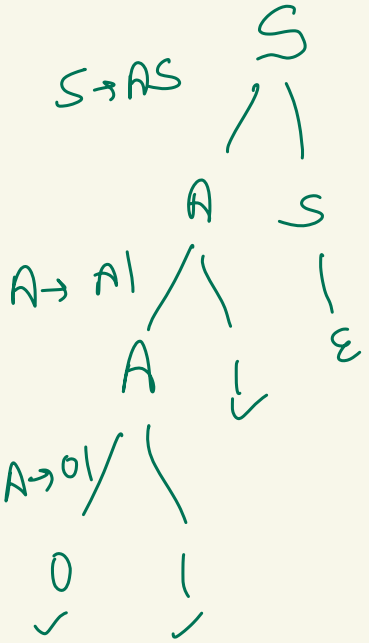
$\Sigma = \{0, 1\}$

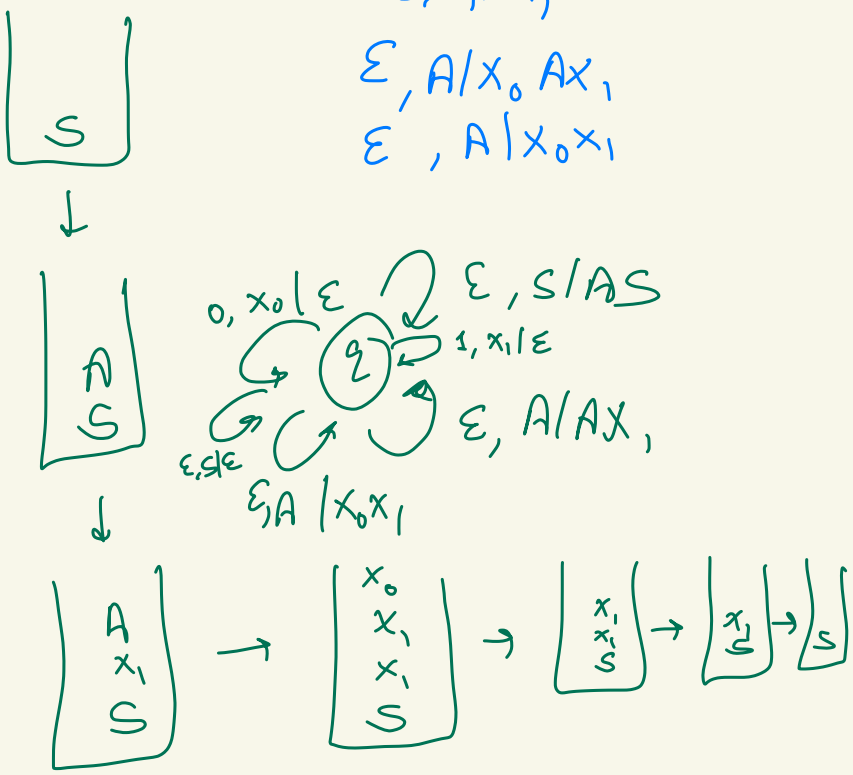
$\Gamma = \{S, A, X_0, X_1\}$

$S \to A \cdot S \mid \varepsilon$

$A \to A1 \mid 0A1 \mid 01$







start symbol

$S \to ABS \mid 0A1 \mid 01$ (0S1, 0D1) → remove

$A \to 01 \mid 1A0 \mid \varepsilon \mid 10 \mid S$ (1S0, 1D0) — Now no need for this

$B \to 1B \mid BB0$ → no way to derive terminal string from B

$C \to A0 \mid 01$ → useless

$D \to S \mid 0AD$

⇒ useless

⇒ Any rule using B has to go

Arbitrary CFG → preprocessing → Remove useless symbols/rules.

$S \rightarrow 0A1 \mid 01$ (0S1, 0D1)

$A \rightarrow 011 \mid 1AD \mid 10$ (1DD, 1SD)

$D \rightarrow S \mid 0AD \mid 0AS$

$S \rightarrow X_0SX_1 \mid X_0DX_1 \mid X_0AX_1 \mid X_0X_1$

$X_0 \rightarrow 0$

$X_1 \rightarrow 1$

$A \rightarrow X_0X_1 \mid X_1AX_0 \mid X_1DX_0 \mid X_1SX_0 \mid X_1X_0$

$D \rightarrow X_0AD \mid X_0AS$

$D \rightarrow X_0D_1$

$D_1 \rightarrow AS$

$S \rightarrow X_0S_1$

$S_1 \rightarrow SX_1$

Chomsky Normal Form (CNF)

- No useless symbols
- No ε production
- No unit production
- All prod rules

$X \rightarrow a$

$X \rightarrow Y$

$A \rightarrow BC$

$D \rightarrow 0$

CNF — G ---- n non-terminals

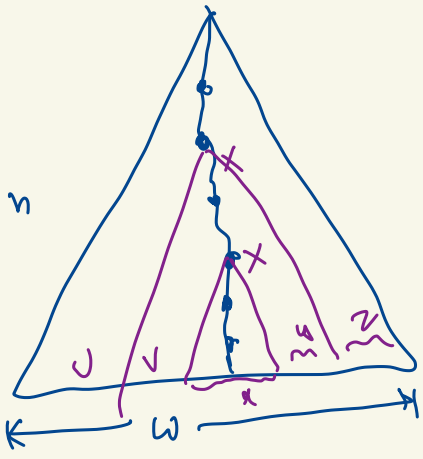
Binary tree → see chomsky conditions

n-non terminals

but path length $> n$

↓

some non-terminal repeating



$w = u.v.x.y.z$

$|w| > 2^n$

$u.v^i.x.y^i z \longrightarrow$ Also in the language

(Like pumping Lemma)

PL for CFLs

$|vxy| \leq 2^n$

$|x| \geq 1$

Every CFL $\longrightarrow$ PL

Not PL $\longrightarrow$ not CFL

$L = \{ 0^n 1^n 2^n \mid n \geq 0 \}$ $\quad \Sigma = \{0, 1,$

$0^{2k} 1^{2k} 2^{2k}$ $\quad k = 2^{\#NT \text{ in CFG grammar}}$

Tutorial 6: Context-Free Tutorial

---

1. Solve the following problems. Assume the alphabet to be $\Sigma = \{a, b, c\}$

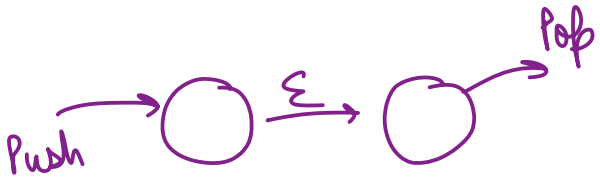
   (a) Consider the language of non-palindromes (words that are not palindromes). A palindrome is a word that spells the same way forward and backwards. For example, 'abcba' is a palindrome but 'abbaa' is not). Describe a PDA and a CFG for the language.

   (b) Consider the language $\{w \mid w \neq uu \text{ for any } u \in \Sigma^*\}$. Describe a PDA and a CFG for the language.

The outer letters are same → induct forward
different → we are done

$S \to aSa \mid bSb \mid cSc \mid aTb \mid bTa \mid aTc \mid cTa \mid bTc \mid cTb$

$T \to Ta \mid Tb \mid Tc$



$w = a_1 a_2 a_3 \dots a_n$

first odd letter detection

$S \to aSa \mid bSb \mid \dots \mid a \mid b \mid c$

or maybe: $A \to bAb \dots \mid a$

$B \to \mid b$

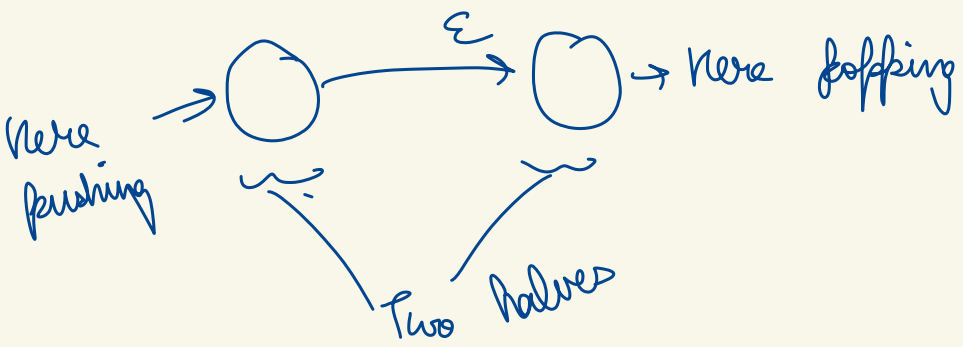
$C \to \mid c$

1. (a)

$S \to aTa \mid bTc \mid cTa \mid aTc \mid bTa \mid cTb$

$aSa \mid bSb \mid cSc$

$T \to Ta \mid Tb \mid Tc \mid \varepsilon$



(b) $|w| =$ odd $\quad w \in L$.

$S'_{odd} \to$

CFG with odd words with a at centre

$A \to aAb \mid aAa \mid bAc \mid a$ } a in the middle

$B \to b$ } b in the middle

$C \to c$ } c in the middle

$|w| =$ even $= 2n$

$a_1 a_2 \ldots a_n \mid a_{n+1} \ldots a_{2n}$

$\exists k \quad a_k \neq a_{n+k}$

$a_1 a_2 \ldots a_k a_{k+1} \ldots a_{2k-1}$

$a_{2k} \ldots a_{n+k} \ldots a_{2n}$

$S \to AB \mid AC$

$BC \mid a$

$CB \mid BA$

$A \mid BC$

2. Determine if the following languages are context-free or not. If yes, provide a CFG and PDA for the same, else prove, using the Pumping Lemma, that they are not Context Free

   (a) $L_1 = \{w \mid w = uu \text{ for any } u \in \Sigma^*\}$

   (b) $L_2 = \{0^p | \text{p is prime}\}$

   For more on L_2, look at the takeaway problems

If L is context free then $\exists p \in \mathbb{N}$ s.t. for all $x \in L$

$\exists\ u, v, x, y, z \in \Sigma^*$ s.t. v and y are both non-empty and $|vxy| \leq p$ and $t = uxyz$ and $\forall k \in \mathbb{N}$

$uv^k x y^k z \in L$

(a) $0^n 1^n 0^n 1^n$ for any $k > p$

(b) $L = \{0^q \mid q \text{ is prime}\}$

take $q > p$

$|uv^k xy^k z| = |u| + k|y| +$

$= \text{in AP}$ - .

↳ one of them = composite

3. **Deterministic Context-Free Languages**

We know that Context-Free Languages (CFL) are accepted by Push-Down Automata (NPDA), where we had allowed non-determinism in PDA transitions. In this question, we will explore Deterministic Push-Down Automata (DPDA). Recall that a (not necessarily deterministic) PDA M can be defined as a 7-tuple:

$$M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$$

h where Q is a finite set of states. Σ is a finite set of input symbols. Γ is a finite set of stack symbols. $q_0 \in Q$ is the start state. $Z_0 \in \Gamma$ is the starting stack symbol. $A \subseteq Q$, where A is the set of accepting, or final, states δ is a transition function, where $\delta : (Q \times (\Sigma \cup \varepsilon) \times \Gamma) \to \mathcal{P}(Q \times \Gamma^*)$ Here, $\mathcal{P}(X)$ is the power set of a set X, and ε denotes the empty string. We say that M is a deterministic PDA (or DPDA) if it satisfies both the following conditions:

For any $q \in Q, a \in \Sigma \cup \varepsilon, x \in \Gamma$, the set $\delta(q, a, x)$ has at most one element.

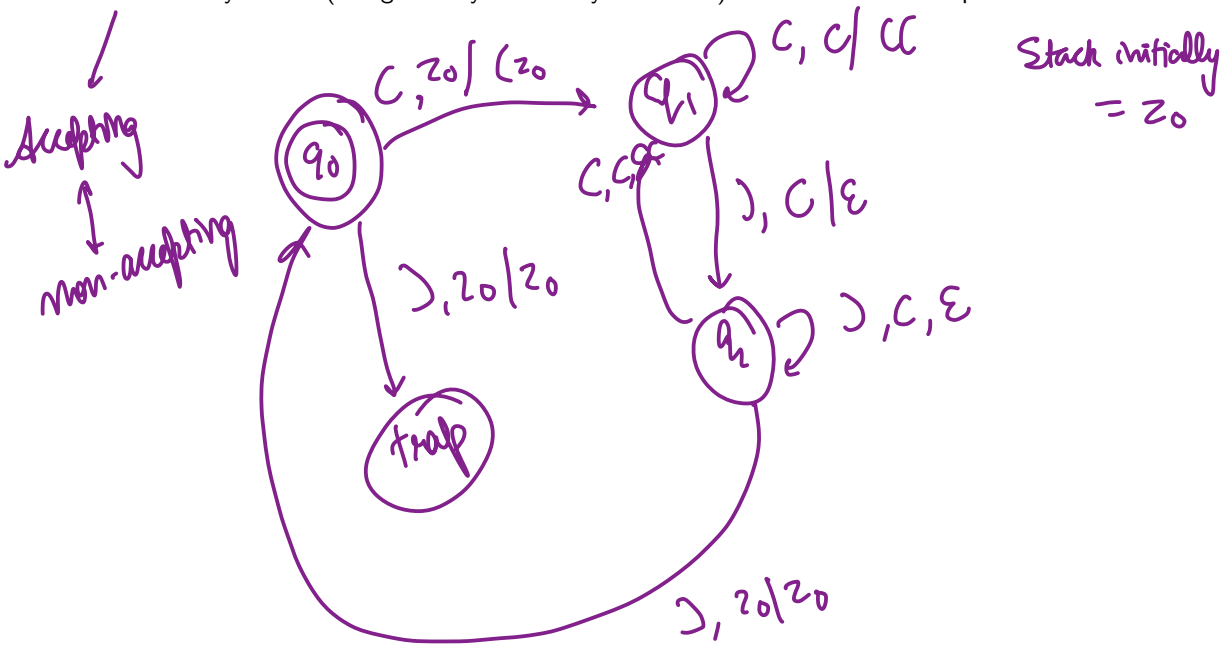
For any $q \in Q, x \in \Gamma$, if $\delta(q, \varepsilon, x) \neq \emptyset$, then $\delta(q, a, x) = \emptyset$ for every $a \in \Sigma$.

We call the languages accepted by DPDAs as DCFLs (Deterministic Context Free Languages). We have studied in class that PDAs can accept by *empty stack* or by *final state*, and that these provide equivalent accepting power. Interestingly, this is not so for DPDAs, so we need to make up our mind about which acceptance criterion to use. For purposes of this question, we will use acceptance by *final state*. We investigate acceptance by empty stack in a takeaway question at the end of this tutorial.

1. Consider the language of balanced parentheses. A string of parentheses is balanced if the number of opening parentheses in any proper prefix is at least as much as the number of closing parenthesis in the same prefix. Also, the total number of opening and closing parenthesis in the entire string must be equal.

   Draw a DPDA (accepting by final state) that accepts the language of balanced parentheses strings.

2. Construct a deterministic PDA for the complement of the above language. Does this give you an idea why DCFLs (recognized by DPDAs by final state) are closed under complementation?



4. **Takeaway: The Curious Case of the Unary Alphabet** Prove that any language over a unary alphabet (the alphabet has exactly one element) is context-free if and only if it is regular.

5. **Takeaway: Expressions in Intermediate Code**
   Consider the following language for expressions in some (familiar) programming languages

   $\langle expr \rangle$ ::= $\langle term \rangle$ '+' $\langle term \rangle$
   | $\langle term \rangle$

   $\langle term \rangle$ ::= $\langle factor \rangle$ '*' $\langle factor \rangle$
   | $\langle factor \rangle$

   $\langle factor \rangle$ ::= '(' $\langle expr \rangle$ ')'
   | $\langle number \rangle$

   $\langle number \rangle$ ::= [0-1]+

   Now, though expressions can be a sum of as many terms, it is essential, during an intermediate step of compilation, that every expression must be a sum of at most two terms and every term must be a product of at most two terms. Draw a Deterministic PDA (Definition provided in an earlier question) to recognise strings which are of the form as stated above. Note that the alphabet is $\Sigma = \{0, 1, (,), *, +\}$.

6. **Takeaway: Null-stack DPDAs**

   In Problem 3, we used acceptance by final state for a DPDA. Let's see what happens if we now allow a DPDA to accept by emptying its stack (regardless of which state it is in, when the stack becomes empty). We will call such a DPDA a *null-stack DPDA*, i.e. it's a DPDA just like we had earlier, but it accepts by emptying its stack.

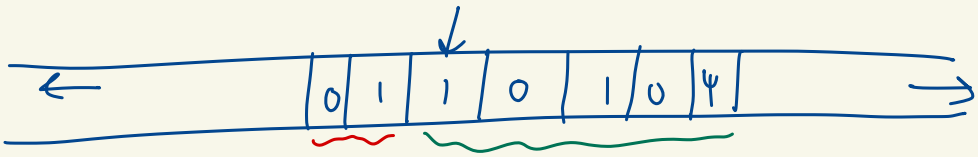
   1. Prove that no null-stack DPDA can accept the language of balanced parentheses. Recall that a string of parentheses is balanced if the number of opening parenthesis in any prefix of the string is at least as much as the number of closing parenthesis in the same prefix, and the total number of opening and closing parenthesis ar equal.

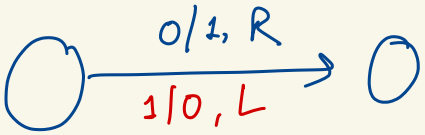
      [Hint:] Can a null-stack DPDA accept two strings u and $u.v$, where one is a proper prefix of the other?

      Note: This is quite damaging news in the DPDA world, since we saw in Problem 3 that the language of balanced parentheses can be accepted by a DPDA accepting by final state. The above proof should now convince you that unlike normal PDAs, acceptance by final state and acceptance by empty stack are not equally powerful in the DPDA world.

   2. A string is said to be *minimally balanced parentheses* if the number of opening parenthesis in any proper prefix is strictly more than the number of closing parenthesis in the same prefix, and the total number of opening and closing parenthesis in the entire string are equal. Thus, $(()())$ is a minimally balanced parentheses string, but $()()$ and ε are not. Any string of balanced parentheses can be written as either ε or a concatenation of a finite number of minimally balanced parentheses strings. Show that for every given value of $k > 0$, we can construct a null-stack DPDA that accepts the language of balanced parentheses strings containing exactly k minimal valid parentheses substrings. Can we construct a null-stack DPDA if we want to accept the language of balanced parentheses containing upto (instead of exactly) k minimally valid parentheses substrings?

TURING MACHINE





Config:

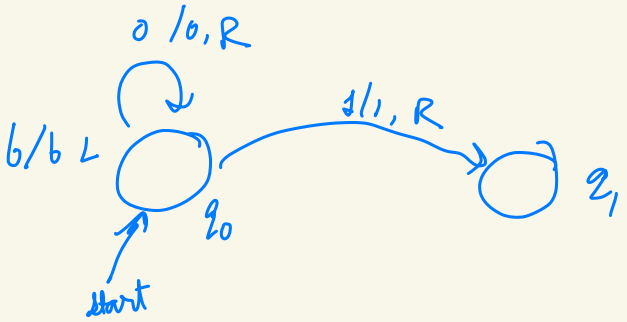
blank

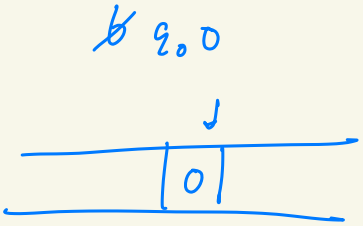
01 q_5 1016 ₿0

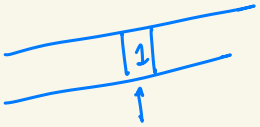
0 q_5 0010₿0

$\alpha_0 q_0 \beta_0 \vdash \alpha_1 q_1 \beta_1 \vdash \alpha_2 q_2 \beta_2$

$\alpha_0 q_0 \beta_0 \vdash^{*} \alpha_i$ done β_i

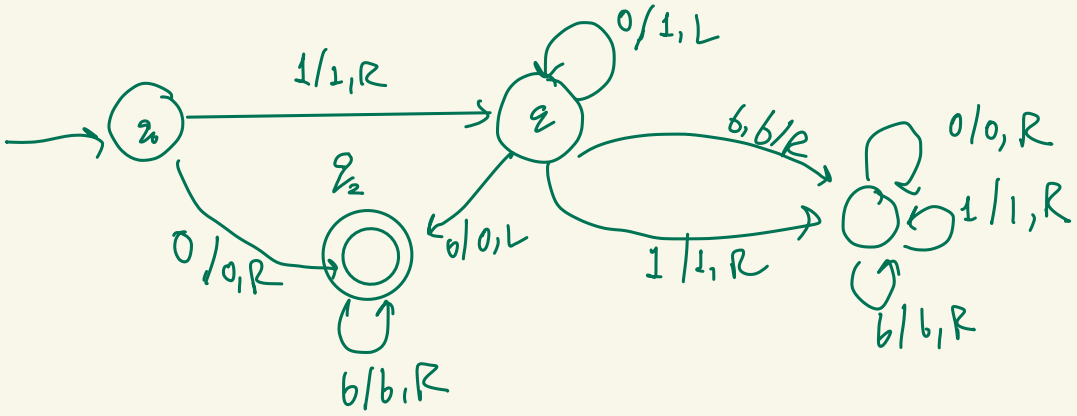






1 q_0 ₿

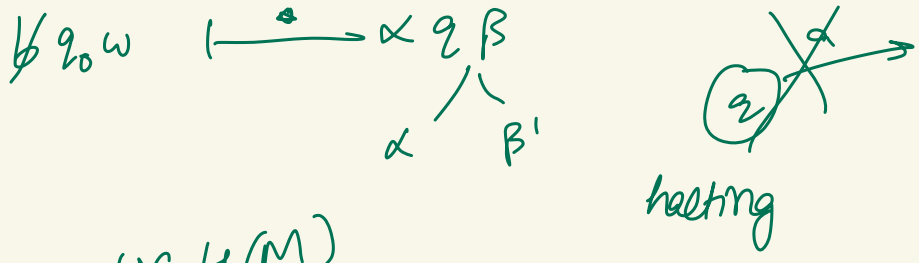
Acceptance of strings by final state
Acceptance of strings by halting state



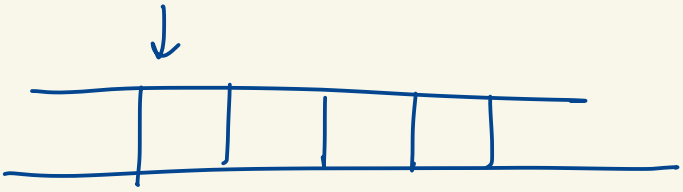
₿ q_0 w $\vdash$ $\alpha q_f \beta$

$w \in L(m)$

m = Turing Machine

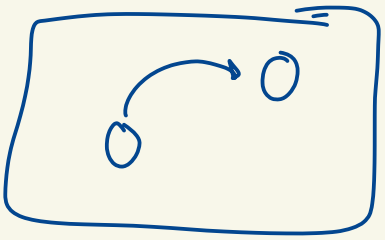


$w \in H(M)$

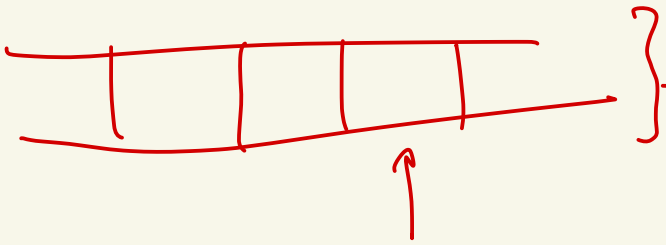


M = turing machine

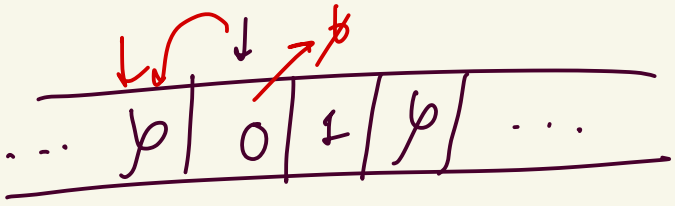
$L(M)$
$= \{ w \mid M$ enters a final state while processing $w \}$



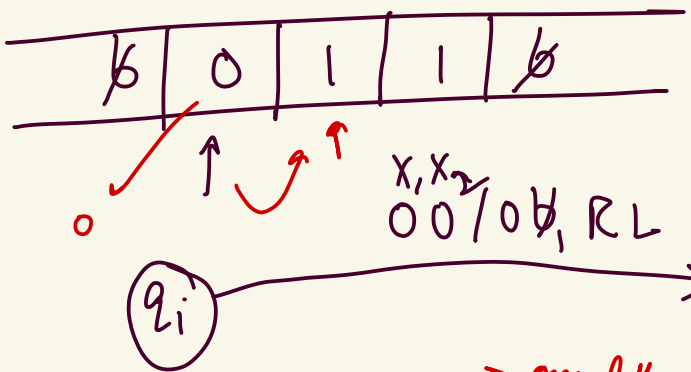
$H(M) = \{ w \mid M$ halts while processing $w \}$

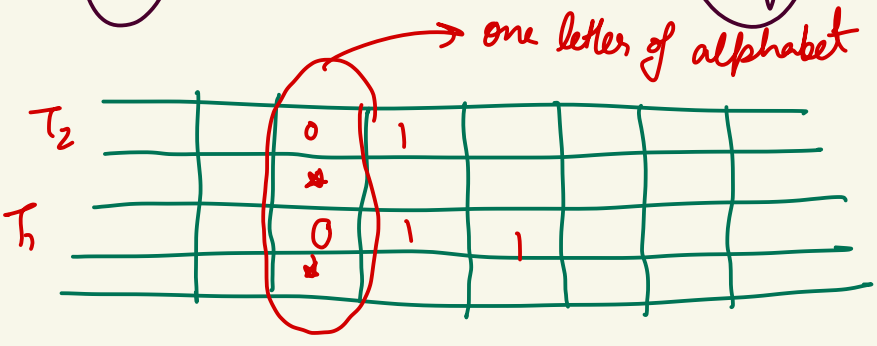


→ Exactly the same power as having a single tape.

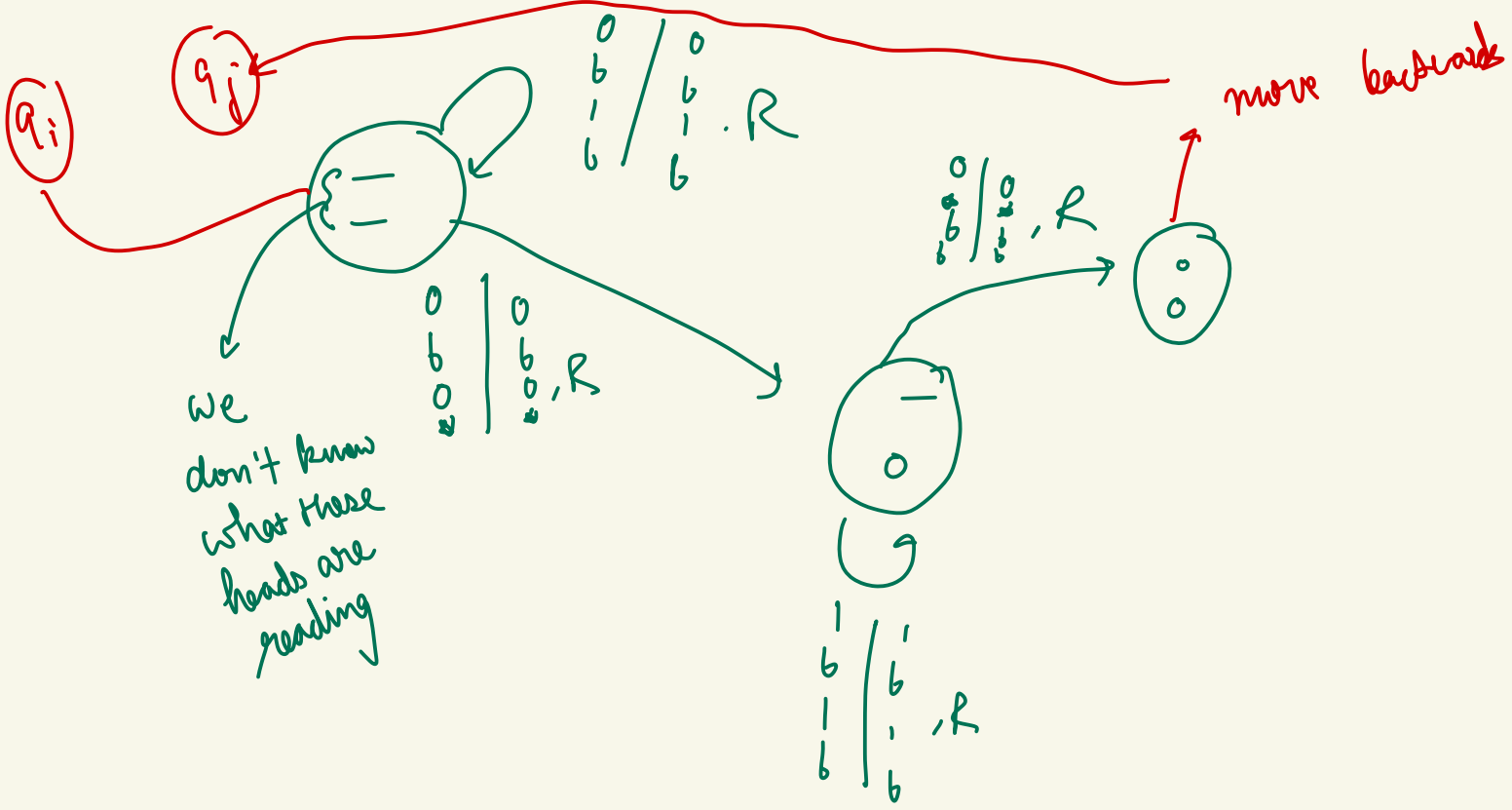


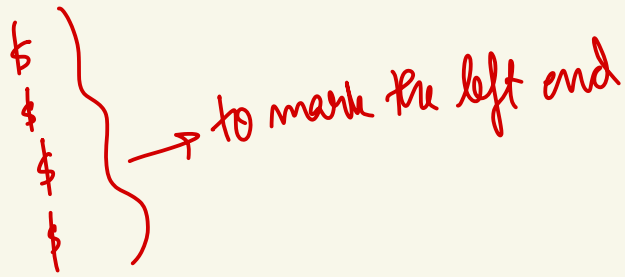
T_2

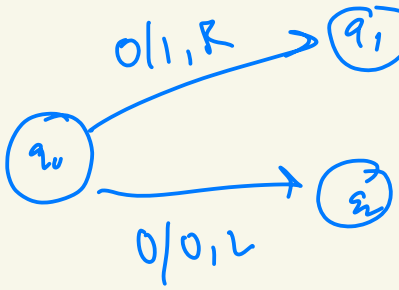


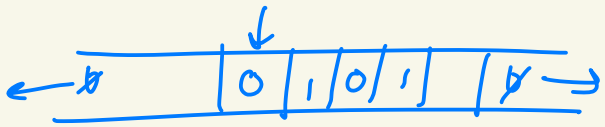


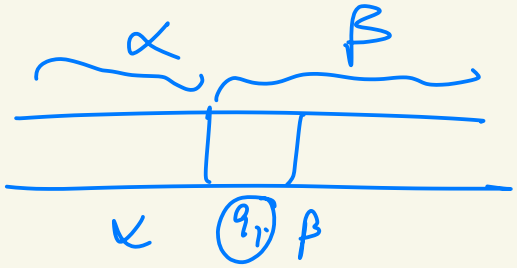
* = head



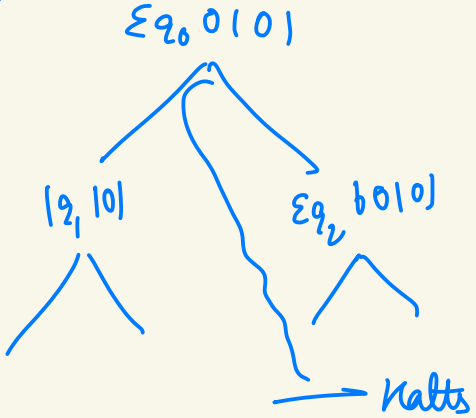






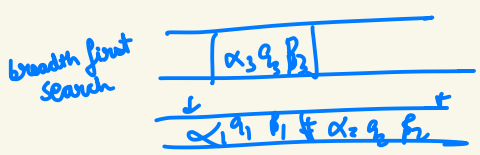


non-deterministic



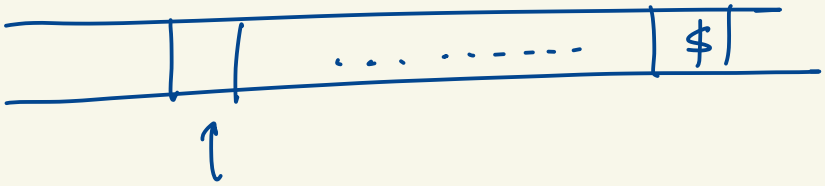
If halts in one of the configuration → then accept.

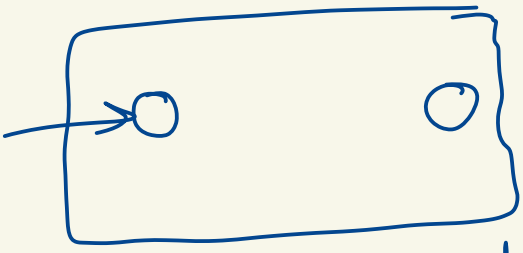
Deterministic Turing Machine (2 tapes)



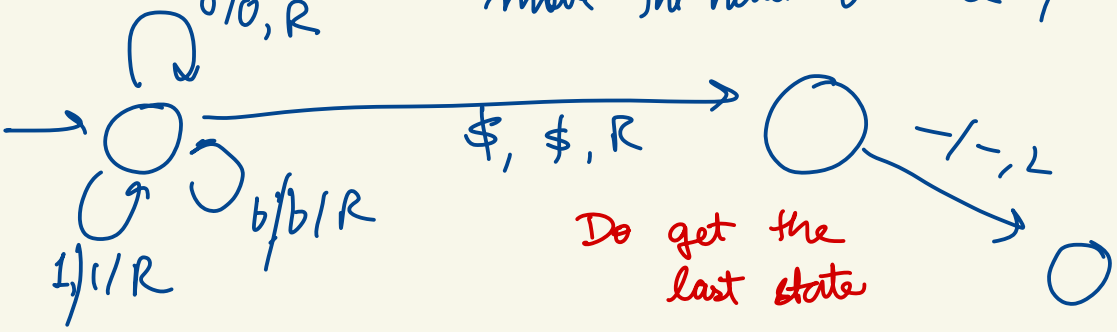
NTM → DTM

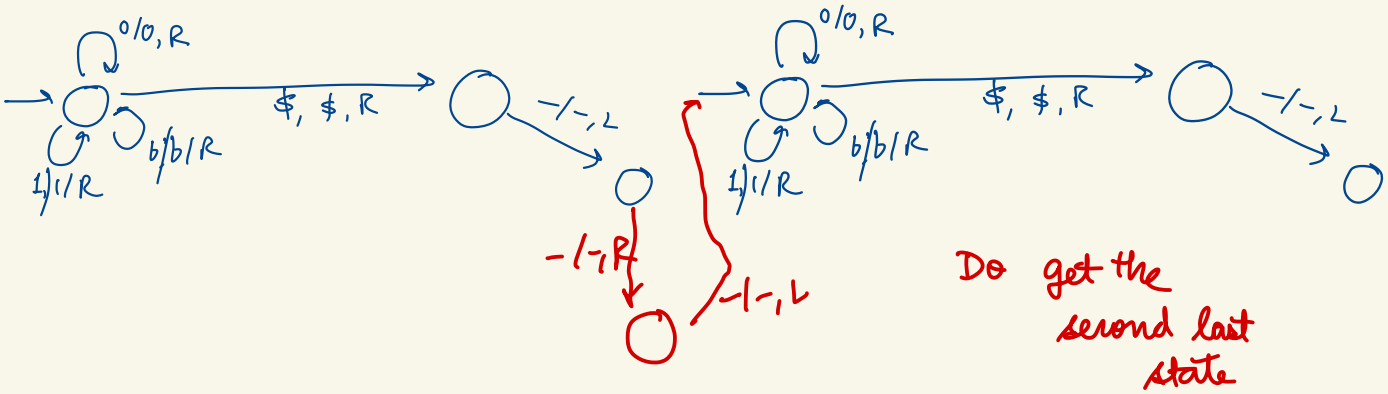
Non-deterministic → Deterministic

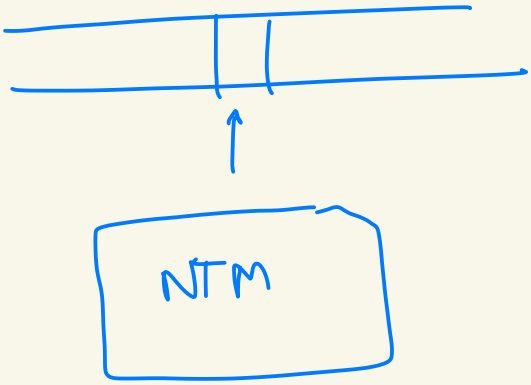


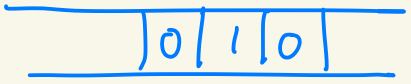


move the head right till $



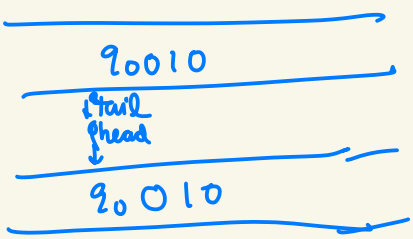




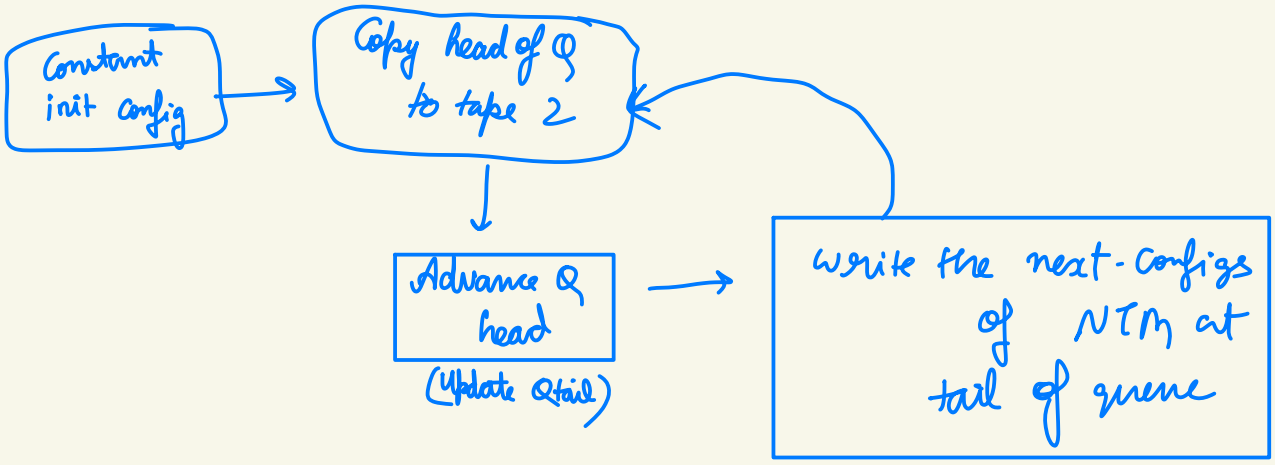


Current config of NTM

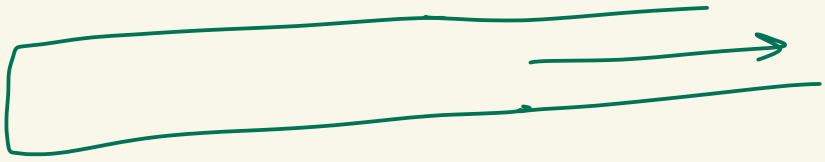
Queue of configs of NTM



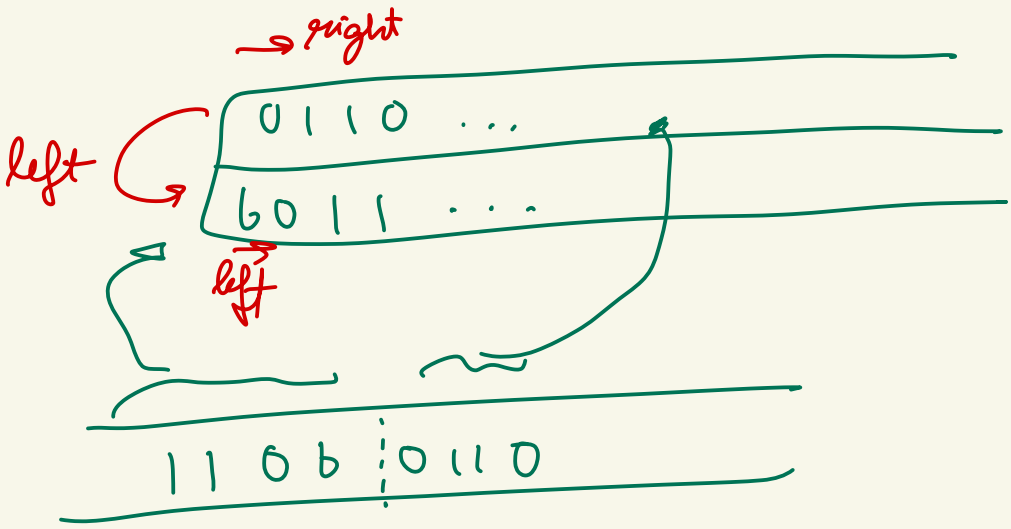
DTM



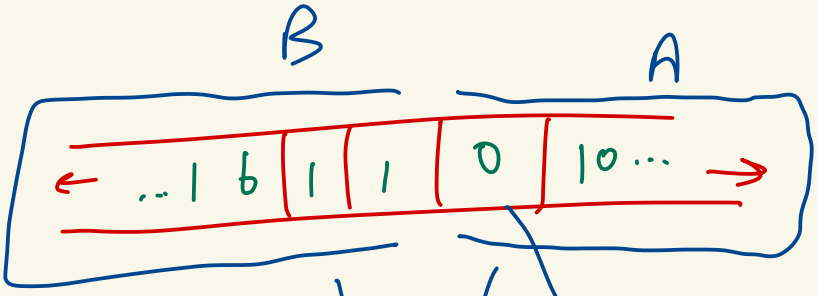
Single-sided Tape

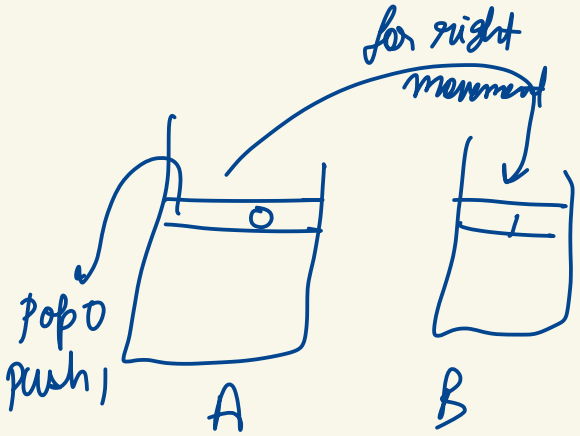


state = upper track/ lower track



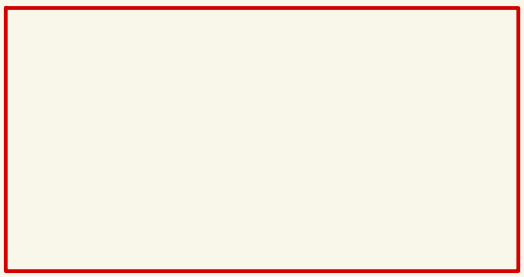
Single sided tape same power

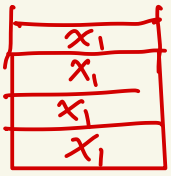




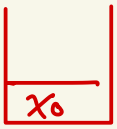
stacks change to 1

Two stacks == Turing Machine

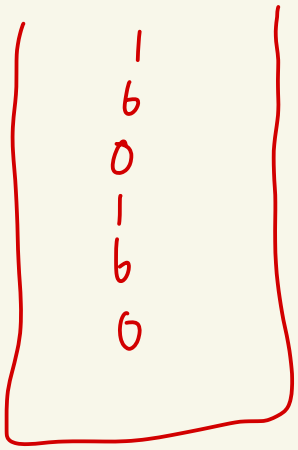


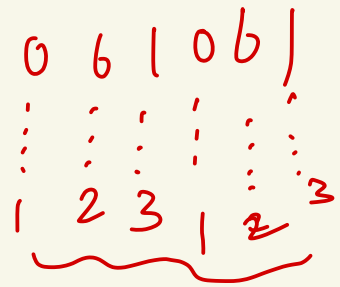


C_1



C_2





numbers in base 4
uniquely defines
what this there
on stack

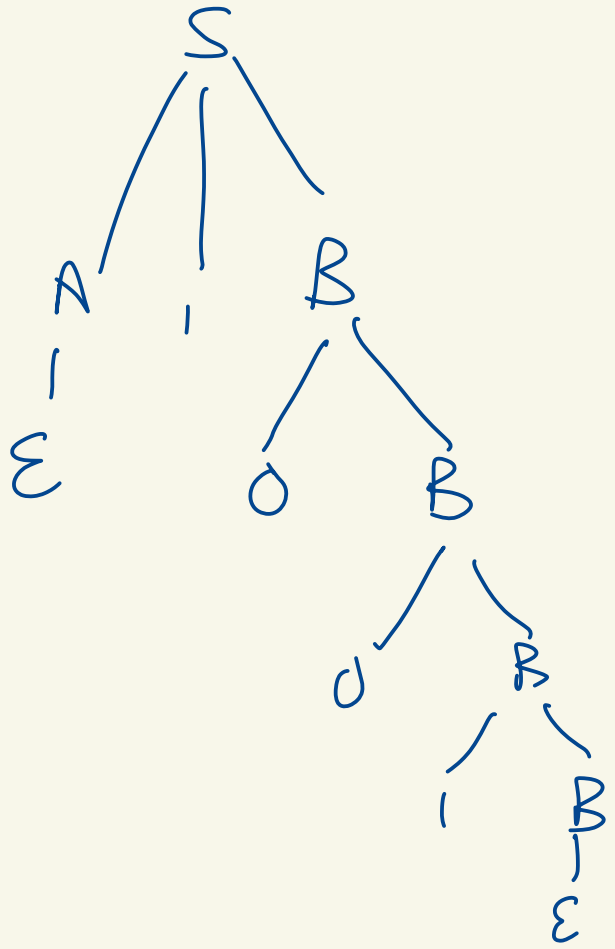
push pop

right & left shift

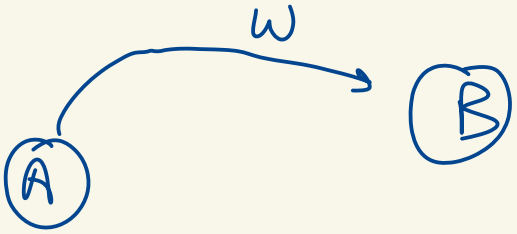
multiplication & division
↓
repeated addition
& subtraction

Practice Problem Set - 4

5.1-2 (b) 1001



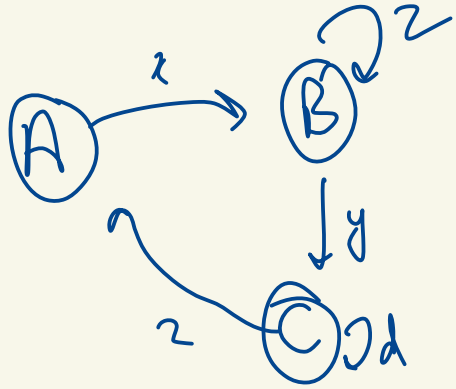
5.1.4 (a) Every non-terminal → represented by a state



If w has more than one letters then have multiple intermediate states

S → accepting state

(b)



$A \to xB$

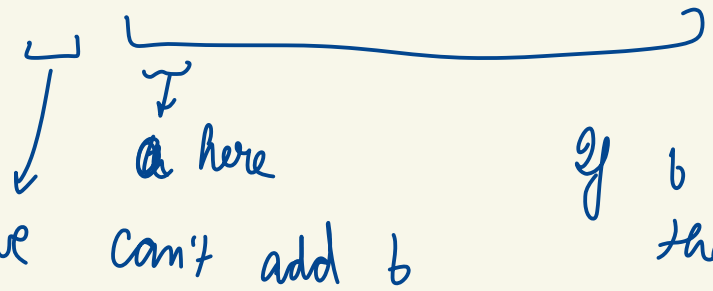
$B \to yC \mid zB$

$S \to F_1 \mid F_2 \mid F_3$

Accepting state : S

5.1.7 (a) Induction length 2 : not possible
(cannot start with b)

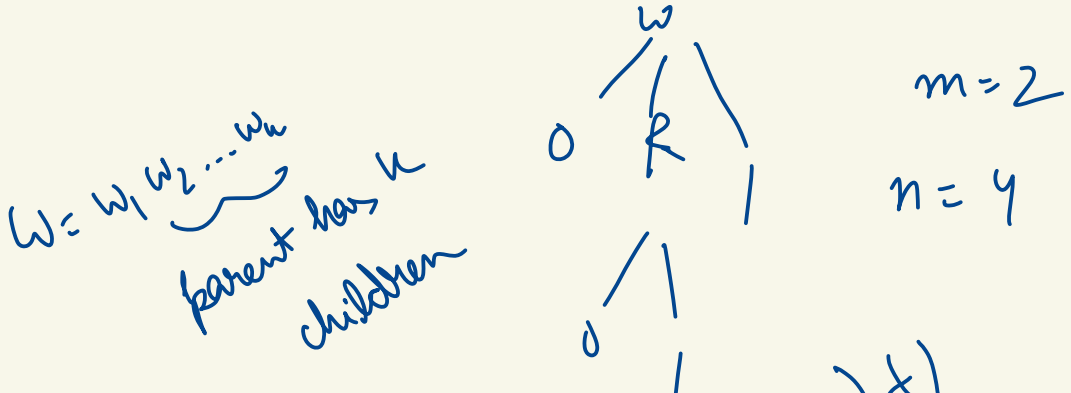
length $(k-1)$ not possible



(b) $a^* b^*$

5.2.2 $\text{len}(w) = n$ n steps in derivation

number of nodes $= n+m$



$\text{numnodes}(w) = n_{\text{nodes}}(w_1) + \ldots + n_{\text{nodes}}(w_k) + 1$

↓ gets added after every step

at the end $n =$ number of terminals

$= m+n$

5.2.3

ε at some places

number of derivation steps $= m$ (given)

maximum alternate ε

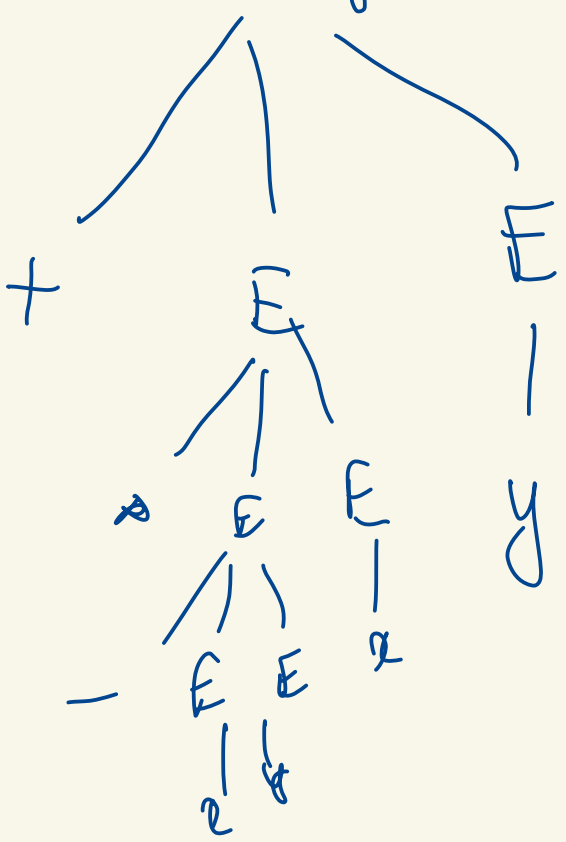
$$w \overset{\varepsilon+}{=} w_1 \overset{\varepsilon+}{+_\wedge} w_2 \overset{\varepsilon+}{+_\wedge} w_3 \cdots + w_n + \varepsilon$$

$$f(w) \Leftarrow f(w_1) + \cdots + f(w_k) + 1 + (k+1)$$

every w_i should lead to some symbol except ε

(because consecutive ε can be merged to I)

5.4.7 (a) $+* - xy\,xy$



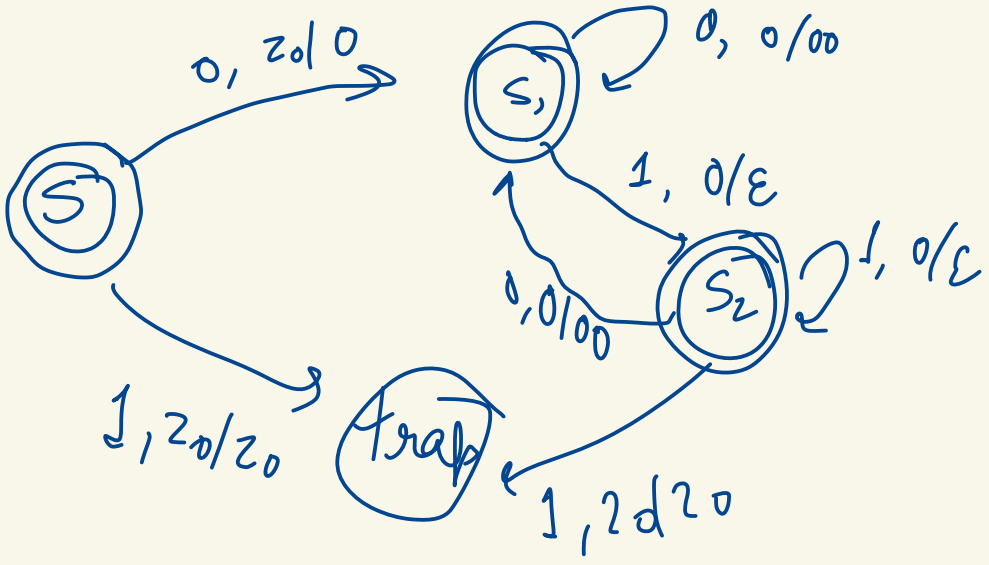
starting letter is fixed

↓

unambiguous

one option for every starting letter

6.2.5 (b)

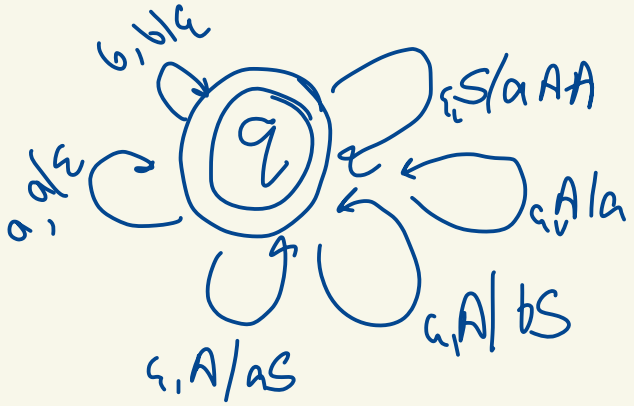


6.2.7 

Give alphabets some numbers and code

push and pop accordingly

6.3.2



Accepted by empty stack

6.3.5 (a) For every a b push two X_0 then pop while seeing c

(b) $a \to 4x$

$S \to XC \mid AY$ ← form using ε transition

$X \to a^i b^j \quad i = 2j$

$Y \to b^j c^k \quad j = 2k$

Construct PDAs

(c) $0^n 1^n$ $0^n 1^{2n}$

Non deterministically push one as two 0s and then start popping 1s see if this accepts for some procedure

6.4.2 (a) keep pushing → Z_0 visible

(b) keep pushing → Z_0 not visible

(c) $0^m 1^m 0^n$ — push, do nothing, pop

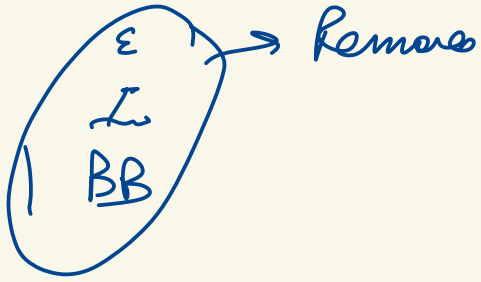
6.4.4 Bit doubtful

7.1.2 $S \to OAO \mid 1 \mid BB$

$A \to C$

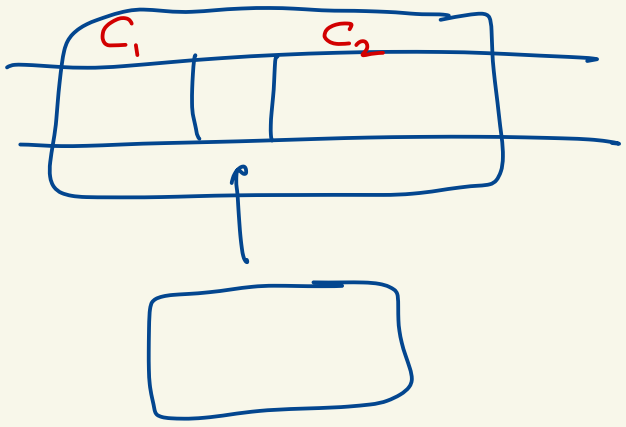
$B \to S \mid 1$

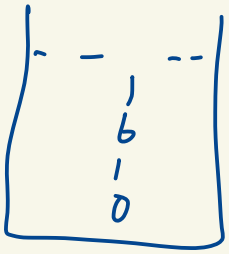
$C \to S$



7.2.4 $a^{2p} b^{2p} c^i$ it will be a as b

7.2.1 (f) $0^p 1 1 0^p 0^p 1$





with base = 3
the most significant
bit = 0 won't
make sense

div by k+1 mult by k+1
add 1,2...k

MSB				LSB
0	1	b	1	0
i	2	3	2	1

Pop : div by k+1
Push : { mult by k+1
add 2,... k
top : rem by k+1

3 counters → equivalent
2 counters → temporary counter, another one → same as turing machine

C_1 C_2 C_3
i j k

$p_1^i p_2^j p_3^k$
$C = 2^i 3^j 5^k$
(i, j, k) represented uniquely

+1
-1
=0?

Is $C_1(i) = 0$?
$3^i \times 5^k$

$i+1 \Leftrightarrow 2^{i+1} \times 3^j \times 5^k$
$= (2^i \times 3^j \times 5^k) \times 2$

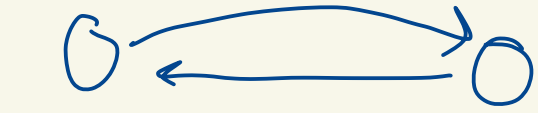
multiplication using temporary counter.

c^* $\hat{c}$

30

& $c^* = 0$?

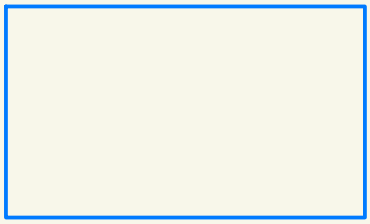
$c^* = c^* - 1$ $\hat{c} = \hat{c} + 1$

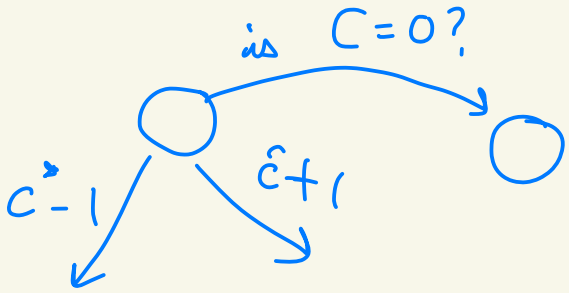


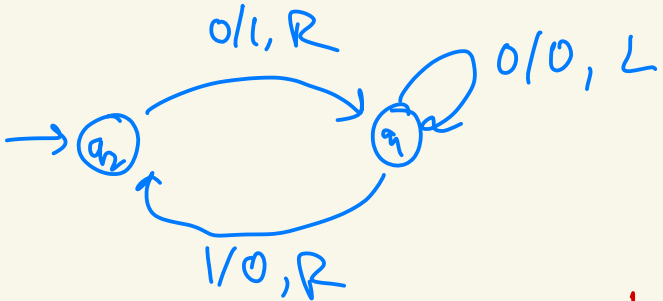
mod 2 = 0 mod 2 = 1

$c^* = c^* - 1$ $\hat{c} = \hat{c} + 1$

c^* $\hat{c}$







blank symbol id

11 0 11 0 111 0 1 0

states | Start State id | # tape symbols

L = 1

R = 11

11 0 11 0 111 0 10 0 11 0 11 0 1 0 111 0 11 00

Initialization part

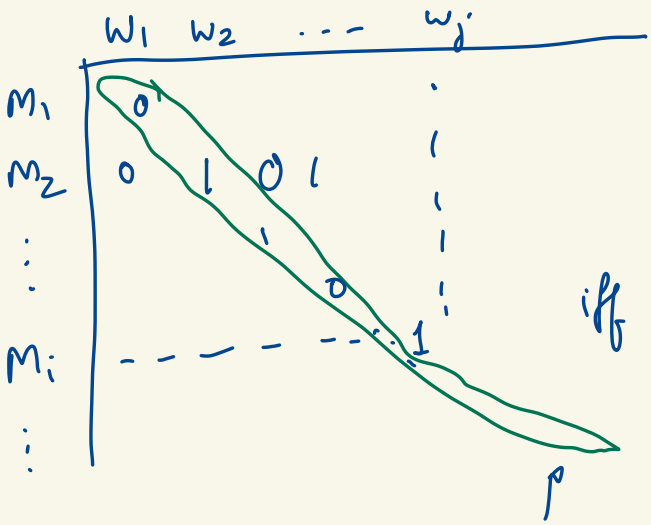
Transition q_2 0 q_1 1 R

q_2 0 q_1 1 R Transition

id of src state / 0 / id of symbol read / 0 /

id of destination state / 0 / id of sym written /

/ 0 / id of head movement / 00 ↓ Demarcator



iff M_i halts on w_j

this diagonal has countably infinite entries

1 0 0 1 ...

reverse and write

Does there exist any TM M_k such that

$$L(M_k) = L_d \; ?$$

Suppose Yes.

So $w_k \in L(M_k)$?

yes → $w_k \notin L_d$

No → $w_k \notin L_d$ X

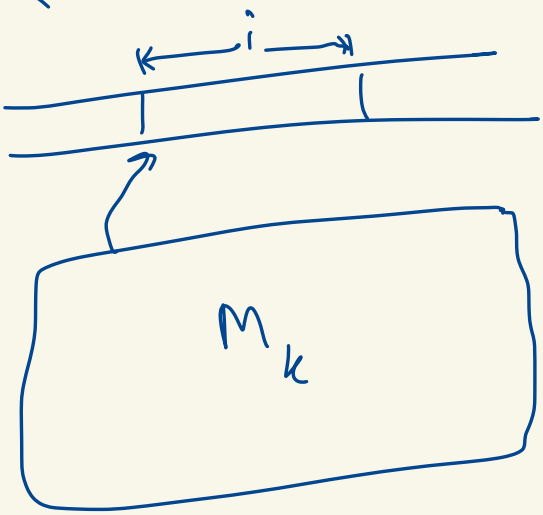
Recursively Enumerable Languages

L is REL iff $\exists$ TM M such that $L = H(M)$

$L_d \notin REL$

$L_d = \{ i \mid M_i \text{ does not halt on } w_i \}$

$\overline{L_d} = \{ i \mid M_i \text{ halts on } w_i \}$



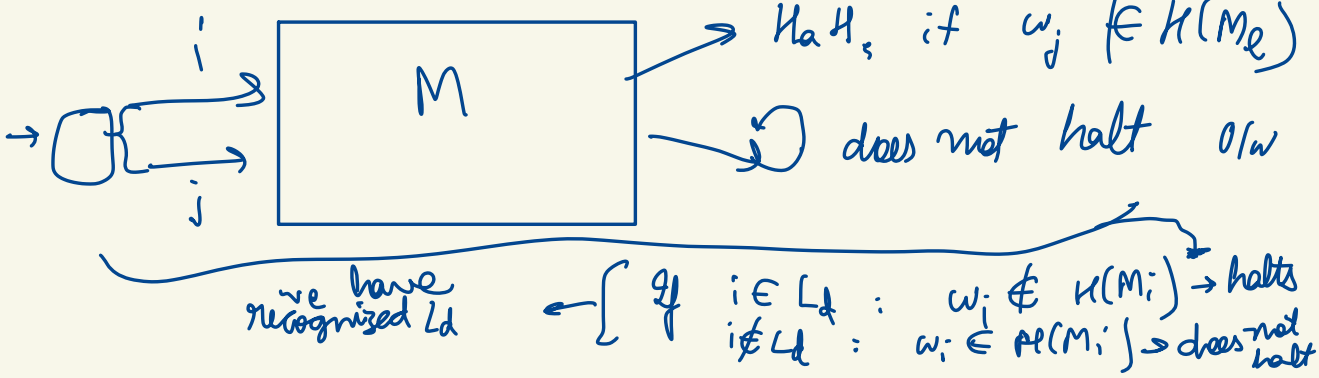
Universal TM:

$L_u = \{ i \# j \mid w_i \in H(M_i) \}$ Is L_u r.e.?

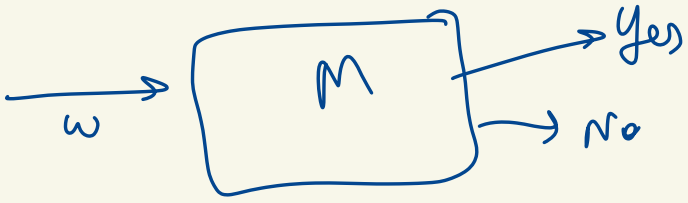
$\exists M_k$ s.t. $H(M_k) = L_u$?

$\overline{L_u} = \{ i \# j \mid w_j \notin H(M_i) \}$ Is $\overline{L_u}$ r.e.?

Suppose $H(M) = \overline{L_u}$



Recursive Languages



always halts, but can either halt in an accepting state or non-accepting state

Given L, is L rec?

Does $\exists M$ s.t. $H(M) = \Sigma^*$
$L(M) = L$.

Is L_d recursive?

L_d is not r.e
T_d is r.e.

Is L_u recursive?

L_u is r.e.
T_u is not r.e.

Recursive languages are closed under complementation.

1/1

Decidable Languages

Tutorial 7: Turing machines all around

---

1. **Turing machines and halting programs**

   Let $\Sigma = \{a, b\}$ and $L \subseteq \Sigma^*$ be a language over Σ.

   - Define PREFIX$(L) := \{w \mid \exists y \in \Sigma^*$ such that $wy \in L\}$. Thus, PREFIX(L) is the the set of all prefixes of all words in L.
   - Define HALF$(L) := \{w \mid \exists y \in \Sigma^*$ such that $|w| = |y|$ and $wy \in L\}$. Thus, HALF(L) is the set of all first-halves of words of even length in L.

   (a) Your best friend has constructed a Turing machine M, such that its halting language $H(M) = L$. Show how to build a Turing machine that has PREFIX(L) as its halting language. Your Turing machine can use M as a sub-machine.

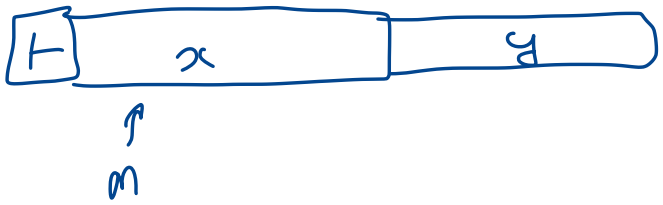
   (b) Your second best friend has constructed a Turing machine M' that enumerates all and only the words in L on an output tape. Show how to build an enumerator for HALF(L) using M.

   [Hint: Try to construct a simple pseudo-code (with loops, if statements, assignments, jump statements etc.) that achieves the desired task. Then think about whether every construct in this pseudo-code can be converted into a (possibly multi-tape) Turing machine, all of which can then be "strung" together to give you an overall Turing machine that achieves the desired task.]

recursively enumerate over (y, n) ↑ number steps to halt if one found then move to next y

Check the length of y

1.(a)



$y \in \Sigma^*$

Total TM + halts on every input

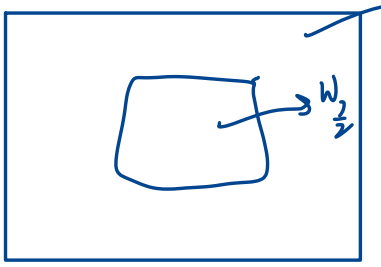
L is recursive if $\exists$ TM M such that $L(M) = L$

↓ recursively enumerable ↓ TM

$L(M') = \text{PREFIX}(L)$

$L = L(M)$

(b)



2. **Closure properties of languages accepted by Turing machines**

Recall from our discussion in class that a language L is *recursively enumerable* if there exists a Turing machine M such that its halting language $H(M)$ equals L. Recall also that we discussed that for every such language, we can build another Turing machine that takes no input, but enumerates all and only the strings in L on an output tape.

Finally, recall from our class discussions that language L is *recursive* if there is a Turing machine M' with two designated states q_Y and q_N such that (a) $H(M') = \Sigma^*$, (b) for all $w \in L$, the machine M' halts in state q_Y when it is started with w on its tape, and (c) for all $w \notin L$, the machine M' halts in state q_N when it is started with w. In other words, M' halts on all inputs, but it halts in q_Y for every string in L, and halts in q_N for every string not in L.

Let RE be the class of all recursively enumerable languages, and let R be the class of recursive languages.

Take a word and split into many portions (finite number of splittings) Run machine on all of them.

(a) Is RE closed under Kleene closure? Yes Is R closed under Kleene closure? Yes

(b) A homomorphism is defined by a mapping $h : \Sigma \to \Sigma^*$. With abuse of notation, we use h to also denote a mapping from strings to strings as follows: $h(\varepsilon) = \varepsilon$ and $h(a_1.a_2.\ldots.a_k) = h(a_1).h(a_2).\ldots.h(a_k)$, where each $a_i \in \Sigma$ and $a_1.a_2.\ldots.a_k \in \Sigma^*$.

Is RE closed under homomorphisms? Is R closed under homomorphisms?

*[**Food for thought:** Define the inverse homomorphism of a language $L \subseteq \Sigma^*$ as $h^{-1}(L) = \{w \mid w \in \Sigma^*, h(w) \in L)\}$. Is* RE *closed under inverse homomorphism? What about* R*?]*

$L = \{L_1, L_2 \ldots\}$ $L \in \Sigma^*$

$h(\hat{L}) = L_1, L_2 \in L$

$L_1 L_2 \in L$

$L_1 \in L \Rightarrow L_1^* \in L$

$K \subset (RE) = RE$

$h : \Sigma \to \Sigma^*$

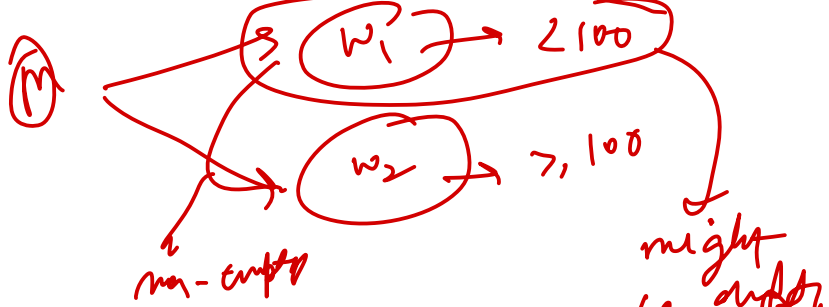
$L = \{(M, w, e_i)\}$ → Some special symbol Eg $\$i$

i = number of transitions

Concatenation of three strings

$h : 0 \to 0$
$1 \to 1$
$\$ \to \varepsilon$

Reduce to L_u for Recursive. $MW \to L_u$ (not recursive)

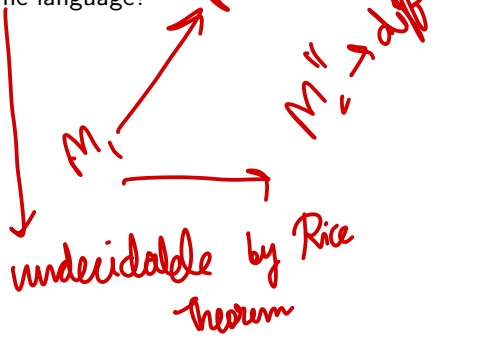


Decidable: $|s| \leq 100$. Just take the prefix if the string is longer

3. **Let's decide (if we can)**

   Find whether the following problems are decidable.

   - Given a Turing machine M, a state q and a string w, does M when started on the string w reach the state q?
   - Given a Turing Machine M, does it halt on all strings in less than 100 transitions?
   - Given two Turing machines M_1 and M_2, do they accept the same language?



$L = \{ (M, q, w) \text{ such that } w \xrightarrow{*}_M \alpha q \beta \}$

M, w & qp $\xrightarrow{\text{mapping}}$ $(M' q_0 w) \in L$

$(M,w) \in HP$

$M' \to M$, enters q_a if M enters q_r q_a (reject and accept state)

$L = \{ M \mid M \text{ halts on all strings in } < 100 \text{ transitions} \}$

$\{ M \mid M \text{ halts on all strings } s, |s| < 100 \text{ transitions} \}$

$L = \{ M \mid L(M) = \phi \}$ is undecidable

4. **Take-away problem: Enumerating recursive languages**

   Show that every infinite recursively enumerable language has an infinite subset that is a recursive language.

5. **Take-away problem: Neither r.e. nor co-r.e.**

   For purposes of this question, let M_i denote the Turing machine encoded by the binary representation of the natural number i, and let $H(M_i)$ denote the language accepted by M_i by halting. Let $L = \{1^i 0^j \mid i, j \in \mathbf{N}, H(M_i)$ is a strict subset of $H(M_j)\}$. In other words, $1^i 0^j \in L$ iff every word in $H(M_i)$ is also in $H(M_j)$, but there is at least one word in $H(M_j)$ that is not in $H(M_i)$.

   1. Show that L is not recursively enumerable.
   2. Show that L is not co-recursively enumerable either.

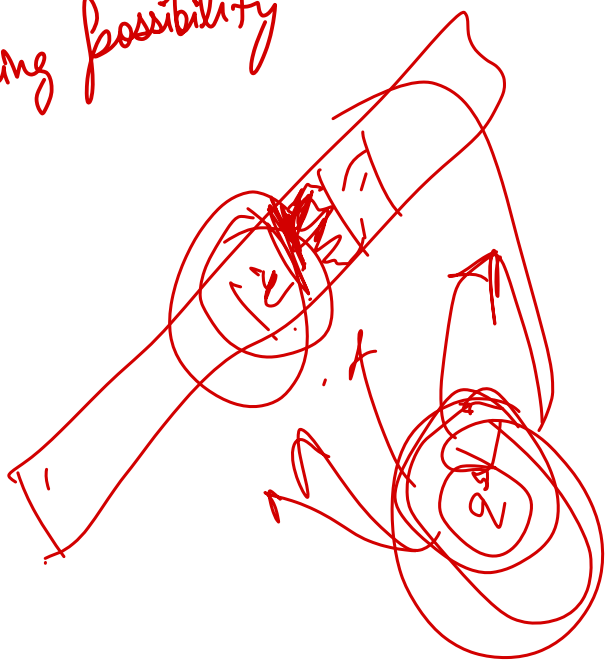
6. **Take-away problem: Almost undecidable, but not quite!**

Let $\langle M \rangle$ denote the encoding of Turing Machine M as a string over $\Sigma = \{0,1\}$, as studied in class. Let w be a string in $\{0,1\}^*$ and let $L = \{(\langle M \rangle, w) \mid M$ when run with w as the initial string on its tape visits less than $|w|$ tape squares$\}$.

(a) Show that L is decidable.

(b) Give an upper bound of the time complexity (i.e. count of steps taken by a halting Turing machine) of deciding if a given $(\langle M \rangle, w)$ pair is in L. Your complexity expression should be a function of $|\langle M \rangle|$ and $|w|$.

Think about the number of states and symbols M has and then a looping possibility



Undecidability

Turing machines → halt
↳ may run forever

L is recursively enumerable if $L = L(M)$ for some TM M.

(M, w) — M accepts w.
↓ ↳ string of 0s and 1s
Turing machine coded in binary

L_d = diagonalization language = all strings w such that the TM represented by w does not accept the input w.

w = binary string $\Rightarrow$ $\underbrace{1w}$ integer representing w.

Codes for turing machines:

States $q_1 \dots q_r$ q_1 = start q_2 = accepting

$X_1 \dots X_s$ = tape symbols $X_1 = 0$, $X_2 = 1$, $X_3 = B$

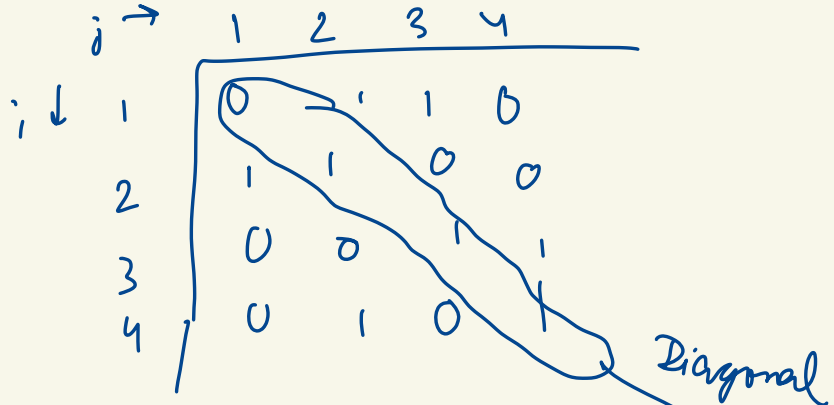
direction L as D_1
R as D_2

$\delta(q_i, X_j) = (q_k, X_l, D_m)$

$0^i 1 0^j 1 0^k 1 0^l 1 0^m$

$C_1 11 C_2 11 \dots$
C_i = transition

(M, w)
↓
111 to separate



M_i accepts w_j

1 means yes

0 means 0

Take the complement of diagonal for L_d

L_d is not RE. There is no TM that accepts L_d.

suppose $L_d = L(M)$ for some M

In our coding $M = M_i$.

If $w_i \in L_d$ then M_i accepts w_i $\Rightarrow$ $w_i \notin L_d$ (by definition)

If $w_i \notin L_d$ then M_i does not accept w_i

$\leftarrow$ $w_i \in L_d$

Contradiction!

Recursive Languages:

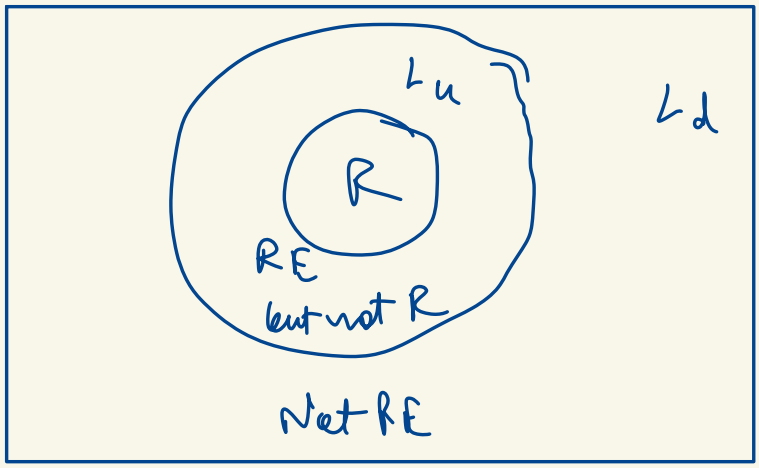
Language L = recursive:

(i) If $w \in L$ M accepts (and therefore halts)

(ii) If $w \notin L$ M eventually halts (never enters an accepting state)

recursive languages are decidable

undecidable if not recursive



If $L \xrightarrow{M}$ recursive then so is $\bar{L} \xrightarrow{\bar{M}}$.

The accepting states of M are made non-accepting states of $\bar{M}$

$\bar{M}$ has new accepting state γ. (No transitions from γ)

(non-accepting, take-symbol) $\longrightarrow \gamma$ } Add these.

take-symbol: at which that state halts.

transition

If both L and its complement are RE, then L is recursive. (So is $\bar{L}$)

Two tape → one-tape TM

M_1 , M_2

$\downarrow$ $\downarrow$

L $\bar{L}$

If w to M is in L, M_1 will halt

else M_2 will

M will accept

M will reject

$L(M) = L$

$\Rightarrow$ M = recursive

The universal language:

L_u = universal language

$= (M, w)$

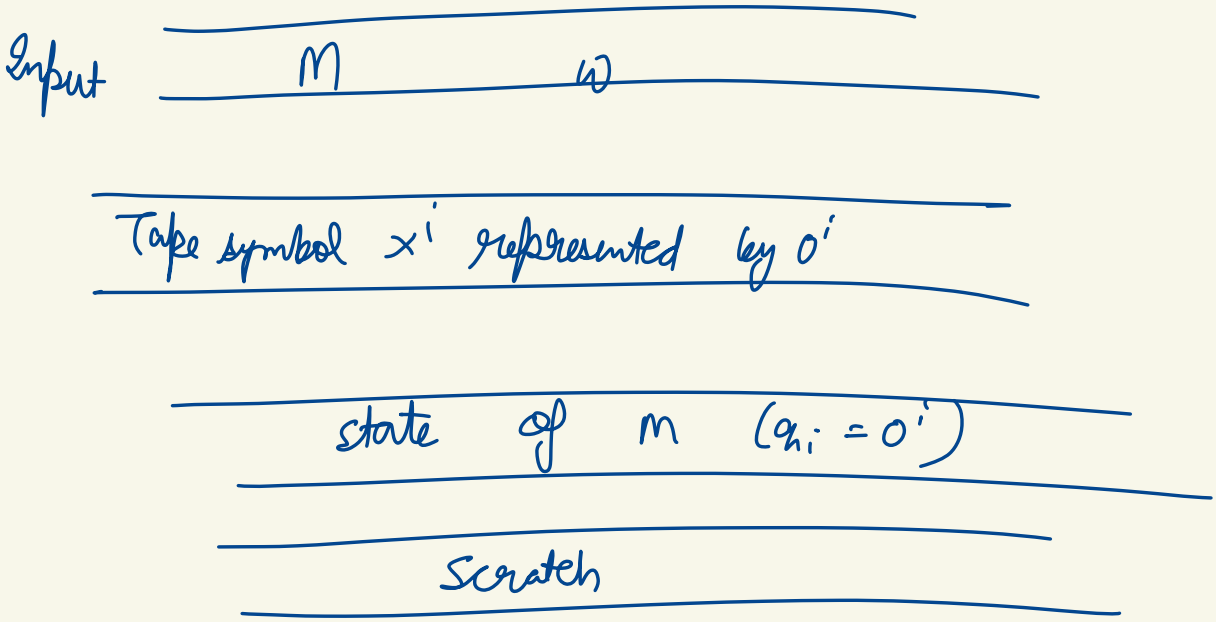
M → T_M (encoding of M)

$w \in L(M)$

$L_u = L(U)$

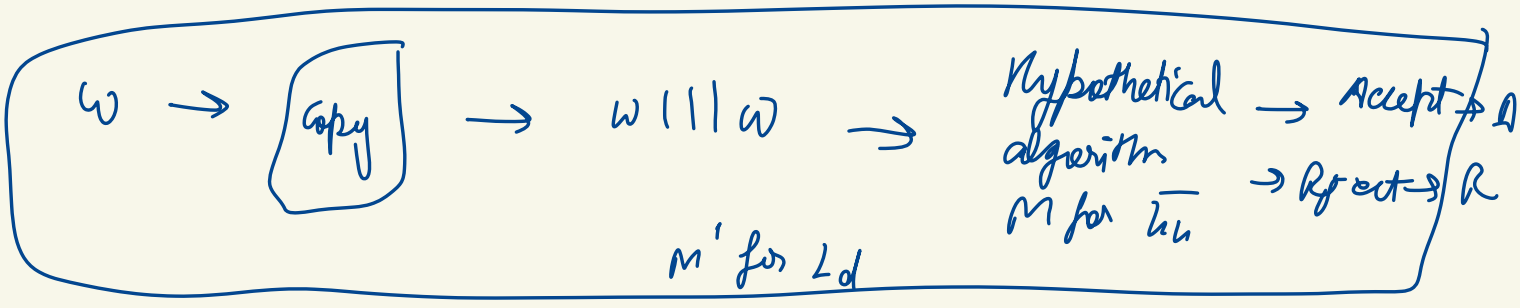
U = universal turing machine

U → multitape Turing Machine



L_u is RE but not recursive

Consider $\overline{L_u}$



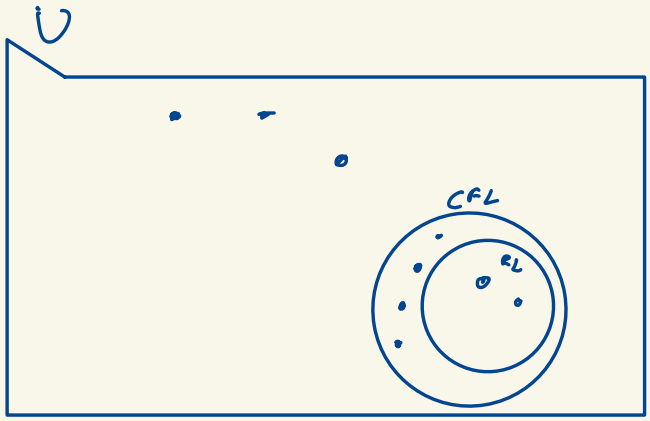
not re : $L_d = \{ i \mid M_i$ does not halt on $w_i \}$

not rec { re : $\overline{L_d} = \{ i \mid \ldots M_i$ halts on $w_j \}$

not rec { re : $L_u = \{ (i,j) \mid M_i$ halts on $w_j \}$

not re : $\overline{L_u}$

$L = H(M)$



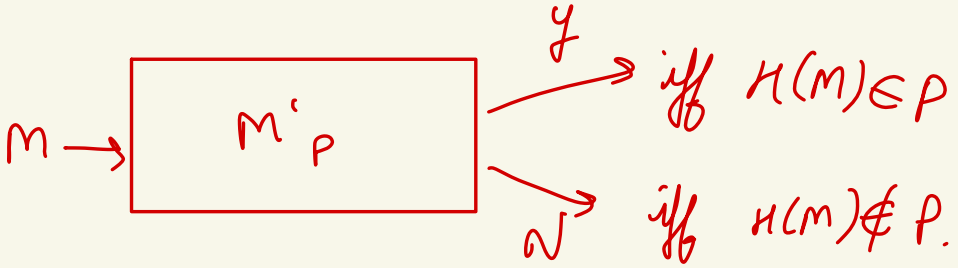
Property of languages defines a subset of U

Set of all possible r.e. languages on Σ

Property P of languages is non-trivial.
iff $\exists L_1$ s.t. $L_1 \notin P$
and $\exists L_2$ s.t. $L_2 \in P$

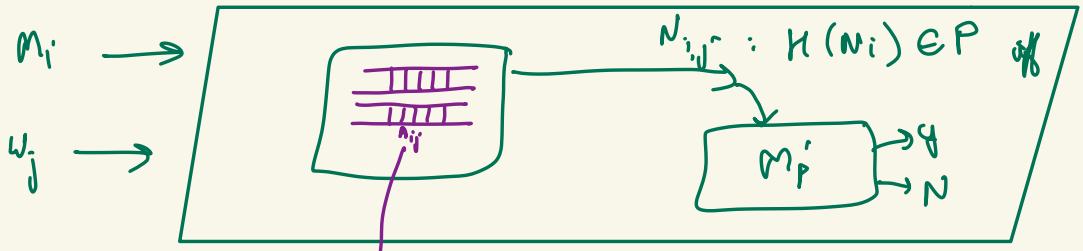
Rice's Theorem { Given a non-trivial prop P of U ($P \subset U$, $P \neq \phi$)
is P decidable?

Given a TM M does $H(M) \in P$?



$\phi \rightarrow \in P$
$\phi \rightarrow \notin P$

w.l.o.G $\phi \in P$ (else it would $\in \overline{P}$, reverse the reasoning)



writes w_j on scratch tape
mimics q_0 on w_j. If this process steps, then goes to first tape.
Mimics M_2 on T_1
$\downarrow$
$H(M_2) \notin P$

T_1 is tape on which A_{ij} processes it's input

If M_i halts on w_j
$H(A_{ij})$ → Language of M_2
$= H(M_2) \notin P$

If M_i halts on w_j
$H(A_{ij}) = \phi \in P$

At the end we will get to know if M_i halts on w_j
↓
Not possible!

CS 208 : Automata Theory and Logic Spring 2024

Homework 4

Due: 20^{th} Apr, 2024 *Max Marks: 30*

Instructions:

- *Please start writing your* ***solution to each homework problem on a fresh page.***
- *For each homework problem, you must scan your solution and upload a separate PDF file on Moodle. Please check Moodle for detailed instructions on file naming and uploading instructions.*
- *Be brief, complete, and stick to what has been asked.*
- *Untidy presentation of answers, and random ramblings will be penalized by negative marks.*
- *Unless asked for explicitly, you may cite results/proofs covered in class without reproducing them.*
- ***If you need to make any assumptions, state them clearly.***
- ***Do not copy solutions from others. All detected cases of copying will be reported to DADAC with names and roll nos. of all involved. The stakes are high if you get reported to DADAC, so you are strongly advised not to risk this.***

---

1. Turing machines and natural numbers 10 points

We have studied in class that every natural number i can be thought of as encoding a Turing machine, say M_i. Similarly, every natural number j can be thought of as encoding a string, say w_j, over an alphabet Σ.

Choose ***any one*** of the languages L_i ($i \in \{1, 2\}$) defined below and determine whether L_i is recursive. If your answer is "Yes", you must describe how to construct a halting Turing machine (i.e. halts on all inputs) that decides L_i. Otherwise, you must give a proof why L_i is not recursive (i.e. undecidable). Answers without a Turing machine or proof will fetch no marks.

Choose any one of the languages below to answer this question. Indicate clearly in your answer which language you are choosing.

(a) $L_1 = \{n \in \mathbb{N} \mid \exists m \in \mathbb{N} \text{ s.t. } M_n \text{ halts on } w_m\}$.

(b) $L_2 = \{m \in \mathbb{N} \mid \exists \text{ infinitely many } n \in \mathbb{N} \text{ s.t. } M_n \text{ halts on } w_m\}$.

2. Post's Correspondence Problem (PCP) and grammars 10 points

We've seen that the halting problem for Turing machines is undecidable. Another very well-known undecidable problem is *Post's Correspondence Problem, also popularly call PCP*. An instance of PCP consists of two finite (ordered) lists, say A and B, of strings over an alphabet Σ, such that the lists are of equal length. Let the lists be $A = (w_1, w_2, \ldots w_k)$ and $B = (v_1, v_2, \ldots v_k)$, where each $w_i, v_i \in \Sigma^*$. For each $i \in \{1, \ldots k\}$, the strings w_i and v_i are called *corresponding strings*. A solution to the PCP instance is a finite sequence of integers $(i_1, i_2, \ldots i_m)$ such that the concatenated strings $w_{i_1} w_{i_2} \cdots w_{i_m}$ and $v_{i_1} v_{i_2} \cdots v_{i_m}$ are identical.

As an example, suppose $\Sigma = \{0, 1\}$ and let $A = (01, 10, 101, 101)$, $B = (0101, 101, 1, 01)$. In other words, $w_1 = 01, w_2 = 10, w_3 = w_4 = 101$, while $v_1 = 0101, v_2 = 101, v_3 = 1, v_4 = 01$. One solution to this instance of PCP is $(3, 1, 2, 4)$, since $w_3 w_1 w_2 w_4 = 1010110101 = v_3 v_1 v_2 v_4$.

On the other hand, if $A = (011, 1101, 110, 111)$ and $B = (001, 001, 00, 010)$ you can convince yourself that this instance of PCP doesn't have a solution, since every w_i has more 1s than the corresponding v_i.

The decision version of PCP can be stated as follows: *Given an instance of PCP, does it have a solution?*

It is known that ***PCP is undecidable***. In other words, there does not exist any halting Turing machine that takes as input an instance of PCP, and halts in a designated "Yes" state if the instance has a solution, and halts in a designated "No" state otherwise. For those interested, you can look up Section 9.4 of Hopcroft, Motwani and Ullman's book for a detailed proof of this result. In this problem, we won't use the details of the proof. Instead, we will simply appeal to the undecidability of PCP.

Show by using a reduction from PCP that the following problem is undecidable:
Given a context-free grammar G, does there exist a terminal string $w \in L(G)$ such that $w^R \in L(G)$ as well, where w^R denotes the string w reversed?

You must explain clearly how having a Turing machine that decides the above problem about CFGs would enable you to construct a Turing machine that decides PCP.

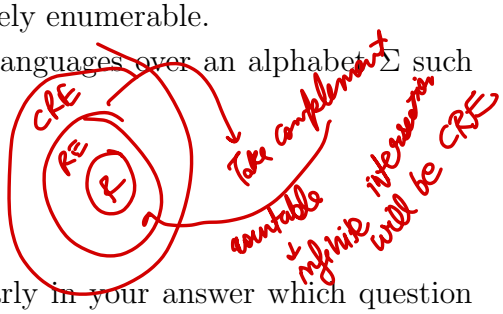
Answers without explanations will fetch no marks.

[Hint: Take an arbitrary instance of PCP and show how to construct a CFG G out of it such that both w and w^R are in $L(G)$ iff the PCP instance has a solution.]

3. Co-recursively enumerable languages 10 points

A language $L \subseteq \Sigma^*$ is said to be *co-recursively enumerable* if its complement (i.e. $\Sigma^* \setminus L$) is recursively enumerable. We have seen examples of co-recursively enumerable languages in class. For example, the diagonalization language L_d is co-recursively enumerable.

Let $\mathcal{F} = \{L_i \mid L_i \subseteq \Sigma^*,\ i \in \mathbb{N}\}$ be an (infinite) family of languages over an alphabet Σ such that



- For every $i \in \mathbb{N}$, L_i is not recursively enumerable
- For every $i \in \mathbb{N}$, $L_i \subseteq \Sigma^*$ is co-recursively enumerable.
- For every i, j s.t. $i \neq j$, $L_i \not\subseteq L_j$.

Answer ***any one*** of the following questions. Indicate clearly in your answer which question you are choosing to answer.

(a) Prove that the (infinite) intersection of all languages in $\mathcal{F}$, i.e. $\{w \mid \forall i \in \mathbb{N}, w \in L_i\}$, is co-recursively enumerable.

(b) Give an example of $\mathcal{F}$ such that the (infinite) union of all languages in $\mathcal{F}$, i.e. $\{w \mid \exists i \in \mathbb{N}, w \in L_i\}$, is not recursively enumerable but is co-recursively enumerable.
You must clearly state what the language L_i is for each $i \in \mathbb{N}$, show that these languages satisfy the three properties listed above, and prove that the infinite union is not recursively enumerable, although its complement is recursively enumerable.

Answers without proofs will not fetch any marks.

$L_d = \{ w_{k_1}, w_{k_2} \dots w_{k_i} \dots \}$

= set of all w_i such that w_i is not in $M(i)$

Not RE

$L_i = L_d \setminus \{ w_{k_i} \}$

L_i is Co-RE

Code M_i : 0

not RE

Union = L_d = not RE

complement = RE

1) $i \to M_i$ $\exists m \in$

$j \to w_j$

3) (a) $L_1, L_2 \ldots$

$L_1 \cap L_2 \ldots$

infinite intersection: Co-recursively enumerable

$\overline{L_1} \cup \overline{L_2} \cup \ldots \cup$

recursively enumerable

We need to find all w such that

$w \in \overline{L_1} \cup \overline{L_2} \cup \ldots$

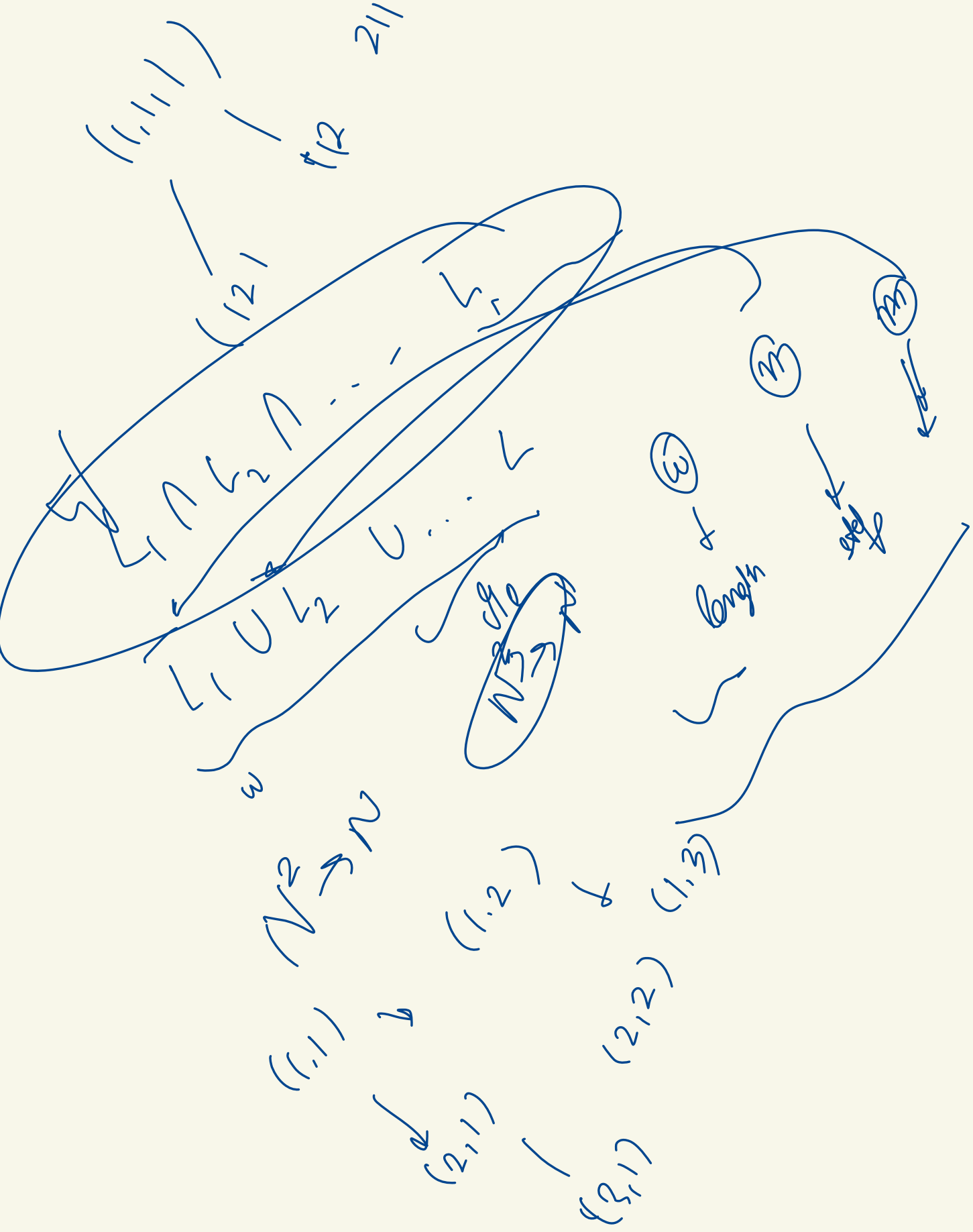
We will have three dimensions : number of steps to halt
encoding of string
which language we are considering

construct turing machine

11b (a) Rice Theorem

$L(M) = \phi$ → non-trivial property

No algorithm



CS 208

HW4-Q1

SAKSHAM RATHI

22B1003

1) (a) $L_1 = \{n \in \mathbb{N} \mid \exists m \in \mathbb{N} \text{ s.t. } M_n \text{ halts on } w_m\}$

So, L_1 is basically the set of all turing machines M_n which halt on atleast one input string ($H(M) \neq \phi$)

Claim: L_1 is undecidable

Proof: It is known that the Halting Problem is undecidable. So, if we reduce halting problem to L_1, then we can show that L_1 is undecidable.

(Halting problem = set of pairs (M, w) such that w is in $H(M)$ i.e. M halts on w.)

We will describe an algorithm that transforms (M, w) into an output M', the code for another turing machine, such that w is in $H(M)$ iff $H(M') \neq \phi$.

We can make M' ignore its input and instead simulate M on input w. If M halts, then M' accepts its own input.

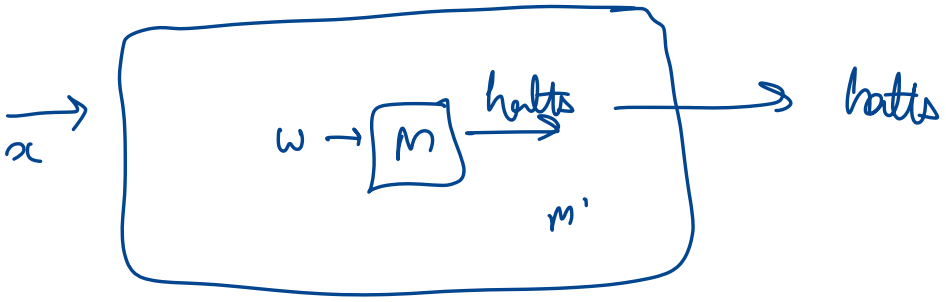
As we can see, if M does not halt on w then M' accepts none of its inputs i.e. $H(M') = \phi$. However, if M halts on w then M' accepts every input and thus $H(M') \neq \phi$

[By ignoring its input x, we mean M' replaces (x) by (M, w). this can be accomplished by some extra q_n states where n = length of the pair (M, w)]

Now using these additional states, M' simulates the turing machine for the halting problem.

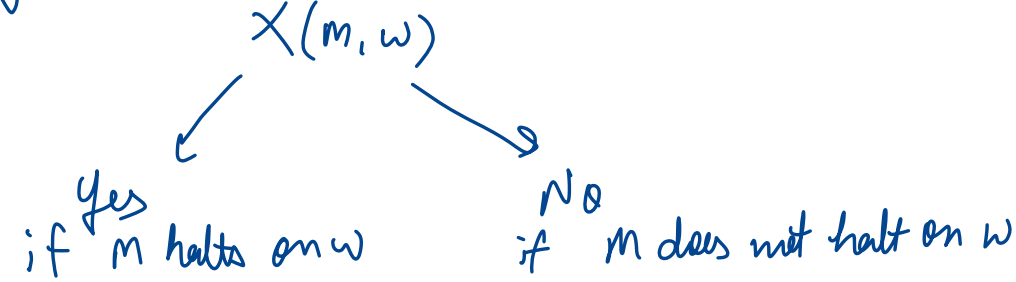
Therefore, we have reduced the halting problem to L_1.

Now, since halting problem is not recursive, and L_1 is as hard as halting problem, we can deduce that L_1 is undecidable.

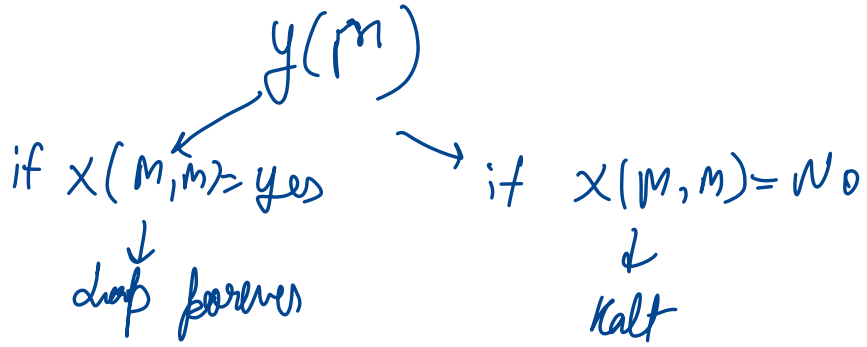


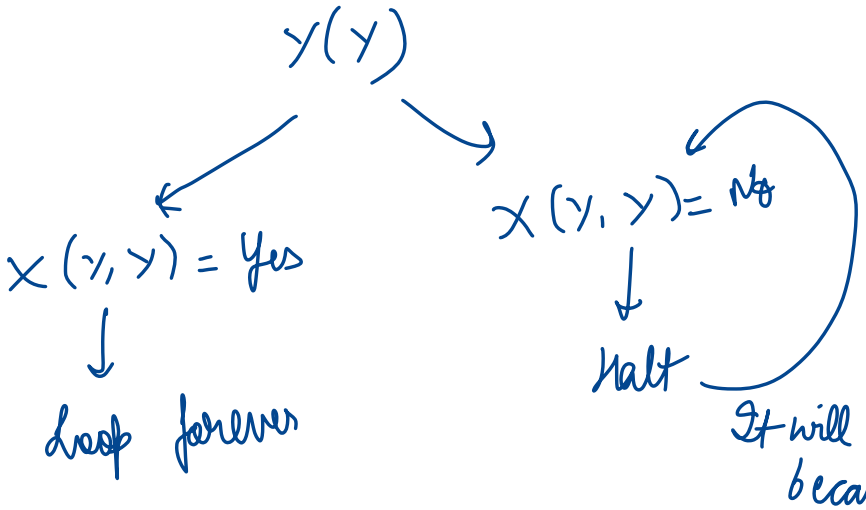
Proof of halting problem being undecidable:

Let us assume that we have a turing machine which moves to an accepting state if M halts on w. Call this machine X.



Consider Y such that:





Therefore, the halting problem is undecidable.

* Another smaller proof for this will be using Rice theorem.
We consider the property that $H(M) \neq \phi$. (Clearly a non-trivial property, because we can create machines that never halt and also machines which always halt.)
Now, using Rice theorem, we can deduce that this property is undecidable so is our language L_u.

CS 208

HW4-Q2

SAKSHAM RATHI

22B1003

2) Given two lists of strings $u_1, u_2 \dots u_n$ and $v_1, v_2 \dots v_n$ over an alphabet Σ, does there exist a sequence of indices $i_1, i_2 \dots i_k$ such that $u_{i_1} u_{i_2} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$

(This is the PCP problem)

We will take this instance of PCP and try to construct a grammar G out of this. Now, the PCP will have a solution iff w and w^R are in $L(G)$.

start symbol = S

$$S \to A \mid B$$

$$A \to u_1 A a_1 \mid u_2 A a_2 \mid \dots \mid u_n A a_n \mid \varepsilon$$

$$B \to a_1 B v_1^R \mid a_2 B v_2^R \mid \dots \mid a_n B v_n^R \mid \varepsilon$$

where $a_1 \dots a_n$ are extra symbols (different from the elements of Σ) (single letter symbols and distinct from each other, as long as n is finite, we can find such a_is)

A will have strings of the form:

$$u_{i_1} u_{i_2} \dots u_{i_k} a_{i_k} a_{i_{k-1}} \dots a_1 = w_A$$

B will have strings of the form:

$$a_{i_1} a_{i_2} \dots a_{i_k} v_{i_k}^R v_{i_{k-1}}^R \dots v_{i_1}^R = w_B$$

Consider

$$w_B^R = v_{i_1} v_{i_2} \dots v_{i_k} a_{i_k} \dots a_{i_1}$$

Now if w_B^R has to belong to $G_1(S)$ then it should be part of $G_1(A)$.

(Because B has a_i in the start and a_is are different from Σ alphabet letters.)

$$w_B^R = w_A' \quad \text{for some } w_A' \in G_1(A)$$

$$v_{i_1} v_{i_2} \dots v_{i_k} a_{i_k} \dots a_{i_1} = v_{j_1} \dots v_{j_\ell} a_{j_\ell} \dots a_{j_1}$$

These parts must match because they are different from Σ.

$\Rightarrow k = \ell$

and $i_t = j_t \quad \forall 1 \le t \le k$ → Because a_i distinct from a_j $i \neq j$.

From this we get that:

$$v_{i_1} \dots v_{i_n} = v_{i_1} \dots v_{i_k} \quad \text{for some set } (i_1 \dots i_k)$$

Similarly, we can take w_A^R and prove it to be equal to w_B

Therefore, we have deduced the following:

If PCP has a solution $i_1 \dots i_n$ then we can find w and w^R belonging to G.

Similarly, if we can find w and $w^R \in G$, we can have a solution to the PCP problem instance.

Since we have proved that PCP reduces to our grammar G, proving the existence of a terminal string $w \in L(G)$ such that $w^R \in L(G)$ is undecidable.

* One might think that for the set $i_1 \dots i_n$, i_{l_1} can be equal to i_{l_2} for $l_1 \neq l_2$. But it can be shown that for such cases we can remove all the repitions and our solution will still be valid.

CS 208

HW 4 - Q3

SAKSHAM RATHI

22B1003

3)(b) Consider $L = \{ i \mid M_i$ does not halt on any input $\}$

L = set of all turing machines which do not halt at all (for any inputs, $H(M) = \phi$)

$L_i = L \setminus \{ M_i \}$

M_i = turing machine which moves right by i steps and then moves left indefinitely (on seeing any input on tape) (does not halt on any input)

Infinite union of $L_i = L$

$\bar{L}$ = complement of L

$= \{ i \mid M_i$ halts on atleast one input $\}$

This string is similar to (Q1a) of HW4 (Already proved that it is not recursive)

But $\bar{L}$ is RE.

This can be proved by an enumerating turing machine. (seen in lecture).

We will take a turing machine M as input and iterate over two parameters:

(i) string w

(ii) number of steps to execute.

If M has some string w on which it halts, then it will do so after some n steps and we will stop after finding a single w.

Since, we are able to enumerate, it is clear that

$\overline{L}$ is RE.

It is known from Q1(a) that L is not R.

If L = RE and $\underbrace{\overline{L} = RE}_{\text{known}}$ $\Rightarrow$ $L = R$ (Contradiction!)

$\Rightarrow$ L is not RE

Therefore, both the constraints for the union are satisfied.

Now, we will consider L_i:

* $\overline{L_i} = \overline{L} \cup \{M_i\}$

Now since $\overline{L}$ and $\{M_i\}$ are both RE.

($\{M_i\}$ is RE because we can define it using encoding)

their union must be RE $\Rightarrow$ $\overline{L_i}$ = RE

* $L_i \not\subseteq L_j$ $\forall\ i \neq j$

L_i has M_i and L_j has M_j extra in their language sets. Hence they can't be subsets of each other.

* $L_i \neq$ RE

If L_i = RE, we already know that M_i = RE (because we can encode it in string form)

then $L = L_i \cup \{M_i\}$ = RE (Contradiction!)

$\Rightarrow$ $L_i \neq$ RE

Thus, all the constraints on L_i are satisfied.

$\therefore$ Hence, we have given a suitable example of $F = \{w \mid \exists i \in \mathbb{N}, w \in L_i\}$

3)(b) We need to take the infinite union of all languages in f

i.e. $\{w \mid \exists i \in \mathbb{N}, w \in L_i\}$

L_d = diagonalization Language

$= \{w_i \mid w_i \notin L(M_i)\}$

$= \{w_{d_1}, w_{d_2} \ldots\}$ (Countably infinite set, hence we can enumerate) (because of injection to $\mathbb{N}$)

Consider $L_i = L_d \setminus \{w_{d_i}\}$

[We are removing the string at the i^{th} position from L_d to obtain L_i]

* L_i is not recursively enumerable. Proof by contradiction:

$L_i = L(M_j)$ for some turing machine M_j

w_j = encoding of M_j in string form

If $w_j \in L_i$ then

then M_j accepts w_j but then by definition of L_i

$w_j \notin L_i$

If $w_j \notin L_i$ then M_j does not accept w_j. then $w_j \in L_i$

Contradiction!

* $\overline{L_i} = \overline{L_d} \cup \{w_{d_i}\}$

It is known that $\overline{L_d}$ is recursively enumerable.

Now, if we add one more string to $\overline{L_d}$, $\overline{L_i}$ will still remain RE (closed under union, we can use two tapes to prove).

* For every $i \neq j$ $L_i \not\subseteq L_j$

Clearly, L_i and L_j both have one string extra from each other, hence this is satisfied.

Therefore, we have proved that L_i satisfies all three constraints given in the question.

Infinite union = L_d (all strings which were removed are added back)

* It is known that L_d is not RE
(Proof similar to that of L_i done before.)
* Also, $\overline{L_d}$ = recursively enumerable.

This can be proved using the Machine for L_u (universal language).
Just, give (w_j, w_j) as input instead of (w_i, w_j).

Therefore, we have proved that L_i and their infinite union satisfy the constraints.

3b

$F = \{ L_i \mid L_i \subseteq \Sigma^*, \; i \in \mathbb{N} \}$

L' = infinite intersection of all languages in $F = \{ w \mid \forall i \in \mathbb{N}, w \in L_i \}$

= Co-recursively enumerable.

This is equivalent to proving that the complement of L' is recursively enumerable

$L' = L_1 \cap L_2 \cap \ldots$

$\overline{L'} = \overline{L_1} \cup \overline{L_2} \cup \ldots$

Now, we need to enumerate $\overline{L'}$ using a turing machine. (Hence, we will prove that $\overline{L'}$ is RE.)

The enumeration which we saw in class was based on two dimensions. Here will have three dimensions.

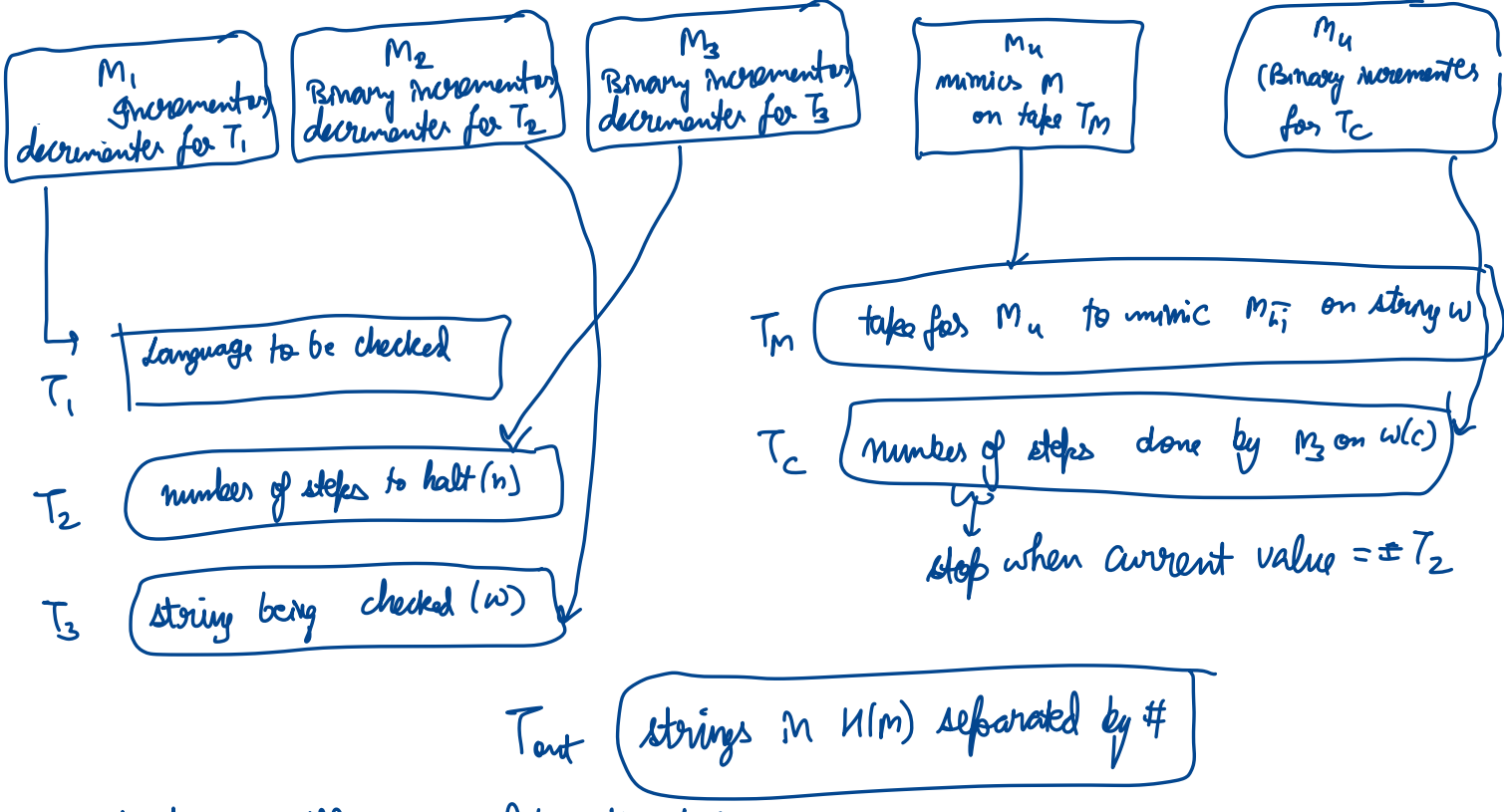
(i) which language to choose ($\overline{L_i}$)

(ii) encoding of the string (w)

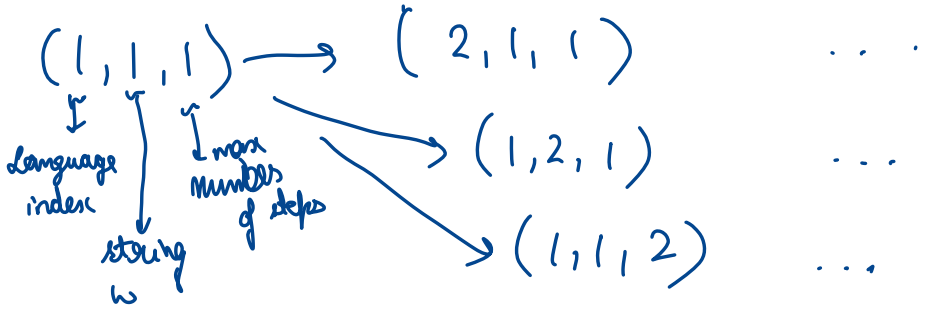
(iii) number of steps to halt (n)

Every string $w \in L'$ will belong to some $\overline{L_i}$. Also, it will have a finite number of steps n after which $M_{\overline{L_i}}$ halts since $w \in \overline{L_i} = H(M_{\overline{L_i}})$

Here is how our turing machine will look like:



So, the checking will go on like the following:



(Basically a bijection for N^3 to N)

All words in the language are written on the output tape T_{out} and since M does not halt for all the strings which are not in the language, they will not be written on T_{out}.

Hence, M enumerates exactly all the strings in L'.

$\Rightarrow$ $L' = RE$

$\Rightarrow$ $\overline{L}$ = Co-recursively enumerable

Hence proved!