Finite State Automata

1. Construct FSA A, such that $L(A) = \{x \in \{0,1\}^* \mid x \text{ contains at least three adjacent } 0\}$

2. Construct FSA A, such that $L(A) = \{x \in \{0,1\}^* \mid x \text{ ends with } 00\}$

3. Construct FSA A, such that $L(A) = \{x \in \{0,1\}^* \mid \text{ the second last symbol of } x \text{ is } 0\}$

4. Construct FSA A, such that $L(A) = \{a^{2n}b^{3m}c \mid a, b \in I, n \geq 1, m \geq 0\}$

5. Construct FST T, that implements the translation

$$\tau_T((ab)^n ccc(ba)^m) = d^{2n}ef^{\lfloor \frac{m}{2} \rfloor},\ n \geq 1,\ m \geq 0$$

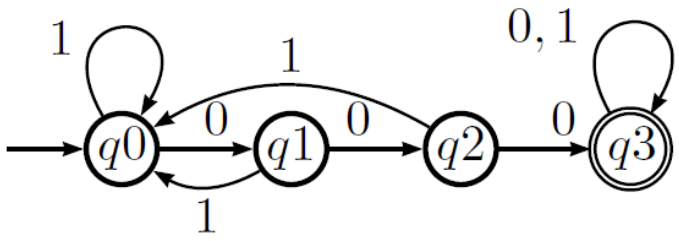
6. Construct FSA A, such that $L(A) = \{x \in \{0,1\}^* \mid x \text{ ends with odd number of 1s}\}$

7. Define the indistinguishability relation '$\approx_L$' for the strings of the language

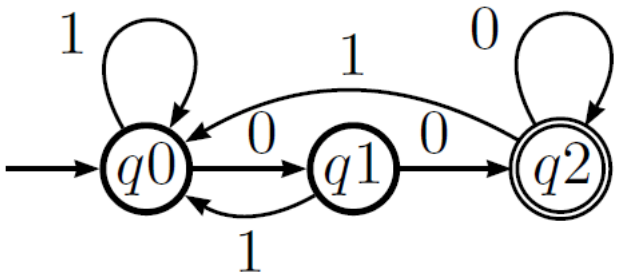
$L = \{ x \in \{0, 1\}^* \mid even(\#_0(x)) \text{ and } odd(\#_1(x)) \}$

--strings with even number of 0's and odd of 1's

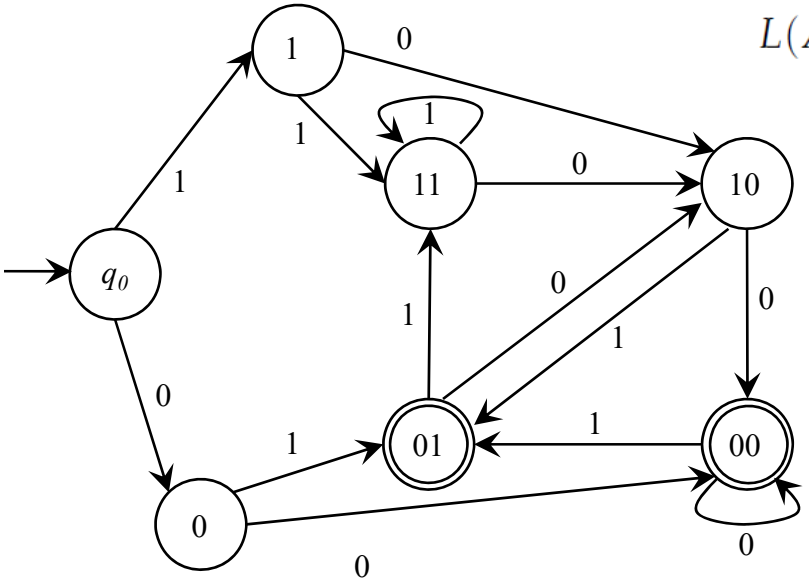
1. Construct FSA A, such that $L(A) = \{x \in \{0,1\}^* \mid x \text{ contains at least three adjacent } 0\}$



2. Construct FSA A, such that $L(A) = \{x \in \{0,1\}^* \mid x \text{ ends with } 00\}$

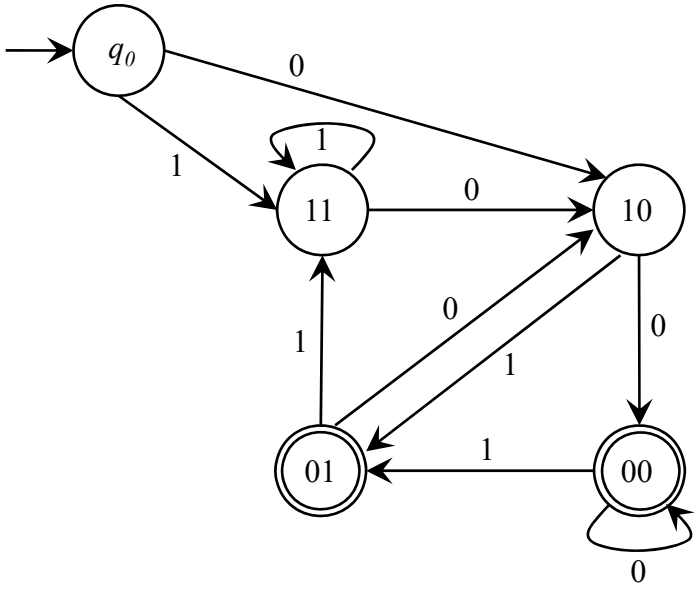


$L(A) = \{x \in \{0,1\}^* \mid \text{the second last symbol of } x \text{ is } 0\}$

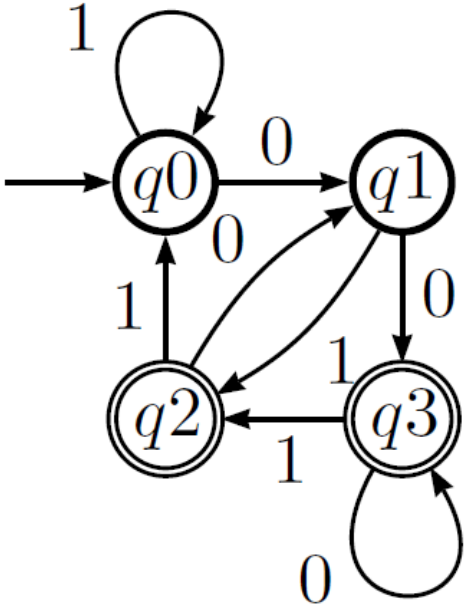


need to remember 2 symbols
- the second last one for accepting/rejecting
- the last one to accept/reject at the next move

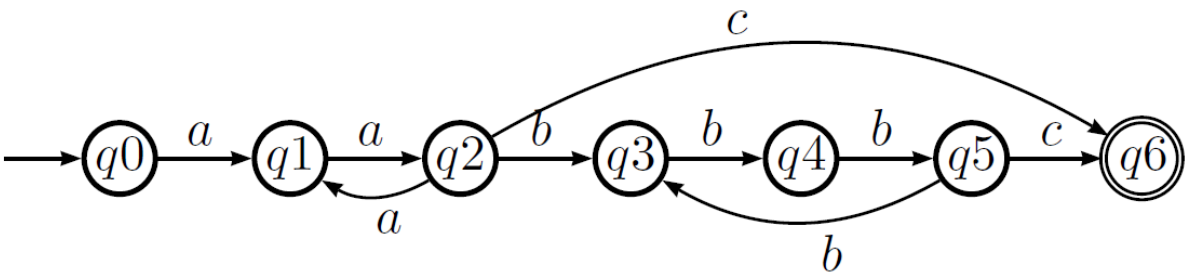
equivalent states
1 ≡ 11, 0 ≡ 10
NB: not 0 ≡ 00 because 00 is final and 0 is not



equivalent states
$q_0 \equiv 11$

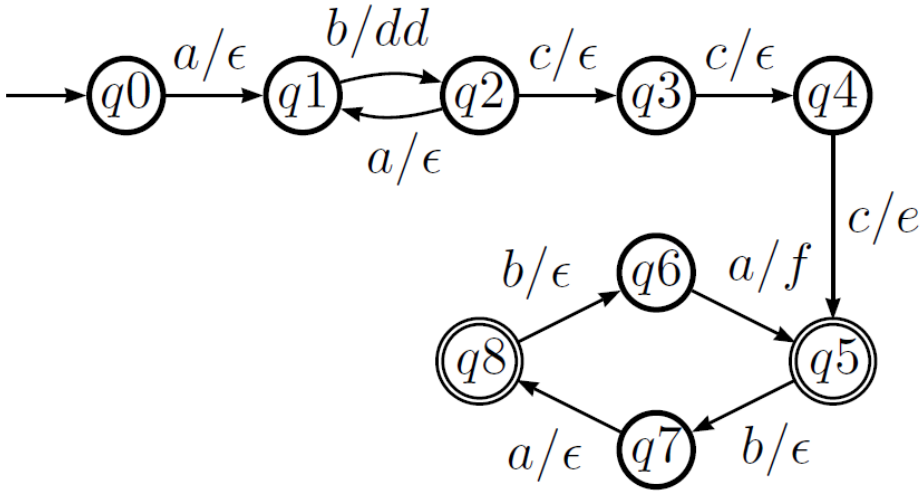


4. Construct FSA A, such that $L(A) = \{a^{2n}b^{3m}c \mid a, b \in I, n \geq 1, m \geq 0\}$



NB: when in a state there is no transition for the current input the computation terminates and the input string is rejected

5. $\tau_T((ab)^n ccc(ba)^m) = d^{2n}ef^{\lfloor \frac{m}{2} \rfloor},\ n \geq 1,\ m \geq 0$



$\tau(ababcccbabababa) = d^4 e f^2$

$\tau(ababcccbababa) = d^4 e f$

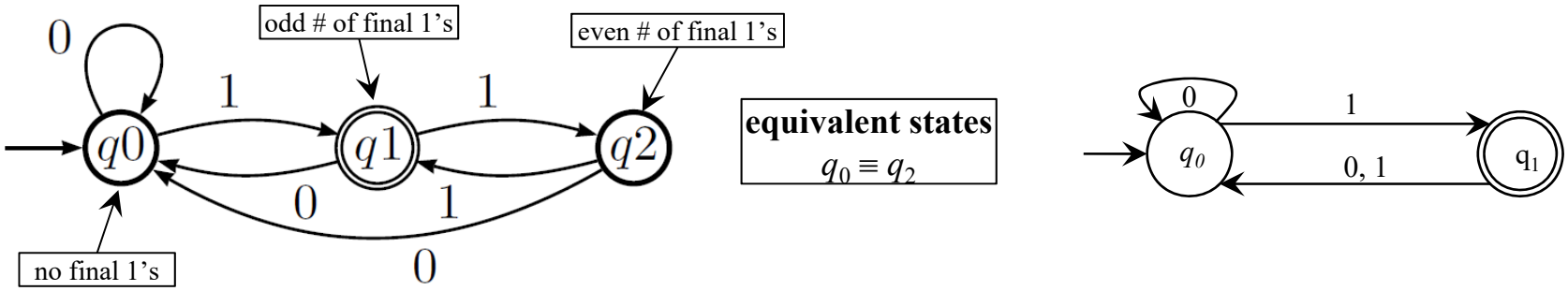
$\tau(abcccba) = d^2 e$

$\delta^*(q_0, abcccba) = q_8 \in F$ **accept**

$\delta^*(q_0, abcccb) = q_7 \notin F$ **reject**

$\tau(abcccb) = \perp$ **because** ***abcccb*** **rejected**

6. Construct FSA A, such that $L(A) = \{x \in \{0,1\}^* \mid x \text{ ends with odd number of 1s}\}$



FSA cannot count in general, but can up to a certain given bound

Ex.: $L = \{ a^n \mid n \text{ MOD } 5 = 3 \}$

Construct FSA A, such that $L(A) = \{x \in \{0,1\}^* \mid x \text{ the third last symbol is a } 0\}$

7. Define the indistinguishability relation '$\approx_L$' for the strings of the language

$L = \{ x \in \{0, 1\}^* \mid even(\#_0(x)) \text{ and } odd(\#_1(x)) \}$

--strings with even number of 0's and odd of 1's

[next we abbreviate $even(\#_0(.))$ into $even_0$, $odd(\#_1(.))$ into odd_1 etc…..]

Recall: $x, y \in L$ are ***indistinguishable*** $x \approx_L y$ iff $\forall z \in I^* \; x \cdot z \in L \Leftrightarrow y \cdot z \in L$
i.e., x and y share a common destiny }

for every prefix string v, what occurs next depends on the parity of the number of 0's and 1's

$x \approx_L y$ iff $\text{even}(\#_0(x)) \Leftrightarrow \text{even}(\#_0(y)) \wedge \text{even}(\#_1(x)) \Leftrightarrow \text{even}(\#_1(y))$

There are four cases, depending on whether $even_0$ and $even_1$ hold or not

Hence $\approx_L$ has four equivalence classes, which constitute a partition of $\{0, 1\}^*$

$[\varepsilon] = \{ x \in \{0, 1\}^* \mid even(\#_0(x)) \text{ and } even(\#_1(x)) \}$ … 0 is even …

$[0] = \{ x \in \{0, 1\}^* \mid odd(\#_0(x)) \text{ and } even(\#_1(x)) \}$

$[1] = \{ x \in \{0, 1\}^* \mid even(\#_0(x)) \text{ and } odd(\#_1(x)) \}$

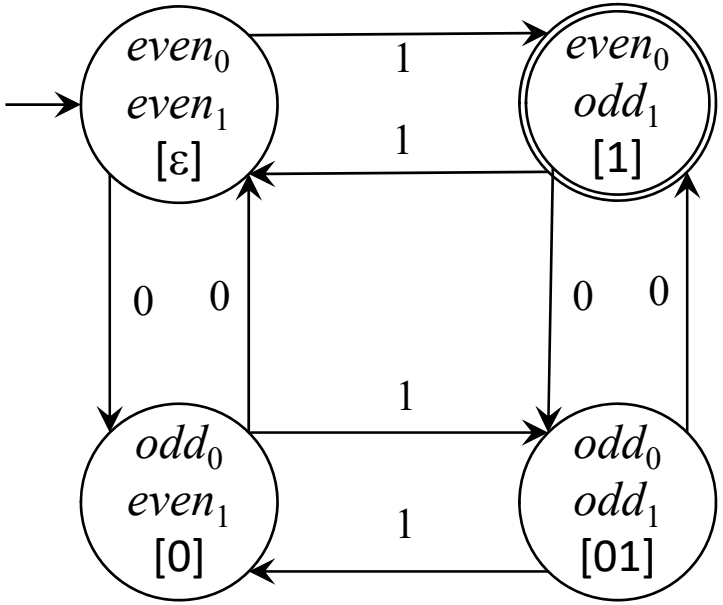
$[01] = \{ x \in \{0, 1\}^* \mid odd(\#_0(x)) \text{ and } odd(\#_1(x)) \}$

NB $[\varepsilon] = \{ \varepsilon, 00, 11, 0011, 0110, \ldots, 000011, 100001, \ldots \}$ is an infinite set of strings

$L = \{ x \in \{0, 1\}^* \mid even(\#_0(x)) \text{ and } odd(\#_1(x)) \}$ --even 0's and odd 1's

From $\approx_L$ obtain the accepting automaton

- $q_0 = [\varepsilon]$
- $Q = \{ [x] \mid x \in I^* \}$
- for every equiv. class $[x]$, $\forall a \in I \quad \delta([x], a) = [x \cdot a]$
  $\delta([01], 0) = [1]$ i.e., $\delta([odd_0 \wedge odd_1], 0) = [even_0 \wedge odd_1]$ … etc. …
- $F = \{ [x] \mid x \in L \} = \{ [1] \}$



Push Down Automata

1. Construct a PDA A, such that $L(A) = \{a^n b^{2n} \mid n > 0\}$

   provide two solutions, with or without ε-moves

2. Construct an automaton A, such that $L(A) = \{x \in \{a, b\}^* \mid \#_a(x) = \#_b(x)\}$

3. Construct an automaton A, such that

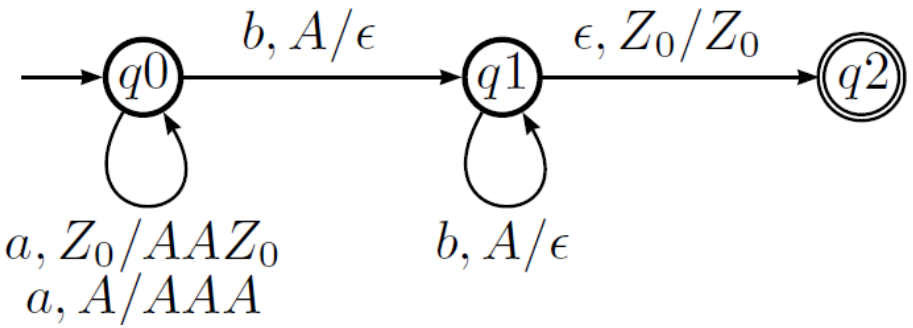
$$L(A) = \{a^n a^m a^m a^n \mid m \geq 1, n \geq 1, n \text{ is even and m is odd } \}$$

Build the deterministic pushdown transducer that computes the transduction τ defined by:

$$\tau(a^n b^m c^m) = d^{3m} e^n, \text{ with } m \geq 1, n \geq 1$$

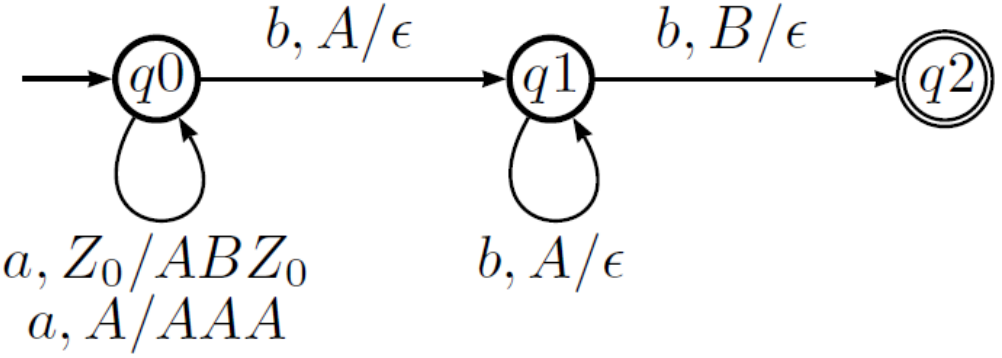
1. Construct a PDA A, such that $L(A) = \{a^n b^{2n} \mid n > 0\}$

provide two solutions, with or without ε-moves

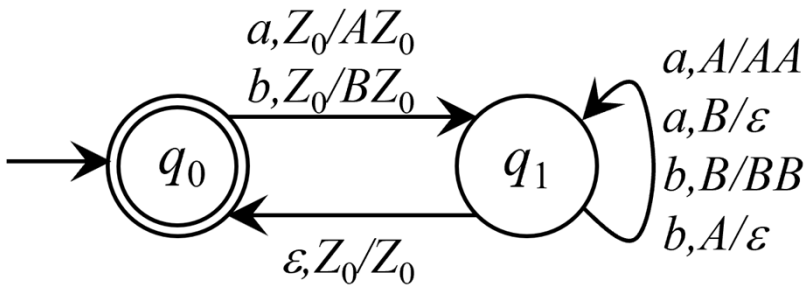


show computations on strings:
aabbbb, ***aabb***, ***abbb***

show computations on string
aabbbb



2. Construct an automaton A, such that $L(A) = \{x \in \{a,b\}^* \mid \#_a(x) = \#_b(x)\}$



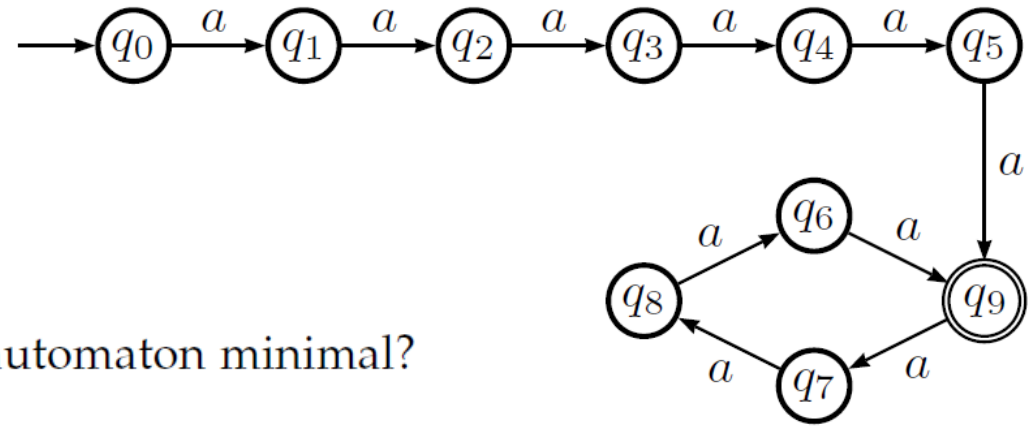
show computations on strings:
ba, ***aba***, ***baaabb***

3. Construct an automaton A, such that

$$L(A) = \{a^n a^m a^m a^n \mid m \geq 1, n \geq 1, n \text{ is even and m is odd } \}$$

deceitful question: it is a regular language

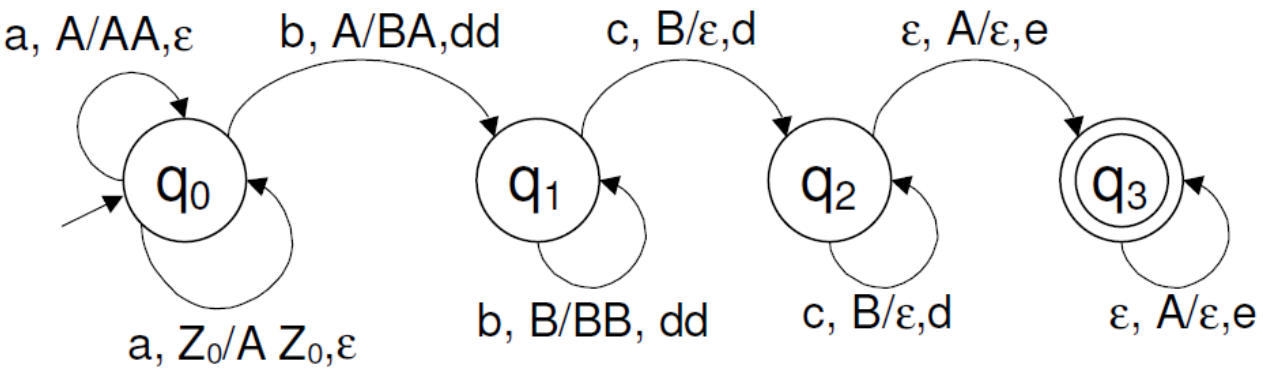
$L = \{ a^k \mid k = 2\,(2i + (2j + 1)) \ \wedge i \geq 1 \wedge j \geq 0 \ \} =$ (NB: even + odd = odd)
$= \{ a^k \mid k = 2\,(2i + 1) \ \wedge i \geq 1 \} =$
$= \{ a^k \mid k \in \{6, 10, 14, ...\} \}$



Is the automaton minimal?

Build the deterministic pushdown transducer that computes the transduction τ defined by:

$$\tau(a^n b^m c^m) = d^{3m} e^n, \text{ with } m \geq 1,\ n \geq 1$$



A pushdown automaton can use the states of the control unit for (bounded) counting
(much like a Finite State Automaton)

Example: design minimal power devices computing

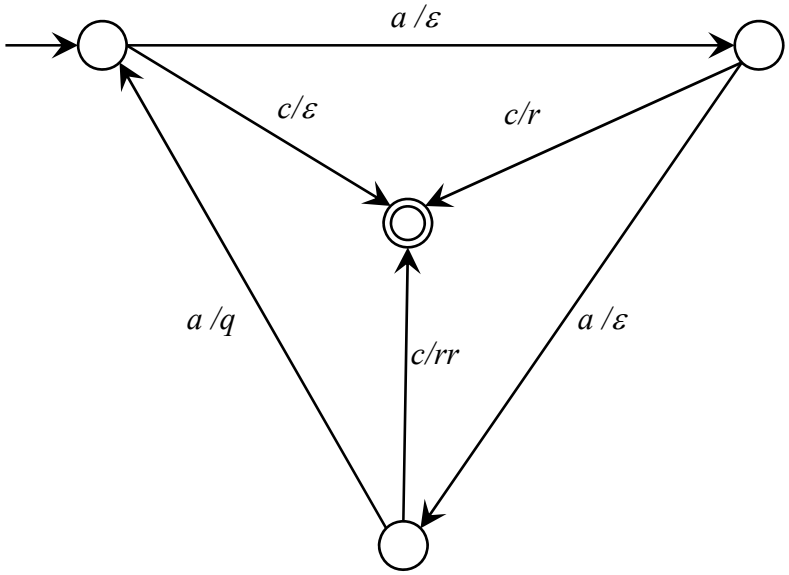
$$\tau(a^n) = q^{\,n \text{ DIV } 3}\ r^{\,n \text{ MOD } 3} \qquad \tau(a^n) = r^{\,n \text{ MOD } 3}\ q^{\,n \text{ DIV } 3}$$

Or better, adding an endmark symbol c,

$$\tau(a^n c) = q^{\,n \text{ DIV } 3}\ r^{\,n \text{ MOD } 3} \qquad \tau(a^n c) = r^{\,n \text{ MOD } 3}\ q^{\,n \text{ DIV } 3}$$

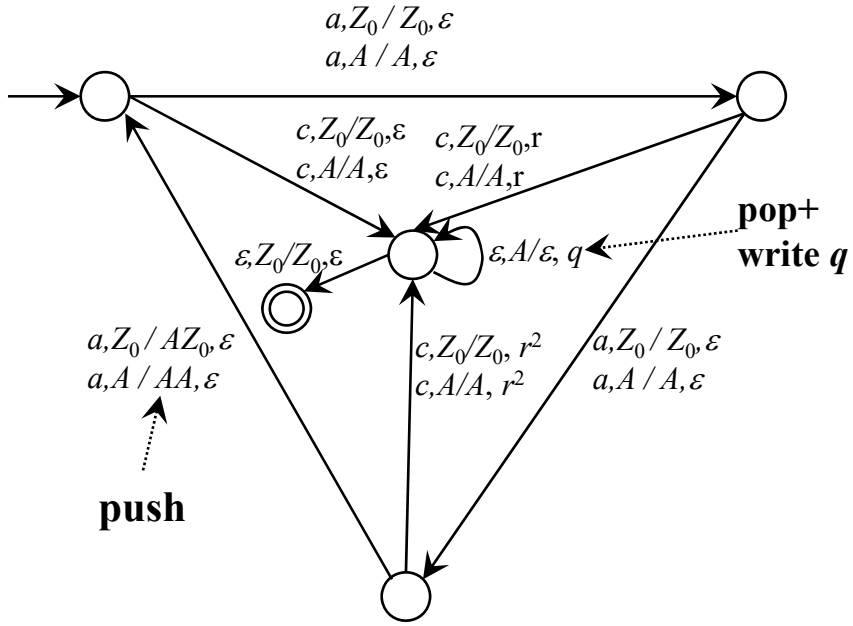
output first the quotient then the remainder
a FSA suffices

$$\tau(a^n c) = q^{\,n \text{ DIV } 3}\, r^{\,n \text{ MOD } 3}$$



output first the remainder then the quotient
a PDA is needed

$$\tau(a^n c) = r^{\,n \text{ MOD } 3}\, q^{\,n \text{ DIV } 3}$$



Turing Machines

$$L_2 = \{a^n b^n \mid n > 0\} \cup \{a^n c^n \mid n > 0\}$$

$$L_3 = \{a^n b^n \mid 0 < n < 21\}$$

$$L_5 = \{a^n b^{2n} a^n \mid n > 0\}$$

Construct an automaton A, such that $L(A) = \{a^n b^{n^2} \mid n > 0\}$

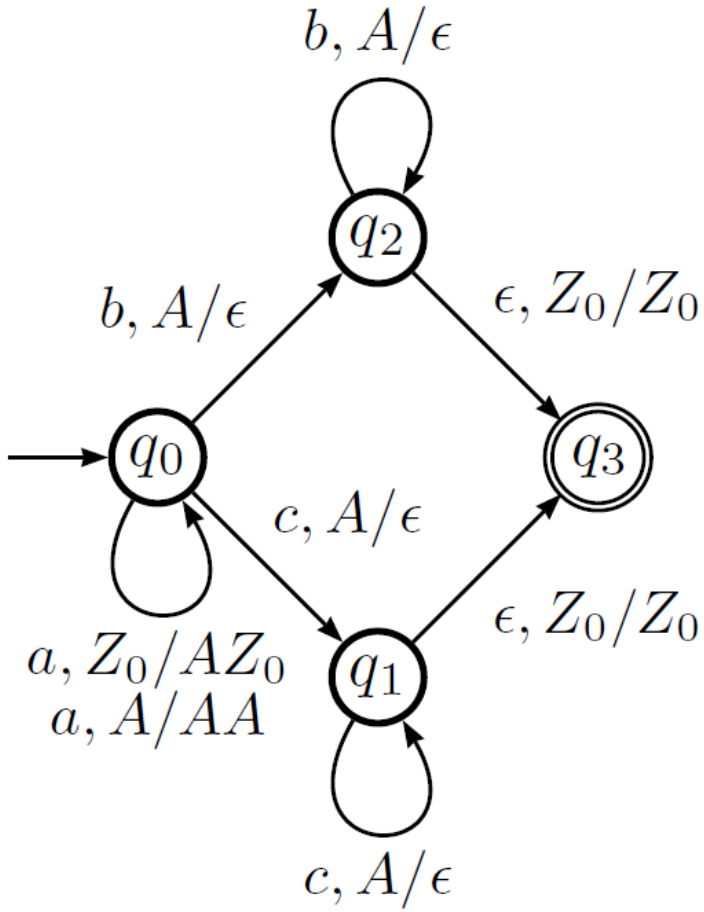
Construct a single tape TM M, such that $L(M) = \{a^n b^n \mid n > 0\}$

Construct an automaton A, such that

$$L(A) = \{ x \in \{a, b, c\}^+ \mid \#_a(x) = \#_b(x) \vee \#_a(x) = \#_c(x) \}$$

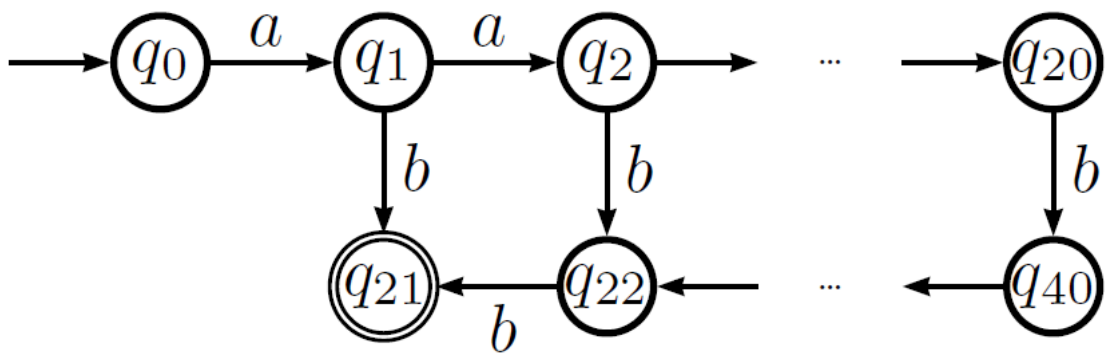
$$L_2 = \{a^n b^n \mid n > 0\} \cup \{a^n c^n \mid n > 0\}$$

A (deterministic) PDA suffices (deterministic, despite union of languages)



$$L_3 = \{a^n b^n \mid 0 < n < 21\}$$

Counting up to a bounded value: a FSA suffices

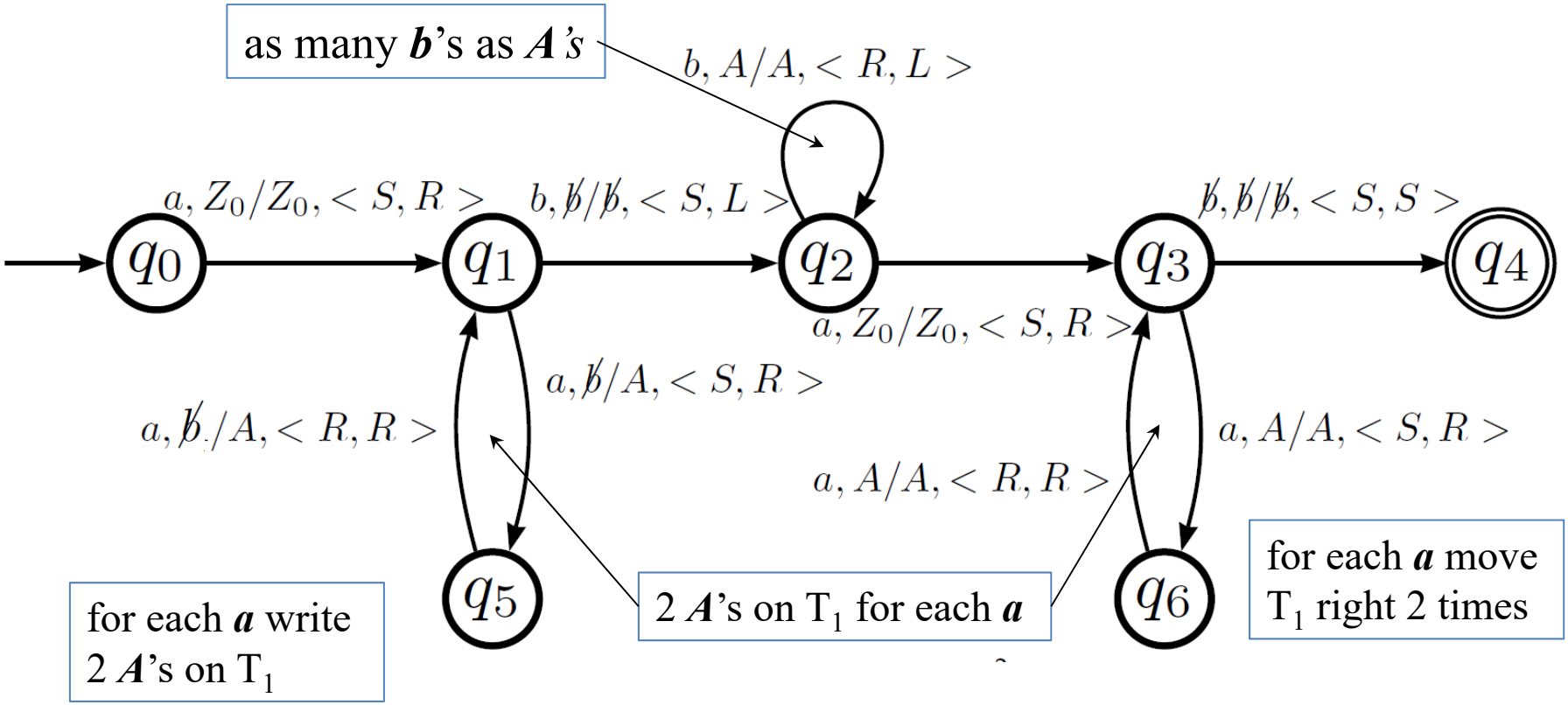


$$L_5 = \{a^n b^{2n} a^n \mid n > 0\}$$

Can't use a PDA: it could not count separately the ***b***'s and the ***a***'s

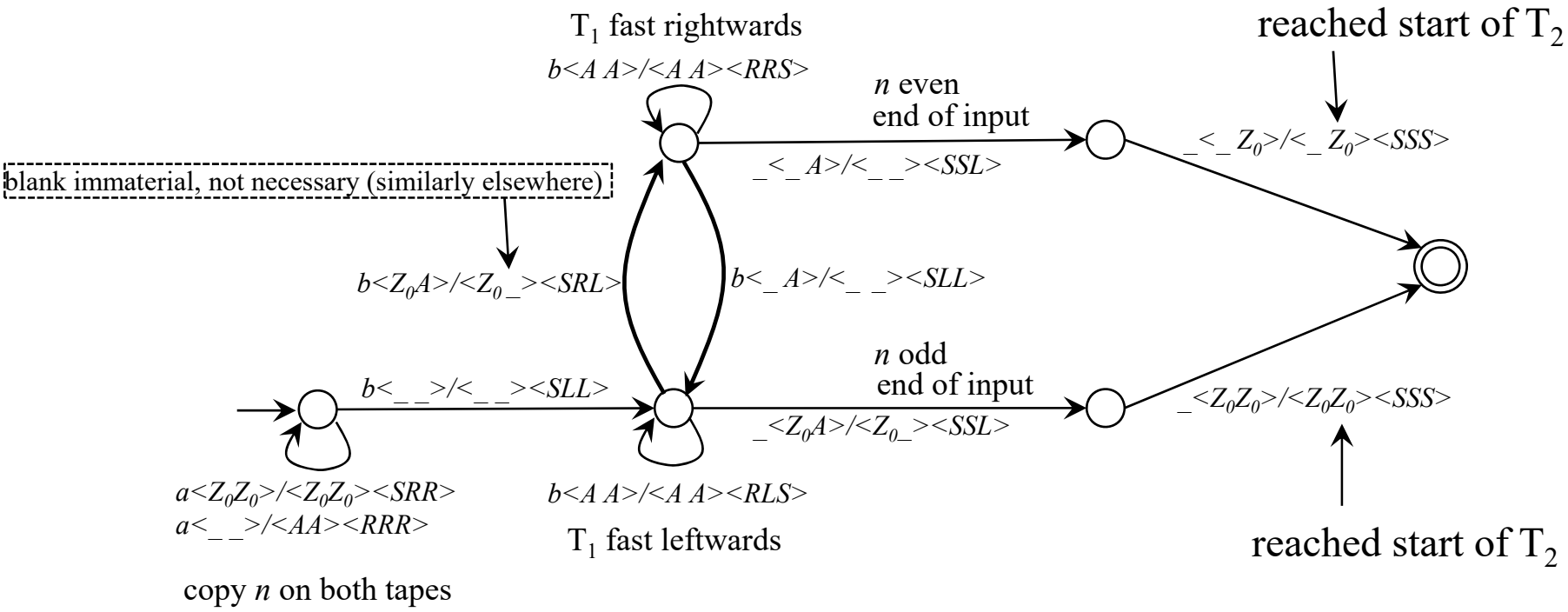
a PDA that pushes 3 symbols for each a : why doesn't it work?

(it accepts a ***superset*** of L_5)

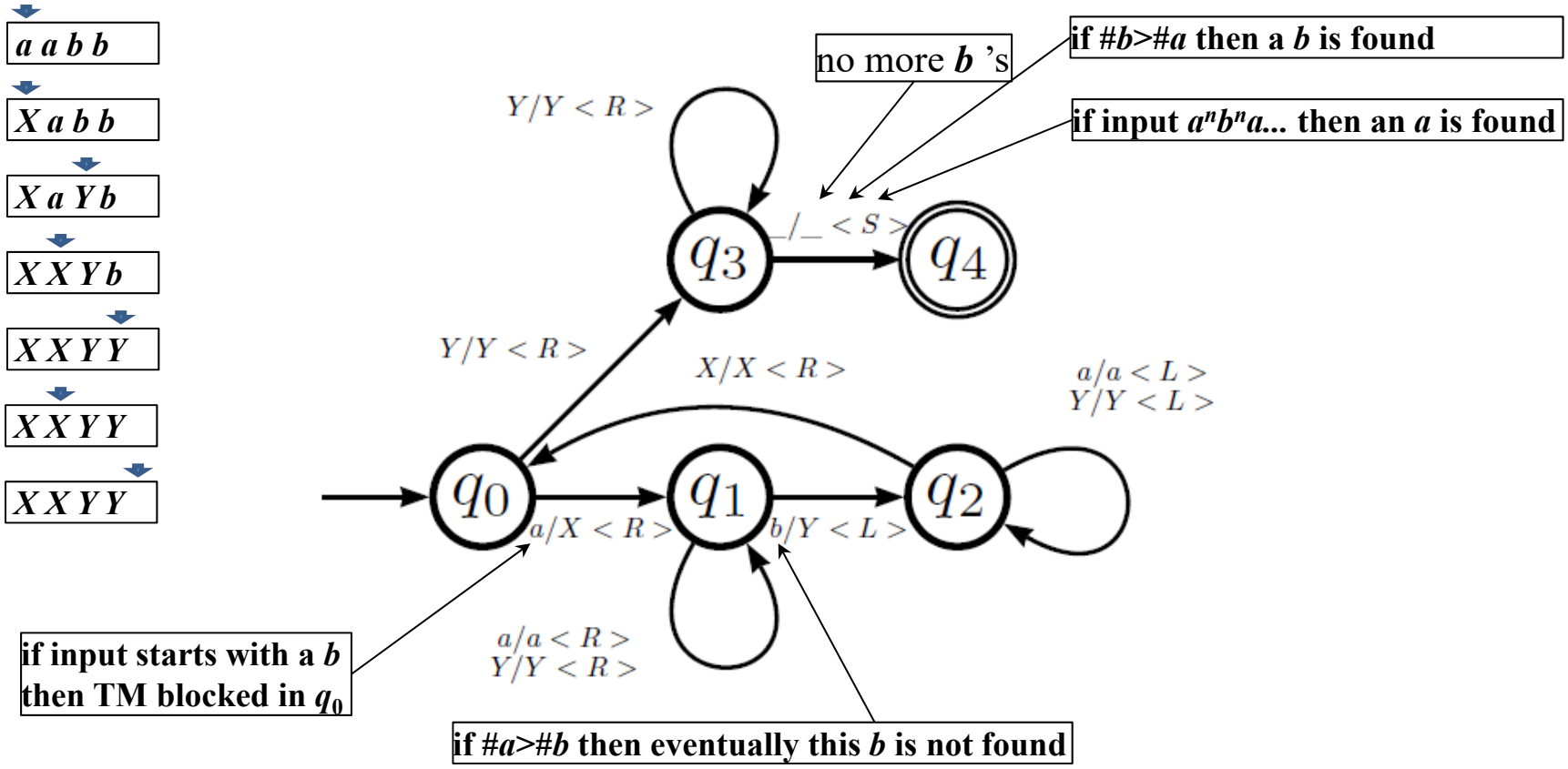


Construct an automaton A, such that $L(A) = \{a^n b^{n^2} \mid n > 0\}$

Two memory tapes: T_1 «fast» counter, T_2 «slow» counter



Construct a single tape TM M, such that $L(M) = \{a^n b^n \mid n > 0\}$

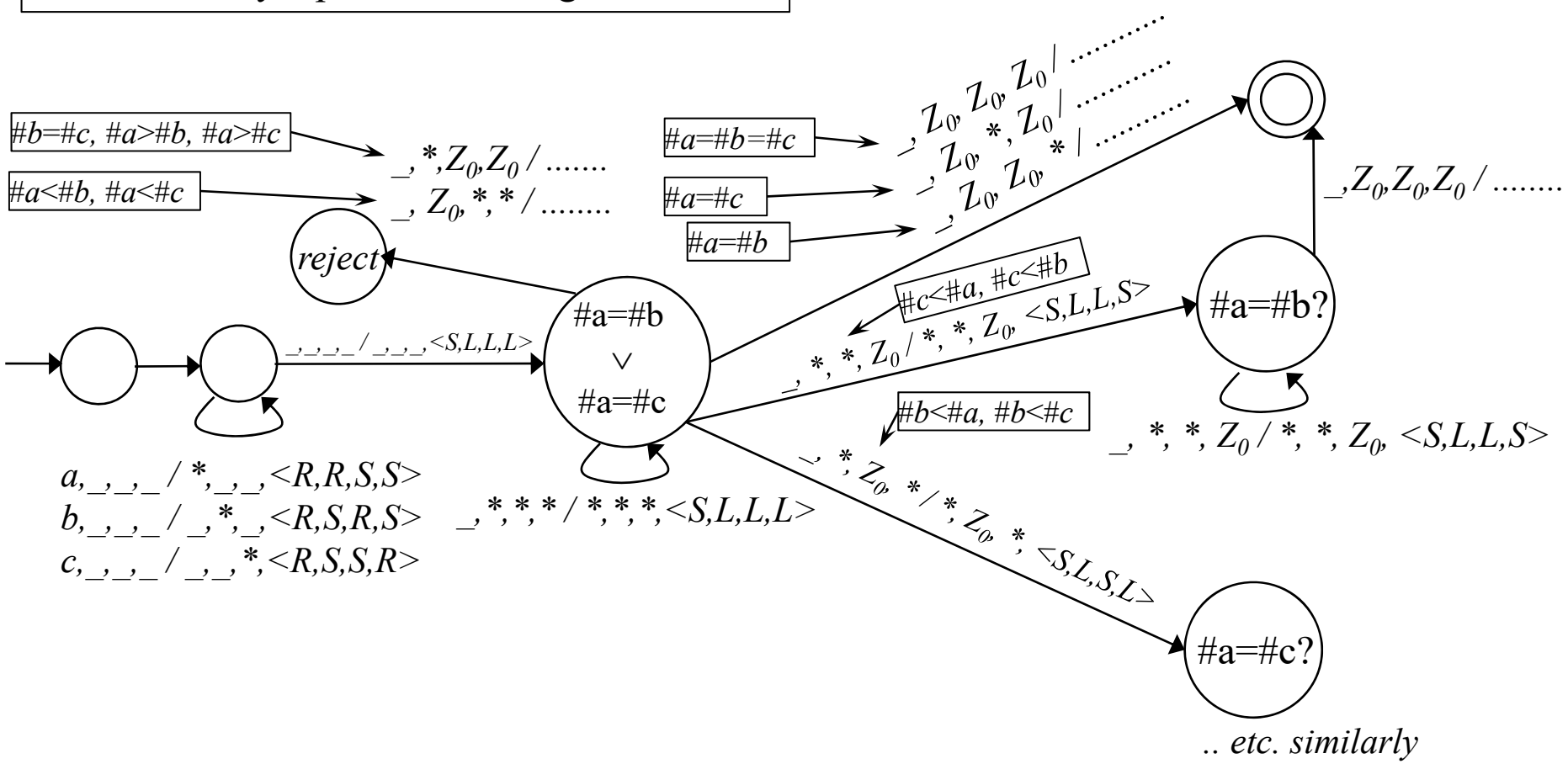


Number of moves (time complexity) ?

$$L(A) = \{\, x \in \{a, b, c\}^+ \mid \#_a(x)=\#_b(x) \vee \#_a(x)=\#_c(x) \,\}$$

Just a sketch of the solution

Three memory tapes for counting *a*'s, *b*'s, *c*'s



Nondeterministic Models

Construct a (possibly nondeterministic) automaton N suh that

$L(N) = \{ x \in \{0, 1\}^* \mid$ the second last symbol of x is a 0 $\}$

Derive an equivalent deterministic automaton

$L(A) = \{x \in \{0,1\}^* | x$ contains a pair of 0s separated by a string with length $n = 4i,\ i \geq 0\}$

Construct an automaton A such that $L(A) = \{ww^R \mid w \in \{a,b\}^+\}$

Design a minimal-power automaton that accepts the language

$\{ a^n b^m \mid n > 0$ and $n \leq m \leq 2n \}$

Construct a nondeterministic automaton N s.t. $L(N) = \{ x \in \{0, 1\}^* \mid$ the second last symbol of x is a 0 $\}$

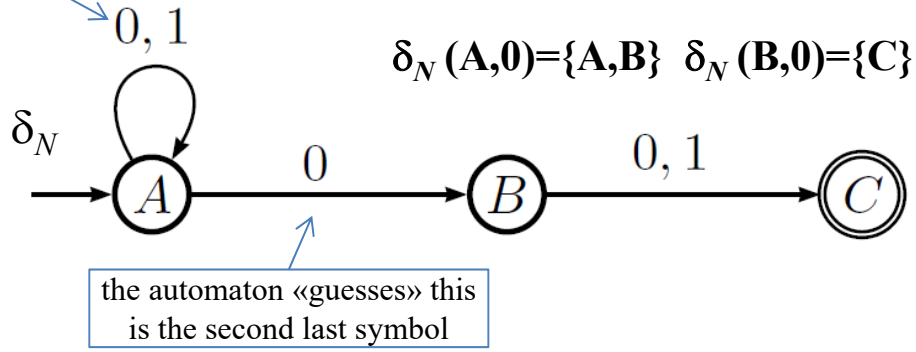
Derive an equivalent deterministic automaton D

$\delta_D : \wp(Q) \times I \to \wp(Q)$ $\delta_D(S, a) = \bigcup_{q \in S} \delta_N(q, a)$

the automaton «waits»

$\delta_N : Q \times I \to \wp(Q)$

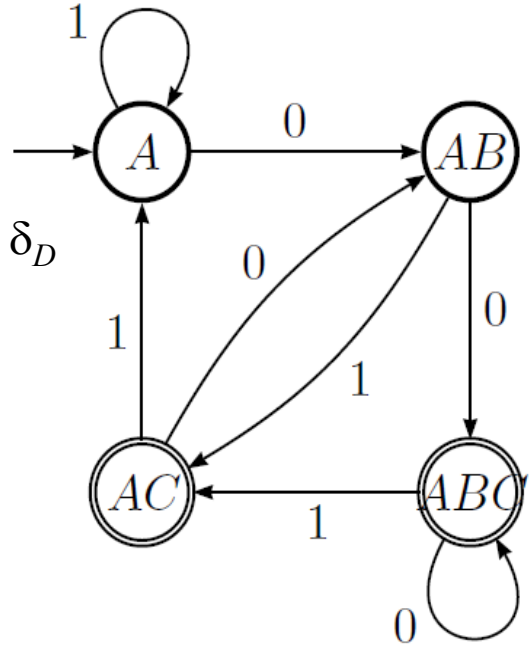
δ_N (A,0)={A,B} δ_N (B,0)={C}



the automaton «guesses» this is the second last symbol

δ_D({A},0)=δ_N(A,0) ={A,B}
δ_D({A},1)=δ_N(A,1) ={A}

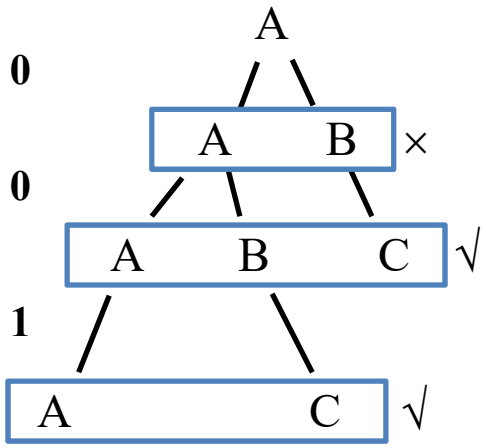
δ_D ({A,B},0)=δ_N (A,0)∪δ_N (B,0)=
= {A,B}∪{C}={A,B,C}



computations for string 001: $A \xrightarrow{0} A \xrightarrow{0} A \xrightarrow{1} A\times$

$A \xrightarrow{0} A \xrightarrow{0} B \xrightarrow{1} C\ \surd$

$A \xrightarrow{0} B \xrightarrow{0} C \times$



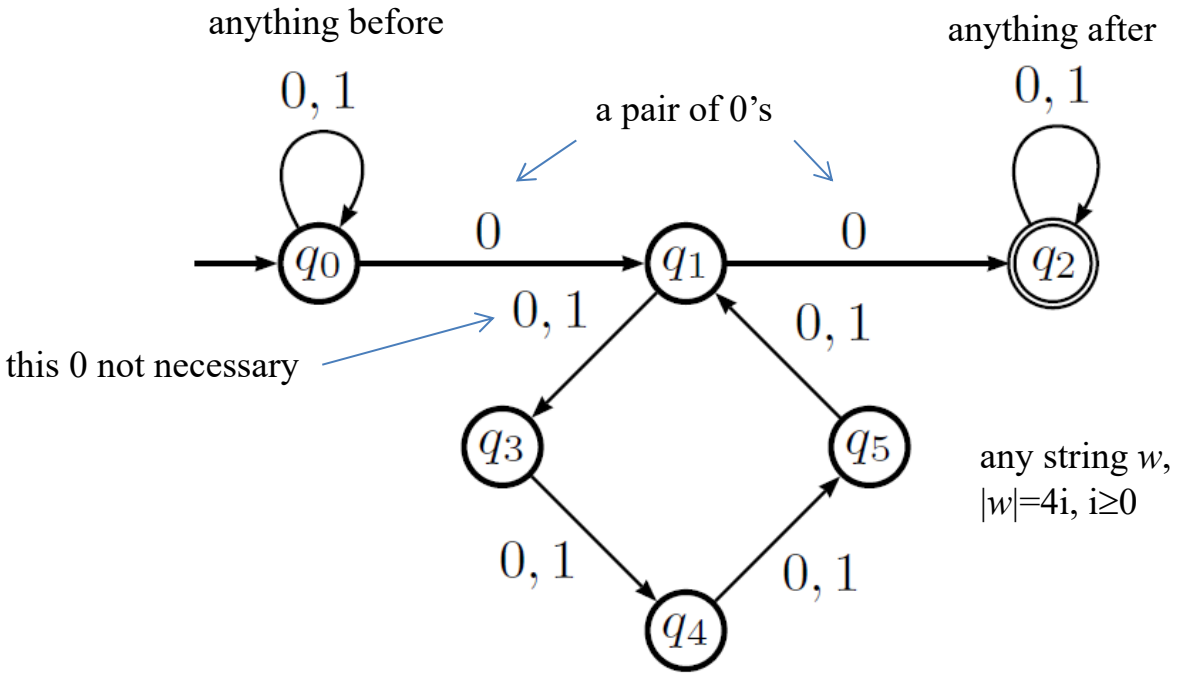
Computation tree on input 001: a collective representation of traces horizontal slices are states of the deterministic machine

δ_D ({A,B,C},1)=δ_N(A,1)∪δ_N(B,1)∪δ_N(C,1)=
{A}∪{C}∪∅={A,C}

δ_D({A,C},0)=δ_N(A,0)∪δ_N(C,0)= {A,B}∪∅={A,B}

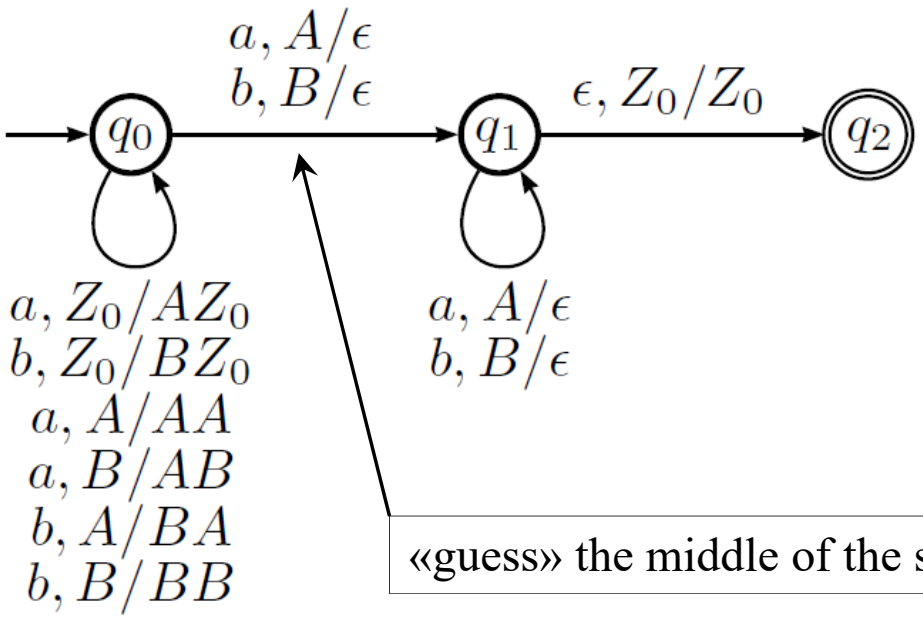
δ_D({A,C},1)=δ_N(A,1)∪δ_N(C,1)= {A}∪∅={A}

$$L(A) = \{x \in \{0,1\}^* | x \text{ contains a pair of 0s separated by a string with length } n = 4i,\ i \geq 0\}$$



Construct an automaton A such that $L(A) = \{ww^R \mid w \in \{a,b\}^+\}$

It is the archetypal nondeterministic context free language



Two computations for string *abba*

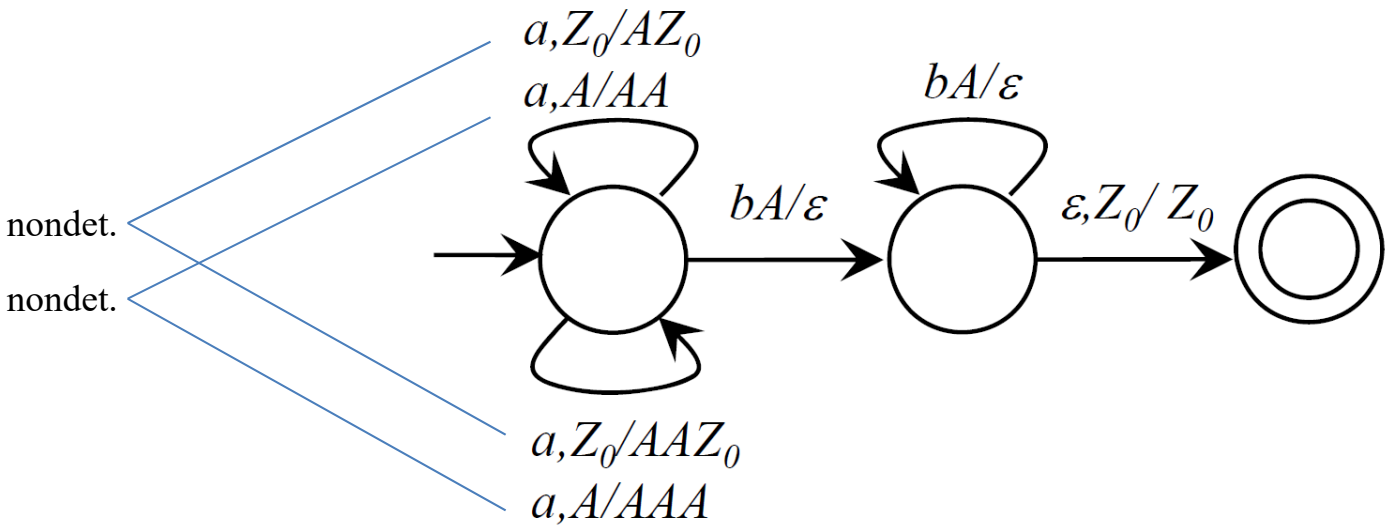
$\langle q_0 \text{ abba } Z_0\rangle \vdash \langle q_0 \text{ bba } AZ_0\rangle \vdash \langle q_0 \text{ ba } BAZ_0\rangle \vdash \langle q_0 \text{ a } BBAZ_0\rangle \vdash \langle q_0 \text{ } \varepsilon \text{ } ABBAZ_0\rangle$ **reject**

$\langle q_0 \text{ abba } Z_0\rangle \vdash \langle q_0 \text{ bba } AZ_0\rangle \vdash \langle q_0 \text{ ba } BAZ_0\rangle \vdash \langle q_1 \text{ a } AZ_0\rangle \vdash \langle q_1 \text{ } \varepsilon \text{ } Z_0\rangle \vdash \langle q_2 \text{ } \varepsilon \text{ } Z_0\rangle$ **accept**

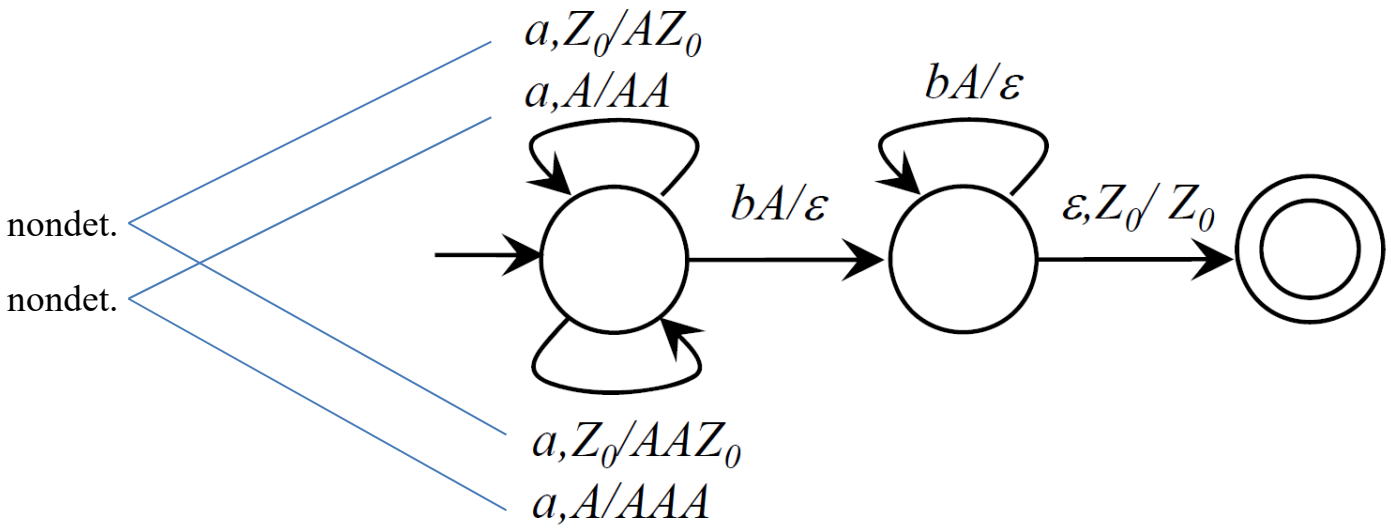
Design a minimal-power automaton that accepts the language

$L=\{\ a^n b^m \mid n>0 \text{ and } n \le m \le 2n\ \}$ e.g., $aabbb$, $a^3b^5 \in L$

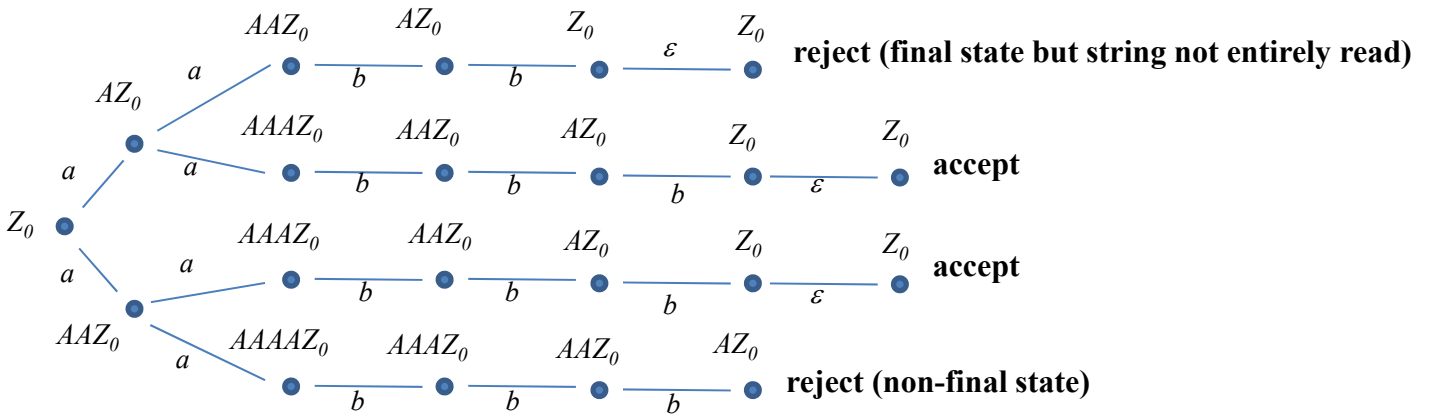
A nondeterministic PDA suffices:
for every *a* in the input, it pushes in the stack both one or two *A*'s
(by means of two distinct nondeterministic moves, originating two distinct computations),
and then pops one A for every *b*



$L=\{\ a^n\, b^m\ \mid n>0 \text{ and } n \le m \le 2n\ \}$ $\quad aabbb \in L$



How many computations for the string $aabbb \in L$?
Two nondeterministic moves for each *a* read from the inut $\Rightarrow$ 4 computations



FORMAL GRAMMARS

Write a grammar G such that $L(G) = \{a^n b^n c^m a^m \mid n \geq 0, m \geq 1\}$

Write a grammar G such that $L(G) = \{x \in \{a, b\}^* \mid \#_a(x) = \#_b(x)\}$

Write a grammar G such that $L(G) = \{a^{2^i} \mid i \geq 0\}$

Design a grammar that generates, and an automaton that accepts,

$$L1 = \{\ w \in \{a, b\}^* \mid\ (\#_a(w) + 2 \cdot \#_b(w))\ \text{MOD}\ 5 = 0\ \}$$

For instance, $abb \in L1$, $\varepsilon \in L1$, $aba \notin L1$

Design a grammar that generates

$$L2 = \{\ a^n b^m c^k\ \mid\ m, n, k \geq 0\ \text{ and }\ m = n + k\ \}$$

For instace, $a^2b^5c^3 \in L2$, $\varepsilon \in L2$, $a^2b^4c^3 \notin L2$

Write a grammar G such that $L(G) = \{a^n b^n c^m a^m \mid n \geq 0, m \geq 1\}$

$$S \rightarrow S_1 S_2$$

$$S_1 \rightarrow \epsilon \mid a S_1 b$$

$$S_2 \rightarrow ca \mid c S_2 a$$

Variant: design a grammar that generates

$$L = \{ a^n b^m c^m d^n \mid n \geq 0 \wedge m \geq 1 \}$$

Write a grammar G such that $L(G) = \{x \in \{a,b\}^* \mid \#_a(x) = \#_b(x)\}$

$$S \to \epsilon \mid aSbS \mid bSaS$$

Correctness of G: $L(G) = L$ where $L = \{ x \in \{a, b\}^* \mid \#_a(x) = \#_b(x) \}$

we must prove the two inclusions: $L(G) \subseteq L$ and $L \subseteq L(G)$

grammar for $L = \{ x \in \{a, b\}^* \mid \#_a(x) = \#_b(x) \}$ $S \rightarrow \epsilon \mid aSbS \mid bSaS$

Part 1: $L(G) \subseteq L$ i.e., $S \Rightarrow^* x$ implies $x \in L$
proof by induction on the length n of the derivation of x
Base: n=1, $S \Rightarrow \varepsilon$, $\varepsilon \in L$
Ind. step: if $S \Rightarrow^* a\,x\,b\,y$ then by the ind.hyp. $x \in L$ and $y \in L$ hence $a\,x\,b\,y \in L$
if $S \Rightarrow^* b\,x\,a\,y$ then by the ind.hyp. $x \in L$ and $y \in L$ hence $b\,x\,a\,y \in L$

Part 2: $L \subseteq L(G)$ i.e., $x \in L$ implies $S \Rightarrow^* x$
If $x \in L$ then $|x| = 2 \cdot n$ for some $n \in \mathbb{N}$
proof by induction on n
Base: n=0, x=ε, $S \Rightarrow \varepsilon$
Ind. step: if $n = k > 0$ then $x = a\,y\,b\,z$ or $x = b\,y\,a\,z$
for some $y, z \in L$ s.t. $|y| = 2 \cdot h$, $h < k$, and $|z| = 2 \cdot j$, $j < k$
then by the ind.hyp. $S \Rightarrow^* y$ and $S \Rightarrow^* z$
hence either $x = a\,y\,b\,z$ and $S \Rightarrow a\,S\,b\,S \Rightarrow^* a\,y\,b\,z$
or $x = b\,y\,a\,z$ and $S \Rightarrow b\,S\,a\,S \Rightarrow^* b\,y\,a\,z$

QED

Write a grammar G such that $L(G) = \{a^{2^i} \mid i \geq 0\}$

$$S \to XAY$$

$$X \to XC \mid \epsilon \qquad \boldsymbol{X \overset{*}{\Rightarrow} C^n}$$

$$CA \to AAC \qquad \textbf{\#}\boldsymbol{A}\textbf{ doubled for every }\boldsymbol{C}$$

$$CY \to Y \mid \epsilon \qquad \boldsymbol{C}\textbf{'s disappear}$$

$$A \to a$$

$$Y \to \epsilon \qquad \boldsymbol{Y}\textbf{ disappears}$$

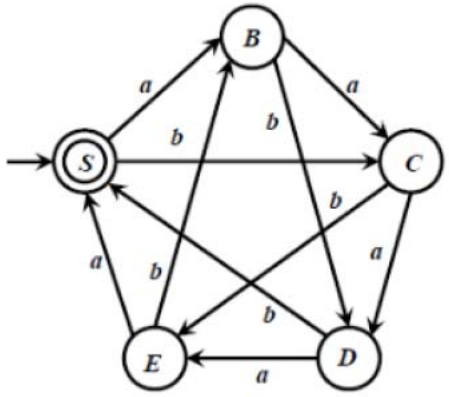
$\boldsymbol{S \overset{*}{\Rightarrow} a^4}$

$\boldsymbol{S \Rightarrow \underline{X}AY \Rightarrow \underline{X}CAY \Rightarrow \underline{X}CCAY \Rightarrow C\underline{CA}Y \Rightarrow CAA\underline{C}Y \Rightarrow CAAY \Rightarrow AA\underline{CA}Y}$
$\boldsymbol{\Rightarrow AAAACY \overset{*}{\Rightarrow} a^4}$

Design an automaton that accepts, and a grammar that generates

$$L1 = \{\ w \in \{a, b\}^* \mid (\#_a(w) + 2 \cdot \#_b(w))\ \text{MOD}\ 5 = 0\ \}$$

$abb \in L1$, $\varepsilon \in L1$, $aba \notin L1$



Computation accepting *abb*: $S \xrightarrow{a} B \xrightarrow{b} D \xrightarrow{b} S$

$S \rightarrow \varepsilon$
$S \rightarrow a\ B \mid b\ C$
$B \rightarrow a\ C \mid b\ D$
$C \rightarrow a\ D \mid b\ E$
$D \rightarrow a\ E \mid b\ S$
$E \rightarrow a\ S \mid b\ B$

Derivation generating *abb*: $S \Rightarrow aB \Rightarrow abD \Rightarrow abbS \Rightarrow abb$

Design a grammar that generates

$$L2 = \{\ a^n b^m c^k \ \mid\ m, n, k \geq 0 \quad \text{and} \quad m = n + k\ \}$$

$a^2b^5c^3 \in L2$, $\varepsilon \in L2$, $a^2b^4c^3 \notin L2$

REMARK: $a^n\ b^m\ c^k$ = … ($m = n + k$) … = $a^n\ b^n\ b^k\ c^k$

$$S \rightarrow A\ C$$

$$A \rightarrow a\ A\ b \mid \varepsilon$$

$$C \rightarrow b\ C\ c \mid \varepsilon$$

BTW: what's wrong with this solution?

$S \rightarrow a\ S\ b$

$S \rightarrow b\ S\ c$

$S \rightarrow \varepsilon$

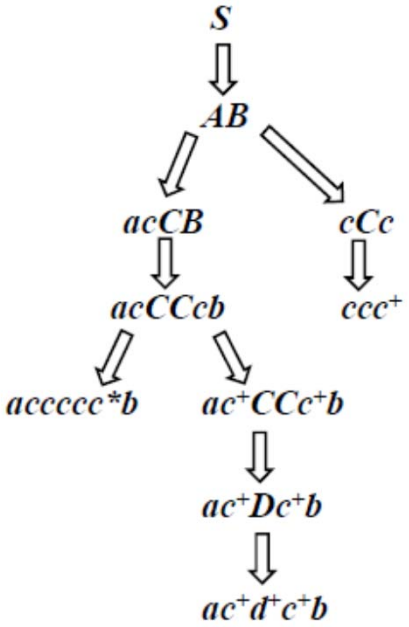
Consider the following grammar G:

$S \rightarrow A\,B$
$A \rightarrow a\,c\,C$
$B \rightarrow C\,c\,b$
$C \rightarrow c\,C \mid C\,c \mid c$
$C\,C \rightarrow D$
$D \rightarrow d\,D \mid d$
$A\,B \rightarrow c\,C\,c$

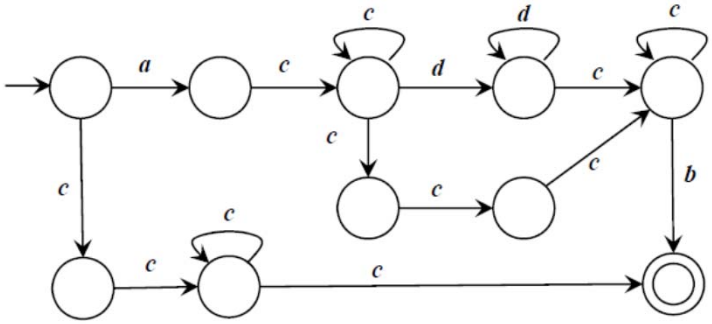
a) Identify the **smallest *language* family** of $L(G)$; provide a precise and concise description of $L(G)$.

b) Design an automaton, of the least powerful type, that accepts $L(G)$.

a. **$L(G)$ is the regular language $\{accccc^*b\} \cup \{ac^+d^+c^+b\} \cup \{ccc^+\}$**



b) A (possibly nondeterministic) finite state automaton suffices.



REMARK (moral of the story): a general grammar may generate a language that belongs to a simpler (i.e., more constrained) family (e.g., a regular language)

MFO and MSO Logic to define Languages

Write a FOL[S] formula that defines language containing words such that every letter “e” is preceded by a letter “s” only with letters “p” between them.

Write a FOL[S] formula that defines language containing words where every letter “b” is preceded by letter “a” in one of the three previous positions. VARIANT: “...in one and only one of the three previous positions”.

Write a FOL[S] formula that defines language containing words where in between any two “a”-s there is at least one “b”.

Write an appropriate formula over the word structure that defines language containing words such that between every two “a”-s there are odd number of letters.

Write an appropriate formula over the word structure that defines language containing words in which “a”-s, “b”-s and “c”-s occur at most once.

Write an appropriate formula over the word structure that defines language containing words in which for every 10 consecutive letters there is at most 2 "a"-s.

Write an appropriate formula over the word structure that defines the language containing words such that the number of ***b***'s is a multiple of 3, and each ***b*** is between letters ***a*** and ***c*** (not necessarily adjacent to ***b***), and between every ***a*** and ***c*** there is exactly one ***b***.

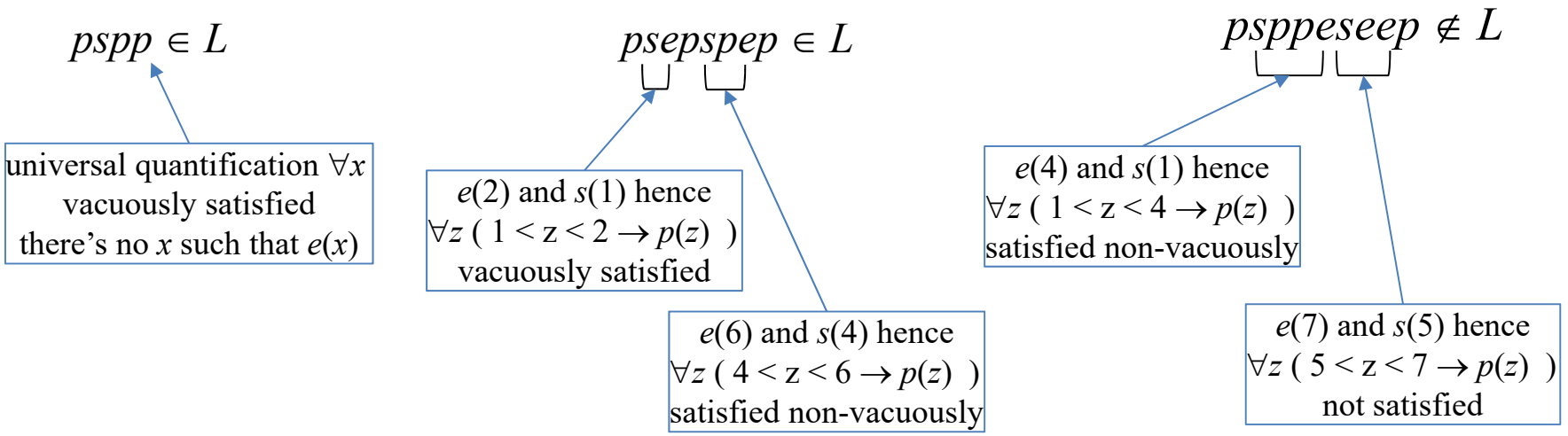
Write an appropriate formula over the word structure that defines language containing words such that "a" appears in continuous sequences of at least 4.

Write a formula that defines a language $\{a^n r^{n \mod 3} \mid n > 0\}$, use the standard word structure.

Write a FOL[S] formula that defines language containing words such that every letter "e" is preceded by a letter "s" only with letters "p" between them.

$$I = \{ e, p, s \}$$

$$\forall x.(e(x) \rightarrow \exists y.(y < x \wedge s(y) \wedge \forall z.(y < z < x \rightarrow p(z))))$$



Write a FOL[S] formula that defines language containing words where every letter "b" is preceded by letter "a" in one of the three previous positions. VARIANT: "...in one and only one of the three previous positions".

$$\forall x.(b(x) \rightarrow \exists y.\begin{pmatrix} (y = x - 3 \wedge a(y)) \vee \\ (y = x - 2 \wedge a(y)) \vee \\ (y = x - 1 \wedge a(y)) \end{pmatrix})$$

Write a FOL[S] formula that defines language containing words where in between any two "a"-s there is at least one "b".

$$\forall x, y.(x < y \wedge a(x) \wedge a(y) \rightarrow \exists z.(x < z < y \wedge b(z)))$$

Write an appropriate formula over the word structure that defines language containing words such that between every two "a"-s there are odd number of letters.

$$\forall x, y.(x < y \wedge a(x) \wedge a(y) \rightarrow \exists X.(X(x) \wedge X(y) \wedge \forall z.(\neg last(z) \rightarrow (X(z) \leftrightarrow \neg X(z+1)))))$$

Write an appropriate formula over the word structure that defines language containing words in which "a"-s, "b"-s and "c"-s occur at most once.

$$\forall x, y.(a(x) \wedge a(y) \rightarrow x = y) \wedge$$
$$\forall x, y.(b(x) \wedge b(y) \rightarrow x = y) \wedge$$
$$\forall x, y.(c(x) \wedge c(y) \rightarrow x = y)$$

Equivalent alternative: $\neg\exists x\, \exists y\, (\, a(x) \wedge a(y) \wedge x \neq y\,) \wedge \ldots$

Write an appropriate formula over the word structure that defines language containing words in which for every 10 consecutive letters there is at most 2 "a"-s.

$$\forall x, y.(x = y + 9 \rightarrow \neg\exists z_1, z_2, z_3.(y \leqslant z_1 < z_2 < z_3 \leqslant x \wedge a(z_1) \wedge a(z_2) \wedge a(z_3)))$$

Consider the two language expressions

$$L_1 = ((a \mid b) (c \mid d))^+ \quad \text{and} \quad L_2 = ((a \mid b) (b \mid c))^+$$

Notice that the two languages are not disjointed and none of the two is included in the other; in fact, the following string inclusions hold (where the operator '-' here denotes set difference):

$$x_1 = ad \in L_1 - L_2, \quad x_{12} = ac \in L_1 \cap L_2, \quad x_2 = ab \in L_2 - L_1.$$

1. Write a monadic logic sentence φ_1 (first-order, or second-order only if necessary) such that $L_1 = L(\varphi_1)$. Verify that $x_2 = ab \notin L(\varphi_1)$ and explain in logical terms the reason why $ab \not\models \varphi_1$.

2. Write a monadic logic sentence φ_2 (first-order, or second-order only if necessary) such that $L_2 = L(\varphi_2)$. Verify that $x_1 = ad \notin L(\varphi_2)$ and $y = abb \notin L(\varphi_2)$ and explain in logical terms the reason why $ad \not\models \varphi_2$ and $abb \not\models \varphi_2$.

$L_1 = ((a \mid b) (c \mid d))^+$ **and** $L_2 = ((a \mid b) (b \mid c))^+$

$x_1 = ad \in L_1 - L_2, \quad x_{12} = ac \in L_1 \cap L_2, \quad x_2 = ab \in L_2 - L_1$

φ_1 **such that** $L_1 = L(\varphi_1)$

$$\forall x \, (\; (\mathit{first}(x) \rightarrow a(x) \vee b(x)) \wedge$$
$$(\mathit{last}(x) \rightarrow c(x) \vee d(x)) \wedge$$
$$(a(x) \vee b(x) \rightarrow c(x+1) \vee d(x+1)) \wedge$$
$$((c(x) \vee d(x)) \wedge \neg \mathit{last}(x) \rightarrow a(x+1) \vee b(x+1))$$
$$)$$

$x_2 = ab \notin L(\varphi_1)$ **because for this string** $b(1)$ **holds, therefore for** x**=1 the subformula (** $\mathit{last}(x) \rightarrow c(x) \vee d(x)$ **) evaluates to false.**

In addition: the standard clause for ruling out the empty string ε**:** $\exists x \, (a(x) \vee \neg \, a(x))$

$L_1 = ((a \mid b) (c \mid d))^+$ and $L_2 = ((a \mid b) (b \mid c))^+$

$x_1 = ad \in L_1 - L_2$, $x_{12} = ac \in L_1 \cap L_2$, $x_2 = ab \in L_2 - L_1$

φ_2 such that $L_2 = L(\varphi_2)$

$$\exists X \, \forall x \, (\neg X(0) \wedge (\neg last(x) \rightarrow (X(x) \leftrightarrow \neg X(x+1))) \wedge$$
$$(\neg X(x) \rightarrow a(x) \vee b(x)) \wedge$$
$$(X(x) \rightarrow b(x) \vee c(x)) \wedge$$
$$(last(x) \rightarrow X(x))$$
$$)$$

$x_1 = ad \notin L(\varphi_2)$ because $d(1)$ holds, therefore for x=1 the subformula $(X(x) \rightarrow b(x) \vee c(x))$ evaluates to false.

$y = abb \notin L(\varphi_2)$ because $|y|$=3, therefore for x=2
the subformula $(last(x) \rightarrow X(x))$ evaluates to false.

Suggested further exercise: design automata accepting
L_1, L_2, $L_1 \cap L_2$, $L_1 \cup L_2$

Write an appropriate formula over the word structure that defines the language containing words such that each ***b*** is between ***a*** followed by ***c*** (not necessarily adjacent to ***b***), and the number of ***b***'s is a multiple of 3, and between every ***a*** and ***c*** there is exactly one ***b***.

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

each ***b*** is between ***a*** followed by ***c***

$$\forall x.(b(x) \rightarrow \exists u, v.(u < x \wedge x < v \wedge a(u) \wedge c(v))) \wedge$$
$$\forall x, y.(x < y \wedge a(x) \wedge c(y) \rightarrow \exists z.(x < z < y \wedge b(z) \wedge \forall k.(x < k < y \wedge z \neq k \rightarrow \neg b(k)))) \wedge$$
$$\exists N_0, N_1, N_2.(\forall x.(\neg(N_0(x) \wedge N_1(x)) \wedge \neg(N_0(x) \wedge N_2(x)) \wedge \neg(N_1(x) \wedge N_2(x))) \wedge$$

mutual excl.

$$(first(x) \wedge b(x) \rightarrow N_1(x)) \wedge (first(x) \wedge \neg b(x) \rightarrow N_0(x)) \wedge$$

value at first pos.

$$\forall y.(S(x, y) \rightarrow (N_0(x) \wedge b(y) \rightarrow N_1(y)) \wedge$$
$$(N_0(x) \wedge \neg b(y) \rightarrow N_0(y)) \wedge$$
$$(N_1(x) \wedge b(y) \rightarrow N_2(y)) \wedge$$
$$(N_1(x) \wedge \neg b(y) \rightarrow N_1(y)) \wedge$$
$$(N_2(x) \wedge b(y) \rightarrow N_0(y)) \wedge$$
$$(N_2(x) \wedge \neg b(y) \rightarrow N_2(y))) \wedge$$
$$(last(x) \rightarrow N_0(x)))$$

	a b c b a b a b c $\notin L$
N_0	T F F F F T T F F
N_1	F T T F F F F T T
N_2	F F F T T F F F F

	b c a b c a b c $\in L$
N_0	F F F F F F T T
N_1	T T T F F F F F
N_2	F F F T T T F F

$N_0(x)$ (resp. $N_1(x)$, $N_2(x)$) means no. of ***b***'s up to position ***x*** included **MOD 3 = 0** (resp **=1**, **=2**)

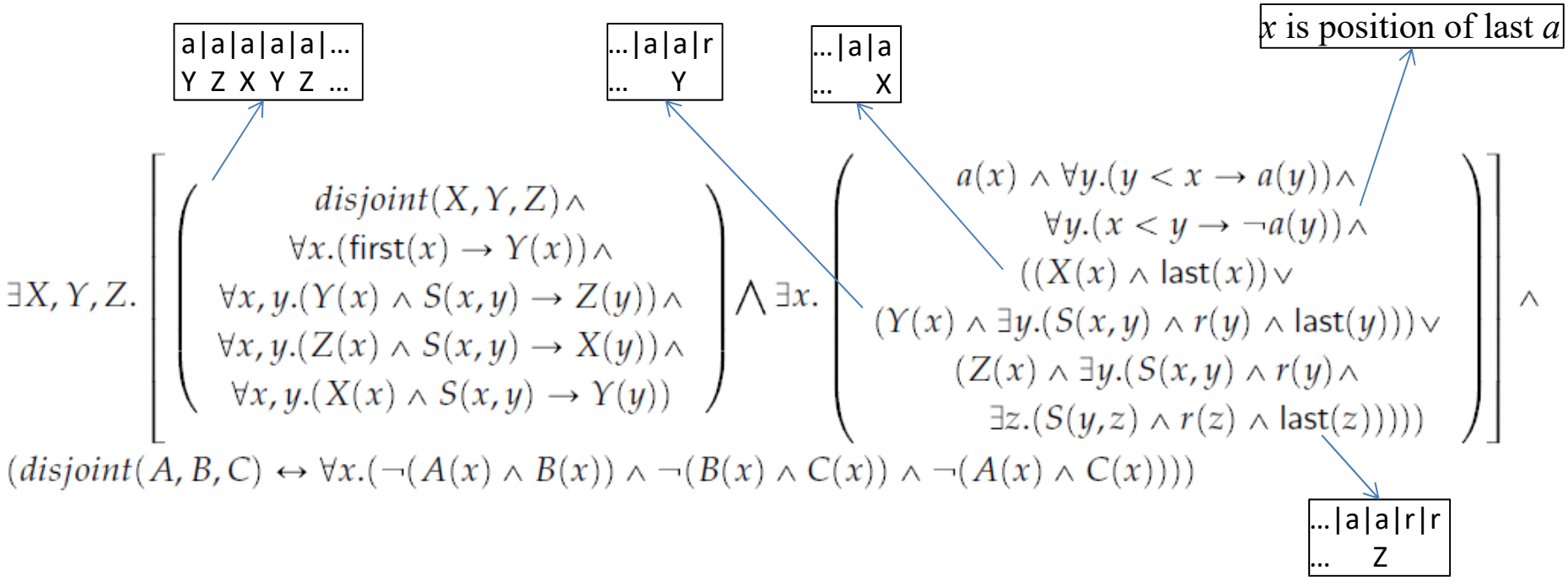
Write an appropriate formula over the word structure that defines language containing words such that "a" appears in continuous sequences of at least 4.

$$\forall x.(a(x) \wedge \neg\exists y.(y = x - 1 \wedge a(y)) \rightarrow \exists y.(y = x + 3 \wedge a(y) \wedge a(x + 1) \wedge a(x + 2)))$$

or, more simply,

$$\forall x\ (\ a(x) \wedge \neg a(x\text{-}1)\ \ \rightarrow\ \ a(x{+}1) \wedge a(x{+}2) \wedge a(x{+}3)\ \ )$$

Write a formula that defines a language $\{a^n r^{n \bmod 3} \mid n > 0\}$, use the standard word structure.



X*(*x*)** (resp. ***Y*(*x*)**, ***Z*(*x*)**) means no. of ***a's up to position ***x*** included **MOD 3 = 0** (resp. **=1**, **=2**)

Logic to specify program requirements

(+ minimal background on Computability Theory)

1

Write pre- and post- conditions for a program P that implements a function $f(x, y, z) = 1$ if a parameter can be expressed as a product of the other two and $f(x, y, z) = 0$ otherwise. Parameters x, y and z are integer numbers.

2

Specify, by means of suitable pre- and post-conditions, a subprogram $P(a, n, v)$ where: a and n are input parameters representing, respectively, an array of integer numbers and the number of its elements (assumed to be ≥3); v is an output Boolean parameter. The parameter v must be set equal to **true** if and only if the elements of array a are placed in increasing order from the first position up to a certain intermediate position and in decreasing order from that point on; both the increasing and decreasing array segments must not be empty. For instance: if a=[1 3 7 5 2] then v=**true**; if a=[1 3 7] then v=**false**; if a=[5 4 6 8] then v=**false**. Please assume that the indexing in the array starts from 0.

3

Consider the function INSREV, which takes as input three arguments: a nonempty array a of integers sorted either in increasing or decreasing order, a positive integer n representing the number of the elements of a, and an integer e, to be inserted into a; it is assumed that e is not an element of a.

The function INSREV outputs an array b, that represents the result of inserting e into a at a non-deterministic position; no other element is inserted into a, and no element is canceled, but the order of the elements previously present in a is changed as follows. All elements before the position where e is inserted are placed in reverse order, while the elements following that position are kept in the same order as in a. For example, if a=[0,1,2,3,4,5,6] and e=8 then b may be the following array: [3,2,1,0,8,4,5,6].

Specify formally, using mathematical logic formulas with the usual predicates and relations on the integers, the pre- and post-conditions for the function INSREV; assume that array indexing starts from 0.

4

A procedure *sumArr*(*a*, *b*, *dim*, *p*) has 3 input parameters *a*, *dim*, *p*, and one output parameter *b*. Parameters *a* and *b* are arrays of integers having *dim* elements, and it can be assumed that *dim*≥3, while parameter *p* can be assumed to be equal to either 0 or 1.

If *p*=0 then after the execution of *sumArr* the first two elements of *b* are equal to the elements of *a* in the same positions, and every element from the third position on

- is equal to the sum of the two preceding elements if the second preceding element is greater or equal to the immediately preceding element;
- is equal to the product of the two preceding elements if the second preceding element is less than the immediately preceding element.

If *p*=1 then after the execution of *sumArr* every element of *b* is equal to the sum all elements of *a* having position less than or equal to it.

For instance, if *a*=[7 3 5 6 6 9 5], *dim*=7, *p*=0 then *b*=[7 3 10 15 30 12 54], while

if *a*=[7 3 5 6 6 9 5], *dim*=7, *p*=1 then *b*=[7 10 15 21 27 36 41].

Specify the procedure *sumArr* by means of a pre-condition and a post-conditions expressed in first-order logic with the usual relational and arithmetic operators typical of common programming languages (NB: no use of the summation operator like "$\sum_{\ldots}^{\ldots}$..." is allowed).

5

A procedure P receives as input a natural number $n > 1$ and two arrays, *in* and *out*, both cointaining $2 \cdot n$ natural numbers (the index of both arrays is in the range $1 \,..\, 2 \cdot n$).

As a preliminary, warm-up question, specify the predicates *odd*(n) and *even*(n), which mean than the natural number n is odd, respectively even, using mathematical logic formulas with integer variables and constants, the equality predicate and the sum and multiplication operations.

1) Specify the following preconditions, which can be assumed to hold for the execution of P.
   a) Every element of array *in* is different from all the other elements of the same array *in*.
   b) All elements of array *in* in odd positions (i.e., for index values 1, 3, … etc.) are odd, and all elements in even positions are even (in specifying that you can use the predicates *odd*(n) and *even*(n) previously defined).

The procedure copies all elements of array *in* into the array *out*, placing the odd numbers in the first n positions, and the even numbers in the subsequent n positions.

2) Specify the following postconditions.
   a) All elements in the lower part of the array *out* (index values $1 \,..\, n$) are equal to some odd element of array *in*.
   b) All elements in the higher part of the array *out* (index values $n+1 \,..\, 2 \cdot n$) are equal to some even element of array *in*.
3) Optionally, add any postcondition that you think is necessary to make the specification of the procedure precise and unambiguous.

1

Write pre- and post- conditions for a program P that implements a function $f(x, y, z) = 1$ if a parameter can be expressed as a product of the other two and $f(x, y, z) = 0$ otherwise. Parameters x, y and z are integer numbers.

Solution:

Precondition: $\top$

Program: $k = P(x, y, z)$

Postcondition: $(k = 1 \wedge (x = yz \vee y = xz \vee z = xy)) \vee (k = 0 \wedge (x \neq yz \wedge y \neq xz \wedge z \neq xy))$

2

Specify, by means of suitable pre- and post-conditions, a subprogram $P(a, n, v)$ where: a and n are input parameters representing, respectively, an array of integer numbers and the number of its elements (assumed to be ≥3); v is an output Boolean parameter. The parameter v must be set equal to **true** if and only if the elements of array a are placed in increasing order from the first position up to a certain intermediate position and in decreasing order from that point on; both the increasing and decreasing array segments must not be empty. For instance: if a=[1 3 7 5 2] then v=**true**; if a=[1 3 7] then v=**false**; if a=[5 4 6 8] then v=**false**. Please assume that the indexing in the array starts from 0.

Pre: $n \geq 3$

$P(a, n, v)$

Post: $v \leftrightarrow \exists p \, (\; 0 < p < n\text{-}1 \; \wedge \; \forall i \, (\, 0 \leq i < p \rightarrow a[i] < a[i+1] \,) \; \wedge \; \forall i \, (\, p \leq i < n\text{-}1 \rightarrow a[i] > a[i+1] \,) \;)$

3

Consider the function INSREV, which takes as input three arguments: a nonempty array a of integers sorted either in increasing or decreasing order, a positive integer n representing the number of the elements of a, and an integer e, to be inserted into a; it is assumed that e is not an element of a.

The function INSREV outputs an array b, that represents the result of inserting e into a at a non-deterministic position; no other element is inserted into a, and no element is canceled, but the order of the elements previously present in a is changed as follows. All elements before the position where e is inserted are placed in reverse order, while the elements following that position are kept in the same order as in a. For example, if a=[0,1,2,3,4,5,6] and e=8 then b may be the following array: [3,2,1,0,8,4,5,6].

Specify formally, using mathematical logic formulas with the usual predicates and relations on the integers, the pre- and post-conditions for the function INSREV; assume that array indexing starts from 0.

PRE:

$$n > 0 \land (\forall i\ (0 \le i < n-1 \rightarrow a[i] \le a[i+1]) \ \lor \ \forall i\ (0 \le i < n-1 \rightarrow a[i] \ge a[i+1])\ ) \land \neg \exists i\ (0 \le i < n \land a[i] = e)$$

POST:

$$\exists k\ (\ 0 \le k < n+1 \land b[k] = e \land$$

$$\forall i\ (\ 0 \le i < k \rightarrow b[i] = a[k-1-i]) \land$$

$$\forall i\ (\ k+1 \le i < n+1 \rightarrow b[i] = a[i-1])\ )$$

Pre: $dim \geq 3 \wedge (p=0 \vee p=1)$

Post:

If *p*=0 then after the execution of *sumArr* the first two elements of *b* are equal to the elements of *a* in the same positions, and every element from the third position

- is equal to the sum of the two preceding elements if the second preceding element is greater or equal to the immediately preceding element;
- is equal to the product of the two preceding elements if the second preceding element is less than the immediately preceding element.

$(p=0 \rightarrow b[0]=a[0] \wedge b[1]=a[1] \wedge$

$\forall i (1<i<dim \rightarrow ((a[i-2] \geq a[i-1] \rightarrow b[i]= a[i-2]+a[i-1]) \wedge$

$(a[i-2] < a[i-1] \rightarrow b[i]= a[i-2]*a[i-1])))) \wedge$

If *p*=1 then after the execution of *sumArr* every element of *b* is equal to the sum all elements of *a* having position less than or equal to it.

$(p=1 \rightarrow b[0]=a[0] \wedge$

$\forall i (1 \leq i<dim \rightarrow b[i]= b[i-1]+a[i]))$

$$\forall x\ (\ odd(x) \leftrightarrow \exists k\ (\ x = 2{\cdot}k + 1\ )\ )$$
$$\forall x\ (\ even(x) \leftrightarrow \exists k\ (\ x = 2{\cdot}k\ )\ )$$

1) Specify the following preconditions, which can be assumed to hold for the execution of *P*.
   a) Every element of array *in* is different from all the other elements of the same array *in*.
      $\forall i\ (1 \le i \le 2n \rightarrow \neg\exists j\ (1 \le j \le 2\text{n} \wedge i \ne j \wedge in[i] = in[j]\ )\ )$
   b) All elements of array *in* in odd positions (i.e., for index values 1, 3, … etc.) are odd, and all elements in even positions are even (in specifying the you can use the predicates $odd(n)$ and $even(n)$).
      $\forall i\ (1 \le i \le 2n \rightarrow (\ (\ odd(i) \rightarrow odd(in[i])\ ) \wedge (even(i) \rightarrow even(in[i])\ )\ )$

2) Specify the following postconditions.
   a) All elements in the lower part of the array *out* (index values 1 .. n) are equal to some odd element of array *in*.
      $\forall i\ (1 \le i \le n \rightarrow \exists j\ (1 \le j \le 2n \wedge odd(in[j]) \wedge out[i] = in[j]\ )\ )$
   b) All elements in the higher part of the array *out* (index values n+1 .. 2·n) are equal to some even element of array *in*.
      $\forall i\ (n{+}1 \le i \le 2n \rightarrow \exists j\ (1 \le j \le 2n \wedge even(in[j]) \wedge out[i] = in[j]\ )\ )$

3) Optionally, add any postcondition that you think is necessary to make the specification of the procedure precise and unambiguous.

   The array *out* must contain all the elements of array *in* (equivalently, all elements of array *in* are present at some position in the array *out*).

   $\forall i\ (1 \le i \le 2n \rightarrow \exists j\ (1 \le j \le 2n \wedge out[j] = in[i]\ )\ )$

6.
Consider a subprogram
mulDiv (***a***, ***n***, ***k***, ***m***)
whose parameters have the following meaning
a is an array of positive integer numbers that includes at least 3 elements

n is the actual number of elements of array *a* (notice that ***indices*** inside array *a* **start from 1**)
k is an index position inside array ***a***, strictly included between the first and last position

In implementing the subprogram one can assume that the elements of array ***a*** are strictly increasing from the initial position up to index ***k***, and strictly decreasing from index ***k*** up to the last position.

Parameter ***m*** is a ***boolean*** output, that is a *boolean* result returned to the caller; it must be true if and only if

•for all positions from the initial one up to position ***k*** included, in every pair of consecutive elements the second one (the one with the higher position) is an integer multiple of the first one (the one with inferior position), and
•for all positions from ***k*** included up to the last position, in every pair of consecutive elements the second one (the one with the higher position) is an integer divisor of the first one (the one with inferior position).

Example: when ***a*** = [1, 3, 6, 24, 12, 4, 2], ***n*** = 7, ***k*** = 4, ***m*** = true then the program behavior is correct.

- Write formulas for the pre- and post-conditions of program ***mulDiv***
- For each of the following sets of values of the parameters of the *mulDiv* subprogram, say if they are compatible with a correct implementation of the subprogram with respect to its specification by means of the pre- and post-condition.

a = [2, 4, 12, 6, 3], *n* = 5, *k* = 3, *m* = true
a = [2, 8, 8, 4, 2], *n* = 5, *k* = 2, *m* = false
a = [2, 4, 12, 6, 4], *n* = 5, *k* = 3, *m* = true

a = [2, 8, 8, 4, 2], *n* = 5, *k* = 3, *m* = true
a = [2, 4, 12, 6, 4], *n* = 5, *k* = 3, *m* = false
a = [2, 4, 12, 6, 3], *n* = 5, *k* = 3, *m* = false

precondition

$$n \geq 3 \ \wedge 1 < k < n \ \wedge \\ \forall i \left(\begin{array}{l} (1 \leq i \leq n \rightarrow a[i] > 0) \\ \wedge \\ (1 \leq i < k \rightarrow a[i] < a[i+1]) \\ \wedge \\ (k \leq i < n \ \rightarrow \ a[i] > a[i+1]) \end{array} \right)$$

Postcondition for the value of the out parameter ***m***

$$\forall i \left(\begin{array}{l} (1 \leq i < k \rightarrow \exists q \, (q > 1 \wedge \ a[i+1] = q \cdot a[i])) \\ \wedge \\ (k \leq i < n \ \rightarrow \ \exists q \, (q > 1 \wedge \ a[i] = q \cdot a[i+1])) \end{array} \right)$$

or equivalently

$$\forall i \, \exists q \left(\begin{array}{l} (1 \leq i < k \rightarrow \ (q > 1 \wedge \ a[i+1] = q \cdot a[i])) \\ \wedge \\ (k \leq i < n \ \rightarrow \ \ (q > 1 \wedge \ a[i] = q \cdot a[i+1])) \end{array} \right)$$

a = [2, 4, 12, 6, 3], *n* = 5, *k* = 3, *m* = true

precondition and postcondition both satisfied, implementation correct

a = [2, 8, 8, 4, 2], *n* = 5, *k* = 3, *m* = true

NB: precondition is ***not*** satisfied (two maximal elements **8**), implementation correct independently of the value of *m*

a = [2, 8, 8, 4, 2], *n* = 5, *k* = 2, *m* = false

NB: precondition is ***not*** satisfied (two maximal elements **8**), implementation correct independently of the value of m

a = [2, 4, 12, 6, 4], *n* = 5, *k* = 3, *m* = false

precondition and postcondition both satisfied, implementation correct

a = [2, 4, 12, 6, 4], *n* = 5, *k* = 3, *m* = true

precondition satisfied, postcondition not sastisfied (4 is not a divisor of 6, hence *m* should be false), implementation incorrect

a = [2, 4, 12, 6, 3], *n* = 5, *k* = 3, *m* = false

precondition satisfied, postcondition not sastisfied (*m* should be true), implementation incorrect

minimal background on Computability Theory

(refer to Ch.0 of textbook)

cardinality of a set S, $|S|$ or $card(S)$: intuitively, «the number of its elements»

two sets X and Y are ***equinumerous***, $X \approx Y$, iff there is a bijection $f: X \to Y$

equinumerous sets are said to ***have the same cardinality***

property '$\approx$' is an equivalence relation

X has smaller cardinality than Y, $card(X) < card(Y)$, iff

- X is equinumerous with a subset of Y
- Y is not equinumerous with any subset of X

a set X is ***finite*** iff it is empty or it is equinumerous with $\{ i \mid i \in \mathbb{N} \wedge i \leq k \}$ for some fixed $k \in \mathbb{N}$

a set X is ***infinite*** iff it is not finite

a set X if ***denumerable*** iff it is equinumerous with $\mathbb{N}$ (the natural numbers)

hence there is a bijection $f : \mathbb{N} \to X$ and X can be ***enumerated***,

i.e., listed without skipping any element $X = \{ x_0, x_1, x_2, \ldots \}$

a denumerable set is said to have ***cardinality*** $\aleph_0$ ($\aleph$ is «aleph»)

a set is ***countable*** iff it is finite or denumerable

hence every subset of a denumerable set is countable (because it is denumerable or finite)

For a set S, $\wp(S)$ is the powerset of S (the set of all its subsets)

the cardinality of $\wp(S)$: $|\wp(S)| = 2^{|S|}$

it follows from:

- the cardinality of the set of total functions from a domain A to a range B is $|B|^{|A|}$

(to identify a function, choose one of $|B|$ values for $|A|$ times)

- a subset S' of S may be viewed as a function $f_{S'} : S \rightarrow \{ False, True \}$

Any set equinumerous with the set of subsets of a denumerable set

is said to have cardinality $2^{\aleph_0}$ which is called the ***cardinality of the continuum***

NB: here $2^{\aleph_0}$ is used as a symbol, it is not a natural number

Fundamental fact: $2^{\aleph_0} > \aleph_0$, i.e., the cardinality of the continuum is greater than that of $\mathbb{N}$

Celebrated proof by Cantor using a ***proof by DIAGONALIZATION***

(clearly, $2^{\aleph_0}$ is not smaller than $\aleph_0$: there are trivial bijections between $\mathbb{N}$ and a subset of $\wp(\mathbb{N})$)

Suppose by contradiction that $\mathbb{N}$ and $\wp(\mathbb{N})$ have the same cardinality
then there is a bijection $s : \mathbb{N} \to \wp(\mathbb{N})$ i.e., an enumeration $\{s_0, s_1, s_2, ...\}$ of $\wp(\mathbb{N})$

	0	1	2	3	4	...
s_0	×	×	×	×	×	...
s_1	√	×	×	×	×	...
s_2	×	√	×	×	×	...
s_3	√	√	×	×	×	...
s_4	×	×	√	×	×	...
...	...	...	...	...	...	...

just as an example of how such enumeration might look like:

$s_0=\varnothing$,
$s_1=\{0\}$,
$s_2=\{1\}$,
$s_3=\{0, 1\}$,
$s_4=\{2\}$,
...
$s_i=\{k \mid k\text{-th}$ digit in the binary encoding of i is 1 $\}$

Assume there is a bijection $s : \mathbb{N} \to \wp(\mathbb{N})$ i.e., an enumeration $\{s_0, s_1, s_2, ...\}$ of $\wp(\mathbb{N})$

For any fixed enumeration s, consider set $D \subseteq \mathbb{N}$ defined as $D = \{ n \mid n \notin s_n \}$

i.e., D is the set of the numbers n not included in $s_n \in \wp(\mathbb{N})$

	0	1	2	3	4	...
s_0	×	×	√	×	×	...
s_1	√	√	×	×	√	...
s_2	×	√	×	×	×	...
s_3	√	√	×	√	×	...
s_4	×	×	√	×	×	...
...	...	...	...	...	...	...

In example of the table on the left
(the enumeration s differs from the previous one):

s_0={2, ...}, s_1={0, 1, 4, ...}, s_2={1, ...},
s_3={0, 1, 3, ...}, s_4={2, ...}, ...

D = {0, 2, 4, ... }

If the bijection $s : \mathbb{N} \to \wp(\mathbb{N})$ exists, then $D = s_i$ for some $i \in \mathbb{N}$

Then we ask the question: $i \in D$ or $i \notin D$?

If $i \in D$ then $i \notin s_i$ by the definition of D; but then $i \notin D$ because $D = s_i$; a contradiction

If $i \notin D$ then $i \in s_i$ by the definition of D; but then $i \in D$ because $D = s_i$; a contradiction

QED

Diagonalization is a fundamental proof technique in the theory of computability

COMPUTABILITY – 1

DOVETAILING SIMULATION (A cura di ChatGPT)

Problema: Dato un input che rappresenta una coppia (programma, input), determinare se il programma termina (si arresta) quando eseguito con l'input fornito.

Simulazione Dovetailing:
Costruisci la Macchina di Turing Universale (UTM):

Immagina di avere una UTM che può simulare il comportamento di tutte le macchine di Turing possibili.

Simulazione Dovetailing:
L'idea chiave del dovetailing è eseguire passi di calcolo alternati per tutte le possibili coppie (programma, input).
Ad ogni passo, la UTM esegue un passo di calcolo su ciascuna coppia (programma, input) in modo alternato.

Controllo dell'Arresto:
Ogni volta che una simulazione raggiunge uno stato in cui il programma terminerebbe (esegue un passo finale), controlla se l'output della funzione rappresentante indica che l'input appartiene a P (ad esempio, restituisce 1). Se sì, accetta l'input.
Se una simulazione continua all'infinito senza terminare, l'input non appartiene a P.

Conclusioni:
Se l'input appartiene a P, una delle simulazioni dovrebbe accettare in un numero finito di passi.
Se l'input non appartiene a P, nessuna delle simulazioni dovrebbe mai accettare.
Questo dimostra la semidecidibilità del problema dell'arresto di Turing. La simulazione dovetailing permette di esplorare in modo "parallelo" tutte le possibili coppie (programma, input), ma non garantisce che la UTM termini sempre. Se un'input appartiene al problema, alla fine la UTM lo riconoscerà; se non appartiene al problema, la UTM potrebbe non terminare mai.

Show that the function g defined as follows

$g(x, y) =$ if $f_y(x) = 1$ then 1 else 0

is not computable. Use preferably a diagonalization proof.

Starting from *g*, let us define the function of an argument h, in this way:

$h(x) =$ if $g(x, x) = 1$ then 0 else 1

If g is computable, then *h* is also computable, hence $h = f_{x_h}$ for some x_h.

Let us compute the value $h(x_h)$ of *h* applied to the index of the Turing Machine that computes it.

$h(x_h) =$ if $g(x_h, x_h) = 1$ then 0 else 1;

Then, using the definition of g: $h(x_h) =$ if ((if $f_{x_h}(x_h) = 1$ then 1 else 0) = 1) then 0 else 1:

NB: the condition "(if $f_{x_h}(x_h) = 1$ then 1 else 0) = 1"

is equivalent to "$f_{x_h}(x_h) = 1$" (it is true if $f_{x_h}(x_h) = 1$, it is false if $f_{x_h}(x_h) \neq 1$), hence

$h(x_h) =$ if $f_{x_h}(x_h) = 1$ then 0 else 1;

then (since f_{x_h} is h):

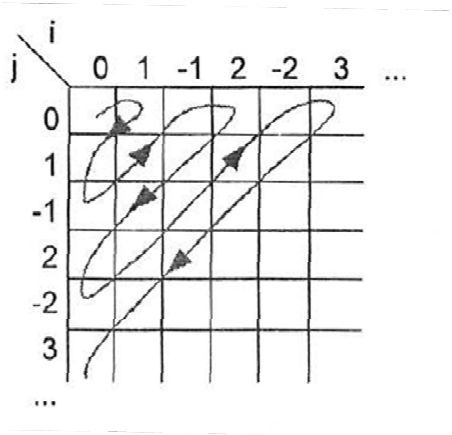
$h(x_h) =$ if $h(x_h) = 1$ then 0 else 1 $\Rightarrow$ contradiction!

Since we reach a contradiction, the hypothesis of computability of the function g is wrong.

Hilbert's Tenth problem:
Given a polynomial P with integer coefficients in integer variables (for instance $5xy + 23z^3w^5 - 13x^8$). Check whether there exist integer solution for the equation P (x, y, z, w) = 0.
Hilbert's Tenth problem was shown to be undecidable.
Check whether it is at least semi-decidable, justifying your answer.

Enumeration of 3-tuples obtained from an enumeration of pairs,
Enumeration of 4-tuples obtained from enumeration of three-tuples, etc.

Hilbert's Tenth problem is semi-decidable. In fact, let *n* be the number of variables of a given polynomial, it is possible to recursively enumerate all the n-tuples of integer numbers (for instance a recursive enumeration of integer pairs is the following one: {<0,0>, <1,0>, <0,1>, <0,-1>, <1,1>, <-1,0>, <2,0>, <-1, 1>, <1, -1>, <0,2>, <0, -2>, ...}).
For every pair, it is easy to verify if it is a root for the polynomial. Thus, if there exists a solution, sooner or later it will be found by the former computation. However this computation will never end if there does not exist any integer root of the polynomial.



Consider the following sentence ***A***:
Given any possible positive integer x, if x is even divide it by 2. Otherwise multiply it by 3 and sum 1 to the result. Then divide it by 2. Repeat this procedure a finite number of times, soon or later we will obtain the value 1.
Show if it is decidable the problem to determine if ***A*** is true or false, explaining the answer.
(Note that: it is not asked to decide if ***A*** is true or false, but it is asked the decidability of its truth).
$5 \to 16 \to 8 \to 4 \to 2 \to 1$; $6 \to 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$
$7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \ldots$

The problem is decidable. In fact, ***A*** is necessarily true or false. So its decision is equivalent to compute a constant function (the constant function 0 or 1), hence computable. This does not mean that is possible to determine the truth of ***A***: ***A*** is considered a likely true conjecture on natural numbers but it has not yet been proved (to the best of our knowledge).

Consider the set S_A of the positive integer numbers such that, by applying to them the above described procedure the value 1 is eventually obtained. Is S_A semidecidable ? What can we say about the decidability of S_A ?

S_A can be enumerated by applying the dovetailing technique to the above described procedure. Hence it is semidecidable.
S_A is decidable if ***A*** is true, because in this case S_A is the set of all positive integers.

Show that the following function g

$g(x) =$ 1 if I_{fx} is finite

0 otherwise

where I_{fx} denotes the *image* of function f_x is not computable.

The theorem is a simple consequence of the Rice theorem. Consider the set

I = {computable functions with finite image}.

Obviously, $I \neq \varnothing$ and $I \neq$ {computable function }, so the set $S = \{x \mid f_x \in I\}$ is not recursive, from the Rice theorem.

But $g(x)$ is the characteristic predicate of set S, therefore is not computable.

Show that, given a (fixed) value z_o, the function

$$g(x) = \begin{cases} 1 \text{ if } z_o \in I_{fx} \\ 0 \text{ otherwise} \end{cases}$$

is not computable.

The theorem immediately follows from the Rice theorem, considering, for a given z_o, the set $Z_o=\{x \mid z_o \in I_{fx}\}$, that is the set of the computable functions which have z_o in their image.

Show that the function

$g(x, y)$ = if f_x coincides with f_y then 1 else 0

is not computable.

The proof is obtained by reducing to g a function g_u that we show is not computable.
Consider the everywhere undefined function u(x): $\forall x\ u(x) = \perp$.
u is obviously computable
Consider the index x_u of a TM that computes u.
Define function $g_u(x)$ that determine if a given TM, identified by its index, computes the function u. Function g_u is defined as follows

$g_u(x)$ = if $f_x = u$ then 1 else 0

Clearly the function g_u is a "specialization" of g: in fact $g_u(x) = g(x, x_u)$. So if g is computable then g_u is also computable. In other words, g_u is reduced to g.
But g_u is not computable, from the Rice theorem: in fact, consider the set U of computable functions defined as follows

$U = \{ u \mid \forall x, u(x) = \perp \}$

U is not empty and it does not coincide with the set of all computable functions
Hence the set

$S = \{x \mid f_x \in U\}$

is not recursive, because of the Rice theorem.
But g_u is the characteristic function of S, so g_u is not computable, hence g is also not computable.

Consider the set S of computable functions whose value is a prime number for every value of their argument for which they are defined, i.e., $S=\{ k \mid \forall x (f_k(x) \neq \perp \rightarrow f_k(x) \text{ is prime }) \}$. Determine whether S is recursive or recursively enumerable. Provide adequate justifications for your answer.

S is not recursive, based on the Rice theorem: it is neither empty (consider the computable constant function $g(x)=13$) nor it is the set of all computable functions (consider, say, $h(x)=x^2$, which is computable and such that $h \notin S$). On the other hand, the complement $\neg S$ of set S, which consists of the computable functions that have a non-prime value for some argument for which they are defined (i.e., $\neg S = \{ k \mid \exists x (f_k(x) \neq \perp \wedge f_k(x) \text{ is not prime }) \}$) can be enumerated using the well-known dovetailing technique, therefore the set S in not recursively enumerable (otherwise both S and $\neg S$ would be recursive).

Answer the following questions providing brief but clear and precise justifications.

1. Is the function g defined as follows:
   $g(x) = 1$ if $\forall y\, f_x(y) = 2 \cdot y$; $g(x) = 0$ otherwise
   computable?
2. Is the function defined as follows:
   $h(x) = 1$ if x is even and $\forall y\, f_{x/2}(y) = 2 \cdot y$; $h(x) = 0$ otherwise
   computable?

1. The function
   $g(x) = 1$ if $\forall y\, f_x(y) = 2 \cdot y$; $g(x) = 0$ otherwise
   is not computable, due to the Rice theorem, because it is the characteristic function of the set of TMs that compute the function $p(x) = 2 \cdot x$.
2. The function
   $h(x) = 1$ if x is even and $\forall y\, f_{x/2}(y) = 2 \cdot y$; $h(x) = 0$ otherwise
   is not computable. In fact the computation of function $g(x)$ above can be ***reduced*** to that of function $h(x)$; in fact, $g(x) = h(2 \cdot x)$.
   $h(2 \cdot x) = 1$ if $2 \cdot x$ is even and $\forall y\, f_{2 \cdot x/2}(y) = 2 \cdot y$; $h(2 \cdot x) = 0$ otherwise
   $= 1$ if $\forall y\, f_x(y) = 2 \cdot y$; 0 otherwise $= g(x)$
   Therefore if $h(x)$ was computable also $g(x)$ would be computable.

Consider the set of S of Turing machines, acting as language acceptors, accepting languages that do not include any string of even length. State, providing suitable justification, whether S is decidable or semidecidable.

By adopting any reasonable encoding convention, the Turing machines acting as language acceptors can be viewed as computing functions (the function's value could be, say, 1 for accepted strings and $\perp$ for rejected strings); therefore set S satisfies the hypotheses of Rice's theorem and is undecidable. On the other hand $\neg S$, the complement of S, which includes the Turing machines that accept some even-length string, can be enumerated by simulation (using the dovetailing technique), hence it is semidecidable, so that S cannot be semidecidable.

Let M be a generic Turing machine, considered as a language acceptor, so that $L(M)$ denotes the language accepted by M. Answer the following questions, providing concise justifications.

1. Is the set $S_1 = \{ M \mid |L(M)| = 13 \}$ decidable?
2. Is the set $S_2 = \{ M \mid |L(M)| \geq 13 \}$ decidable? Is it semidecidable?
3. Is the set $S_3 = \{ M \mid |L(M)| \leq 13 \}$ decidable? Is it semidecidable?
4. Is the set $S_4 = \{ \langle M, x \rangle \mid x$ is accepted by M in ≤ 13 steps $\}$ decidable? Is it semidecidable?

1. No, as a consequence of the Rice theorem
2. No, as a consequence of the Rice theorem; yes, by using the customary dovetailing technique.
3. No, as a consequence of the Rice theorem; no, because the set $\{ M \mid |L(M)| > 13 \}$, of which S3 is the complements, is undecidable and semidecidable (for the same reasons as S2), therefore if S3 was semidecidable both S2 and S3 would be decidable.
4. S_4 is decidable: since the length of the computation is bounded it suffices to simulate the execution of M on x up to that number of steps.

Preliminarily, prove that the union of two semidecidable sets, S_1 and S_2, is also semidecidable. Your argument should be as formal and precise as possible.

Preliminarily: yes, the union of the two semidecidable sets S_1 and S_2 is semidecidable.

In fact, if we call g_{S1} and g_{S2} the generating functions of S_1 and S_2, then any total computable function having as *image* the union of the images of the two generating functions g_{S1} and g_{S2} would be a generating function, call it $\boldsymbol{g_{1\cup 2}}$, for $S_1 \cup S_2$.

E.g., $g_{1\cup 2}(x) =$ ***if*** $odd(x)$ ***then*** $g_{S1}(x \text{ DIV } 2)$ ***else*** $g_{S2}(x \text{ DIV } 2)$

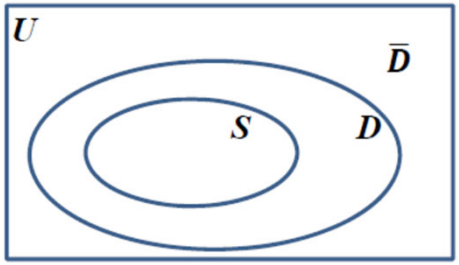
Another argument consist of defining the enumeration of $S_1 \cup S_2$ based on that of S_1 and S_2. Let $enum_1$={ $n1_0$, $n1_1$, $n1_2$, …} be an enumeration of S_1, and $enum_2$={ $n2_0$, $n2_1$, $n2_2$, …} be an enumeration of S_2; then an enumeration $enum_{12}$ of $S_1 \cup S_2$ is defined as follows $enum_{12}$={ $n1_0$, $n2_0$, $n1_1$, $n2_1$, $n1_2$, $n2_2$, …}. If there exist total computable functions generating the $enum_1$ and $enum_2$ then certainly there exists a total computable function generating $enum_{12}$ hence $S_1 \cup S_2$ is semidecidable.

Prove the following property: the difference set, $D - S$, between any decidable set D and ***any*** semidecidable and not decidable set S strictly included in D (i.e., such that $S \subset D$ and $S \neq D$) is *not* necessarily semidecidable.

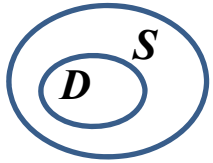
Prove the following property: the difference set, $D - S$, between any decidable set D and any semidecidable and not decidable set S strictly included in D (i.e., such that $S \subset D$ and $S \neq D$) is *not* necessarily semidecidable.

Proof of the stated property.

Since D is decidable, its complement $\overline{D}$ is also decidable, hence semidecidable. If $D - S$ was semidecidable, then also (D-S)$\cup\overline{D}$, would be semidecidable, being the union of two semidecidable sets. But the set (D-S)$\cup\overline{D}$ is equal to the complement of S. Hence, being both S and $\overline{S}$ semidecidable, S would be decidable, contradicting the hypothesis that it is semidecidable and not decidable.



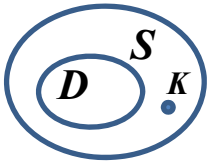
- Consider a semidecidable, non-empty set S (so that it admits a generating function $g_S(x)$), and a decidable set D (so that its characteristic predicate $c_D(x)$ is computable), D being strictly included in S. Prove that the difference set S - D is semidecidable. To this end, please define function $g_{S-D}(x)$, the generating function of set S - D.
- How does the answer change if set D is not necessarily strictly included in S?



Since S is non-empty and D is strictly included in S, then there exists an element K such that $K \in S$ - D. Then the generating function of S - D is defined as follows

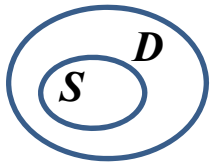
$$g_{S-D}(x) = \text{if } c_D(g_S(x)) = 0 \text{ then } g_S(x) \text{ else } K$$

(notice that $c_D(g_S(x)) = 0$ means that $g_S(x) \notin D$, which implies $g_S(x) \in S\text{-}D$)

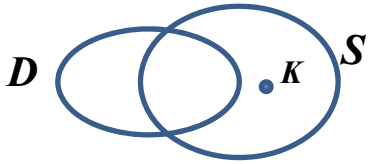


If set D is not strictly included in S the answer does not change. Two cases arise.

1. $S \subseteq D$: then S - $D = \varnothing$ and S - D is semidecidable by definition.



2. $(S \cap D) \subset S$ and $(S \cap D) \neq S$; hence there exists an element $K \in S$ - D and the argument follows the same lines as above.

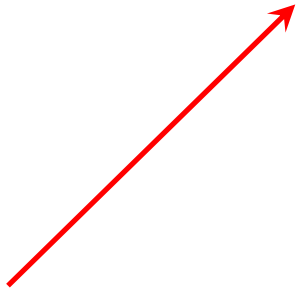


COMPUTABILITY – 2

Are the following functions computable?

1. $g(x, y) = \text{if } (\exists z\ f_x(z) \neq \perp \wedge f_y(z) \neq \perp \wedge f_x(z) \neq f_y(z)) \text{ then } 1 \quad \text{else } \perp$
2. $g(x, y) = \text{if } (f_x \neq f_y) \text{ then } 1 \quad \text{else } 0$

Please provide adequate justifications for your answers.



NB: "a TM computes the value ⊥"
does NOT mean the TM stops and outputs '⊥' !!
it means that the TM never stops so it does not output anything

1. g is computable because we can perform a dovetailing simulation over triples (A,B,C) from N × N × $\{x, y\}$ where A denotes the input string, B denotes the length of the simulation and C represents pairs of machine x and y. If both machines eventually terminate on some input and produce different values, then the simulation returns 1, otherwise it continues.
2. g is not computable. Otherwise we could, say, take a fixed Y and use $g(x, Y)$ to decide the set $F = \{x \mid f_x = f_Y\}$, which contradicts the Rice theorem.

Consider the following problem:

Given a program P that computes the function f_P: N → N (N is the set of natural numbers) establish if $f_P(x) \neq 10$ for all x in its domain.
Say, justifying briefly your anwser, if the problem is:

- decidable
- Semidecidable but not decidable
- Not even semidicidable.

The problem is not decidable. In fact the set of functions for which we can apply the specified property is obviously neither empty nor the universe set. Therefore we can apply the Rice theorem.
However the complement of the problem is semidecidable. In fact, using the typical technique of diagonal enumeration it is possible to identify an x such that $f_p = 10$, if such x exists.
Consequently, the original problem is not semi decidable too, otherwise it would be also decidable.

For each of the following sets S:

a) $S = \{ y \mid$ for all x for which $f_y(x)$ is defined, $f_y(x)$ is an even number $\}$

b) $S = \{ y \mid$ the TM y halts on input 0 within no more than 1,000 computation steps $\}$

c) $S = \{ \langle y, x \rangle \mid f_y(x) \neq \perp \}$

1. state whether or not Rice's Theorem is relevant for establishing the decidability of S;
2. state whether or not S is decidable, or S is semidecidable, or $\neg S$, the complement of S, is semidecidable.

In the answer, D denotes the family of decidable sets, SD the family of semidecidable sets.

a)

1. Rice's theorem applies because S corresponds to a set of computable functions;
2. $S \notin D$; $\neg S \in SD$ ($\neg S = \{ y \mid \exists x (f_y(x) \neq \perp \wedge f_y(x)$ is odd $) \}$) can be enumerated by simulation; therefore $S \notin SD$.

b)

1. Rice's theorem does not apply because S specifies property of the computations, not of the computable functions;
2. $S \in D$, it can be decided by simulation; hence $S \in SD$ and $\neg S \in SD$.

c)

1. Rice's theorem does not apply *immediately*, but the undecidability of S can be established by reducing to it the decidability, for instance, of the set $S_a = \{ y \mid f_y(a) \neq \perp \}$, for any fixed ***a***, which is undecidable by Rice's theorem;
2. $S \in SD$, as usual by simulation; hence $\neg S \notin SD$.

John Hackitt, a software developer, is trying to acquire a compiler to translate C code into the assembler language of his preferred hardware. He knows that not all the available software is 100% reliable, hence before acquiring a given compiler he wants to have sufficient technical assurance of its correctness. Therefore, before looking for a compiler that fits his needs, he searches for a tool or a testing benchmark able to certify the correctness of any compiler that he might decide to acquire.

Can the search of the tool or the testing benchmark (not the successive one for the compiler) be successful? Explain briefly the reason for your reply.

Optional part. John Hackitt, in his search for a reliable compiler, finds a compiler that is claimed to be accompanied by a correctness certification (provided in a mathematical way by a team of experts in compilation and software verification technology) which is specific for that compiler and hardware. Should John Hackitt trust this claim? On which basis?

The correctness problem for a compiler as well as the correctness problem for any program, can be formalized as the problem of determine if a generic algorithm computes a given computable function; therefore the Rice theorem can be applied to this case (the compilation of C programs into assembler corresponds to a well defined computable function, different from both the empty set and the set of all computable functions).

Similarly, a well designed testing benchmark can be useful to catch errors in a compiler, if they are present, but it cannot guarantee that the compiler is free of errors, hence it cannot provide any absolute guarantee that a compiler is correct.

Optional part: in principle, it is possible (though not very likely at the current state of the art of software verification technology) that there exists a proof of correctness of a specific compiler for a specific hardware, which John could check by himself (if he is enough skilled) or by using some proof checker (a tool that can check the correctness of proofs with reference to a given formal system).

Please indicate, providing suitable justifications, which of the following problems are decidable:

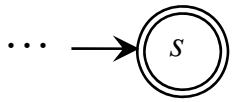
1. Determine whether a generic Turing machine immediately stops (therefore executing 0 moves) for every possible input.
2. Determine whether a Turing machine stops after one move for every possible input.
3. Determine whether a Turing machine stops (eventually) for every possible input.

1. Problem 1 is decidable: the TM stops in 0 moves iff the initial state is also final, which can be determined by means of a check on the machine state-transition diagram. (Notice that the problem is independent of the input).
2. Problem 2 is decidable: it suffices to check that the TM stops within one move for the strings of length ≤ 1, that is for a finite number of cases: if fact within one move the TM can at most inspect one symbol of the input string.
3. Problem 3 is undecidable: it is equivalent to determine whether the TM computes a total function, which is well known to be undecidable.

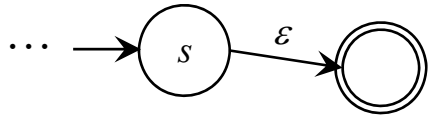
1. Consider the problem of determining whether a generic Turing Machine, with a generic input, can reach, during its computation, one of its states (the state is also generic, in that it must be considered, like the TM and the input, a parameter of the problem). Determine whether the problem is solvable.
2. Consider the variant of question 1 above, where the state is bound to be *not final*. Does the result or its justification change? Provide brief, concise explanations.

1. The problem is unsolvable, because the halting problem can be reduced to it by choosing a final state as the state whose reachability is investigated.
2. If the state whose reachability is investigated is bound to be not final, the problem is still undecidable. This can also be proved by reducing to it the halting problem, although in a more elaborate way. For any given Turing machine, T1, for which one wants to determine if it halts, define a derived Turing machine T2 which is identical to T1 except that any final state *s* of T1 is not final in T2, and when T2 reaches *s* instead of halting it moves into another, newly added, final state. Clearly, the Turing machine T1 halts if and only if T2 reaches the non-final state *s*.

T1:



T2:



Answer the following questions providing brief but clear and precise justifications

1. Consider the following set $S = \{ y \mid \forall x \, (f_y(x)\neq\perp \rightarrow f_y(x)=x^2) \}$. Is set S decidable? Is it semidecidable?
2. Consider the following function.
   $g(x, y) = 1$, if $\forall z \, (f_x(z)\neq\perp \wedge f_y(z)\neq\perp \rightarrow f_x(z)= f_y(z))$, 0 otherwise
   Is function g computable?
3. Consider the following function g', variation of function g above
   $g'(x, y, z) = 1$, if $f_x(z)\neq\perp \wedge f_y(z)\neq\perp \wedge f_x(z)= f_y(z))$, $g'(x, y, z) = \perp$ otherwise
   Is function g' computable?

1. Set S is not decidable, as a direct consequence of the Rice theorem. The complement S' of S
   $S' = \{ y \mid \exists x \, (f_y(x)\neq\perp \wedge f_y(x)\neq x^2) \}$
   is semidecidable (its elements can be enumerated by a systematic procedure which considers in turn all computable functions and executes them for each possible input values and for increasing number of steps, and lists those y for which the condition $f_y(x)\neq x^2$ holds). Therefore set S is not semidecidable (otherwise S and S' would be both decidable).
2. By taking constant K equal to the Goedel number of a TM that computes function $f(x)=x^2$, we have $g(x, K)=p(x)$, where p is the characteristic function of set S above, hence function $g(x, K)$ is not computable, and so is $g(x, y)$.
3. Function g' is computable: a TM computing g' simulates the execution of $f_x(z)$ and $f_y(z)$ and, if they both terminate, checks whether $f_x(z)= f_y(z)$, otherwise it continues the computation indefinitely.

Consider the following set

$S = \{ i \mid$ Turing machine M_i terminates its execution in less than 100 steps for some input$\}$

$= \{ i \mid \exists x\, T_{M_i}(x) < 100 \}$ -- $T_{M_i}(x)$ is the notation for denoting time complexity

Is S recursively enumerable? Is S recursive?

S is recursively enumerable, with the usual simulation technique (with dovetailing)

Complement set: $\neg S = \{ i \mid \neg \exists x\, T_{M_i}(x) < 100 \} = \{ i \mid \forall x\, T_{M_i}(x) \geq 100 \}$
It does not seem to be obviously semidecidable, with the usual simulation techniques

S is recursive (hence also recursively enumerable). It suffices to verify that the machine stops within 99 steps for some string of length <100. Such strings are in finite number, hence this check can be executed in a finite number of steps. It is not necessary to check longer strings because if there exists a string, longer than 100, such that the machine stops within 100 moves, hence having read a prefix of the input (let's call it ***s***) that is shorter than 100, then necessarily when having as input the string ***s*** the machine would stop within the same number of moves.

Notice that in this question the Rice theorem does not apply, because the computation terminating within 100 steps is ***not*** a property of the computed function, but a property of the computations of the machine.

define a ***monotonic grammar*** as an unrestricted grammar s.t., for every rule of the form $\alpha \rightarrow \beta$:

- $\alpha \in V_N^+$ where V_N is the set of nonterminal symbols
- $\beta \in (V_N \cup V_T)^+$ where V_T is the set of terminal symbols
- $|\alpha| \leq |\beta|$

The notions of derivation and language generated by the grammar are defined as usual.

Answer the following questions precisely and clearly, providing suitable justifications.

1. Is the language generated by a monotonic grammar decidable? Is it semidecidable?
2. What changes concerning the decidability and semidecidability of the generated language if the above constraints are removed?

1. Is the language generated by a monotonic grammar decidable? Is it semidecidable?

1. The language generated by a monotonic grammar is decidable. A ***nondeterministic*** Turing machine that, having as input a string x, decides if it belongs to the language, works as follows. It nondeterministically explores all derivations, starting from the axiom S. For every obtained string in $(V_N \cup V_T)^+$ it nondeterministically executes all possible derivation steps, while it keeps track, along every computation path representing a derivation, of all the strings generated so far.

   If a string $s \in (V_N \cup V_T)^+$ obtained by a derivation step is such that $|s|>|x|$, or s has already been generated previously in the same computation path, then the generation terminates with rejection (NB: only for that computation path);

   if a terminal string is obtained then the generation terminates, and x is accepted if and only if it is equal to the generated string.

   Notice that the check to rule out that a string is obtained twice in the same derivation is necessary because the length of successively derived strings does not necessarily increase at every step.

2. What changes concerning the decidability and semidecidability of the generated language if the above constraints are removed?

2. If the constraints are removed then the grammar is a generic unrestricted grammar, whose generated language is notoriously semidecidable but undecidable.

Consider any family F of sets such that F has the following properties

a. F is not empty
b. F is *closed under complementation* (for every set $S \in F$ also $\neg S \in F$, $\neg S$ being the complement of set S)
c. $\emptyset \notin F$ (all sets in F are non-empty)

For each of the following statements,

1. All sets of the family F are decidable
2. All sets of the family F are semidecidable and not decidable
3. No set of the family F is semidecidable
4. Some set of the family is semidecidable and not decidable

say if it is:

– necessarily true (it holds for every family F with the above features), or
– necessarily false (it does not hold for any family F with the above features), or
– possibly true/false (it holds for some family but not for some other).

Suitably motivate your answer.

Do the answers you provided above change if, in addition, the elements of the family F are sets of indices of computable functions? Suitably motivate your answer.

1. All sets of the family F are decidable: this can be true for some family and false for some other, because the complement of a decidable set is decidable, hence the complement of an undecidable set is undecidable. A family may include some decidable sets (and their complement sets, that are decidable) and some undecidable sets (and their complement sets, that are undecidable).
2. All sets of the family F are semidecidable and not decidable: this is false in all cases, because for every set in the family, its complement set is also in the family, and if a set and its complement are semidecidable, then they are both decidable.
3. No set of the family F is semidecidable: possible (for instance, if all the sets of F are neither decidable nor semidecidable), but not necessary (the family may include decidable –and hence also semidecidable – sets without violating the properties of F).
4. Some set of the family is semidecidable and not decidable: possible (in this case the complement of the semidecidable and not decidable set would not be semidecidable), but not necessary (for instance all sets in F may be decidable).

If, in addition, the elements of the family F are sets of indices of computable functions, then case 1 above is impossible (because all the sets of F satisfy the hypothesis of the Rice theorem) and the other cases are unchanged.

Consider the following set S_1

$$S_1 = \{ y \mid \forall n \text{ the Turing machine } M_y \text{ does not accept any string of length } n \}$$

State, providing suitable clear and precise justifications, whether set S_1 is decidable or semidecidable.

Set S_1 is undecidable, which can be justified based on the Rice Theorem.

Let us consider the complement set $\neg S_1$

$$\neg S_1 = \{ y \mid \exists n \text{ such that the Turing machine } M_y \text{ accepts some string of length } n \}$$

S_1 is the set of TM with empty accepted language, hence $\neg S_1$ is the set of TM with nonempty accepted language, and such set is obviously semidecidable by dovetailing simulation. Therefore S_1 is not semidecidable.

Consider next the following set S_2

$S_2 = \{ y \mid \forall n$ the Turing machine M_y accepts some string of length n in less than n steps $\}$

State, providing suitable clear and precise justifications, whether at least one of the two sets, S_2 and $\neg S_2$, is semidecidable (remember that $\neg S_2$ denotes the complement of set S_2).

Let us give a more formal definition of set S_2 (where $T_{My}(x)$ denotes the time complexity of machine M_y on input x)

$$S_2 = \{ y \mid \forall n \, \exists x \, (|x|=n \wedge f_y(x) \neq \perp \wedge T_{My}(x) < n) \ ) \ \ \}$$

Let us consider the complement set $\neg S_2$

$$\neg S_2 = \{ y \mid \exists n \, \neg \exists x \, (|x|=n \wedge f_y(x) \neq \perp \wedge T_{My}(x) < n) \ ) \ \ \}$$ or, in words

$\neg S_2 = \{ y \mid \exists n$ s.t. Turing machine M_y does not accept any string of length n in less than n steps$\}$

$\neg S_2$ is semidecidable (by means of dovetail simulation): for a given machine y an exhaustive simulation, over all possible integers n, of the computation of machine M_y for all possible inputs x of length $|x|=n$ can be carried out to check if the computation terminates in less that n steps; if there exists a value n for which a string with such a computation does not exist then that specific value of n is identified and y can be stated to belong to $\neg S_2$.

Notice that a similar reasoning *cannot* be applied to S_2 because of the universal quantification over variable n in the definition of S_2 (a simulation cannot check that the required property is satisfied *for all* the infinite possible values of n).

Is it decidable whether a generic sentence of the Monadic First Order logic on strings is satisfiable or not? Explain clearly your answer.

The satisfiability of an MFO sentence is decidable: it can be REDUCED to the non-emptiness of the language accepted by a Finite State Automaton.

In fact, there is an algorithm that, for any given MFO sentence *F*, builds an equivalent FSA, that is, an FSA that accepts exactly the set of strings that satisfy the sentence *F*.

Therefore the sentence *F* is satisfiable if and only if the FSA equivalent to it accepts a non-empty language; this latter problem is notoriously decidable.

COMPLEXITY – 1

A bit of (gross and earthy) mathematics to evaluate sums

- **arithmetic series:** $\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \in \Theta(n^2)$
- **quadratics series:** $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \in \Theta(n^3)$
- **cubic ...:** $\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4} \in \Theta(n^4)$
- **these can all be proved easily by induction**

- **useful «rule of thumb»:** $\sum_{k=1}^{n} f(k) \cong \int_{1}^{n} f(x)dx$
- **applications:**

since $\int \log x \, dx = x \cdot \log x - x$, **then** $\sum_{k=1}^{n} \log k \cong n \log n - n \in \Theta(n \log n)$

since $\int x^m \log x \, dx = x^{m+1}\left(\frac{\log x}{m+1} - \frac{1}{(m+1)^2}\right)$, **then** $\sum_{k=1}^{n} k \log k \in \Theta(n^2 \log n)$

$\sum_{k=1}^{n} k^h \log k \in \Theta(n^{h+1} \log n)$

- Hence $\log n! = \log\left(\prod_{k=1}^{n} k\right) = \sum_{k=1}^{n} \log k \in \Theta(n \log n)$

- this also follows from **Stirling's approx of** $n!$ $n! \cong \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

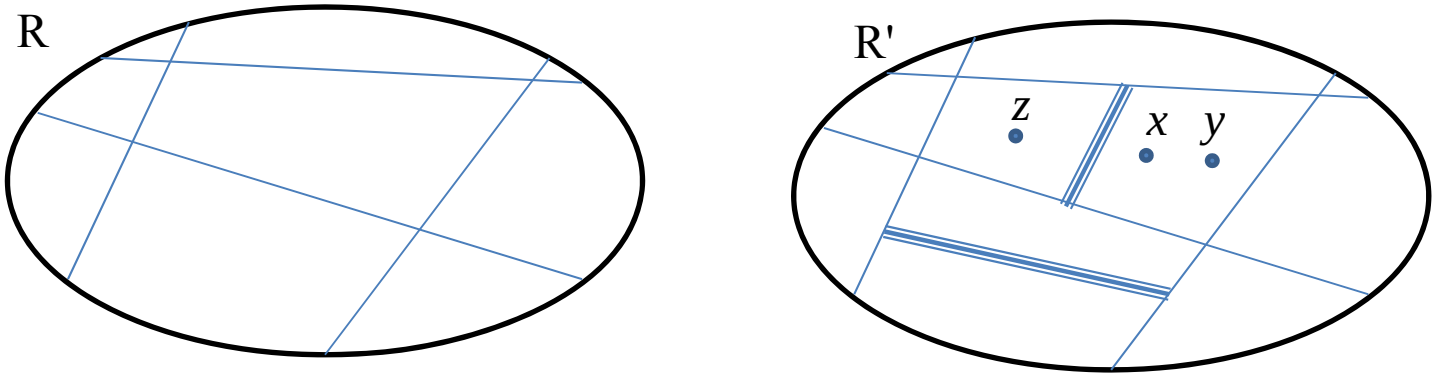
from which $\log n! \cong \frac{1}{2} \log n + n(\log n - \log e)$

hence $\log n! \cong n \log n - n \in \Theta(n \log n)$

relation R' is ***finer*** than relation R if

$$\forall x, y \left((x \, R' \, y) \rightarrow (x \, R \, y) \right)$$

R' is ***coarser*** than R if R is finer than R'



define two relations, one finer and one coarser than Θ,
useful for the study of the complexity of the computation

A relation Θ' finer than Θ is the following:

$f \in \Theta'(g)$ if and only if $\lim_{x \to +\infty} \frac{f(x)}{g(x)} = 1$

equivalence classes of Θ' include functions whose behavior when $x \to +\infty$ differs in a "negligible" way.

E.g., for $f(n) = n + \log(n)$ and $g(n) = n$, $f \in \Theta'(g)$ (hence $f \in \Theta(g)$)
while, for $f(n) = 2n$ and $g(n) = n$, $f \notin \Theta'(g)$ (but $f \in \Theta(g)$)

A relation Θ" coarser than Θ is the following:

$f \in \Theta''(g)$ iff $\exists\, k, h1, h2$ s.t. $\forall n \left(n > k \rightarrow \left(f(n) \leq g(n)^{h1} \wedge g(n) \leq f(n)^{h2} \right) \right)$

Θ" distinguishes algorithms with polynomial complexity from algorithms with exponential or logarithmic complexity

E.g., for $f(n) = n$ and $g(n) = n^3$, $f \in \Theta''(g)$ (but $f \notin \Theta(g)$)
while, for $f(n) = n^3$ and $g(n) = 2^n$, $f \notin \Theta''(g)$ (hence $f \notin \Theta(g)$)

Exercise: prove that both Θ' and Θ" are equivalence relations.

Design a single-tape TM ***M*** that recognizes the language $L(M)=\{ wcw^R \mid w \in \{a, b\}^* \}$. Evaluate its time and space complexity

NB: $\{ wcw^R \mid w \in \{a, b\}^* \}$ can be recognized by a PDA
in time $T(n) = 2n+2 \in \Theta(n)$ and space $S(n) = n \in \Theta(n)$, where $n = |w|$

For the single-tape TM
let's just consider a simulation
and count the moves:

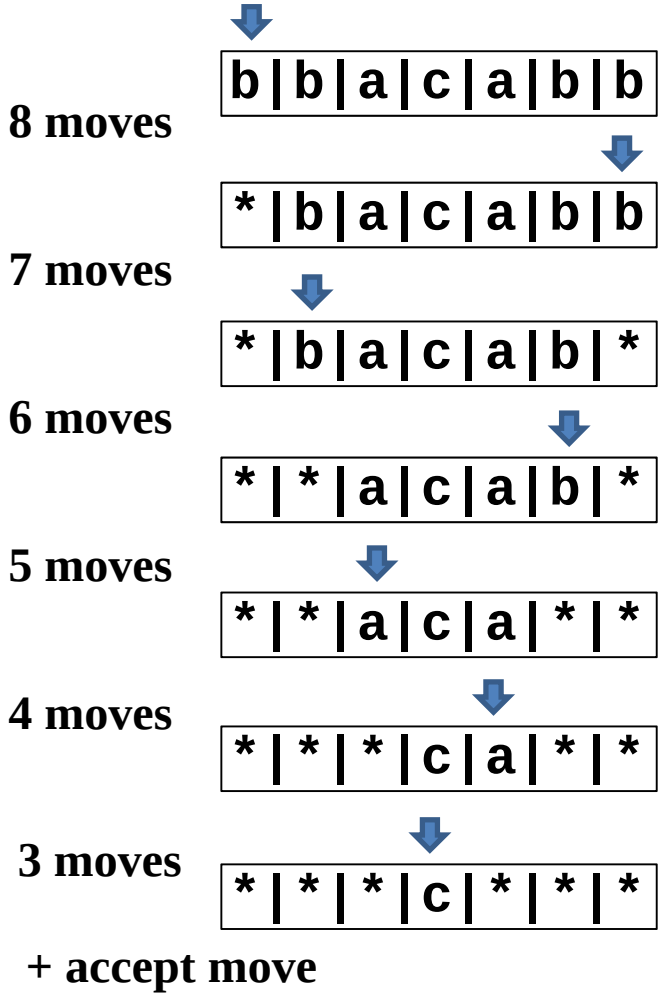
$$T(n) = 1 + \sum_{i=1}^{n} \big((2i + 2) + (2i + 1)\big)$$

$$= 1 + \sum_{i=1}^{n} (4i + 3)$$

$$= 1 + 3n + 4 \sum_{i=1}^{n} i$$

$$= 1 + 3n + 4 \frac{n(n+1)}{2} \in \Theta(n^2)$$

$S(n) = 2n+1 \in \Theta(n)$



Construct a TM M such that $L(M) = \{a^n b^n | n \geq 1\}$ and space complexity is $S_M(n) = \Theta(log(n))$

What is the time complexity of the solution?

- Use 2 tapes for counting ***a***'s and ***b***'s
- Encode counters on the tape in binary
  - to increase by 1 a counter i : $\approx 2 \cdot \log i$ (including repositioning the head)
- Check the exponents for equality by comparing the encodings on the tapes: $\log n$
- Overall dominant term:

$$\sum_{i=1}^{n} \log i \in \Theta(n \log n)$$

Say briefly justifying the answers if the following statements are true or false

(a) Deterministic CF languages can be recognized by a k-tape TM with time complexity $\Theta(n)$ where n is the length of the string.

TRUE. k-tape TM can simulate deterministic PDA by using one memory tape like a stack. Deterministic PDA always have linear time complexity.

(b) Deterministic CF languages can be recognized by a single-tape TM with time complexity $\Theta(n)$ where n is the length of the string.

FALSE. Language wcw^R needs at least a complexity $\Theta(n^2)$ to be recognized by single-tape TM.

Say briefly justifying the answers if the following statements are true or false

(c) Regular languages can be recognized by a single-tape TM with time complexity $\Theta(n)$ where n is the length of the string.

TRUE. Single tape can simulate FSA (without using the tape).
FSA always use liner time to recognize strings.

(d) No CF language can be recognized by a k-tape TM with space complexity $\Theta(log(n))$ or less where n is the length of the string.

FALSE. Regular languages are recognized without using memory. A single-tape TM can simulate the FSA with the control unit and reading the tape from left to right. Technically it needs one extra transition that should go from any original final states of the FSA to a single final state of the TM upon reading a blank symbol. However asymptotically this has no effect.

recall: relation R' is ***finer*** than relation R iff $\forall x, y\ (\ (x\ R'\ y) \rightarrow (x\ R\ y)\ )$

Given two complexity functions $f(n)$ and $g(n)$, $g(n)$ is an ***asymptotic upperbound*** for $f(n)$, in symbols $f(n) \in \mathcal{O}(g(n))$ or $f\ \mathcal{O}\ g$, iff there exist a positive integer k and a positive constant c such that

$$\forall n\ (\ n \geq k \rightarrow f(n) \leq \mathrm{c} \cdot g(n)\ )$$

1. Discuss, providing a suitable justification, possibly supported by simple examples, the following questions
   a. is relation Θ finer than $\mathcal{O}$?
   b. is relation $\mathcal{O}$ finer than Θ?

If $f\ \Theta\ g$ then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some c such that $0 < \mathrm{c} < \infty$, therefore(*) $\exists$k, $\exists c'$ such that $\forall$n (n $\geq$ k $\rightarrow f(n) \leq c' \cdot g(n)$), hence ($f\ \mathcal{O}\ g$) and Θ is finer than $\mathcal{O}$.

For instance $f(n)=3n^2+n$ and $g(n) = 2n^2-n$: both the following hold ($f\ \Theta\ g$) and ($f\ \mathcal{O}\ g$).

On the other hand, however, ($f\ \mathcal{O}\ g$) does no in general imply ($f\ \Theta\ g$); as a counterexample, consider $f(n) = n$ and $g(n) = n^2$: then ($f\ \mathcal{O}\ g$) holds, but ($f\ \Theta\ g$) does not hold.

(*) $\left|\frac{f(n)}{g(n)} - c\right| < \varepsilon$ implies $\frac{f(n)}{g(n)} < c + \varepsilon$ hence $f(n) < (c + \varepsilon)g(n)$

2. Relation $\mathcal{O}$ does not enjoy all the algebraic properties of relation Θ that are useful in the analysis of (time and space) complexity: specifically, which property of the Θ relation is not enjoyed by relation $\mathcal{O}$?

Relation $\mathcal{O}$ is ***not*** symmetric (i.e., ($f \; \mathcal{O} \; g$) does not imply ($g \; \mathcal{O} \; f$)): as a counterexample consider again $f(n) = n$ and $g(n) = n^2$: then it holds ($f \; \mathcal{O} \; g$) but ($g \; \mathcal{O} \; f$) does not hold.

COMPLEXITY – 2

Write a program for a RAM machine that reads an integer value $n \geq 1$ and computes the value n^n; evaluate the obtained time/space complexity with both the cost criterions.

variables used: M[1]: input *n*, M[2]: counter, M[3]: cumulated product

Code		Description	Uniform cost	Logarithmic cost
	READ 0	M[0]=input		l(0)+l(n)
	STORE 1	M[1]=M[0]		l(1)+l(n)
	STORE 2	M[2]=M[0]	5	l(2)+l(n)
	LOAD= 1	M[0] = 1		l(1)
	STORE 3	M[3] =M[0]		l(3)+l(1)
LOOP:	LOAD 1	M[0]=M[1]		l(1)+l(n)
	MULT 3	M[0]=M[0]·M[3]		$l(3)+l(n)+l(n^{i-1})$
	STORE 3	M[3]=M[0]		$l(3)+l(n^i)$
	LOAD 2	M[0]=M[2]	$7n$	l(2)+l(n-i+1)
	SUB= 1	M[0]=M[0]-1		l(1)+l(n-i+1)
	STORE 2	M[2]=M[0]		l(2)+l(n-i)
	JGZ LOOP	if(M[0]>0) then PC=loop		l(n-i)
	WRITE 3	output = M[3]	1	$l(3)+l(n^n)$
	HALT	stop	1	1
		Total time:	$7n+7 \in \Theta(n)$	$\Sigma_{i=1}^{n} l(n^i) = l(n)\Sigma_{i=1}^{n} i \in \Theta(n^2 log(n))$
		Total memory:	$4 \in \Theta(1)$	$2log(n) + 2log(n^n) \in \Theta(nlog(n))$

Sketch in pseudocode an algorithm (to be executed by the RAM) to compute function

$f: \mathbb{N} \to \mathbb{N}$ s.t. $f(x) =$ if x is even then $\sqrt{x^x}$ else x^3

Evaluate the time and memory complexity under the both cost criteria, with reference to both the ***value*** encoded in the input string, and the ***size*** of the input string

Sketch in pseudocode

```
int f (int x){

   result = 1;
   if (x \% 2 == 0){
      m = x \ 2;
      for (int i = 0; i<m;i++){
         result *= x
      }
      return result
   }
   else{
      return x*x*x
   }
}
```

NB:

if x even $\Rightarrow \sqrt{x^x} = x^{x/2}$

input size $n = \log x \quad \Rightarrow \quad x = 2^n$

```
int f (int x){

  result = 1;
  if (x \% 2 == 0){
    m = x \ 2;
    for (int i = 0; i<m;i++){
      result *= x
    }
    return result
  }
  else{
    return x*x*x
  }
}
```

NB:

if n even $\Rightarrow \sqrt{x^x} = x^{x/2}$

input size $n = \log x \quad \Rightarrow \quad x = 2^n$

Constant cost criterion:

time complexity: worst case when loop executed $m = x/2$ times

$$T(n) \in \Theta(m) = \Theta(x) = \Theta(2^n) , \quad n \text{ is the } \textbf{\textit{size}} \text{ of the input}$$

space complexity: $S(n) = \Theta(1)$, number of variables does not depend on input size

```
int f (int x){

  result = 1;
  if (x \% 2 == 0){
    m = x \ 2;
    for (int i = 0; i<m;i++){
      result *= x        ⇐ line 7
    }
    return result
  }
  else{
    return x*x*x
  }
}
```

NB:

input size $n = \log x \Rightarrow x = 2^n$

Logarithmic cost criterion:

Time complexity: line 7 executed m times

each time: read result and x vrbl. and write product back to the result vrbl.

$$\sum_{i=1}^{m}(l(x) + l(x^{i-1}) + l(x^i)) \cong l(x)\sum_{i=1}^{m} i \in \Theta(log(x)m^2) = \Theta(n2^{2n})$$

because $m \in \Theta(x) = \Theta(2^n)$

Space complexity: biggest stored value is $x^{x/2}$ whose size in memory is $\log(x^{x/2})$

$S(n) \in \Theta(x \log x) = \Theta(n\, 2^n)$, because $x \cdot \log x = 2^n \cdot \log 2^n = 2^n \cdot n$

Exercise 1

Consider the language $L = L_1 \cap L_2$, where $L_1 = \{ww^R \mid w \in \{a, b\}^*\}$ and $L_2 = \{a^n b^* a^n \mid n \geq 0\}$

a) List at least the first 5 strings of the lexicographic enumeration of L.

b) Define a grammar, of the least general type, that generates L.

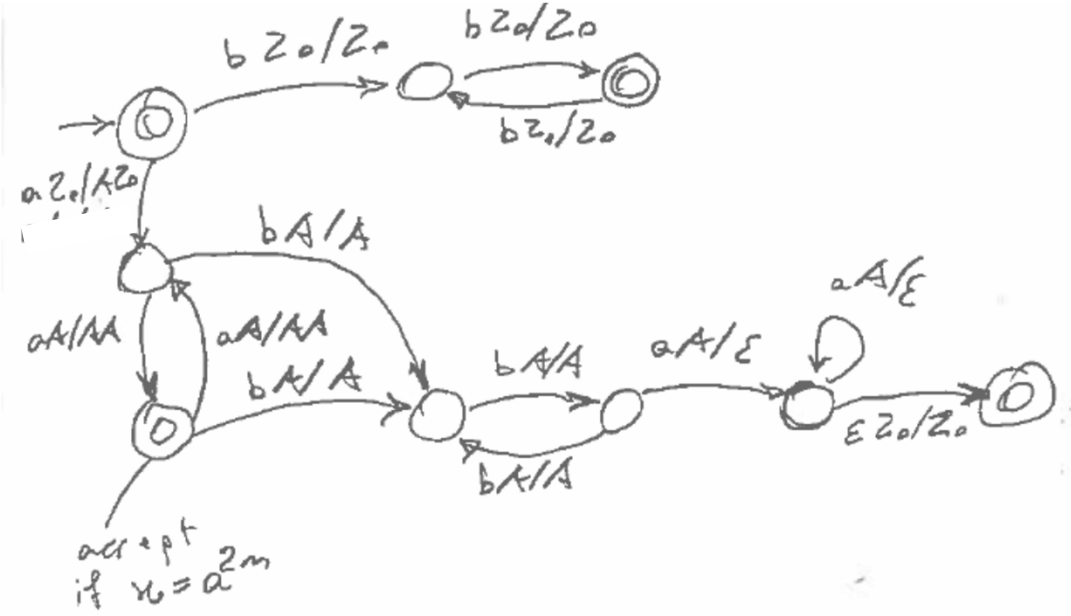
c) Define an automaton, of the least powerful type, that accepts L.

d) Prove that L is not regular, or at least sketch a rigorous argument.

a) ε, aa, bb, aaaa, abba, bbbb, …

b) $S \rightarrow a\ S\ a \mid B \qquad B \rightarrow b\ B\ b \mid \varepsilon$

c) A deterministic PDA suffices.



d) Prove that *L* is not regular, or at least sketch a rigorous argument.

Informally, the automaton accepting L must be able to count the a's in strings of the form $a^n b^+ a^n$. More formally, the fact that L is not regular can be proved by contradiction, based on the Pumping lemma, using an argument similar to the proof that the language $\{ a^n b^n \mid n>0 \}$ is not regular. The proof is a bit more complex than in that case, because various cases on the length and form of the string w to be "pumped" must be considered.

Exercise 4

With reference to language L of Exercise 1:

1. design, preferably in full detail, or at least sketch in a precise and unambiguous way, a Turing machine that accepts L, and evaluate its time and space complexity;
2. sketch, in a precise and unambiguous way, a RAM machine that accepts L, and evaluate its time and space complexity both with reference to the uniform and the logarithmic cost criterion;
3. determine which of the two above computing models (TM and RAM) is more efficient, considering also the logarithmic cost criterion. Provide justifications for your answers.

(Sketchy solutions)

1. A 1-tape TM can work in the same way as the PDA of exercise 1, using the memory tape to simulate the stack of the PDA. The time complexity is $T(n)=n$ in all cases. The space complexity is $S(n)=n$ in the worst case, when the input string is $x=a^n$. Both the time and space complexities are therefore $\Theta(n)$.
2. The RAM machine counts the number of *a*'s in the input; if there is no following *b* it just checks the parity of this number. If the input includes some *b*'s, then it counts them, checks the parity of the number of *b*'s, counts the number of the following *a*'s and verifies that the two parts composed of *a*'s have the same length. Hence under the uniform cost criterion the space complexity is constant (only the storage for the counters is needed) and the time complexity is $T(n)=n$. Adopting the logarithmic cost criterion, the space complexity is however $\Theta(\log n)$ and the time complexity is $\Theta(n \log n)$.
3. Under the uniform cost criterion the RAM machine is preferable to TM, because the time complexity is the same for the two models, while the space complexity is better for the RAM. Adopting the logarithmic cost criterion the RAM is better than the TM for what concerns space, but it is worse w.r.t. time.

Describe a k-tape Turing machine M that, given as input a positive natural number *n*, *coded in unary*, prints in the output tape the first *n* powers of 4 i.e., 4^0, 4^1, 4^2, ... 4^{n-1}, *coded in binary* and each followed by the symbol #. For instance, if the input is 111 (i.e., number 3 coded in unary) then the output is 1#100#10000#.
Determine the temporal and spatial complexities of M, justifying your answer.

To produce the desired values a 1-tape TM suffices; the memory tape stores a list (initially empty) of symbols '0'; the length of the string is equal to the exponent of the power of 2 that must be printed, in binary, at each iteration; at each iteration the TM works as follows.

- Read a symbol from the input
- Write a '1' symbol on the output tape
- Copy all the '0' (while reading them from left to right) from the memory to the output tape, and add the '#' symbol at the end
- Write ***two*** additional '0' symbols on the memory tape (now the memory head is in the rightmost position)
- Go back to the initial, leftmost position on the memory tape, reading the '0' symbols from right to left, then repeat from the initial step.

For the time complexity, the cost of the *i*-th iteration is $\Theta(4i)$, hence the total cost is

$\Theta(\sum_{i=\{1..n\}} 4(i))$, which is $\Theta(n^2)$.

To determine the space complexity only the memory tape is relevant, therefore such complexity is $\Theta(n)$.

Consider the language $L=\{a^n b^m c^{n\times m} \mid n, m>0\}$

1. design, preferably in full detail, or at least sketch in a precise and unambiguous way, a Turing machine that accepts L, and evaluate its time and space complexity;
2. design, preferably in full detail, or at least sketch in a precise and unambiguous way, a RAM machine that accepts L, and evaluate its time and space complexity both with reference to the uniform and the logarithmic cost criterion;
3. determine which of the two above computing models (TM and RAM) is more efficient, considering also the logarithmic cost criterion. Provide justifications for your answers.

The Turing machine reads the *a* symbols and copies them on a (first) memory tape, then it reads the *b* symbols and copies them on a (second) memory tape; the two memory tapes are then used to count the *c* symbols while reading them (e.g., the head on the first memory tape is moved once for every *c* symbol read from the input, and the head of the second memory tape is moved by one position only when the leftmost or rightmost position is reached on first memory tape).

Clearly a single pass on the input is sufficient, and the input is checked, in the worst case, in real-time (plus a small, constant number of moves). Therefore: $T(|x|) = \Theta(|x|)$. The worst case for space complexity occurs when $x \notin L$ or when $n=1$ or $m=1$; in that case, if $\boldsymbol{x}$ is the input then $S(|x|) = \Theta(|x|)$.

The RAM machine reads and counts the a's and the b's, then it computes the value $n \times m$, it reads and counts the c's and finally checks that the number of symbols is correct.

Therefore, with the uniform cost criterion, $S(|x|) = \Theta(1)$, $T(|x|) = \Theta(|x|)$; the RAM is superior to the Turing machine for what concerns space complexity, and it is comparable regarding time complexity.

Adopting the logarithmic cost criterion, $S(|x|) = \Theta(log\ |x|)$, $T(|x|) = \Theta(|x|\ log\ |x|)$; the RAM is still superior to the Turing machine for what concerns space complexity, whereas time complexity is better for the Turing machine.

Consider the language L = $\{w \cdot c \cdot w \cdot c \cdot w \mid w \in \{a, b\}^+\}$ (defined over alphabet A=$\{a, b, c\}$). Describe, informally but precisely, a RAM machine that recognizes language L with a minimal ***space*** complexity according to the constant criterion; also evaluate the space complexity according to the logarithmic cost criterion. For the same machine also determine the time complexity; for this figure does the adopted criterion (uniform or logarithmic) make any significant difference?

One can see the substrings *w* as binary numbers, for instance by considering *a*=0 and *b*=1. One memory cell (say, M[1]) can therefore be used to store the numerical value encoded in *w*, which is computed while reading from the input the symbols composing *w* (multiply by 2 when a new digit is read, add 1 if the digit is *b* etc.).

E.g., string *w=baabb* results into the number

$((((((0) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1) = 19$ (binary encoding: 10011)

b *a* *a* *b* *b*

After reading the first *c* the second substring *w* is encoded similarly (say, in M[2]), then the third one in M[3].

Finally a simple check is made that M[1]= M[2]= M[3].

(Actually, to distinguish strings «w» with a different number of leading zeros, M[1], M[2], and M[3] should not be initialized to 0, but to 1.)

Space complexity is $\Theta(1)$ with the constant criterion;

with the logarithmic cost criterion, $S(n) \cong l(2^n) \in \Theta(n)$, where ***n*** is the **input length**.

The time complexity: linear with the constant criterion

Time complexity with the logarithmic criterion: the dominating factor comes from operation

M[1] := M[1] * 2 + 1

LOAD 1	$l(2^i) + l(1)$	$= i + 1$
MULT =2	$l(2^i) + l(2)$	$= i + 2$
ADD =1	$l(2^{i+1}) + l(1)$	$= i + 2$
STORE 1	$l(2^{i+1}) + l(1)$	$= i + 2$

which adds up to $4i + 7$

Thus in the worst case $T(n) \cong \sum_{i=1}^{n}(4i + 7) \in \Theta(n^2)$.

In fact, time complexity figures change significantly with the logarithmic criterion, as the computed numerical values are exponential w.r.t. the input length.

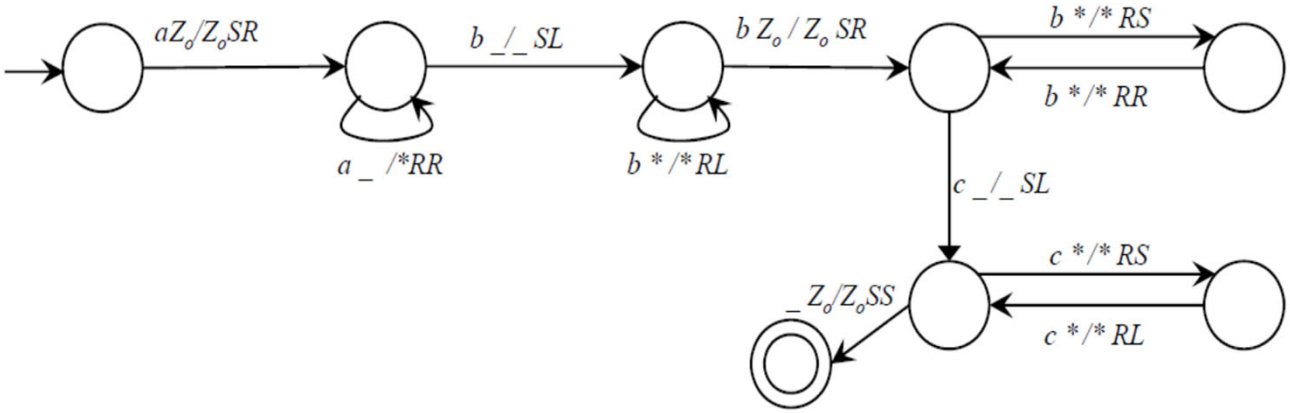
Consider language $L = \{ a^n b^m c^k \mid m, n, k \geq 1 \text{ and } k = 2 \cdot n \text{ and } m = n + k \}$.

Design a Turing machine (preferably having only one memory tape) that recognizes the language with complexity *class* $\Theta(n)$. For this machine, determine in a precise way the time complexity $T(n)$ as a *function* of the input string length.

Sketch a ***single*** tape Turing machine that recognizes the language with a minimal complexity class, providing an adequate justification for your reply.

NB: $m = n+k = n+2n = 3n$, strings are of type $a^n b^{3n} c^{2n}$

The TM below counts the ***a***'s in unary on the memory tape, and uses this count to check the number of the ***b***'s (two memory tape scans) and of the ***c***'s (one leftward memory tape scan)



time complexity function is $T(y) = y + 5$, where $y = |x|$ is the input length

The same language can be accepted by a single tape Turing machine with complexity class $\Theta(n^2)$, because the machine necessarily must go up and down the string, canceling the symbols, a number of times that is proportional to the string length.

Consider the language, over the alphabet {0, 1} that includes strings of the type xy, where $|x| = 5$ and x encodes in binary the position of a character, inside string y, which must be equal to 1.

Describe, in a clear, complete and unambiguous way, a ***single*** tape Turing machine and a RAM machine that accept the above language; evaluate the space and time complexity using all the available cost criteria.

The length of string x is fixed, hence the largest possible denoted position is 11111 in binary, that is, 31 in decimal. Hence the correct strings can be identified by considering only the prefix of length 5+31; and the number of such prefixes is finite, therefore the language can be accepted by a finite state automaton, which terminates after 36 steps at most.

Thus the TM and the RAM can work in the same way, simulating a finite state automaton, with the following complexity figures.

Single tape TM: $T(n) = \Theta(1)$, $S(n) = \Theta(n)$ (the latter because the input string is on the tape)

RAM, with both constant and logarithmic cost criteria: $T(n) = \Theta(1)$, $S(n) = \Theta(1)$.

Design a Turing machine that accepts ***in real time*** the language

$$\boldsymbol{L} = \{\, \boldsymbol{x\#y\#} \mid \boldsymbol{x, y} \in \{\boldsymbol{a, b, c, d}\}^{+} \text{ and } \boldsymbol{x} \text{ is an anagram of } \boldsymbol{y} \,\}$$

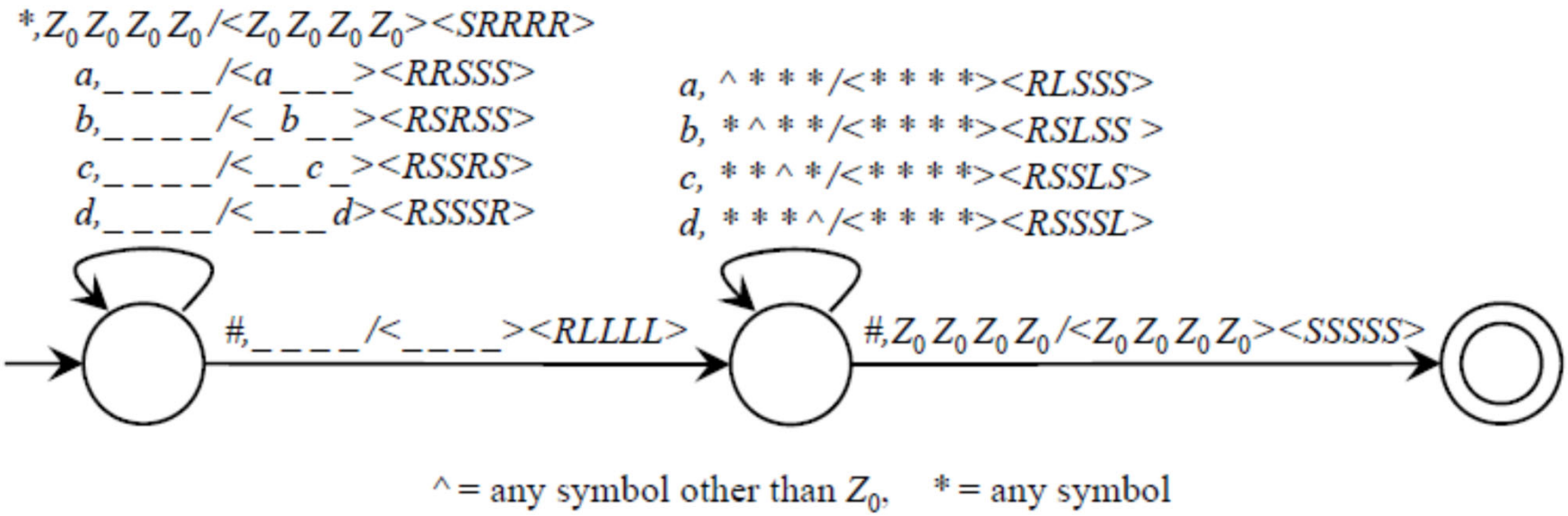
(Let us remind that a string x is an anagram of a string y if the two strings contain the same letters, each letter having in each string the same number of occurrences, e.g., as in *badacdbbd* and *bdcadbadb*).

Sketch a RAM machine that accepts the same language, and compare the time and space complexity of the two machines, according to both the constant and the logarithmic cost criterion.

For the Turing machine below the space complexity for an input string w, with $|w|=n$, is

- $S_T(n) = n$ if $w \notin L$
- $S_T(n) = (n\text{-}2)/2$ if $w \in L$

hence it is $\Theta(n)$.



The RAM machine scans the input string while counting the occurrences of the various symbols in the substrings x and y, and finally compares them. Its time complexity according to the constant criterion is $\Theta(n)$, but adopting the logarithmic cost criterion the time complexity is greater than n, because of the steps necessary for computing and comparing the values of the counters: it is $\Theta(n\ log(n))$, therefore worse than that of the Turing machine. The space complexity is determined by the variables storing the counters, therefore it is constant with the constant criterion, and $\Theta(log(n))$ with the logarithmic cost criterion.

The advantage of the Turing machine over the RAM machine concerning the time complexity is obtained at the price of an increased space complexity due to the unary encoding of the counters in the TM tapes: a typical space-time tradeoff.